Application and Implementation of Transducer Tools in Answering Certain Questions About Regular Languages

By

Meng Yang

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

December 3, 2012, Halifax, Nova Scotia

Abstract

Application and Implementation of Transducer Tools in Answering

Certain Questions About Regular Languages

By

Meng Yang

Abstract: In this research, we investigate, refine, and implement algorithmic tools that allow us to answer decision questions about regular languages. We provide a thorough presentation of existing algorithmic tools to answer the satisfaction questions of whether a given language satisfies a given property described by an input-preserving transducer, which is equivalent to the question of whether a given language is error-detecting for the channel realized by the same input-preserving transducer; whether a given language is error-correcting for the channel realized by an input-preserving transducer; whether a given regular language satisfies the code property. In the process, we give a thorough presentation of an existing algorithm to decide whether a transducer is functional and an algorithm about how to translate a normal form transducer into a real-time transducer. We also introduce our method to provide counterexamples in cases where the answers to the satisfaction questions are negative. In addition, we discuss our new method to estimate the edit distance of a regular language by the error-correction property, which is much faster than the existing method of computing the edit distance via error-detection. Finally, we deliver an open implementation of these algorithms and methods via a web interface – I-LaSer, and add the implementation of transducer classes into our copy of the FAdo libraries.

December 3, 2012.

Acknowledgements

I would like to thank my supervisor, Dr. Stavros Konstantinidis for his guidance, kind advice, and his support for my study and research at Saint Mary’s University. I also appreciate the support given to me by my thesis committee members, Dr. Cezar Campeanu and Dr. Wendy Finbow Singh, and the external examiner Dr. Rogerio Reis.

Special thanks go to Dr. Pawan Lingras, who not only provided the access of the super computer for us to run the performance tests at Saint Mary’s University, but also gave me many suggestions about my research and career choosing. Thank you for your help.

I would also like to thank my fiancée, Nancy, for her understanding and supporting during my research. In addition, many thanks also give to my parents who supported me to come to Canada to take this program.

Many thanks go to all my friends and colleagues in department of mathematics and computing science. I appreciate your help and kind advice very much.

Finally, many thanks go to faculty of graduate studies and research and Saint Mary’s University for provide this precious opportunity for me to make my dream of studying abroad come true.

Halifax, Nova Scotia Meng Yang

December 2012

Table of Contents

Chapter 1
Introduction

1.1 About our research

In computer science and information transmission, the information that needs to be stored or transmitted usually has to be encoded into a certain format. For example, sending an image via a network requires encoding the image into a certain binary word whose bits are interpreted as signals that can be transmitted over the network. In most applications, such as data compression and signal transmission, the information involved is represented by words of a formal language over some alphabets, which are required to follow a certain restriction or posses a certain common feature. A language property is the set of all languages that follow a certain restriction or posses a certain common characteristic.

There are many language properties, such as the code property, the prefix code property, the suffix code property, etc. A regular language may satisfy particular properties, which can be described by transducers. Researchers [15, 16] have investigated algorithms and methods to answer the question of whether a given language satisfies a particular code related property, and have delivered an implementation of these algorithms and methods which is accessible via a web interface – LaSer [36]. The motivation of our research is to enhance the capabilities of the existing LaSer. The limitation of LaSer is that it can only answer the satisfaction question for language properties described by input-altering transducers. Some language properties are described by input-preserving transducers, and we usually use input-preserving transducers to simulate channels, in order to decide the error-detection and error-correction properties of a regular language. The question of deciding whether a language satisfies a given language property described by an input-preserving transducer is not addressed in [15, 16]. In addition, LaSer does not solve the problem of deciding whether a language satisfies the code property. Also, the error-detection property of a regular language is investigated in [10, 11] only for sequential transducers, and the question of deciding the error-correction property is not addressed there.

