A system for describing and deciding properties of regular languages using input altering transducers

By

Krystian Dudzinski

A Thesis Submitted to
Saint Mary's University, Halifax, Nova Scotia
in Partial Fulfillment of the Requirements for
the Degree of Master of Science in Applied Science

May, 2011, Halifax, Nova Scotia

Approved:	Dr. Stavros Konstantinidis Supervisor Department of Mathematics and Computing Science
Approved:	Dr. Mark Daley External Examiner Department of Computer Science University of Western Ontario
Approved:	Dr. Shyamala C. Sivakumar Supervisory Committee Member Department of Finance, Information Systems, and Management Science
Approved:	Dr. John Irving Supervisory Committee Member Department of Mathematics and Computing Science
Approved:	Dr. Paul Muir Program Representative
Approved:	Dr. Pawan Lingras Graduate Studies Representative
Date:	July 19, 2011

Your file Votre référence
ISBN: 978-0-494-83259-2
Our file Notre référence
ISBN: 978-0-494-83259-2

A system for describing and deciding properties of regular languages using input altering transducers

By

Krystian Dudzinski

A Thesis Submitted to
Saint Mary's University, Halifax, Nova Scotia
in Partial Fulfillment of the Requirements for
the Degree of Master of Science in Applied Science

May, 2011, Halifax, Nova Scotia

Approved: _____
Dr. Stavros Konstantinidis
Supervisor
Department of Mathematics and Computing Science

Approved: _____
Dr. Mark Daley
External Examiner
Department of Computer Science
University of Western Ontario

Approved: _____
Dr. Shyamala C. Sivakumar
Supervisory Committee Member
Department of Finance, Information Systems,
and Management Science

Approved: _____
Dr. John Irving
Supervisory Committee Member
Department of Mathematics and Computing Science

Approved: _____
Dr. Paul Muir
Program Representative

Approved: _____
Dr. Pawan Lingras
Graduate Studies Representative

Date: _____

Acknowledgements

I would like to take this opportunity to express my sincere gratitude to all the people who, both directly and indirectly, made this thesis possible.

My supervisor, Dr. Stavros Konstantinidis, a great professor, a great mentor, and a great person. For his commitment, patience, and guidance. It was a pleasure and an honour to work with him.

The supervisory committee, Dr. Shyamala C. Sivakumar and Dr. John Irving for their contribution.

The external examiner, Dr. Mark Daley, for the thorough revision of this thesis, an insightful feedback, and helpful comments.

The Department of Mathematics and Computing Science, the Faculty of Science, and the Faculty of Graduate Studies and Research for giving me the opportunity to study at Saint Mary's University and the financial support.

All the faculty and the staff in the Department of Mathematics and Computing Science. In particular I would like to thank Dr. Paul Muir and Dr. Pawan Lingras. They are both wonderful professors, truly committed to their students, both inside and outside the classroom.

My colleagues and fellow graduate students who significantly contributed to making my experience at Saint Mary's University memorable.

The faculty and staff of Saint Mary's University. Tom Webb, for the opportunity to work with him. Janet Stalker, for her help and commitment to improving my communication skills. Dr. Dawn Jutla for the opportunity to look at computer science from a different perspective.

My friends and colleagues that were there for me when I needed them.

Finally I would like to thank my family and loved ones. It is their faith and support that helped me to get where I am right now. My parents, my grandmother, Kasia, my friends, thank you!

Abstract

A system for describing and deciding properties of regular languages using input altering transducers

By

Krystian Dudzinski

Abstract: We present a formal method for describing and deciding code related properties of regular languages using input altering transducers. We also provide an implementation of that method in the form of a web application. We introduce the concept of an input altering transducer. We show how to use such transducers to describe properties of languages and present examples of transducers describing some well known properties (like suffix codes, prefix codes, infix codes, solid codes, and others). We discuss some limitations of our method. In particular, all properties that can be described using input altering transducers are 3-independence properties. We also give an example of a 3-independence property that cannot be represented using a transducer. We explain how our method is a specialisation of a more general method based on language in-equations. We also discuss the relation between our method and a method that uses sets of trajectories to describe properties. In particular, we show how, for any given set of trajectories describing some property, to build an input altering transducer describing the same property. We introduce the concept of counterexample, which is a pair of words that, if a given language does not belong to a given property, illustrate that fact. We show how we can incorporate extracting such counterexample into our method. Finally, we provide some details on the implementation and usage of the web application that was built as a part of this research.

July 19, 2011.

Table of Contents

Chapter 1

Introduction

1.1 About our research

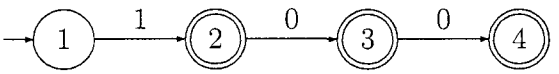
Information is contained in various structures, like pictures, sound, text, speech, database records, and many more. Regardless of how the concept of information is defined, we often need means of storing and transmitting those structures. For instance, consider speech. A means of storing and transmitting human speech is writing it down. There are two aspects to that operation: something that we would consider a hardware, which might be simply pen and paper, and some form of representing speech in a graphical form in a way that it can be transformed back to its original form. For the second aspect we require some form or representation that is suitable for the hardware used, some form of a language. In computer science we use the concept of languages, in particular formal languages, for representing information. A formal language is a set of words over some alphabet. Given an alphabet $\Sigma = \{0, 1\}$, an example language over this alphabet might be: $L_1 = \{1, 10, 100\}$. However, we cannot just use an arbitrary set of words for representing information. Particular applications, like data communication or DNA computing, require using languages that satisfy certain constraints. For example some application in data communication may require a language in which no word is a prefix of any other word in this language.

An example would be a language consisting of three words: $L_2 = \{1, 01, 001\}$. We say that the language L_2 satisfies the prefix property. Note that the language L_1 does not satisfy the prefix property because 1 is a prefix of 10. The prefix property is only one of many properties of languages. Investigating those properties is an important branch of theoretical computer science. In particular, we investigate methods for testing if a given language satisfies a particular property.

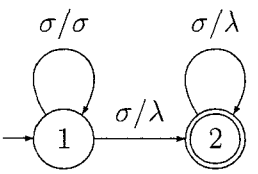
There is an abundance of algorithms in the literature for deciding language properties. Most of those algorithms are specific to the property being tested. Lately, however, there has been a trend toward developing uniform methods for deciding and describing properties of languages. Each of those methods introduces a different approach for expressing properties. However, to our knowledge there is a lack of available implementation of any of those methods. We attempt to fill that gap.

In this thesis we introduce our method for describing and deciding properties of languages. Our method requires a language in the form of a finite automaton (Fig. 1.1a) and is therefore applicable only to regular languages. The properties in our method are described using transducers (Fig. 1.1b), which are well established objects in theoretical computer science.

Provided with a language in the form of a finite automaton and a property in the form of a transducer, we can use our method to test if the given language satisfies that property (Fig. 1.2)



(a) An automaton representing a language $L_1 = \{1, 10, 100\}$. Circles denote states of the automaton. In particular a double circle denotes a final state and an arrow with no origin state denotes a start state. Arrows denote transitions and the labels by transitions denote the input symbols.



(b) A transducer describing the prefix property. The main difference from an automaton is that every transition is labelled with a pair of *input/output* symbols.

Figure 1.1: Sample input machines for our method.

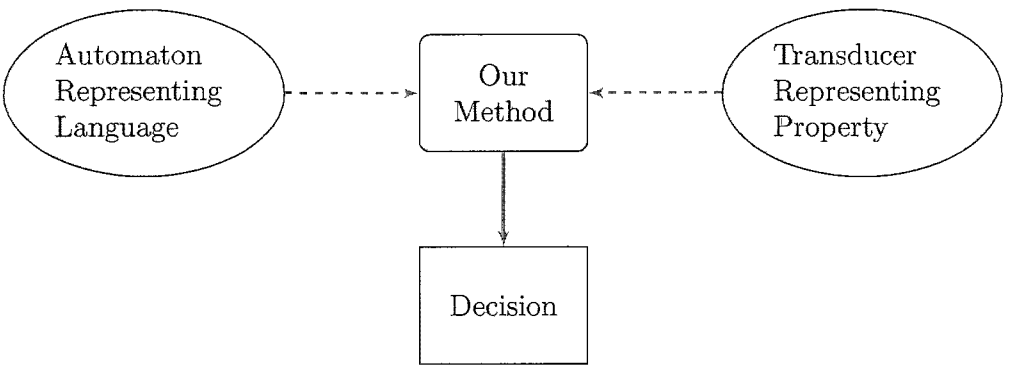


Figure 1.2: A conceptual diagram showing the functionality of our method

As a part of this research we deliver an implementation of our method in the form of a web application. This application is hosted on the university server and is available for unrestricted access.

1.2 Thesis structure

Following we present an overview of the structure of this thesis.

In Chapter 2 we present basic definitions and conceptual tools that are crucial for our research. We define the concepts of regular language, automaton, transducer, and the Cartesian Product construction.

In Chapter 3 we take a look at known properties of languages. We focus on code related properties. In the same chapter we review existing methods for describing and deciding properties of languages. In particular, we look at methods based on implication conditions, regular trajectories, and language in-equations

Chapter 4 provides a detailed description of our method for defining and describing properties of languages. We explain how input altering transducers are used to describe properties in the context of our method. We also show how the product construction is applied in our method. Finally we present example transducers describing many known properties of regular languages.

In Chapter 5 we discuss some limitations of our method. We establish what type of properties our method can be applied to. In particular, all properties that can be described using our method fall in to the class of independence properties. We take a closer look at the concept of input altering transducers, and we provide an example and a proof of an independence property that can not be represented using an input altering transducer.

Chapter 6 covers further details regarding our method. We investigate the complexity of our method, the relation between a transducer and its inverse in the context of our method, and dependencies between our method and some other existing methods presented in Chapter 3. We also introduce the concept of a counterexample, that is a pair of words with particular relation to each other that is a subset of a tested language

if that language does not satisfy the given property. We explain how extracting such conuterexample works in our method.

One of the main goals of our research was to provide a usable implementation of our method in the form of a web application. Chapter 7 contains details of that implementation. We discuss both the implementation of algorithms as well as the user interface. Finally, the format for describing automata and traducers in our system is explained and some examples are provided.

The final chapter contains a summary of our work and a direction for future research and implementation.

Chapter 2

Basic Concepts

In this chapter we introduce the basic concepts and notation used to investigate regular languages.

2.1 Set Notation

If a set is denoted by S then $|S|$ denotes the cardinality of that set, and 2^S denotes all possible subsets of that set.

2.2 Alphabet, Word, Language

An alphabet, denoted by Σ, is a finite nonempty set. Elements of an alphabet are called symbols. A word is a finite sequence of symbols over a given alphabet Σ. A language is a set of words. It may contain the empty word, which is denoted by λ. The basic operation on words is concatenation. If u and v are words then their concatenation is uv and consists of the symbols of u followed by those of v. Obviously, $u\lambda = \lambda u = u$.

***Example**: An alphabet consisting of two symbols : $\Sigma = \{a,b\}$. An example language over that alphabet consisting of 3 words: $\{abb,\ ab,\ a\}$. Also, if $u = abb$ and $v = ab$,*

then $uv = abbab$.

2.3 Regular Languages

The set of all possible languages is denoted by 2^{Σ^*}. Among those languages we can distinguish four hierarchical groups, depending on the type of grammar or machine used to describe the languages [7]:

- nonrestricted grammars - accepted by Turing machines
- context-sensitive grammars - accepted by linear bounded Turing machines
- context-free grammars - accepted by pushdown automata
- regular grammars - accepted by finite automata

In this paper we investigate languages from the last group. By definition a regular language is a formal language accepted by a finite state automaton.

2.4 Finite State Automata

We base our definition of a Finite State Automaton (FSA) on the one given in [42]. The only difference is that our definition of FSA allows multiple start states. It is known that such modification does not affect the computational power of the model. A finite state automaton (also referred to as a finite automaton) is a quintuple (Q, Σ, δ, S, F) where:

- Q is the finite set of states

- Σ is the input alphabet
- δ is the transition function or the transition set
- S is the set of start states
- F is the set of final states

An FSA has a graphical representation (Fig. 2.1). States are denoted by circles, transitions are denoted by arrows with labels that connect states. Start states are denoted by an arrow with no origin state and final states are denoted by double circles.

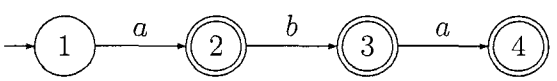


Figure 2.1: An example FSA with $Q = \{1, 2, 3, 4\}$, $\Sigma = \{a, b\}$, start states $S = \{1\}$ and final states $F = \{2, 3, 4\}$.

We say that an automaton accepts a word if that word is created by concatenating the transition labels encountered when moving along a path from a start state to a final state. Such path is called an accepting path.

Example: *Let us consider the automaton in Fig. 2.1.* 1 *is the single start state in this automaton. The first transition, labelled with a, takes us from state* 1 *to state* 2 *while accepting symbol a. From state* 2 *the transition labelled with b takes us to state* 3 *which is a final state. Along the path from the start state to one of the final states*

the automaton accepted the word ab. Therefore ab is one of the words accepted by this automaton.

We say that an automaton A accepts or represents a language L, if every word of this language is accepted by A and every word accepted by A belongs to L.

Example: *Let us again consider the automaton in Fig. 2.1. This automaton accepts the following set of words:* $\{a, ab, aba\}$. *Therefore, this automaton represents the language consisting of these three words.*

We define the size of an automaton A to be:

$$|A| = |Q| + |\delta| \tag{2.1}$$

where $|Q|$ is the number of states in A, and $|\delta|$ is the number of transitions in A.

There are two types of automata that we often deal with in the literature [42]:

- Deterministic Finite Automaton (DFA)
- Nondeterministic Finite Automaton (NFA).

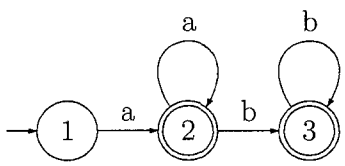
In a DFA the next state is always uniquely determined by the current state and input symbol. Hence the transition function has the following form:

$$\delta : Q \times \Sigma \to Q \tag{2.2}$$

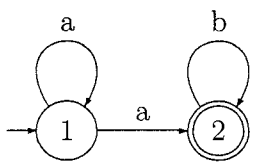
An NFA is a generalization of a DFA. Formally an NFA is defined in the same way as a DFA. The difference is in the transition function. In an NFA, it has the following form:

$$\delta : Q \times \Sigma \rightarrow 2^Q \tag{2.3}$$

where 2^Q denotes all subsets of the states in an NFA. This means there might be more than one transition from a state with the same label. Consequently, the next state is not uniquely determined by the pair of the current state and the input symbol.



(a) DFA - every state has at most one transition going out with the same label



(b) NFA - more than one transition with the same label originating from state 1

Figure 2.2: Two equivalent Finite Automata: deterministic (2.2a) and nondeterministic (2.2b).

Both types of automata accept the same family of regular languages [34]. Furthermore, two automata are called equivalent if they accept the same language. For every NFA we can build an equivalent DFA [42], however for a particular NFA with n states an equivalent DFA can have up to 2^n states.

In order to accept a given word, an automaton has to perform a certain computation, which is simply an accepting path. We denote that computation by listing the transitions that are visited during the computation in the order that they are visited. We

use the following notation to describe a single transition in an automaton:

$$(p, \sigma, q) \tag{2.4}$$

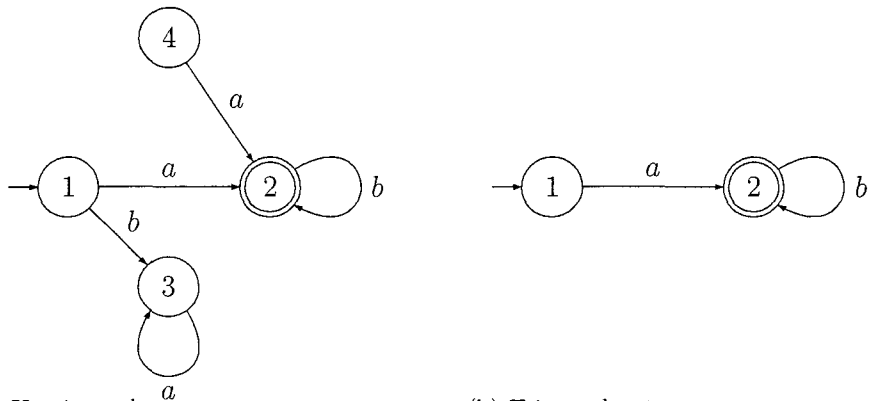
where p is the origin state, q is the destination state, σ is the label of that transition, and $q \in \delta(p, \sigma)$. For example the automaton in Fig. 2.2b in order to accept the word *aaab* has to perform the following computation:

$$(1, a, 1), (1, a, 1), (1, a, 2), (2, b, 2)$$

An automaton can contain some states that do not belong to any accepting path. Such states are either unreachable from any start state of the automaton via any computation, or none of the final states can be reached from that state regardless of the input word. The process of removing such states and all corresponding transitions from an automaton is called trimming. It does not affect the language accepted by an automaton and can decrease the size of such automaton.