The objectives of our research are to refine and implement algorithmic tools that allow us to answer the above unresolved and related questions, and to strengthen the capabilities of LaSer. These algorithmic tools involve automaton tools and transducer tools. The main contributions of our research are:

1. Thorough presentation of an existing algorithm to decide whether a transducer is functional. This algorithm applies to different types of transducers: restricted sequential transducer, standard form transducer, and real-time transducer. In the case of real-time transducers, we introduce a pre-functionality test to speed up, in some cases, the decision process.
2. Thorough presentation of existing algorithmic tools to answer the following satisfaction questions:

   A. Whether a given language satisfies a given property described by an input-preserving transducer, which is equivalent to the question of whether a given language is error-detecting for the channel realized by the same input-preserving transducer.
   B. Whether a given language is error-correcting for the channel realized by an input-preserving transducer.
   C. Whether a given regular language satisfies the code property.
3. Propose an algorithm to generate counterexamples in cases where the answers to the above satisfaction questions are negative.
4. Present an algorithm to translate a given transducer in normal form into an equivalent real-time transducer, based on the mathematical method of [42].
5. Provide a new method to estimate the edit distance of a regular language in quadratic time, improving the previous known method in terms of time complexity.
6. Implementation of transducer classes and their integration into our copy of the FAdo libraries [2, 18]. Implementation of the above algorithmic tools and development of a new web interface – I-LaSer [24], which is an upgraded version of the existing LaSer.

1.2 Thesis structure

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

In Chapter 2 we give some general notions, notation and background information about words, languages, automata, transducers, and Cartesian product operations that are crucial to our research.

In Chapter 3 we look at some known language properties, as well as the methods of describing language properties using input-altering transducers and input-preserving transducers. We explain two language properties for a combinatorial channel: the error-detection property and the error-correction property. We also provide two propositions about how to decide the error-detection property and the error-correction property.

In Chapter 4 we mainly focus on the algorithmic tools and decision algorithms to answer the questions we mentioned above. We explain how to construct product machines and how to decide the functionality of a given transducer. Afterwards, we show how to answer the satisfaction questions, such as whether a given language is error-detecting or error-correcting for a channel, and whether a given language is a code. In addition, we present our algorithm to generate a counterexample in the case where a transducer is not functional. Also, we give an algorithm to construct input-altering transducers that describe certain fixed properties over a given alphabet, and an algorithm to translate a transducer in normal form into an equivalent real-time transducer.

In Chapter 5 we discuss the concept of edit distance, and two methods to compute the edit distance of a regular language using the error-detection property and the error-correction property. We present an algorithm to construct an input-preserving transducer realizing the channel $SID(m,\infty)$, based on a given positive integer number m and a given alphabet Σ. In addition, we explain how to use the algorithmic tools in Chapter 2 and Chapter 4 to estimate the edit distance of a given regular language by the error-correction property. We also provide two performance tests, where the result shows that our new method is much faster than the existing method.

In Chapter 6 we deliver an implementation of our research through a web interface. We discuss the implementation of our methods and the architecture of this web interface. We also list the basic functions of our web interface and illustrate different file formats that our web application will use. In addition, we give examples of the files in different formats in our system.

The final Chapter 7 contains a summary of our research and directions for future research and implementations.

Chapter 2

Basic notions and notation

In this chapter we give some general notions, notation and background information about words, languages, automata, transducers, and Cartesian product operations that are important to our research. Readers are referred to [17, 22, 31, 37, 42, 47] for more information about these concepts.

2.1 Sets, words, languages, relations

2.1.1 Set

The cardinality of a finite set S, denoted by $|S|$, is the number of elements in S. The power set of a set S, denoted by 2^S, is the set of all the subsets of S.

2.1.2 Alphabet, word, and language

An alphabet is a finite, nonempty set of symbols. Conventionally, we use the symbol Σ for an alphabet. A word (also called string or message) w is a finite sequence of symbols chosen from an alphabet. The empty word, denoted by λ, is the string with zero occurrences of symbols. If Σ is an alphabet, then Σ^* is the set of all words over Σ including the empty word λ, and Σ^+ is $\Sigma^* - \{\lambda\}$. The standard notation for the length of a word w is $|w|$. Any set of words is called a language. The basic operation on words is concatenation. Let x and y be words. Then xy denotes the concatenation of x and y.

Example: *The alphabet* $\Sigma = \{0, 1\}$ *consists of two symbols:* 0 *and* 1. *The set* $L = \{0, 11, 01\}$ *is a language consisting of three words over* Σ. *If* $x = 11$ *and* $y = 01$, *then the concatenation of* x *and* y *is* $xy = 1101$, *and the length of* xy *is* $|xy| = |1101| = 4$.

2.1.3 Binary relation

A binary word relation R over two alphabets A and B is a subset of $A^* \times B^*$. The binary relation R consists of pairs of words (u, v) for some $u \in A^*$ and $v \in B^*$. The domain of R is $\{u|(u, v) \in R\}$, and the co-domain of R is $\{v|(u, v) \in R\}$. The inverse of R, denoted by R^{-1}, is the binary relation $\{(b, a)|(a, b) \in R\}$ over B and A. Unless specified otherwise, we use the term relation to refer to binary relation in this thesis.

In this thesis, two relations play an important role. The relation $R \cap (L \times B^*) = \{(a, b) \in R|a \in L\}$, denoted by $R \downarrow L$, is the relation R with its domain restricted to L. Also, the relation $R \cap (A^* \times L) = \{(a, b) \in R|b \in L\}$, denoted by $R \uparrow L$, is the relation R with its co-domain restricted to L.

The relation R is functional if $(a, b_1) \in R$ and $(a, b_2) \in R$ imply that $b_1 = b_2$.

2.2 Regular languages and automata

In this research, we focus on regular languages which are exactly all the languages that can be accepted by finite state automata. In addition, regular languages can be described by regular expressions. Readers are referred to [8, 22, 47] for more information about regular languages and regular expressions.

A λ-NFA (Lambda Nondeterministic Finite Automaton) consists of a finite set of states and a set of transitions. The transitions set the change of the current state when reading a given input. Formally, a λ-NFA is a 5-tuple $A = (Q,\ \Sigma,\ E,\ q_0,\ F)$ such that:

- Q is a finite nonempty set of states.
- Σ is the input alphabet.
- E is the set of transitions, which are 3-tuples of the form (p, x, q) with $p, q \in Q$ and $x \in \Sigma \cup \{\lambda\}$. The element x is called the label of the transition. A transition labeled with the empty word λ is called a λ-transition.
- q_0 is the start state.
- F is the set of final states.

It is convenient to present λ-NFAs as directed graphs as in Figure 2.1. In Figure 2.1, states are portrayed as small circles. Transitions are denoted by edges with arrows pointing from the origin state to the end state. Transitions are labeled with symbols from $\Sigma \cup \{\lambda\}$. The start state q_0 is shown with a short incoming arrow pointing to it. The final states are represented as two concentric circles.

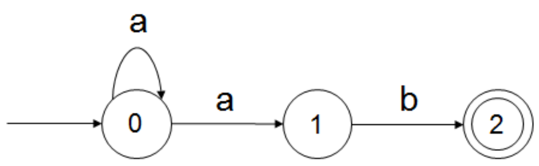


Figure 2.1: An example λ-NFA with $Q = \{0, 1, 2\}$, $\Sigma = \{a, b\}$, the start state $q_0 = 0$ and the set of final states $F = \{2\}$.

If every transition in E of a λ-NFA has a nonempty label, then the automaton is called <u>NFA</u>. The example in Figure 2.1 is also an NFA. In an NFA, if for every input label $x \in \Sigma$ and every state $p \in Q$, there is at most one transition (p, x, q) going to some $q \in Q$, then the NFA is called <u>DFA</u> (Deterministic Finite Automaton). The example in Figure 2.1 is not a DFA because when given a symbol a to the current state 0, there are two different states 0 and 1 that can be reached from to the current state 0. An example of DFA is illustrated in Figure 2.2. Unless specified, we use the term <u>automaton</u> to refer to λ-NFA in this thesis.