***Example**: An automaton accepting the language $L = ab^*$, before and after trimming (Fig. 2.3).*

If a nondeterministic automaton contains any transitions labelled with the empty words then it is called λ-NFA. For every λ-NFA we can build an equivalent NFA with no λ-transitions. Methods for building NFA based on λ-NFA are well known and



(a) Untrimmed automaton (b) Trimmed automaton

Figure 2.3: An automaton accepting the language $L = ab^*$ untrimmed (left) and trimmed (right).

therefore we will not introduce them in this thesis. However, we do refer the reader to [13] and [42] where such a methods are discussed.

There are other well known operations on regular languages crucial for our methods like Cartesian Product.

2.5 Cartesian Product of Two Automata

Using Cartesian product operation we can construct an automaton that represents the intersection of two languages. If an automaton A_1 accepts the language L_1 and an automaton A_2 accepts the language L_2 we can construct an automaton A_3 accepting the language L_3 such that:

$$L_1 \cap L_2 = L_3 \tag{2.5}$$

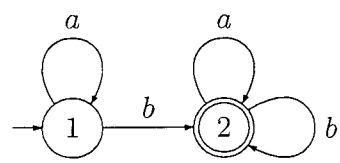
A classical method for building such automaton is a Cartesian Product operation [13].

Let us consider two regular languages. Now let us assume that those two languages

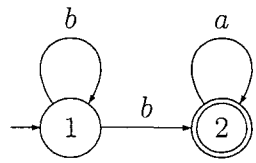
are represented by two NFAs (no λ-transitions), A_1 and A_2. For every transition (p_1, σ, q_1) in A_1 and (p_2, σ, q_1) in A_2 (i.e. transitions with matching labels) we add a transition $((p_1, p_2), \sigma, (q_1, q_2))$ to automaton A_3. The set of start states in A_3 includes all states that are pairs of start states from automata A_1 and A_2. In other words a state (p_1, p_1) becomes a start state in A_3 if and only if p_1 is a start state in A_1 and p_2 is a start state in A_2. Similarly, the set of final states in A_3 consists of states that are pairs of final states of A_1 and A_2. That means a state (q_1, q_2) becomes final state if and only if q_1 is a final state in A_1 and q_2 is a final state in A_2.

The product construction can be inefficient, as in some cases a created transition is not accessible from a start state. In such a case the transition does not affect the represented language but contributes to the size of the product automaton. Therefore a more efficient solution is to build a product automaton incrementally [13]. In this approach we start by defining the set of starting states. Next we build transitions going out of start states. At the same time we build a set of states that are reachable from any start state. In the next iteration we only build transitions going out of this set of reachable states and expand the set of reachable states with states that are constructed in that iteration. The procedure ends when all reachable transitions in the new automaton have been constructed. The automaton constructed in this way is equivalent to the automaton built directly by definition, but it is often smaller.

Example: *Let us consider two automata, A_1 (Fig. 2.4a) and A_2 (Fig. 2.4b).*



(a) The automaton A_1



(b) The automaton A_2

Figure 2.4

In the first step we identify the the set of start states of the product automaton (Fig. 2.5).

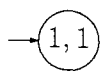


Figure 2.5: A start state of the product automaton

Then we build a set of transitions going out of start states (Fig. 2.6).

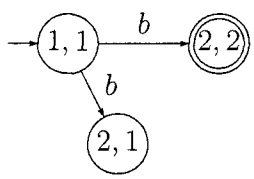


Figure 2.6: Adding transitions from a starting state

We can now identify a set of states that are directly reachable from any start state (in this case state $(1,1)$*). Both state 2 in* A_1 *and state 2 in* A_2 *are final states and therefore the state* $(2,2)$ *becomes a final state in the product automaton. Next we go to the first state in the set of reachable states that have not yet been processed and build all transitions going out from that state (Fig. 2.7).*

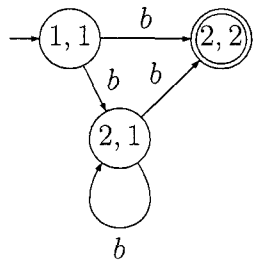


Figure 2.7: Adding transitions going out of state $(2, 1)$

There are no new states added in this step because all added transitions lead to states that have been already established. In the final step we build a transition going out of a final state (Fig. 2.8).

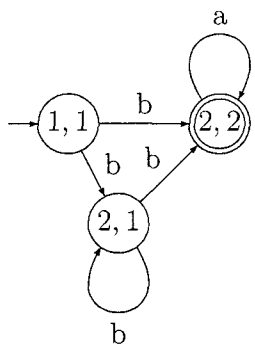
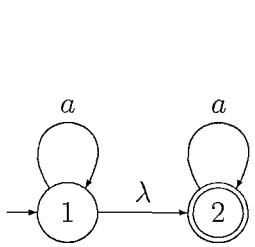
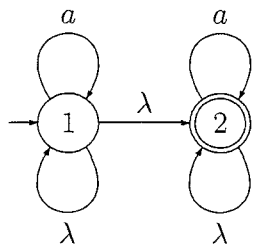


Figure 2.8: Adding transitions that go out of state $2, 1$

What if one of the automata is not in a λ-free form? One approach is to transform it into its λ-free equivalent. A second approach, the one that we follow in this dissertation, is to expand both automata with a set of special λ-transitions (we refer to those transitions as self λ-transitions) that do not affect the accepted language (Fig. 2.9). Every added self λ-transition ends in the same state as it starts. After we apply the modification we proceed with the product construction in the same way as we did in case of λ-free automata, treating λ as any other symbol in the language.



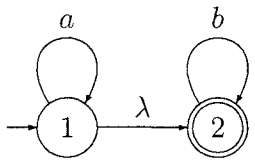
(a) An automaton with λ-transitions



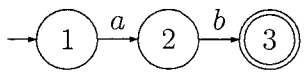
(b) The expanded automaton with self λ-transitions

Figure 2.9

Example: *Let us consider two automata: A_1 (Fig. 2.10a) and A_2 (Fig. 2.10b).*



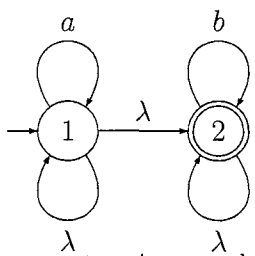
(a) An automaton A_1



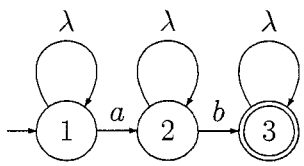
(b) An automaton A_2

Figure 2.10

Because automaton A_1 has a λ-transition we need to expand both automata involved in product construction with self λ-transitions (Fig. 2.11).



(a) The automaton A_1 expanded with self λ-transitions



(b) The automaton A_2 expanded with self λ-transitions

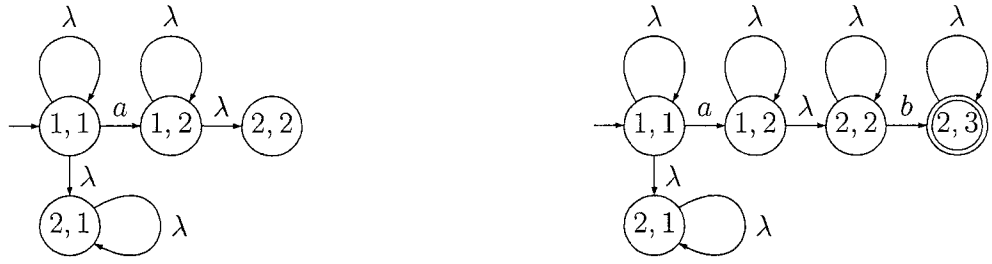
Figure 2.11

Now can we build the product automaton using the incremental approach described before. We start by identifying the set of starting states (Fig. 2.12a). Next we build transitions going out of the start state (Fig. 2.12b). We keep expanding the set of reachable states by building transitions only from reachable states (Fig. 2.12c). The

process is complete once we have processed all the reachable transitions (Fig. 2.12c).

(a) A start state of the product automaton

(b) Transitions going out of the start state



(c) Building new transitions

(d) The final product

Figure 2.12

2.6 Transducers

A transducer (Fig. 2.13) is a 6-tuple $(Q, \Sigma, \Gamma, \delta, S, F)$ where:

- Q is the finite set of states
- Σ is the input alphabet
- Γ is the output alphabet
- δ is the set of transitions

- S is the set of start states
- F is the set of final states

It extends the concept of NFA. For every accepted word, it produces a set of possible output words. The transition set in a transducer has the following form:

$$\delta \subseteq Q \times (\Sigma \cup \{\lambda\}) \times (\Gamma \cup \{\lambda\}) \times Q \tag{2.6}$$

A transducer represents a relation between words. A transducer is called functional or single valued if, for every input word, it produces at most one output word. Nondeterministic Sequential Machine (NSM) is a particular kind of transducer where every transition is limited to one input and one output symbol.

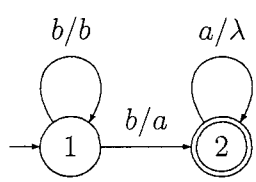


Figure 2.13: An example of a transducer

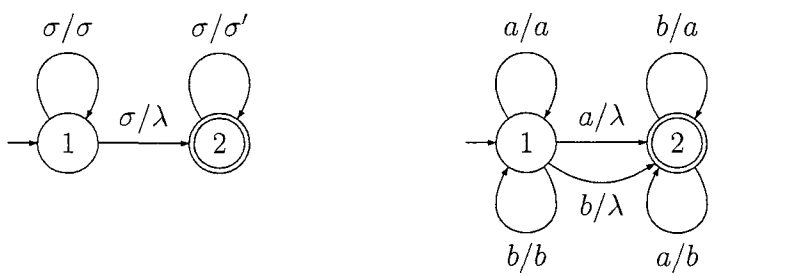
We sometimes present transducers in shorthand form using σ to represent all the possible letters from an alphabet. For instance, if a presented transducer has a transition in the form $(p, \sigma/\lambda, q)$ and we want to apply it where both the input language and the output language are over the alphabet $\{a, b, c\}$ this transition is a shorthand for the following set of transitions: $(p, a/\lambda, q)$, $(p, b/\lambda, q)$, $(p, c/\lambda, q)$. The transition $(p, \sigma/\sigma, q)$ is a shorthand for $(p, a/a, q)$, $(p, b/b, q)$, $(p, c/c, q)$. Finally the transition

$(p, \sigma/\sigma', q)$ is a shorthand for $(p, a/b, q)$, $(p, a/c, q)$, $(p, b/a, q)$ $(p, b/c, q)$, $(p, c/a, q)$, $(p, c/b, q)$. An example of a transducer for the particular alphabet is presented in Fig. 2.14.

Similarly to an automaton, a transducer performs computations. In case of a transducer every accepting computation accepts one word and produces one word. Therefore one computation represents a pair of an input and an output word. Similarly to the case of an automaton, we denote a computation by listing all visited transitions in the order that they were visited. For example for the transducer in Fig. 2.13 to accept the word bba and produce the word ba, this transducer can perform the following computation:

$$(1, b/b, 1), (1, b/a, 2), (2, a/\lambda, 2)$$

In general, $t(u)$ denotes the set of all possible outputs of t on input u. If we assume t to be the transducer in Fig. 2.13 then $ba \in t(bba)$, and also $t(ab) = \emptyset$.



(a) A transducer in shorthand notation (b) The same transducer over the alphabet $\{a, b\}$

Figure 2.14: A transducer in shorthand form and its interpretation over the alphabet $\{a, b\}$

We can perform the trimming operation on transducers. Such a procedure removes transitions and states in a way that the relation represented by a transducer is not affected.

2.7 Cartesian Product of an Automaton and a Transducer

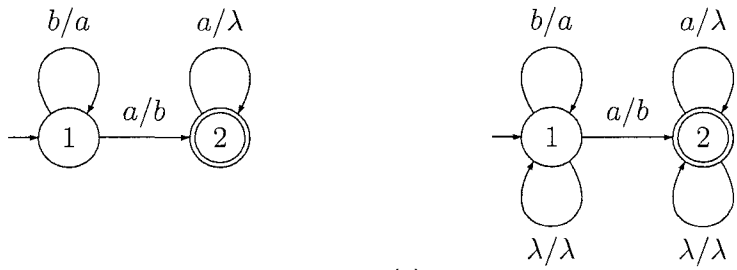
We can apply the relation represented by a transducer not only to a particular word but also to a whole language. Formally such an operation is denoted as follows:

$$t(L_1) = L_2 \tag{2.7}$$

where L_1 is the original language, t is the transducer representing a relation, and L_2 is the language representing all the words that are a result of applying relation t to any word in L_1. We construct an automaton A_2 that represents L_2 using a product-like operation on a pair of an automaton A_1 representing L_1 and a transducer t. For every transition (p_1, σ, q_1) in A_1 and $(p_2, \sigma/\gamma, q_2)$ in t, that is a pair of transitions with matching input labels, we build a transition $((p_1, p_2), \gamma, (q_1, q_2))$ in A_2 where γ is the output symbol of the corresponding transducer transition. We identify the set of start states and final states of A_2 in the same way we do in a case of a product operation of two automata.

Transducers can include transitions with λ as both input or output symbols. A given transducer, unlike an NFA, might not have a λ-free equivalent. Therefore, before applying the product construction, we have to expand both the automaton and the

transducer with self λ-transitions (Fig. 2.15). Transitions added to the transducer have λ as both input and output. Note that adding self λ-transitions does not affect the relation represented by a transducer.

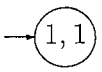


(a) A transducer with λ-transitions

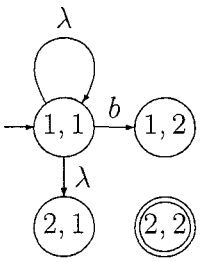
(b) The same transducer expanded with self λ-transitions

Figure 2.15: An example transducer expanded with self λ-transitions

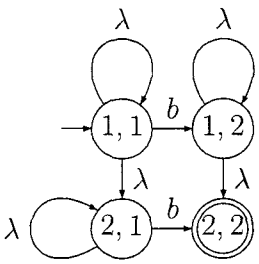
***Example**: Fig. 2.16 presents the process of building the product of the automaton in Fig. 2.9b and the transducer in Fig. 2.15b*



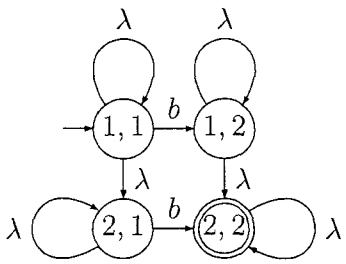
(a) Start state of the product automaton



(b) Transitions going out of the start state



(c) Building new transitions



(d) The final product

Figure 2.16: The product of the automaton in Fig. 2.9b and the transducer in Fig. 2.15b

Chapter 3

Language Properties and Description Methods

We define a language property to be a set of languages. In practice we usually choose a set of languages that posses a certain common characteristic. We say that a language satisfies a property if it belongs to the set of languages defining that particular property.

A common application of regular languages is encoding information. However, not every language may be used for encoding. There are different properties that define whether a language is an appropriate mean of encoding information. An important type of properties are code related properties. There are a few existing methods for deciding and describing such properties but they come with limitations.

In the first part of this chapter, we take a look at some classical and more recently established properties of languages. In the second part, we provide an overview of some existing general and formal methods for describing and deciding properties.

3.1 Unique Decodability

Unique decodablity is one of the most important and widely studied properties. This property consists of languages that we call codes. Satisfying unique decodability is

usually a necessary condition for a language to be a lossless information transfer tool. In the literature, this property is also referred to as the one-to-one property [29] or unique decipherability [16]. We say that that a language belongs to that property if every message over this language has a unique factorization over its code words.

Example: *Let us consider the language* $L = \{0, 01, 10\}$. *The message* 010 *has two possible factorizations over this language:* (0)(10) *and* (01)(0). *Thus* L *is not a code.*

3.2 Code Related Properties

There is no unique definition for what a code related property is. In [21] and [25] code related properties are the independence properties (see Section 5.2 for more information on independence). Some classical code related properties are ([5] and [29]):

- Prefix codes: no word in the language is a prefix of any other word in the same language (*Example:* $L_1 = \{01, 11, 000\}$ *is a prefix code but* $L_2 = \{01, 11, 010\}$ *is not because* 01 *is a prefix of* 010).

- Suffix codes: no word in the language is a suffix of any other word in the same language (*Example:* $L_1 = \{01, 11, 000\}$ *is a suffix code but* $L_2 = \{10, 11, 010\}$ *is not because* 10 *is a suffix of* 010).

- Bifix codes: both prefix and suffix properties are satisfied (*Example: Again* $L_1 = \{01, 11, 000\}$ *is a bifix code but* $L_2 = \{10, 11, 010\}$ *is not because* 10 *is a suffix of* 010).

More recently investigated code related properties are:

- Infix codes [19]: no word in the language appears as part of any other word in the same language (*Example:* $L_1 = \{01, 11, 000\}$ *is an infix code but* $L_2 = \{10, 11, 01101\}$ *is not because* 11 *is part of* 01101).

- Overlap-free languages [6, 15]: no proper prefix of one word is at the same time proper suffix of any word in the language (*Example:* $L_1 = \{10, 100, 1000\}$ *is an overlap free language but* $L_2 = \{101, 100, 1010\}$ *is not because prefix* 10 *of* 100 *is a suffix of* 1010.)

- Solid codes [40, 24, 22] : the infix property is satisfied and the overlap-free property is satisfied (*Example:* $L_1 = \{0111, 01011, 010011\}$ *is a solid code but* $L_2 = \{0111, 1011, 01011\}$ *is not because a suffix of* 1011 *is a prefix of* 0111)

- Thin languages [32] - There are no two words of the same length in the language (*Example:* $L_1 = \{0, 01, 011\}$ *is a thin language but* $L = \{1, 01, 10\}$ *is not*)

Prefix and suffix codes have been well investigated since early days of computer science. Later, the attention of the scientific community was drawn to infix codes and more recently to solid codes. Of course this is only a small subset of existing properties. In fact it can be easily proven that there exists uncountably many properties [11]. It should be noted that not every code related property is a subset of the code property. For example not all thin languages are codes.

3.3 Methods for Describing Code Related Properties

Methods for deciding and describing classes of codes have been under investigation for long time. There are many different algorithms for deciding if a language belongs to a certain class of languages for some well established classes. For instance, there is a variety of algorithms for deciding if a given language is a code for both finite and infinite language. Many of them are based on the Sardinas and Patterson algorithm [5]. A fast algorithm, but applicable only to finite languages is presented in [37]. A different one, applicable to regular languages, is presented in [16]. It uses the concept of functionality of transducers. Algorithms are also known for deciding if a language belongs to the classes of suffix, prefix, and bifix codes [5], solid codes [24], and infix codes [19].

The algorithms in publications listed above use different approaches for every property; In the following, we provide an overview of methods offering unified algorithms applicable to multiple classes of languages. First we a take look at general mathematical methods and then focus on formal methods, that is methods that utilize some form of formal expressions for describing properties.

To our knowledge, [39] is the first work proposing a general mathematical method for defining many code related properties. The authors use the concepts of binary word relation and independent sets. A binary relation p on Σ^* is a subset of $\Sigma^* \times \Sigma^*$. A strict binary relation has to satisfy additional constraints for all $w, v \in \Sigma^*$:

- $(w, w) \in p$ and $(w, \lambda) \in p$

- $(w, v) \in p$ implies $length(w) \geq length(v)$
- $(w, v) \in p$ and $length(w) = length(v)$ implies $w = v$

A subset H of Σ^* is called p-independent (independent with respect to some relation p) if the following statement holds true:

$$\forall u, v \in H : (u, v) \in p \Rightarrow u = v$$

The authors of [39] describe prefix and suffix relations in the context of strict binary relations over some alphabet Σ:

- $p_p = \{(u, ux) | u \in \Sigma^*, x \in \Sigma^*\}$
- $p_s = \{(u, xu) | u \in \Sigma^*, x \in \Sigma^*\}$

Having the relations defined the authors use the concept of p-independence to describe some code related properties. A language is a prefix code if it is p_p-independent and it is a suffix code if it is p_s-independent. Furthermore, the set of all p-independent languages is in fact a property according to our definition of a property. For instance, the set of all p_p-independent languages is the prefix property. In that sense the concept of p-independence provides a mathematical framework for describing language properties. This concept was a foundation for further research - see [41] and [18].

Another, more recent method for defining code related properties is based on the concept of dependence systems [25]. In particular we are interested in the concept

of n-independence. A property is an n-independence property, if a language belongs to such a property if and only if every subset of such a language of cardinality smaller than n also belongs to that property. It turns out that all properties that can be defined using our method are in fact independence properties, in particular 3-independence properties. We further investigate the concept of independence in Section 5.2.

3.4 Existing Formal Methods for Describing Language Properties

In the following part we investigate three existing formal methods for describing (thus also defining) properties of languages:

- Implicational conditions [21], where properties are described by first order logic formulas.
- Regular Trajectories [9, 10], where properties are described by regular expressions.
- Language in-equations [26], where properties are described by binary word operations.

3.4.1 Implicational Condition

The framework presented in [21] provides a method for general algebraic characterisation of classes of languages. It uses first order formulas called implicational independence conditions. The basic implicational independence condition has the following

form:

$$\langle \text{quantifier prefix} \rangle (\langle \text{formula} \rangle \longrightarrow \langle \text{formula} \rangle)$$

where the elements of the condition must satisfy the following:

1. *quantifier prefix* - it involves only universal quantifiers and it may include quantification over the number of variables used.

2. *formula* - the first formula is refereed to as the premise, the second one is the conclusion; both formulas are disjunction of conjunctions of equations and inclusions

Following are some properties discussed in this thesis described using the framework given in [21]:

Prefix codes:

$$I_p = \text{“}\forall u, v : u \in L, uv \in L \rightarrow v = \lambda\text{”}$$

Suffix codes:

$$I_s = \text{“}\forall u, v : u \in L, vu \in L \rightarrow v = \lambda\text{”}$$

Infix codes:

$$I_i = \text{“}\forall u, v, w : u \in L, vuw \in L \rightarrow v = \lambda, w = \lambda\text{”}$$

Hypercodes:

$$\begin{aligned} I_h = \quad & \text{“}\forall n \forall x_0, .., x_n, y_1, ..., y_n \in X^* : \\ & x_0 ... x_n \in L, x_0 y_1 x_1 ... y_n x_n \in L \rightarrow y_1 = \lambda \wedge, ..., y_n = \lambda\text{”} \end{aligned}$$

Overlap-free languages:

$$I_{of} = \text{“}\forall u, v, w : u \in L, uv \in L, vw \in L \rightarrow u = \lambda \wedge v = \lambda \wedge w = \lambda\text{”}$$

Solid codes:

$$I_i \wedge I_{of}$$

In addition to the preceding properties this method is capable of describing properties that are not describable using our method (and also two other methods presented in this section). A good example is the code property:

$I_c = \text{“}\forall m \forall n \forall x_1, .., x_n, y_1, ..., y_m \in X^* :$
$x_1 \in L, ..., x_n \in L, y_1 \in L, ..., y_m \in L, x_1...x_n = y_1...y_n \rightarrow n = m$ and $x_1 = y_1, ..., x_n = y_m\text{”}$

Although this method can be used to express the code property, there are some natural classes of codes the cannot be defined using it. The author of [21] provide as an example the class of block codes.

3.4.2 Regular trajectories

In his paper [9], Domaratzki introduced the concept of T-codes. A T-code is any language L for which the following statement holds true:

$$(L \amalg_T \Sigma^+) \cap L = \emptyset \tag{3.1}$$

where $\amalg_T$ is a word operation called shuffle on trajectories. The shuffle on trajectories operation was first defined by Mateescu in [35].

We give a formal definition of shuffle on trajectories, following the one given in [9] and earlier by [30]. Let us consider two words $x = x_1x_2...x_n$ and $y = y_1y_2...y_m$ where $x_i, y_j \in \Sigma$, and the trajectory $t = t_1t_2...t_k$ where $t_i \in \{0, 1\}$, then

$$x \amalg_t y = \begin{cases} x_1(x_2...x_n \amalg_{t_2...t_k} y_1y_2...y_m) & \text{if } t_1 = 0 \\ y_1(x_1x_2...x_n \amalg_{t_2...t_k} y_2...y_m) & \text{if } t_1 = 1 \end{cases}$$

Example: $aaa\amalg_{010011}bbb = a(aa\amalg_{10011}bbb) = abaabb$ *and* $aaa\amalg_{100011}bbb = b(aaa\amalg_{10011}bb) = baaabb$

If $x = x_1x_2...x_n$ and $y = \lambda$ where $x_i \in \Sigma$, and the trajectory $t = t_1t_2...t_k$ where $t_z \in \{0, 1\}$, then

$$x \amalg_t \lambda = \begin{cases} x_1(x_2...x_n \amalg_{t_2...t_k} y_1y_2...y_m) & \text{if } t_1 = 0 \\ \emptyset & \text{if } t_1 = 1 \end{cases}$$

Example: $aaa \amalg_{000} \lambda = a(aa \amalg_{00} \lambda) = aaa$ *and* $aaa \amalg_{100} \lambda = \emptyset$

If $x = \lambda$ and $y = y_1 y_2 ... y_m$ where $y_j \in \Sigma$, and the trajectory $t = t_1 t_2 ... t_k$ where $t_z \in \{0, 1\}$, then

$$\lambda \amalg_t y = \begin{cases} \emptyset & \text{if } t_1 = 0 \\ y_1(x_1 x_2 ... x_n \amalg_{t_2,...,t_k} y_2 ... y_m) & \text{if } t_1 = 1 \end{cases}$$

Example: $\lambda \amalg_{111} aaa = a(\lambda \amalg_{11} aa) = aaa$ *and* $\lambda \amalg_{011} aaa = \emptyset$

In addition, $x \amalg_\lambda y = \emptyset$ if $x \neq \lambda$ or $x \neq \lambda$. Finally, if $x = y = \lambda$ then $\lambda \amalg_\lambda \lambda = \lambda$ and $\lambda \amalg_t \lambda = \emptyset$ when $t \neq \lambda$

Domaratzki defines a shuffle on a set $T \subseteq \{0, 1\}^*$ of trajectories operation:

$$x \amalg_T y = \bigcup_{t \in T} x \amalg_t y$$

The shuffle on a set of trajectories is further extended by Domaratzki to apply to languages. For $L_1, L_2 \subseteq \Sigma^*$:

$$L_1 \amalg_T L_2 = \bigcup_{x \in L_1, y \in L_2} x \amalg_T y$$

Using a shuffle on set of trajectories operation, Domaratzki proposes a method for

describing T-codes. Consider a language $L \subseteq \Sigma^+$. The language L is a T-code if it is non-empty and Equation 3.1 holds for some T.

Some properties that we investigate in this thesis can be expressed using the T-code framework. Let $P_T(\Sigma)$ denote all T-codes over an alphabet Σ and the set of trajectories T. Using sets of trajectories Domaratzki describes classes of codes that correspond to particular properties that we investigate in this thesis:

- suffix codes: $T = 1^*0^*$
- prefix codes: $T = 0^*1^*$
- bifix codes: $T = 0^*1^* + 1^*0^*$
- outfix and infix codes: $T = 0^*1^*0^*$ and $T = 1^*0^*1^*$
- hypercodes: $T = (1 + 0)^*$

Example: *Let us consider the trajectory set* $T = 0^*1^*$, *some alphabet* Σ *and the language* $L \subseteq \Sigma^*$. *For every* $x \in L$ *the operation* $x \amalg_T \Sigma^+$ *produces every word for which* x *is a proper prefix. The operation* $L \amalg_T \Sigma^+$ *produces the language* L' *consisting of all of words for which a proper prefix is in* L. *The language* L *is a prefix code only if none of the words in* L *is a prefix of any other word in* L. *Therefore* L *is a prefix code if and only if* $(L \amalg_T \Sigma^+) \cap L = \emptyset$

For alternative way of describing T-codes Domaratzki uses deletion along trajectories, a word operation introduced independently in [8] and [27]. Let us consider two words

$x = x_1x_2...x_n$ and $y = y_1y_2...y_m$ where $x_i, y_j \in \Sigma$, and the trajectory $t = t_1t_2...t_k$ where $t_i \in \{i, d\}$, then

$$x \rightsquigarrow_t y = \begin{cases} x_1(x_2...x_n \rightsquigarrow_{t_2...t_k} y_1y_2...y_m) & \text{if } t_1 = i \\ (x_2...x_n \rightsquigarrow_{t_2...t_k} y_2...y_m) & \text{if } t_1 = d \text{ and } x_1 = y_1 \\ \emptyset & \text{otherwise} \end{cases}$$

Example: $aab \rightsquigarrow_{idd} ab = a(ab \rightsquigarrow_{dd} ab) = a$, $aab \rightsquigarrow_{did} ab = (ab \rightsquigarrow_{id} ab) = a$ *and* $aab \rightsquigarrow_{diidi} ba = \emptyset$

Also, $x = x_1, x_2, ..., x_n$ where $x_i \in \Sigma$, and the trajectory $t = t_1t_2...t_k$ where $t_z \in \{i, d\}$, then

$$x \rightsquigarrow_t \lambda = \begin{cases} x_1(x_2...x_n \rightsquigarrow_{t_2...t_k} \lambda) & \text{if } t_1 = i \\ \emptyset & \text{otherwise} \end{cases}$$

Example: $aab \rightsquigarrow_{iii} \lambda = a(ab \amalg_{ii} \lambda) = aab$ *and* $aab \rightsquigarrow_{dii} \lambda = \emptyset$

Finally, $x \rightsquigarrow_\lambda y = \emptyset$ if $x \neq \lambda$. Also $\lambda \rightsquigarrow_t y = \lambda$ if and only if $t = y = \lambda$

Similarly to the shuffle on trajectories the operation of deletion along trajectories is also extended to set of trajectories

$$x \rightsquigarrow_T y = \bigcup_{t \in T} x \amalg_t y$$

The deletion operation is expanded to an operation on languages. For $L_1, L_2 \subseteq \Sigma^*$

we have:

$$L_1 \rightsquigarrow_T L_2 = \bigcup_{x \in L_1, y \in L_2} : x \rightsquigarrow_T y$$

As stated before, classes of T-codes can be expressed in terms of the deletion on trajectories operation (Equation 3.2). There is a relation between a set of trajectories representing a T-code in context of the shuffle operation and a trajectory set representing the same T-code in context of the deletion operation. This relation is given by a morphism $\tau : \{0,1\}^* \rightarrow \{i,d\}^*$ such that $\tau(0) = i$ and $\tau(1) = d$. Given T describing a set of trajectories for shuffle operation, T-codes can be expressed as follows:

$$P_T(\Sigma) = \{L : (L \rightsquigarrow_{\tau(T)} \Sigma^+) \cap L = \emptyset\} \tag{3.2}$$

for every $T \subseteq \{0,1\}^*$ and any Σ.

Example: *Let us consider the trajectory set $T = 0^*1^*$, an alphabet Σ, and language $L \subseteq \Sigma^*$. For every $x \in L$ the operation $x \rightsquigarrow_{\tau(T)} \Sigma^+$ produces every proper suffix of the word x. The operation $L \amalg_{t(T)} \Sigma^+$ produces the language L' consisting of all proper prefixes of every word in L. The language L is a prefix code only if none of the words in L is a prefix of any other word in L. Therefore L is a prefix code only if $(L \rightsquigarrow_{\tau(T)} \Sigma^+) \cap L = \emptyset$*

Domaratziki points out that his framework cannot be used to describe the code property and that not all T- codes are in fact codes. In his paper Domaratzki gives an example of trajectory set for which the T-code is not a code:

Example: *Let us consider the language* $L = \{ab, cb, abcb\}$. *We can see that* L *is not code, as* $abcb = (ab)(cb)$. *Now let us consider the trajectory* $T = (10)^*$. *The language* L *is a* T*-code because* $(L \amalg_T \Sigma^+) \cap L = \emptyset$ *for this particular* T *and* L.

3.4.3 Language in-equations

Some code related properties can be described using language in-equations. To use language in-equation we first have to introduce the concept of binary operations on words. We follow the definition of binary word operations given in [26]. A binary word operation is a mapping:

$$\diamondsuit : \Sigma^* \times \Sigma^* \rightarrow 2^{\Sigma^*} \tag{3.3}$$

where 2^{Σ^*} represents all possible subsets of Σ^*.

In the following, we provide some examples of binary word operation as given in [26]:

- *Concatenation*: $u \cdot v = \{uv\}$
- *Left quotient*: $u \rightarrow_{lq} v = \{w\}$ if $\{u = vw\}$
- *Right quotient*: $u \rightarrow_{rq} v = \{w\}$ if $\{u = wv\}$
- *Insertion*: $u \leftarrow v = \{u_1 v u_2 | u = u_1 u_2\}$
- *Deletion*: $u \rightarrow v = \{u_1 u_2 | u = u_1 v u_2\}$
- *Dipolar deletion*: $u \rightleftharpoons v = \{w | u = v_1 w v_2, v = v_1 v_2\}$

- *Shuffle(scattered insertion)*: $u \amalg v = \{u_1 v_1 ... u_k v_k u_{k+1} | u = u_1 ... u_{k+1}, v = v_1 ... v_{k+1}\}$

A binary operation can be applied to any two languages. For two languages L_1 and L_2:

$$L_1 \diamondsuit L_2 = \bigcup_{u \in L_1, v \in L_2} u \diamondsuit v \tag{3.4}$$

Recall that, according to the definition we follow in this thesis, a property is a set of all languages that share a common characteristic. Some code related properties can be defined as a solution set to a language in-equation involving binary operations. Following are few examples of such properties given in [26]:

1 . *Prefix property* is the solution set of $(X \rightarrow_{lq} \Sigma^+) \subseteq X^c$ with a constraint $X \subseteq \Sigma^+$, and where X^c is the complement of X, that is $X^c = \Sigma^+ - X$

2 . *Suffix property* is the solution set of $(X \rightarrow_{rq} \Sigma^+) \subseteq X^c$ with $X \subseteq \Sigma^+$

3 . *Infix property* is the set of solution of $(X \rightleftharpoons \Sigma^+) \subseteq X^c$ with

4 . *Outfix property* is the set of solution of $(X \rightarrow \Sigma^+) \subseteq X^c$ with $X \subseteq \Sigma^+$

5 . *Hypercode property* is the set of solution of $(X \amalg \Sigma^+) \subseteq X^c$ with $X \subseteq \Sigma^+$

Every language in-equation in the preceding list defines a property which consist of all languages that satisfy that property. For example, a language that satisfies the in-equation described in point 1 is a prefix code.

Language equation can be approached similarly to the algebraic equation. For example $X \diamondsuit L = R$ is to some extent similar to the equation $x + l = r$ where x is the unknown and l and r are constants. However, because a binary word operation is usually not commutative, the concept of left or right inverse has to be applied in order to solve a language equation. Following the definition given in [26] we define the concept of left and right inverse:

- For the binary operation $\diamondsuit$, a left inverse of $\diamondsuit$ denoted as $\diamondsuit^l$ is defined as: $w \in x \diamondsuit v$ iff $x \in w \diamondsuit^l v$ for all $v, w, x \in \Sigma^*$
- For the binary operation $\diamondsuit$, a right inverse of $\diamondsuit$ denoted as $\diamondsuit^r$ is defined as: $w \in \{x \diamondsuit v\}$ iff $v \in x \diamondsuit^r w$ for all $v, w, x \in \Sigma^*$

Similar approach can be applied to language in-equations.

Chapter 4

Our Method

Our method is developed to address the following problem:

Given the descriptions of a property P and a language L decide if L satisfies P.

We are able to decide certain language properties by testing the following condition:

$$t(L) \cap L = \emptyset, \tag{4.1}$$

where L is the given language and t is a transducer that describes the property P. If (4.1) is true, the language L does satisfy (or belongs to) the property P. If (4.1) is not satisfied then that language does not belong to the property P. Testing Equation 4.1 is a simple and effective method for deciding certain properties of languages. The algorithm for that method is a four step process:

1. Describing the property with a transducer
2. Constructing the language $t(L)$, where L is the given language and t is a transducer describing the property
3. Constructing the language $t(L) \cap L$
4. Testing if the language $t(L) \cap L$ is empty or not

The method can be applied to a certain set of properties and comes with some limitations, which are discussed further in the next chapter.

4.1 Describing Properties of Languages with Transducers

Transducers are well understood tools in automaton theory. They can be used to represent relations between words. Some properties are based on a relation between words within a language, or rather avoiding the relation (e.g. suffix codes, where no word can be a proper suffix of any other word within the same language). For some properties we can build a transducer that describes the relation between words that have to be excluded from a language for it to belong to certain property. Then, such transducer describes that property in the context of our method. In particular transducers can be used to describe all properties listed in Section 3.2.

Let us consider the suffix property. It is possible to construct the transducer t_s such that for any given input word the transducer t_s is capable of generating every possible proper suffix of that word (Fig. 4.1). By proper suffix of a word we mean a suffix that does not equal the original word. We say that this transducer represents the proper suffix relation.

Example: *Given the word aaab, the set of possible outputs of the transducer in Fig. 4.1 would be:* $\{aab, ab, b, \lambda\}$. *Note that aaab is not in this set since it not a proper suffix of itself.*

Similarly we can present the transducer t_p for generating proper prefixes (Fig. 4.2).

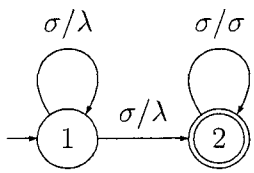


Figure 4.1: The transducer t_s. For every input word, it can produce every proper suffix of this word. As mentioned in Section 2.6, σ represents any symbol in a given alphabet and λ represents the empty word.

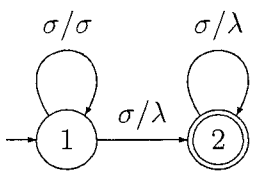


Figure 4.2: The transducer t_p, for every input word, can produce every proper prefix of this word.

Exapmle: *The transducer in Fig. 4.2 can produce all proper prefixes of any input word. Given the input word aaab, the set of possible output words is:* $\{aaa, aa, a, \lambda\}$

In the context of our method and Equation 4.1, we say that the transducer in Fig. 4.1 describes the suffix property and the transducer in Fig. 4.2 describes the prefix property. Using the same approach we can build transducers corresponding to different properties. Note that such a transducer is useful for our method only if it is input altering. A transducer is input altering if, for any input word, the set of output words does not contain that input word. This issue is further discussed in Section 5.1.

Once we have a valid description of a property in a form of an input altering transducer t, the next step is computing the language $t(L)$.

4.2 Constructing $t(L)$

In this step we build the automaton representing the language $t(L)$, where t describes a certain property and L is the given language to be tested. To achieve this, we use the product operation, as explained in Section 2.7. Performing the product operation on the NFA A representing L and the transducer t results in an NFA A', which represents the language of all words that are in relation with any word from L according to t.

***Example**: Let us consider the automaton A_2 (Fig. 4.3) representing the language $L_2 = a^*b$ (Fig. 4.3). The automaton is now expanded with self λ-transitions.*

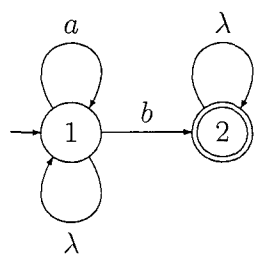


Figure 4.3: The automaton representing the language $L_2 = a^*b$, expanded with self λ-transitions

Next, let us consider a transducer t_p representing the proper prefix relation over the alphabet $\{a, b\}$ (Fig. 4.4).

In order to identify the language that will consist of all proper prefixes of the language L_2 we construct the product of the automaton A_2 and the transducer t_p. The steps in building the product automaton are presented in Fig. 4.5.

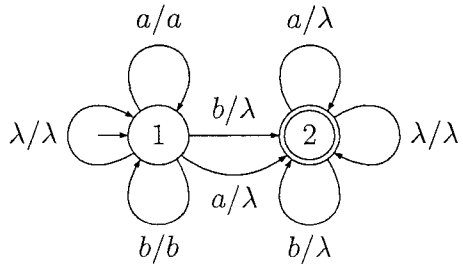
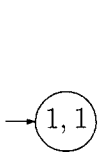
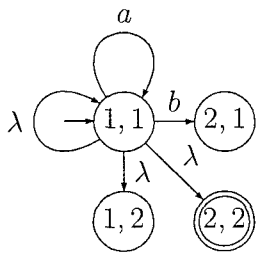


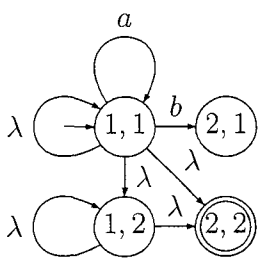
Figure 4.4: The transducer t_p expanded with self λ-transitions, representing the proper prefix relation over the alphabet $\{a, b\}$



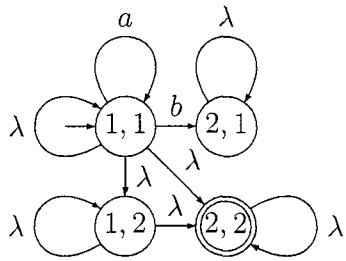
(a) Start state of the product automaton



(b) Transitions going out of the start state



(c) Building new transitions going out of the state $(1, 2)$



(d) The final product

Figure 4.5: The product of the automaton in Fig. 4.3 and the transducer t_p in Fig. 4.4

The automaton in Fig. 4.5d represents the language a^*, which consists of all proper prefixes of the language a^*b.

Now, consider a transducer t_s representing the proper suffix relation over the alphabet $\{a, b\}$ (Fig. 4.6).

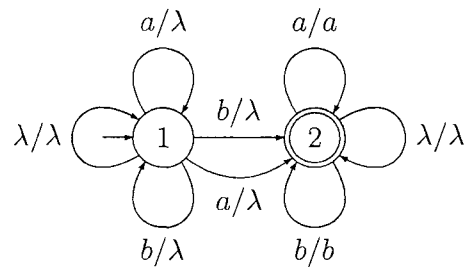
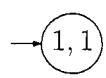
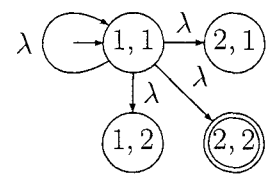


Figure 4.6: The transducer t_s expanded with self λ-transitions, representing the proper suffix relation over the alphabet $\{a, b\}$

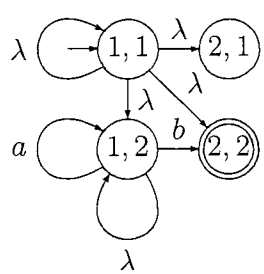
We build the language of all proper suffixes of the language L_2. We construct the product of the automaton presented in Fig. 4.3 and the transducer t_s in Fig. 4.6. The process of building the product automaton is presented in Fig. 4.7.



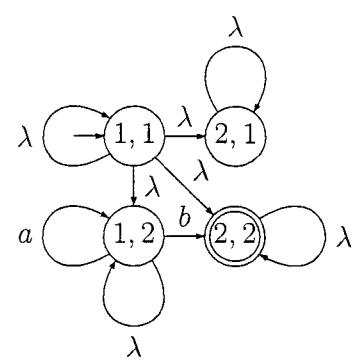
(a) Start state of the product automaton



(b) Transitions going out of the start state



(c) Building new transitions going out of the state $(1, 2)$



(d) The final product

Figure 4.7: The product of the automaton in Fig. 4.3 and the transducer t_s in Fig. 4.6

The automaton in Fig. 4.7 represents the language $a^*b+\lambda$, that is the set of all proper suffixes of the language $L_2 = a^*b$.

We have seen how to construct $t(L)$, given an input altering transducer t and an automaton representing L. The next step is to identify the set of words that exist both in L and $t(L)$.

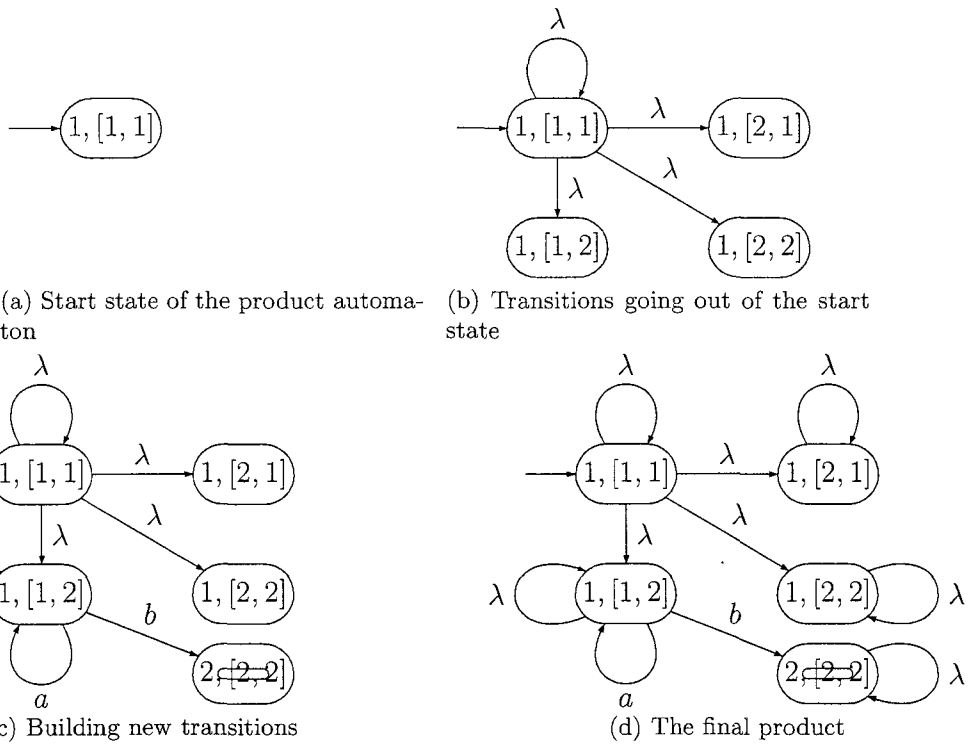
4.3 The Final Product Construction and the Emptiness Test

When we test a given language L for properties like the suffix property or the prefix property, we want to establish if that language contains pairs of words that have a particular relation with one another. In such cases, given an input altering transducer t that represents that relation, we can construct the language $t(L)$. If $t(L)$ has a subset of words that also exist in L, then the property is not satisfied. In other words, if the intersection of the language L and the language $t(L)$ is the empty set, the language L belongs to the property described by t.

In the previous section we showed how to construct the automaton $t(L)$. Next, we construct the automaton accepting the language $t(L) \cap L$ using again the product operation. If the automaton representing $t(L) \cap L$ does not contain any accepting paths, then Formula 4.1 holds and the language L belongs to the property described by t. However if the same automaton contains at least one accepting path, then L does not belong to that property.

***Example**: Let us consider the language L_2 (Fig. 4.3). In the previous section we*

constructed the automaton representing the language $t_s(L_2)$ (Fig. 4.7d) where t_s represents the suffix property. Now we construct the product of L_2 and $t_s(L_2)$ (Fig. 4.8). The resulting automaton is presented in Fig 4.8d. This automaton has an accepting path. This means the language represented by that automaton in not an empty one and the language L_2 does not belong to the property described by t_s.



(a) Start state of the product automaton

(b) Transitions going out of the start state

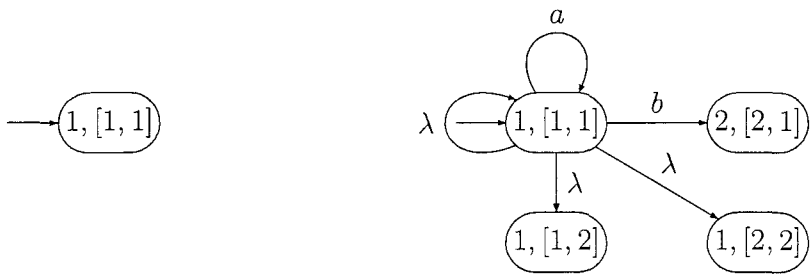
(c) Building new transitions

(d) The final product

Figure 4.8: The process of constructing the automaton representing $L_2 \cap t_s(L_2)$.

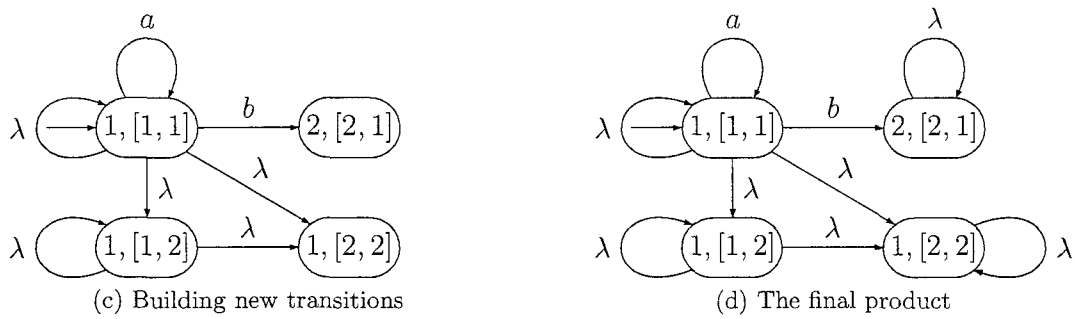
Now consider the language L_2 and the set of all proper prefixes of that language $t_p(L_2)$ (Fig. 4.5d). Constructing a product of L_2 and $t_p(L_2)$ results in the language represented by the automaton in Fig. 4.9d. This automaton does not have an accepting path (lack of an accessible final state clearly indicates lack of an accepting path).

Hence, the language represented by this automaton is empty. Therefore, Equation 4.1 holds for L_2 and t_p, which means that L_2 is a prefix code.



(a) Start state of the product automaton

(b) Transitions going out of the start state



(c) Building new transitions

(d) The final product

Figure 4.9: The process of constructing the automaton representing $L_2 \cap t_p(L_2)$

In order to decide the emptiness problem of languages represented by automata we use the well-known Breadth First Search algorithm [38] for searching for a path from one of the starting states to one of the final states.

4.4 Examples of Transducers Describing Well Known Properties

Next, we present input altering transducers describing some properties listed in Section 3.2 in the context of our method:

- Infix property (Fig. 4.10)
- Thin language property (Fig. 4.11)
- Hypercode property (Fig. 4.12)
- Overlap-free property (Fig. 4.13)
- Solid code property (Fig. 4.14)

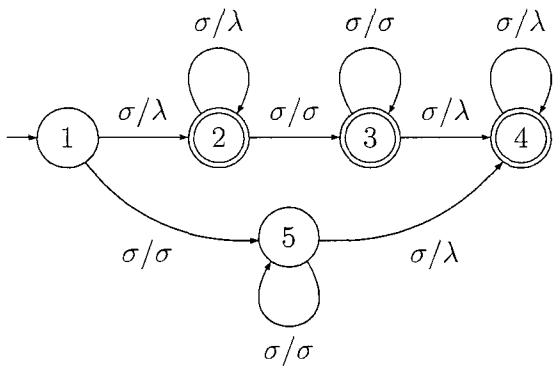


Figure 4.10: Transducer t_i describing the infix property.

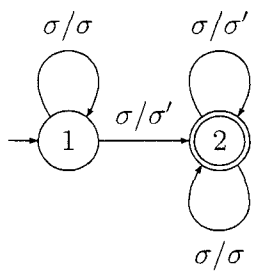


Figure 4.11: Transducer t_{tl} describing the thin language property.

Given a transducer t_1 describing some property P_1 and t_2 describing a property P_2, we can construct a transducer $t_1 \cup t_2$ describing $P_1 \cap P_2$, which is the set of languages that satisfy both P_1 and P_2.

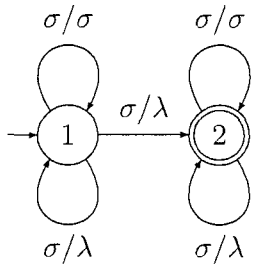


Figure 4.12: Transducer t_h describing the hypercode property.

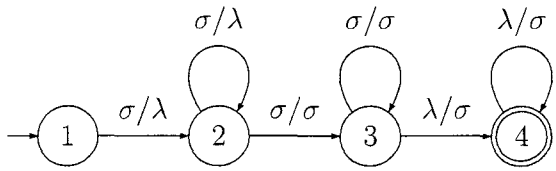


Figure 4.13: Transducer t_{of} describing the overlap-free property.

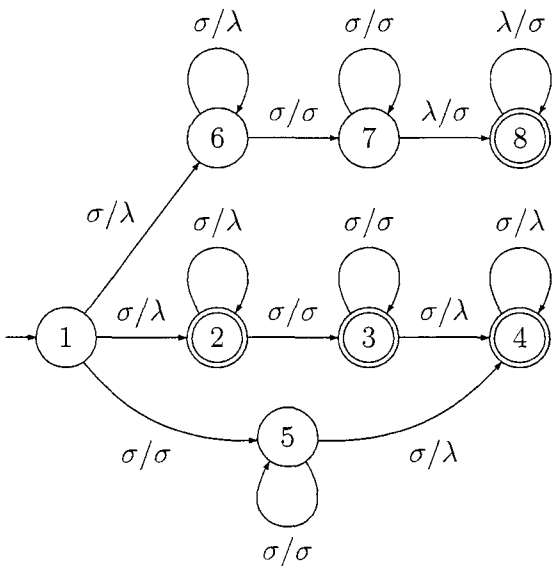


Figure 4.14: Transducer t_{sc} describing the solid code property.

Example: *Consider the solid code property. As explained in Section 3.2, a solid code is a language that belongs both to the infix property and to the overlap-free property. The transducer t_{sc} (Fig. 4.14) is in fact equivalent to $t_i \cup t_{of}$ (Fig. 4.10 and 4.13).*

Chapter 5

Limitations

Although our method is useful for formally describing some properties and testing if languages belong to those properties, it does have limitations. It can only be applied to 3-independence properties. Within that scope of problems our method requires a description of properties in the form of input altering transducers, and not all 3-independence properties can be described using such transducers.

5.1 Input Altering Transducer

The method requires a property to be represented in the form of an input altering transducer. A transducer t is input altering if and only if the following property holds:

$$\forall w \in \Sigma^* : w \notin t(w) \tag{5.1}$$

That means for any given input word over any alphabet, the input word is not contained in the set of output words.

Meeting this requirement is crucial for our method. If even a single word from the input language is preserved by a transducer it means the output language will always contain a word from the input language (Fig. 5.1). As a result Equation 5.1 will never

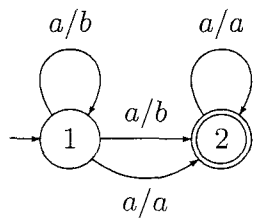


Figure 5.1: An example of a transducer that is not input altering. For the input word aa the set of output words is $\{bb, ba, aa\}$. The original word aa is also found in the set of possible output words.

hold and therefore such transducer cannot be used to test a language for a non-trivial property. Moreover, if the given transducer is input altering then it is also decidable whether a given regular language is maximal with respect to the property described by that transducer - see [11].

Determining if a given transducer is input altering is not a decidable problem [20]. The proof is based on the observation that the question of whether a transducer is input altering can be mapped to the Post Correspondence Problem, which is not solvable [33]. In practice, for the transducers defining properties like those in Section 3.2 it is possible for a human to tell if the transducers is input altering or not. However, the general problem remains algorithmically unsolvable.

5.2 Independence Properties

A property P is an n-independence [23] property, if the following statement holds true for every language L:

$$L \in P \iff \forall L' \subseteq L : 0 < |L'| < n \rightarrow L' \in P \tag{5.2}$$

That is, if a language L belongs to P then every subset of L of cardinality less than n also satisfies that property, and if every subset of L of cardinality smaller than n belongs to P then L satisfies that property. If we want to test a language for a property that is a 3-independence property it is sufficient to test every subset of that language of cardinality less than 3. Using a transducer we represent a relationship between two words and check if a pair of words exists that does not satisfy the property. If no such pair exists it means that all subsets of a language of cardinality less than 3 belong to that property. In the case of 3-inependence property it is sufficient proof to state that a language satisfies a particular property. However, not every 3-independence property can be expressed in the form of input altering transducer.

A property P is called an independence property if it satisfies Formula 5.2 for some n.

5.3 A 3-Independence Property Not Describable by an Input Altering Transducer

There are uncountably many[1] 3-independence properties. Therefore we cannot use finite state machines to represent every property from this set. However this argument does not provide examples of such properties.

An example of a property that is not expressible using a transducer is the set P_{Rev}

[1] In fact, there are uncountably many 3-independence properties consisting of infix codes. This is shown in [11]. The proof is based on the following two claims: (1) There are uncountably many infix codes. (2) For every infix code C, the set 2^C is a 3-independence property whose elements are infix codes.

of languages where each language cannot contain the reverse of any of its words. For instance if *aab* belongs to such language then *baa* cannot belong to it. Formally this property is expressed as follows:

$$L \in P_{Rev} \iff \forall u, v \in L : u \neq v^R \tag{5.3}$$

There is no transducer capable of representing the property P_{Rev} in context of our method. To prove that we use a Pumping Lemma argument: we show that altering a computation performed by t affects the relation represented by it. First we assume that such a transducer exists. If such t exists the following statement holds true:

$$\forall L : t(L) \cap L = \emptyset \iff L \in P_{Rev} \tag{5.4}$$

that is, for any regular language L and the transducer t representing P_{Rev} we can use t in the equation 4.1 in order to decide if L belongs to the property P_{Rev}. Therefore:

$$\forall L : \forall u, v \in L : v \notin t(u) \iff v \neq u^R \tag{5.5}$$

Next let us also consider the language $L = \Sigma^*$ that is the language consisting of all possible words. Then:

$$\forall u, v \in L : v \notin t(u) \iff u \neq v^R \tag{5.6}$$

and, equivalently

$$\forall u, v \in L : v \in t(u) \Longleftrightarrow u = v^R \tag{5.7}$$

Therefore, if such a transducer t exists, it has to be functional, so that $t(u) = u^R$.

In order to disprove the existence of t we prove the following statement:

$$\nexists t : \forall u \in \Sigma^* : t(u) = u^R \tag{5.8}$$

Let us assume that t exists and has M states. Now consider the word $u = a^N b^N$ such that $N > M$. Then $t(u) = b^N a^N$,which means that the transducer t has to realize the following computation:

$(q_0, x_1/y_1, q_1)$,..., $(q_{k-1}, x_k/y_k, q_k)$,..., $(q_{n-1}, x_n/y_n, q_n)$

where q_0 is a start state and q_n is a final state and:

- $x_1 x_2 x_3 ... x_n = a^N b^N$
- $y_1 y_2 y_3 ... y_n = b^N a^N$

In fact some accepted and produced symbols can be λ and it might be the case that $n > |u|$. Therefore, $n \geq 2N$. We can assume that the computation performed by t when accepting the word u has the following structure:

$(q_0, x_1/y_1, q_1)$, ..., $(q_k, x_{k+1}/y_{k+1}, q_{k+1})$, ..., $(q_{k+j-1}, x_{k+j}/y_{k+j}, q_{k+j})$,

$(q_{k+j}, x_{k+j+1}/y_{k+j+1}, q_{k+j+1})$, ..., $(q_{n-1}, x_n/y_n, q_n)$

where the index k is such that $q_k = q_{k+j}$ and $x_{k+i} = a$ the input sequence $x_{k+1}...x_{k+j} = a^p$ where $p \in \{1, .., j\}$ and $j \geq 1$. This is a safe assumption because the transducer t is a finite machine and the word u contains more a's than the transducer t has states, so there is a repeated state $q_k = q_{k+j}$ between the two transitions involving the first and the last input symbol a. In other words there is a repeatable sequence in that computation where the input consist only of a's and λ's and is non-empty, that is contains at least one a. The repeatable sequence in the computation means that the transducer has to have at least one loop in its structure. Therefore we know that t has the one presented in Fig. 5.2, where z_2/z_2' represents the repeated sequence of the computation:

$(q_k, x_{k+1}/y_{k+1}, q_{k+1})$, ..., $(q_{k+j-1}, x_{k+j}/y_{k+j}, q_{k+j})$

where $q_k = q_{k+j}$.

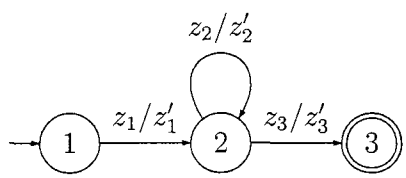


Figure 5.2: A general structure of the transducer representing P_{Rev}

Removing the repeatable sequence from the computation accepting u creates a new computation sequence. The new computation sequence is still a valid computation. The new input word u' remains in the language accepted by the transducer and has

the following form: $a^{N-p}b^N$. Therefore the new computation should still realize the reverse relation and the output v' word should have the following form: $b^N a^{N-p}$. However that leads to a contradiction. To prove that, we look at different scenarios of how the new computation can look like and consider the relation of the input and output word in this new computation:

- If the removed repeated sequence contains a transition of the form $(q_i, a/b, q_{i+1})$ or $(q_i, \lambda/b, q_{i+1})$, then, for the input word $a^{N-p}b^N$, the output word has the form $b^{N-r}a^P$ where $r > 0$, which is not the reverse of the input word.

- If the removed repeated sequence contains any transitions of the form $(q_i, a/\lambda, q_{i+1})$ but does not contain an equal number transitions of the form $(q_j, \lambda/a, q_{j+1})$, then the new computation does not satisfy the reverse relation. The number of non-λ symbols removed form the input word is not equal to the number of non-λ symbols removed from the output word. In effect in the new computation the length of the input word u' is different than the length of the output word v'. That is sufficient to conclude that the new output word is not a valid reverse of the input word for the new computation.

- The last case is that the removed sequence contains equal numbers of transitions in the form $(q_i, a/\lambda, q_{i+1})$ and in the form $(q_i, \lambda/a, q_{i+1})$ and at the same time does not contain any transitions involving b. This means that by removing the repeatable sequence we would remove the same number of a's from the input and from the output word. However this has also other implications. Due to the structure of u we know that in the original computation all a's have to be

accepted before any b's are accepted. Also all b's have to be output before any a's are output . Therefore the existence of a repeatable sequence that both accepts and outputs a's implies that all b's from the output word v has been produced before that sequence appears and all b from the input word u will be accepted after that sequence appears. Because the number of b's in the input word can be arbitrarily large and t is a finite machine we can make further assumption about the structure of t for this case: there has to be a repeatable sequence in original computation, following the repeatable sequence that accepts a's , that accepts b's but does not produce any b's.

In the last case removing the repeatable sequence does not disprove the existence of t because the new computation might still realise the reverse relation. However it provides us with additional knowledge about the original computation:

- the repeated sequence outputs a's and therefore we know that all b's have been output by the computation before that repeated sequence appears
- the repeated sequence accepts a's and thus no b's from the input word have yet been accepted

Based on that knowledge we can draw a new conclusion regarding the structure of t. As stated before we know that t has to be capable of accepting the word u with an arbitrary number of a's and b's. In such situation there must exit at least two more separate repeatable structures:

- a repeated sequence z_4/z_4' preceding the z_2/z_2' that accepts only a's and/or λ and outputs b's and possibly λ

- a repeated sequence z_5/z_5' following the z_2/z_2' that accepts b's and possibly λ and outputs a's and/or λ

The structure of transducer in situation like that is presented in Fig.5.3.

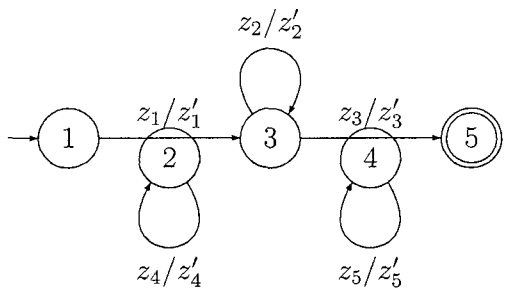


Figure 5.3: A general structure of the transducer that represents the reverse relation

Let us look at the computation performed by t on the input u, assuming t has the structure presented in Fig. 5.3. It has the following form:

$(q_0, x_1/y_1, q_1)$,

...,

$(q_h, x_{h+1}/y_{h+1}, q_{h+1})$,

...,

$(q_{h+c-1}, x_{h+c}/y_{h+c}, q_{h+c})$, $\Big\}\ z_4/z_4'$

...,

$$
\left.\begin{array}{l}(q_g, x_{g+1}/y_{g+1}, q_{g+1}),\\ \ldots,\\ (q_{g+k-1}, x_{g+k}/y_{g+k}, q_{g+k}),\end{array}\right\} z_2/z_2'
$$

$\ldots,$

$$
\left.\begin{array}{l}(q_v, x_{v+1}/y_{v+1}, q_{v+1}),\\ \ldots,\\ (q_{v+s-1}, x_{v+s}/y_{v+s}, q_{v+s}),\end{array}\right\} z_5/z_5'
$$

$\ldots,$

$(q_{n-1}, x_n/y_n, q_n)$

where $q_h = q_{h+c}$, $q_g = q_{g+k}$ and $q_v = q_{v+s}$. The repeated sequence starting from q_v is realized by the structure z_5/z_5'. We know that for this sequence the accepted word for this sequence consist of only b's and/or λ's with at least one b. The output for this sequence consist of only of a's and/or λ'. Again we use the Pumping Lemma argument to analyse this computation. We remove this repeatable sequence and analyse the new computation:

- If the repeatable sequence contains transitions in the form $(q_i, b/a, q_{i+1})$ or $(q_i, \lambda/a, q_{i+1})$, then, in the new computation, the input word has the form $a^N b^{N-p}$ and the output word has the form: $b^P a^{N-r}$ where $r > 0$, which is not the reverse of the input word.

- If the repeated sequence consist transitions $(q_i, b/\lambda, q_{i+1})$, then the new computation produces the output word that is not equal in length to the input word. Hence the output word cannot be the reverse of the input word.

This time in all cases altering the computation alters the relation between input and output words.

We showed that altering the computation realizing the reverse operation alters the relation represented by t while the new input word for the altered computation remains in the language accepted by t. In the new computation $t(u') \neq u'^R$. Therefore the statement 5.8 holds true, which means that it is impossible to construct a transducer that represents the P_{Rev}.

Chapter 6

Further Discussion on Our Method

In this chapter we provide further discussion of our method. We investigate its complexity. We take a look at the relation between a transducer describing a property and its inverse. We also investigate a relation between our method and the one based on sets of trajectories (discussed in Section 3.4.2) and also the one based on language in-equations (discussed in Section 3.4.3). Finally, we present our approach to extracting a counterexample, that is a pair of words belonging to a tested language that shows the language does not belong to a given property.

6.1 Complexity

A theoretical cost of constructing the language $t(L) \cap L$ has a complexity of $O(|A_L|^2|t|)$ where $|A_L|$ is the size of an automaton accepting L and $|t|$ is the size of transducer t representing some property. The cost of performing a Breadth First Search is $O(|A_m|)$ where $|E_m|$ is the size of an automaton accepting $t(L) \cap L$ and $|V_m|$ is the number of states in the same automaton.

Using a product implementation where we progressively build an automaton limits the size of the problem. However, it also makes it more difficult to establish the exact

complexity of such a computation.

6.2 More about Transducers Describing Properties

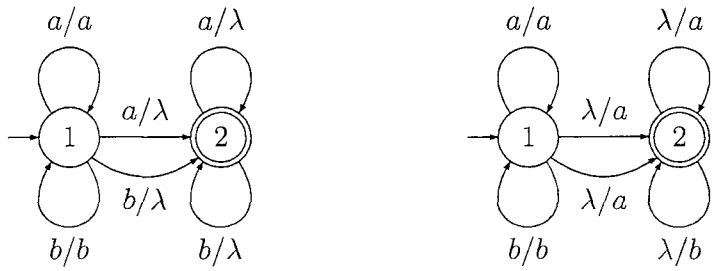
It turns out that if a certain transducer describes a particular property in the context of our method, then the inverse of that transducer represents the same property in context of our method:

$$t(L) \cap L = \emptyset \Longleftrightarrow \nexists x, y \in L : x \in t(y) \Longleftrightarrow \nexists x, y \in L : y \in t^{-1}(x) \Longleftrightarrow t^{-1}(L) \cap L = \emptyset \tag{6.1}$$

where t is an input altering transducer.

We build t^{-1} simply by switching the input symbol with the output symbol on every transition in t. So, for the transition $(q_k, a/b, q_{k+1})$ in t, we put $(q_k, b/a, q_{k+1})$ in t^{-1}.

Example: *Let us again consider the transducer t_p that represents the proper prefix relation over the alphabet $\Sigma = \{a, b\}$ (Fig. 6.1a). In order to construct t_p^{-1} we switch input symbols with output symbols in all transitions in t_p. The transducer t_p produces, for an input word, every proper prefix of that word. The transducer t_p^{-1} (Fig.6.1b) produces, for an input word, every word for which the input word is a proper prefix.*



(a) A transducer describing the prefix property

(b) The inverse of the transducer describing the prefix property

Figure 6.1: A transducer describing the prefix property and the inverse of that transducer

6.3 Transducers and Sets of Trajectories

There are a number of similarities between our method and the one developed by Dommaratzki in [10]. In both methods, for a given language L, we attempt to identify a language L' so that the existence of any words from L' in L would violate the tested property P. In other words, for the given language L, we identify the set L' with respect to some property P so that L belongs to P if and only if:

$$L \cap L' = \emptyset$$

In our method L' is described as follows:

$$L' = t(L)$$

where t is an input altering transducer. In Domaratzki's method the language L' is:

$$L' = L \amalg_T \Sigma^+$$

where T is a set of trajectories. In the context of our method a property is described using an input altering transducer. In the context of Domratzki's method a property is described by a set of trajectories. It turns out that the same approach leads to overlap between properties that can be described by our method and the one from Domaratzki.

6.3.1 Transforming a Set of Trajectories into an Input Altering Transducer

Every regular set of trajectories that describes a property in the Domaratzki's method does have a corresponding input altering transducer in our method. Therefore every T-code can be also tested using our method. In fact, our method can be used to describe properties that cannot be described with trajectories [11]. Recall that in Domaratzki's method a set of trajectories is described by a regular expression over the alphabet $\{0, 1\}$. Next we present a method to transform such a regular expression into an input altering transducer that describes the same property.

The method consists of two steps:

1 Construct an NFA equivalent (accepting the same language) to the given regular expression.

2 Construct an input altering transducer based on the NFA constructed in the previous step.

For the first step we use the classical approach of structural induction [17].

Example: *The automaton in Fig. 6.2 represents the set of trajectories* T=0*1*+1*0*.

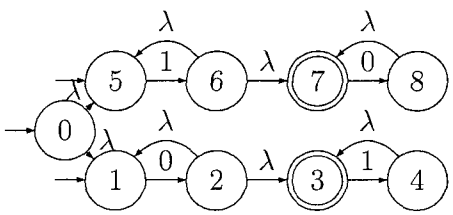
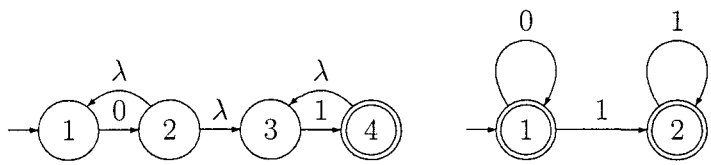


Figure 6.2: The NFA equivalent 0*1* + 1*0*

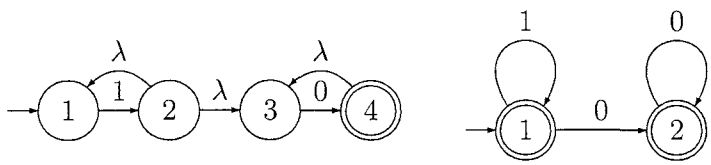
Our on-line application has a feature that allows providing a property in a form of an automaton accepting a set of trajectories. However such automaton has to be in a λ-free form. Only transitions labelled with 0 or 1 are allowed. We leave it up to the user to construct the automaton accepting the set of trajectories in a λ-free form. The algorithm for building NFA equivalent to λ-NFA is a well established one. We also refer the reader to [14] for a fast algorithm for constructing NFAs in λ-free form from regular expressions. Figures 6.3 and 6.4 provide examples of automata in λ-free form, accepting some set of trajectories.

In the second step we build an input altering transducer based on an automaton accepting some particular set of trajectories by implementing the following rules:



(a) λ-NFA (b) NFA

Figure 6.3: Automata accepting $T = 0^*1^*$, which describes the prefix property



(a) λ-NFA (b) NFA

Figure 6.4: Automata accepting $T = 1^*0^*$, which describes the suffix property

- For every transition $(p, 0, q)$ in A we create, if they do not exist already, states (p, No), (p, Yes), (q, No), (q, Yes) in t, and we add the following transitions to t: $((p, Yes), \sigma/\sigma, (q, Yes))$ and $((p, No), \sigma/\sigma, (q, No))$.

- For every transition $(p, 1, q)$ in A we create, if they do not exist already, states (p, No), (p, Yes), (q, Yes) in t, and we add the following transitions to t: $((p, Yes), \lambda/\sigma, (q, Yes))$ and $((p, No), \lambda/\sigma, (q, Yes))$.

- A state (p, No) in t becomes a start state if state p is a start state in A.

- A state (q, Yes) in t becomes a final state if state q is a final state in A.

This construction guarantees that every accepting path in t is input altering. We know that the insertion along the trajectories in Domaratzki's method is an input

altering operation. The input alteration is enforced in his methods by making sure that the inserted word belongs to Σ^+. However developing simple morphism $\iota(0,1) = \{\sigma/\sigma, \lambda/\sigma\}$ to transform an automaton accepting a set of trajectories into a transducer does not guarantee that such transducer corresponding to some set of trajectories will be input altering. On the other hand constructing a transducer using the preceding rules guarantees that every accepting path in such transducer contains at least one transition realizing insertion of a non-empty symbol (that is a transition labelled with λ/σ). This way we make sure that every accepting path in the transducer is input altering. At the same time the transducer t constructed based on some set of trajectories T is such that for any regular language L the language $t(L)$ is equivalent to the language $L \amalg_T \Sigma^+$

Example: *Let us consider the automaton in Fig 6.3b. The automaton consists of the following transitions:* $(1,0,1)$, $(1,1,2)$, $(2,1,2)$. *For every transition in this automaton we construct a pair of transitions in the transducer:*

- *For* $(1,0,1)$, *we have* $((1,Yes),\sigma/\sigma,(1,Yes))$ *and* $((1,No),\sigma/\sigma,(1,No))$
- *For* $(1,1,2)$, *we have* $((1,Yes),\lambda/\sigma,(2,Yes))$ *and* $((1,No),\lambda/\sigma,(2,Yes))$
- *For* $(2,1,2)$, *we have* $((2,Yes),\lambda/\sigma,(2,Yes))$ *and* $((2,No),\lambda/\sigma,(2,Yes))$

The state $(1,No)$ *becomes the start state and* $(1,Yes)$ *and* $(2,Yes)$ *become final states in* t. *This transducer is presented in Fig 6.5*

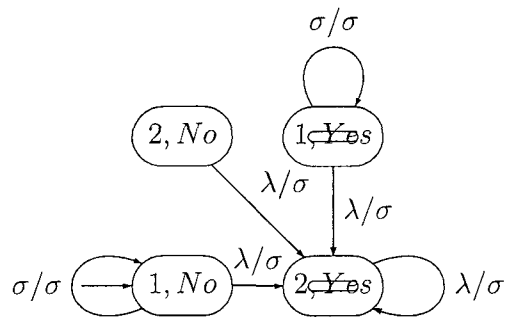


Figure 6.5: The transducer equivalent to $T = 0^*1^*$

The transducer in Fig 6.5 is equivalent to the transducer in Fig. 6.1. For any given input word it produces every possible word of which the input word is a proper prefix.

Similarly, we can take a deletion operation trajectory set to construct a transducer describing the same property. The alphabet for trajectories representing the deletion operation is $\{i, d\}$. In the first step we build an NFA equivalent to a trajectory set, similarly as we did for a trajectory set for shuffle operation. In the next step we build a transducer using the following rules:

- For every transition (p, i, q) in A we create, if they do not exists already, states (p, No), (p, Yes), (q, No), (q, Yes) in t, and we add the following transitions to t: $((p, Yes), \sigma/\sigma, (q, Yes))$ and $((p, No), \sigma/\sigma, (q, No))$

- For every transition (p, d, q) in A we create, if they do not exists already, states (p, No), (p, Yes), (q, Yes) in t, and we add the following transitions to t: $((p, Yes), \sigma/\lambda, (q, Yes))$ and $((p, No), \sigma/\lambda, (q, Yes))$

- A state (p, No) in t becomes a start state if state p is a start state in A

- A state (q, Yes) in t becomes a final state if state q is a final state in A

The motivation for this set of rules is the same as in case of constructing a transducer based on set of trajectories, which is making sure that resulting transducer is input altering.

Example: *Let us consider the trajectory set $T = i^*d^*$, describing the prefix property in the context of the deletion operation. We construct an automaton accepting this set (Fig. 6.4).*

(a) λ-NFA (b) NFA

Figure 6.6: An automaton accepting $T = i^*d^*$, which represents the prefix property

Next, we construct a transducer (Fig. 6.7) based on the automaton in Fig. 6.6b

We can see that the transducer in Fig. 6.7 is in fact the reverse of a transducer in Fig. 6.5 and realizes the same relation as the transducer in Fig 6.1.

6.3.2 Morphism of Trajectories

As we explained in Section 6.2, in the context of our method, if a transducer describes a particular property then the inverse of that transducer describes the same property.

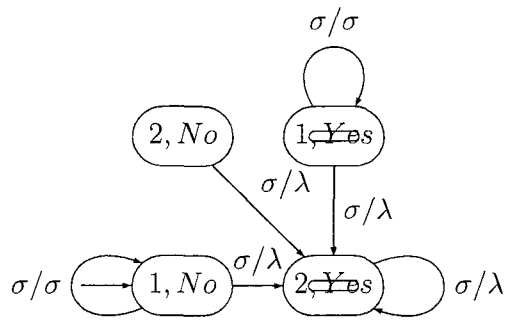


Figure 6.7: A transducer equivalent to $T = i^*d^*$

In his publication [10], Domaratzki defines the following morphism:

$$\tau : \{0,1\}^* \to \{i,d\}^* \tag{6.2}$$

This morphism, when applied to a set of trajectories for the shuffle operation describing a certain property, produces a set of trajectories that describes the same property in the context of the deletion on trajectories operation. In other words, if T describes a certain property in the context of the shuffle operation then $T' = \tau(T)$ describes the same property in the context of the deletion on trajectories operation. It turns out that if we build a transducer t corresponding to the set of trajectories T and a transducer t' corresponding to the set of trajectories $\tau(T)$ then t is an inverse of t' and, as mentioned in Section 6.2, t and t' describe the same property in the context of our method.

6.4 Our Method as a Specialisation of Language In-equations

Earlier in this thesis we presented the concept of describing properties using language in-equations (Section 3.4.3). Here we show that our method is in fact a specialization of that concept.

Every relation represented by a transducer is in fact a binary relation. We know that, for every binary word operation $\diamondsuit$, the binary relation$[\diamondsuit L]$ consists of all pairs (u, v) such that $v \in u \diamondsuit L$. For every input altering transducer t we can identify a binary relation $[\diamondsuit \Sigma^+]$ that is represented by t. We have:

$$L \cap t(L) = \emptyset \Longleftrightarrow L \cap t(L) \subseteq L^c \tag{6.3}$$

***Example**: A language L_p is a prefix code if and only if it satisfies the equation:*

$$L_p \cap t_p(L_p) = \emptyset \tag{6.4}$$

where t_p is the transducer describing the prefix property (Fig. 4.2). At the same time the transducer t_p implements the binary relation $[\longrightarrow_{rq} \Sigma^+]$. Therefore the language L_p satisfies Equation 6.4 if and only if it satisfies the in-equation:

$$L_p \longrightarrow_{rq} \Sigma^+ \subseteq L_p^c \tag{6.5}$$

with the restriction $L_p \subseteq \Sigma^+$.

6.5 Counterxample

The type of properties that our method is designed for makes it possible to identify a two element subset of an investigated language to show that the language does not belong to a particular property. Let us consider the language a^*b. We have already showed that this language is not a suffix code in Section 4.3. To provide a proof we need to show at least one pair of words belonging to that language where one of the words is a suffix of the other one. It might be the pair of (ab, b) because the word b is a suffix of ab and they both belong to a^*b.

In general, in case of independence properties, it is sufficient to find a subset of a language that do not belong to a property to show that the language also does not belong to that property. We investigate 3-independence properties and therefore such a subset has to consist of two elements. We refer to such a set as a counterexample. For a language L and a property described by a transducer t a counterexample consist of two word, v and w such that:

$$v, w \in L \wedge v \in t(w) \tag{6.6}$$

As suggested by Jürgen Dassow we implemented a feature to provide a counterexample to the user in the case that a given language does not belong to a given property. We identified and investigated two ways to provide a user with counterexample. The first one involves only minor modifications to the code we implemented for the previous steps. The drawback of this method is a significant calculation overhead in worst

case scenarios. The second method that we propose has exactly the same output. It has worse space complexity and would require major modification of our main method. Therefore, the first approach has been implemented in our system.

6.5.1 First Approach

In this approach we use methods that have already been implemented for the main method with only slight modifications.

Recall, the main method for testing languages for properties consists of 4 steps. In the final step we investigate if the product language $L \cap t(L)$ is empty or not (Section 4.3). The language is empty if an automaton representing the language has no accepting paths. We use a breadth first search algorithm to identify an accepting path (or lack of such path). The algorithm finishes either after visiting all the states in the automaton or finding an accepting path. If an accepting path is found a corresponding word is extracted. Hence, the method returns the first encountered member of a language if such exists or reports an empty language otherwise. If the word is extracted, then it belongs to a pair that satisfies Formula 6.6. The extracted word is denoted by v.

The second word in the counterexample is calculated based on the one extracted in the previous step. We know that $v \in t(w)$, for some $w \in L$. From that we conclude that $w \in t^{-1}(v)$. We construct the language $L_w = t^{-1}(v)$. In order to do that we first construct an automaton V representing the language consisiting only of the word v. Then we apply the relation t^{-1} to the language represented by V . For that, we use

the product operation described in Section 2.7. A language L_w that is produced by this operation consists of all words that result by applying transformation t^{-1} to the word v. However, not all the words contained in L_w also belong to L. In order to identify a sub of words that is contained in both L_w and L we construct $L'_w = L_w \cap L$ by performing a product operation on L_w and L. L'_w represents a non-empty language that consist of all words that, paired with v, create a counterexample. Only one word from L'_w is required for counterexample and therefore we apply the Breadth First Search algorithm to extract it.

In summary, finding a counterexample when a given language L does not belong to the property described by t consists of following steps:

(i) Extract a single word v from the language $L \cap t(L)$ that was computed in the last step of our method (Section 4.3).

(ii) Construct the language $L_w = t^{-1}(v)$

(iii) Construct the language $L'_w = L_w \cap L$

(iv) Extract a single word w from L'_w

The words $v, w \in L$ provide a simple example to show that property represented by t is not satisfied by L.

***Example**: Let us consider the language $L_2 = a^*b$, represented by the automaton A_2 (Fig. 4.3), and the suffix property described by the transducer t_s (that is the suffix*

property). We have already constructed the language $L_2 \cap t_s(L_2)$ (Fig. 4.8d) and found an accepting path: ((1,2), b, (2,2)). Now we extract a word corresponding to the accepting path that was found. In this example the word consists of one letter b. We build an automaton representing this single word (Fig. 6.8).

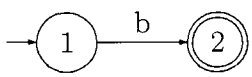


Figure 6.8: An automaton representing the single word b

We need to expand this automaton with self λ-transitions in order to perform the product operation.

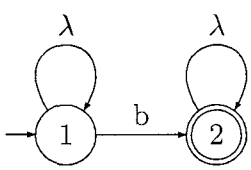


Figure 6.9: The automaton in Fig. 6.8 expanded with self λ-transitions.

Next, we construct t_s^{-1}, that is the inverse of the transducer describing the suffix property (Fig. 6.10).

Similarly to the automaton in Fig. 6.8 we need to expand this transducer with self λ-transitions (Fig. 6.11).

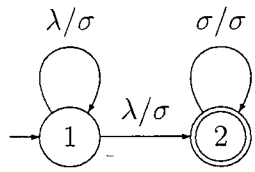


Figure 6.10: Inverse of a transducer describing the suffix property

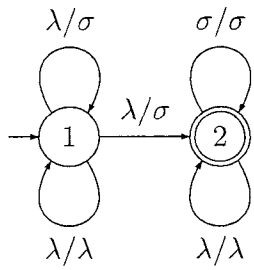
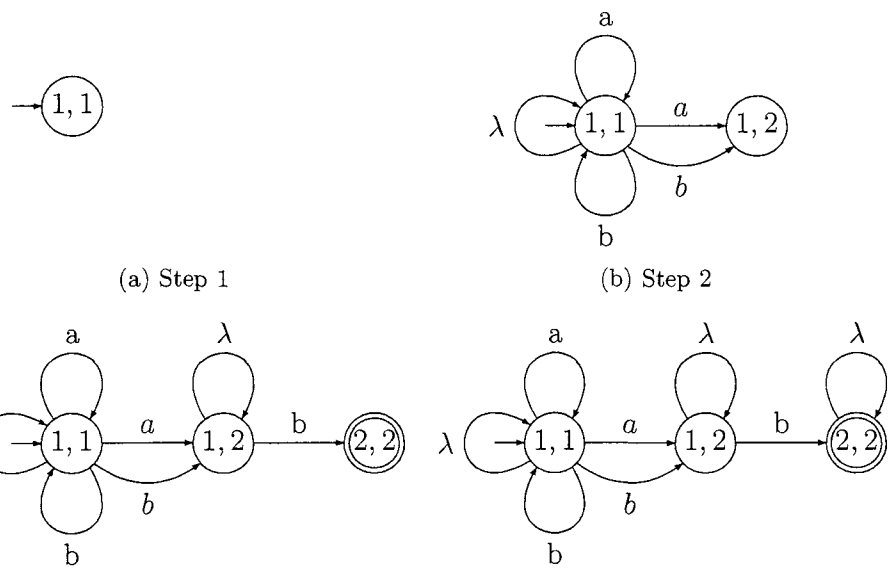


Figure 6.11: The inverse of the transducer t_s describing the suffix property expanded with self λ-transitions.

We use product construction to construct a language $t_s^{-1}(b)$ (Fig. 6.12).



(a) Step 1

(b) Step 2

(c) Step 3

(d) Step 4

Figure 6.12: Steps in building an automaton representing the language $t_s^{-1}(b)$.

The language $t_s^{-1}(b)$ represented by the automaton in Fig. 6.12d consists of all words that are returned by applying t_s^{-1} to the word b, which is $(a+b)^+b$. However, not all the words from this language are in L_2 (for example bab) Therefore we construct an automaton representing $L_2 \cap t_s^{-1}(b)$ (Fig. 6.13). Note that this step does not require expanding automata with self λ-transition as every state in Fig. 6.12d already has these transitions.

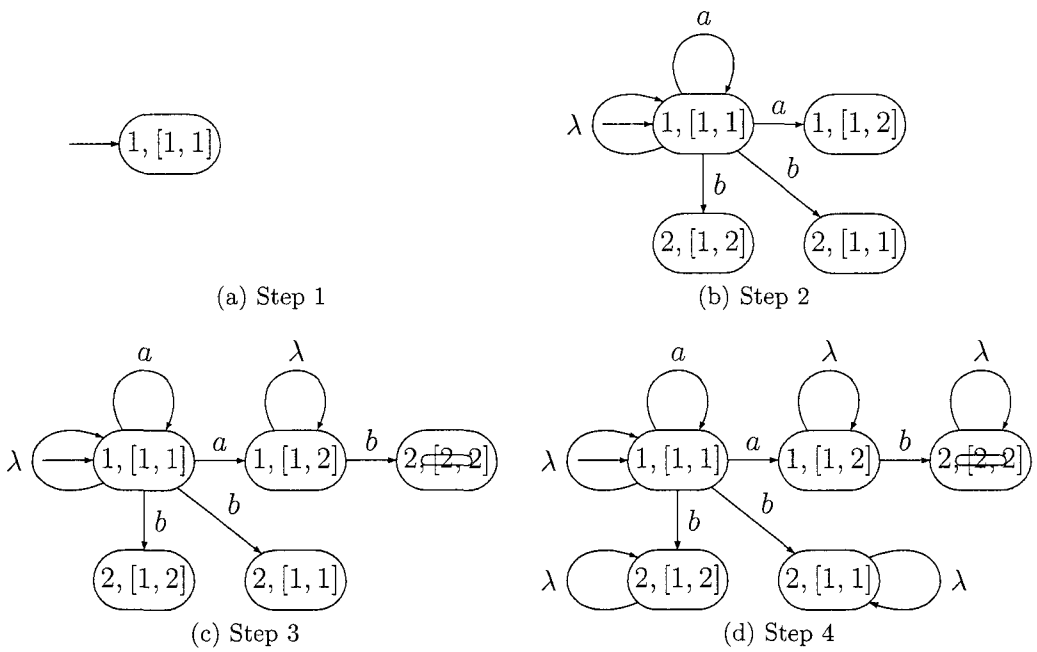


Figure 6.13: Steps in building an automaton representing the language $t_s^{-1}(b) \cap L_2$.

Any of the words in the language represented by the automaton in Fig. 6.13d paired up with word b would create a counterexample for this case. However, we need only one word. Therefore we apply the Breadth First Search algorithm to this automaton to extract a single word. In this, case the extracted word is ab. The pair (ab, b) provides

a desired counterexample. Both words ab and b belong to the language L_2 and b is a suffix of ab. It clearly shows that L_2 is not a suffix code.

The complexity of this method is dependant on the case being consider. If the given language does belong to the given property than the complexity is constant and equal to 0 because the method is not invoked. In case the method is invoked, the complexity rises to $O(|t| * |w| * |A|)$ where $|t|$ is the size of a transducer t describing the property, $|w|$ is the size of the word that was found in the first step and $|A|$ is the size of the automaton accepting the tested language.

6.5.2 Second Approach

Recall our main method. The first major step of that method is building an automaton representing $t(L)$, where t is a transducer describing a property and L is a given language. When building an automaton representing such a language, we look at transitions in the automaton A representing L and transitions in the transducer t. For every transition (p, x, q) in A and $(p', x/y, q')$ in t, we create the transition $((p, p'), y, (q, q'))$ in the automaton representing $t(L)$. Now we modify this step so that not only the output symbol by also the input symbol is preserved. So now we create the transition $((p, p'), y(x), (q, q'))$. We also preserve the input symbols while building the automaton representing $L \cap t(L)$. This modification allows us to build a final automaton with accepting paths corresponding to every possible pair of words u, w where $w \in L$ and $u \in t(w)$. After this modification every accepting path, if such exists, in the automaton accepting $L \cap t(L)$ represents a counterexample.

Example: *Let us again consider the language* $L_2 = a^*b$ *(Fig 4.3) and the transducer* t_s *(Fig. 4.6) describing the suffix property over the alphabet* $\{a, b\}$. *We use the product construction, modified as described above, to construct the automaton* $t_s(L_2)$ *(Fig. 6.14).*

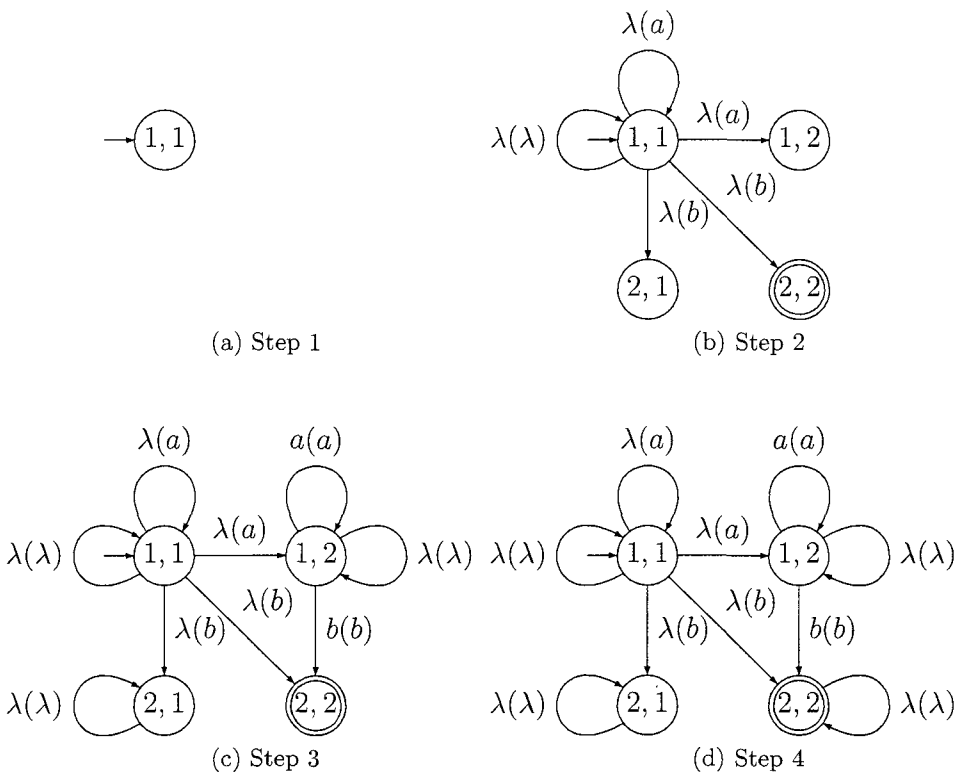
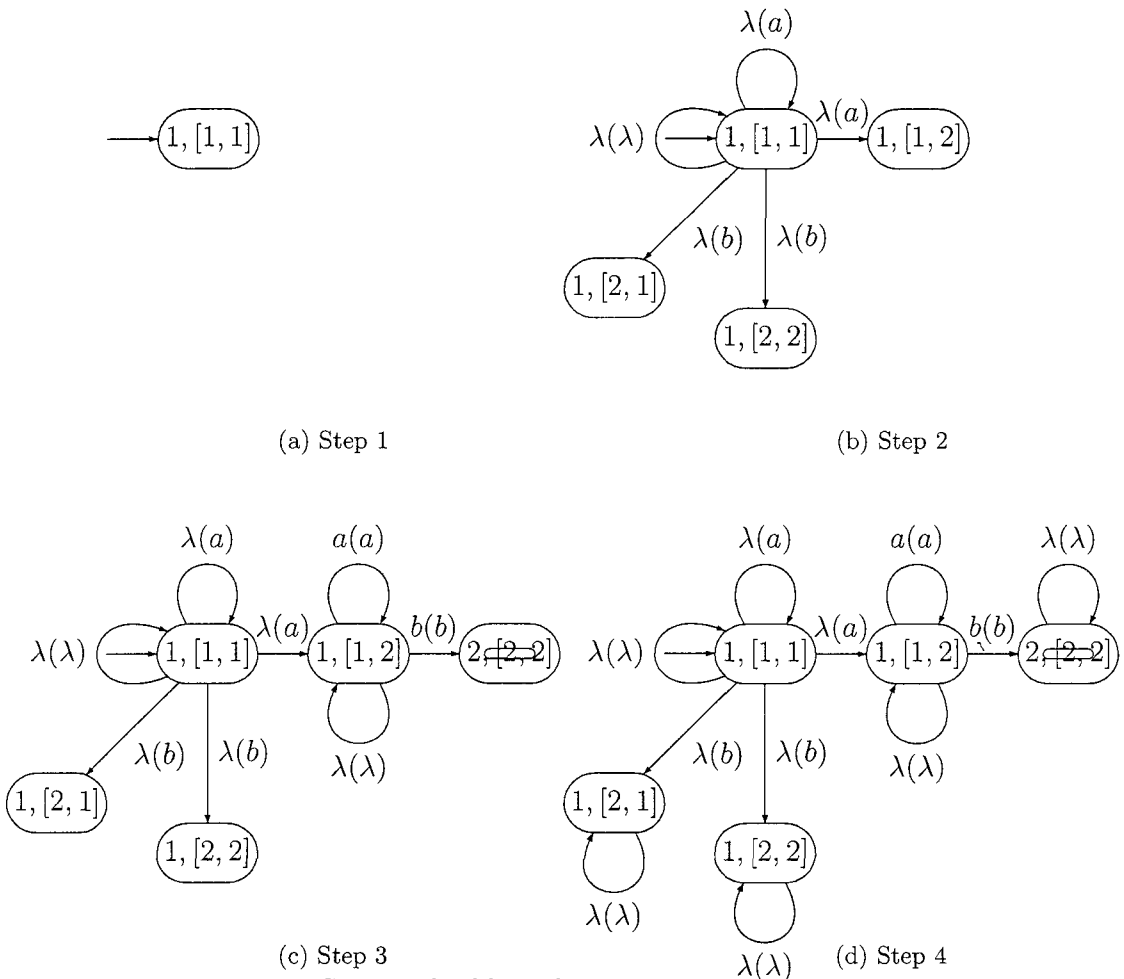


Figure 6.14: Steps in building an automaton representing the language $t_s(L_2)$ while keeping track of input symbols

Next, we construct the automaton representing $t_s(L_2) \cap L_2$ *(Fig. 6.15). In this step we also preserve the input symbol for every transition.*

Every path in the automaton in Fig. 6.15d represents a pair of words (w, v) *with*



(a) Step 1

(b) Step 2

(c) Step 3

(d) Step 4

Figure 6.15: Steps in building the automaton representing the language $t_s(L_2) \cap L_2$ while keeping track of input symbols.

$v, w \in L_2$ *and* $w \in t(L_2)$. *Therefore every accepting path in this automaton represents a counterexample proving that* L_2 *does not satisfy the suffix property . For instance the path* $((1, [11]), \lambda(a), (1, [1, 2])), ((1, [1, 2]), b(b), (2, [2, 2]))$ *represents the pair* (b, ab).

Both the time and the space complexity of the second approach with respect to big O notation are the same as the main method without modifications. However, the exact space complexity is worse because every transition in the automaton representing $t(L)$ and the automaton representing $t(L) \cap L$ would have to store two symbols instead of one. .

Chapter 7

Implementation

In addition to developing a theoretical framework, our goal was to provide an implementation of our methods. The implementation consists of two main elements: a user interface and an implementation of algorithms. The algorithms were implemented in the C++ language with Boost libraries [2]. The user interface was developed using Python with the Django [3] web framework. This combination allowed us to take advantage of both the speed of C++ and the convenience of Django as a rapid web application development solution.

At this point we want to acknowledge the existence of some other packages for manipulating finite automata and regular languages. Among others we investigated FIRE Station [12], Grail++ [1], OpenFST [36], AMORE [28], Vaucanson [4], and Fado [35, 31]. In particular we considered using Grail++ but to our knowledge the project is not maintained any more, the features that we wanted to use do not work correctly, and the package does not provide any features for manipulating transducers. The second package that might be used for our method is Fado. This package is maintained and still growing. However, at the time we started our research all necessary feature were not yet available in this toolset. Also, Fado is implemented in Python with

makes it significantly slower than similar implementations in C++.

7.1 Implementation of the algorithms

The back end functionality and data structures are encapsulated in two general classes:

- *Automaton* - encapsulates the logic of single automaton, which can be an NFA or a transducer
- *AutomatonMethods* - a factory class, that encapsulates the logic of all the operations performed on automata and transducers.

The *Automaton* class stores the structure of an automaton or a transducer as a set of transitions, a set of start states (the implementation allows more than one), and a set of final states.

***Example**: The automaton in Fig. 4.3 would be logically represented as follows:*

- *Set of start states:* $\{1\}$
- *Set of final states:* $\{2\}$
- *Set of transitions:* $\{(1, a, 1), (1, b, 2)\}$

The transducer in Fig 4.2 would be represented as follows:

- *Set of start states:* $\{1\}$
- *Set of final states:* $\{2\}$
- *Set of transitions:* $\{(1, a/a, 1), (1, b/b, 1), (1, a/\lambda, 2), (1, b/\lambda, 2), (2, a/\lambda, 2), (2, b/\lambda, 2)\}$

Following are some important functions in the public interface of the *Automaton* class:

```
// Imitialises the automaton with data contained in the vector
// Each item in vector should contain either one transition or
// definition of start or finish state
Automaton(std::vector<std::string> word);

// Fills the automaton based on the input stram
void fill(std::ifstream & infile);

// Expands the automaton with either a new transition, new start state
// or new final state depending on what is contained in the input line
void fillByLine(std::string line);

// Extract input alphabet of the automaton
// by going through all transitions5
std::set<std::string> getAlphabet();

// Adds p--lambda--p transitions to all states inNFA
// (isTransducer=FALSE)
// and  p--lambda/lambda--p to all states in a transducer
// (isTrandducer=TRUE)
void expandWithSelfLambdaTransitions(bool isTransducer);
```

The *AutomataMethods* is a factory class that encapsulate most of the functionality used in our method. Following are some important functions implemented as a part of public interface of this class:

```
// Generates product of two automata
// If last argument is set to true than second argument is expected to be a
// transducer and the result of operation is a language(in form of
// automaton) generated by providing a language(first) to a transducer
// (second)
bool product(Automaton & first,
             Automaton & second,
             Automaton & result,
             bool isTransducer);

// This function takes automaton and expands every transition that has
// more than one input symbol into set of transitions.
bool expandTransitions(Automaton & automata);

// Check if the automaton has an accepting path
bool acceptingPath(Automaton & automata);

// this method also check if automata has an accepting path
// In addition it returns the first encountered accepted word if such
// exists
bool acceptingPathWithWord(Automaton & automata,
                           std::vector<std::string> & word);

// transforms the automaton accepting set of trajectories
// into a transducer. Only transitions in allownd in
// the input automaton are 0 and 1.
bool automataToTransducer(Automaton & first,
                          Automaton & second,
                          Automaton & alphabetSource);

// Creates transducer that for every transition p---x/y---q in the
// original transducers has a transition p---y/x--q
bool invertTransducer(const Automaton & original, Automaton & inverted);

// The main function that encpsulates our whole method
boost::python::tuple testLanguage(Automaton & testedLanguageAutomaton,
                                  Automaton & propertyTransducer);
```

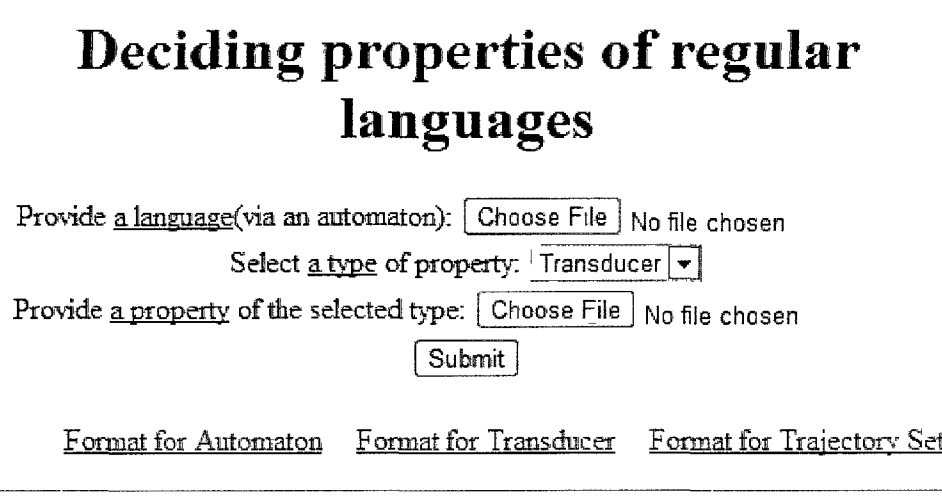
The two classes are exposed to Python using BOOST Python feature. Using this feature we compiled our C++ implementation into a set of dynamic libraries that are

accessible from Python.

7.2 User Interface

Our method is available in the form of web application under following address:

http://laser.cs.smu.ca/transducer/



This website is the user interface for the application based on our method. It was written in Python with a use of Django web development package. To use that interface a user has to provide two files:

- a file containing the description of an automaton that represents a language
- a file containing the description of a transducer that describes a property, or an automaton that accepts the set of trajectories that describes a property; the user has to choose the type of machine

The format of files containing automata and transducers follows the format in Grail++ [1] package. An automaton, both in case of a language and a set of trajectories is described in a file as follows:

- a line of the form $p \ \sigma \ q$ describes a single transition, where p is the origin state of that transition, q is the destination state, and σ is the label of such transition
- a line of the form (START) $|-$ p denotes a start state of the automaton
- a line of the form $q \ -|$ (FINAL) denotes a final state of the automaton

States have to be represented by non negative integers. Labels can consist of a sequence of alphanumerical symbols. However, each label is considered a single letter over the alphabet accepted by given automata (e.g. line 1 $a2$ 4 means that $a2$ is a single symbol in the alphabet of the language accepted by the automaton described in such file). An automaton can have multiple final states and multiple start states defined.

Example: *The content of a file describing the automaton accepting* $aaa(aaa)^*b + aa(ba)^*a(aa(ba)^*a)^*b$*:*

```
(START) |- 1
1 a 2
2 a 3
3 b 2
3 a 4
4 a 2
4 b 8
1 a 5
5 a 6
```

```
6 a 7
7 a 5
7 b 8
8 -| (FINAL)
```

The content of a file describing the automaton accepting the set of trajectories $T =$ $1^*0^*1^*$ *describing the infix property:*

```
(START) |- 1
1 1 1
1 0 2
2 0 2
2 1 3
3 1 3
1 -| (FINAL)
2 -| (FINAL)
3 -| (FINAL)
```

A description of a transducer expands the description of an automaton:

- a line of the form (START) $|-$ p denotes the start state of the transducer
- a line of the form q $-|$ (FINAL) denotes a final state of the transducer
- a line of the form p σ_1 σ_2 q describes a single transition in the transducer, where σ_1 is an input symbol and σ_2 is the output symbol for that transition

Similarly as in case of an automaton, states have to be described by non-negative integers. Both, input and output symbols can be described using any charter sequence except *lambda*. The word *lambda* is reserved for representing the empty word λ.

Example: *An example file that describes a transducer describing the infix property.*

```
(START) |- 1
1 a lambda 2
1 b lambda 2
1 a a 5
1 b b 5
2 a lambda 2
2 b lambda 2
2 a a 3
2 b b 3
3 a a 3
3 b b 3
3 a lambda 4
3 b lambda 4
4 a lambda 4
4 b lambda 4
5 a a 5
5 b b 5
5 a lambda 4
5 b lambda 4
2 -| (FINAL)
3 -| (FINAL)
4 -| (FINAL)
```

After the user submits a file with a language and a file with a property and clicks the *Submit* button, our application computes the answer. If the submitted language belongs to the submitted property then the user simply gets a confirmation of that fact. Otherwise the user receives a message saying that the language does not satisfy the property and a counterexample, that is a pair of words in this language violating the property.

Chapter 8

Conclusion and Future Work

8.1 Conclusion and Discussion

In this thesis we introduced a method for describing and deciding properties of regular languages using input altering transducers. We discussed the relation of our method to other ones, and explored our method limitations. Moreover we implemented our method and created a web application that makes our method available to the research community.

We introduced the concept of input altering transducer and we showed how input altering transducers can be used to describe many classical and more recently established code properties, including but no limited to prefix codes, suffix codes, infix codes, bifix codes. overlap-free languages, thin languages, and solid codes.

There are uncountably many code related properties. Our method is applicable only to a subset of those properties because not all independence properties (and therefore code related properties) can be represented using input altering transducers. As an example we provided an explanation and an extensive proof for the fact that the reverse property cannot be represented using a transducer.

In Chapter 7 we took a closer look at our method. We showed that if a transducer t represents a certain property in the context of our method, t^{-1} represents the same property. We also discussed our method in relation to other methods. In particular we provided a method for transforming a set of trajectories into a corresponding input altering transducer. Moreover, we discussed how to construct a counterexample for a language L and a property P, provided $L \notin P$.

One of our main research goals was to implement our method in the form of a web application. In Chapter 8 we provided some details of the implementation. We also included a brief tutorial on how to access and use our system, along with example input files. To our knowledge, this is the first publicly available implementation of a method for deciding language properties, where the set of properties is not fixed.

8.2 Future Work

Our method was developed for testing if a regular language satisfies a given property. In our opinion the next step is investigating methods for generating a language that satisfies some given properties. Such methods could be developed for finite languages, or if possible for infinite languages where the output would be an automaton accepting the generated language. Also, our method require input altering transducers, which limits the set of properties that can be described. Therefore, the other possible direction for research would be discovering and investigating more powerful methods for describing and deciding properties of languages.

As with any implemented system, ours also is open for improvements and enhancements. As mentioned in Chapter 7, our web application, in addition to transducers, is able to process properties described by trajectory sets. However, an input trajectory set has to be provided in a form of an automaton. A more standard way of describing trajectory sets is via regular expressions. Therefore, a natural enhancement would be to provide the option in our application to input trajectory sets in a form of regular expressions.

When we started thinking about implementing our method as a web application, our goal was to lay a foundation for what we hope will become a more complete application for manipulating languages. Therefore, the more broad direction for future research and development is to implement new features that will provide users of our system with set of useful tools for researching formal languages.

Bibliography

[1] Grail++, October 2009. `http://www.csd.uwo.ca/Research/grail/`.

[2] Boost C++ libraries, October 2010. `http://www.boost.org/`.

[3] Django framework, October 2010. `http://www.boost.org/`.

[4] Vaucanson, June 2011. `http://www.lrde.epita.fr/cgi-bin/twiki/view/Vaucanson/`.

[5] J. Berstel and D. Perrin. *Theory of codes.* Academic Press Orlando, 1985.

[6] A. Carpi. Overlap-free words and finite automata. *Theor. Comput. Sci.*, 115:243–260, July 1993.

[7] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956.

[8] M. Domaratzki. Deletion along trajectories. *Theor. Comput. Sci.*, 320:293–313, June 2004.

[9] M. Domaratzki. Trajectory-based codes. *Acta Inf.*, 40(6-7):491–527, 2004.

[10] M. Domaratzki and K. Salomaa. Codes defined by multiple sets of trajectories. *Theor. Comput. Sci.*, 366(3):182–193, 2006.

[11] K. Dudzinski and S. Konstantinidis. Formal descriptions of code properties: decidability, complexity, implementation. *Int. J. of Foundation of Computer Science*, to appear in 2011.

[12] M. Frishert, L. G. Cleophas, and B. W. Watson. Fire station: An environment for manipulating finite automata and regular expression views. In *CIAA'04*, pages 125–133, 2004.

[13] A. Ginzburg. *Algebraic theory of automata.* ACM monograph series. Academic Press, New York, NY, 1968.

[14] C. Hagenah and A. Muscholl. Computing epsilon-free nfa from regular expressions in o(n log(n)) time. In *Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science*, MFCS '98, pages 277–285, London, UK, 1998. Springer-Verlag.

[15] Yo-Sub Han and Derick Wood. Overlap-free regular languages. In Danny Chen and D. Lee, editors, *Computing and Combinatorics*, volume 4112 of *Lecture Notes in Computer Science*, pages 469–478. Springer Berlin / Heidelberg, 2006.

[16] T. Head and A. Weber. Deciding code related properties by means of finite transducers. *Proc. Sequences II, Methods in Communication, Security, and Computer Science*, pages 260–272, 1993.

[17] J. L. Hein. *Discrete Structures, Logic and Computability.* Jones and Bartlett Publishers, Inc., USA, 1994.

[18] P. T. Hu, K. V. Hung, and D. L. Van. Codes and length-increasing transitive binary relations. In Dang Van Hung and Martin Wirsing, editors, *Theoretical Aspects of Computing ICTAC 2005*, volume 3722 of *Lecture Notes in Computer Science*, pages 29–48. Springer Berlin - Heidelberg, 2005.

[19] M. Ito, H. Jürgensen, H. J. Shyr, and G. Thierrin. Outfix and infix codes and related classes of languages. *J. Comput. Syst. Sci.*, 43(3):484–508, 1991.

[20] J. Johnson. Rational equivalence relations. *Theor. Comput. Sci.*, 47:39–60, November 1986.

[21] H. Jürgensen. Syntactic monoids of codes. *Acta Cybern.*, 14(1):117–134, 1999.

[22] H. Jürgensen. Maximal solid codes. *J. Autom. Lang. Comb.*, 6:25–50, January 2001.

[23] H. Jürgensen and S. Konstantinidis. The hierarchy of codes. In *Fundamentals of Computation Theory*, volume 710 of *Lecture Notes in Computer Science*, pages 50–68. Springer Berlin - Heidelberg, 1993.

[24] H. Jürgensen and S. S. Yu. Solid codes. *J. Inform. Process. Cybern*, 26:563–574, 1991.

[25] H. Jürgensen and S. S. Yu. Dependence systems and hierarchies of families of languages. *Unpublished manuscript*, 1995.

[26] L. Kari and S. Konstantinidis. Language equations, maximality and error-detection. *J. Comput. Syst. Sci.*, 70:157–178, February 2005.

[27] L. Kari and P. Sosík. On language equations with deletion. *Bulletin of the EATCS*, 83:173–180, 2004.

[28] V. Kell, A. Maier, A. Potthoff, W. Thomas, and U. Wermuth. Amore: A system for computing automata, monoids and regular expressions. In *Proceedings of the 6th Annual Symposium on Theoretical Aspects of Computer Science*, pages 537–538, London, UK, 1989. Springer-Verlag.

[29] Al. A. Markov. Nonrecurrent coding. *Problemy Kibernetiki*, (8):169186, 1961.

[30] A Mateescu, G Rozenberg, and A Salomaa. Shuffle on trajectories: Syntactic constraints. *Theoretical Computer Science*, 197(1-2):1 – 56, 1998.

[31] N. Moreira and R. Reis. Interactive manipulation of regular objects with fado. *SIGCSE Bull.*, 37:335–339, June 2005.

[32] G. Paun and A. Salomaa. Thin and slender languages. *Discrete Applied Mathematics*, 61(3):257–270, 1995.

[33] E. L. Post. A variant of a recursivly unsolvable problem. *Bulletin of the American Mathematical Society*, 52, 1946.

[34] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res. Dev.*, 3:114–125, April 1959.

[35] Rogrio Reis and Nelma Moreira. FAdo:tools for finite automata and regular expressions manipulation. Technical Report DCC-2002-2, DCC-FC& LIACC, Universidade do Porto, August 2002.

[36] M. Riley, C. Allauzen, and M. Jansche. Openfst: an open-source, weighted finite-state transducer library and its applications to speech and language. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume: Tutorial Abstracts*, NAACL-Tutorials '09, pages 9–10, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics.

[37] M. Rodeh. A fast test for unique decipherability based on suffix trees. *Information Theory, IEEE Transactions on*, 28(4):648–651, July 1982.

[38] S. J. Russell and P. Norvig. *Artificial Intelligence: a modern approach.* Prentice Hall, 2nd international edition, 2003.

[39] H. Shyr and G. Thierrin. Codes and binary relations. In Marie Malliavin, editor, *Sminaire dAlgbre Paul Dubreil Paris 19751976 (29me Anne)*, volume 586 of *Lecture Notes in Mathematics*, pages 180–188. Springer Berlin / Heidelberg, 1977. 10.1007/BFb0087133.

[40] H. Shyr and S. Yu. Solid codes and disjunctive domains. *Semigroup Forum*, 41:23–37, 1990. 10.1007/BF02573375.

[41] H. J. Shyr. *Free Monoids and Languages.* Hon Min Book Company, Taichung, Taiwan, 1991.

[42] S. Yu. *Regular languages*, volume 1 of *Handbook of formal languages*, pages 41–110. Springer-Verlag New York, Inc., New York, NY, USA, 1997.