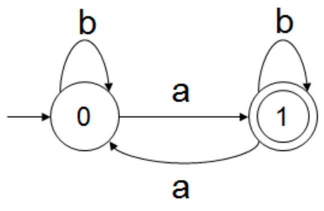


Figure 2.2: An example of DFA.

We say that a word is <u>accepted</u> by an automaton if that word is formed by concatenating the transition labels in a path that begins from the start state to a final state. An automaton A can process the input words and decide whether or not to accept these words. In particular, given any input word w to the start state q_0, the word w is accepted if the automaton reads all the symbols in w by following a sequence of the available transitions

$$(q_0, x_1, q_1),\ (q_1, x_2, q_2), \cdots, (q_{n-1}, x_n, q_n),$$

such that $w = x_1 x_2 \cdots x_n$ and the state q_n is in the set of final states F.

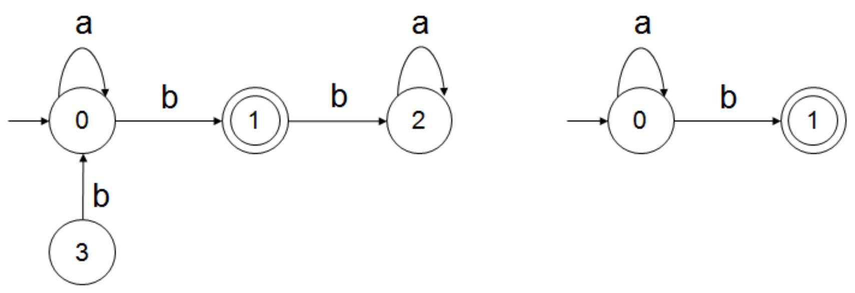
We say that a language L is <u>accepted</u> or <u>represented</u> by an automaton A, if every word in this language L is accepted by A and every word accepted by A belongs to L. We say that the path from the start state to a final state, which accepts a given input word, is an <u>accepting path</u>. The <u>diameter</u> of an automaton A, denoted by $diam(A)$, is the largest number of states in a computation $p_0 a_1 p_1, \cdots, a_n p_n$ for which p_0 is the start state and no state occurs more than once, that is, $i \neq j$ implies $p_i \neq p_j$. For example, the diameter of the automaton A in Figure 2.1 is $diam(A) = 3$.

Example: *Let us consider the automaton A in Figure 2.1. In A, state 0 is the start state. There are three transitions in A. The transition $(0, a, 1)$ takes us from the start state 0 to state 1 while accepting the symbol a. From state 1, the transition $(1, b, 2)$ takes us to state 2, which is a final state. Therefore, ab is one of the words accepted by A. Note that there is also another transition $(0, a, 0)$ taking us from state 0 to state 0. This means that we can accept infinitely many symbols a when we start at state 0. The set of words that the automaton in Figure 2.1 will accept is $\{ab, aab, aaab, aaaab, \ldots\}$. Therefore, the automaton A accepts the language $L = a^*b$.*

For every λ-NFA we can construct an equivalent NFA with no λ-transitions. Methods for constructing NFA based on λ-NFA are well known and therefore we will not present them in this thesis. Readers are referred to [22, 37, 47] where such methods are discussed.

The automaton is said to have accessible states if, for every state q in the state set Q, there is a path from the start state q_0 to q. In some cases, some states in an automaton cannot be reached from the start state, or none of the final states can be reached from such states. The process of erasing such states is called a trimming operation and the obtained automaton is called a trimmed automaton. Technically, an automaton is called trim if it has accessible states and, for every accessible state q, there is a path from q to one of the final states in final state set F. The trimming operation does not affect the language accepted by an automaton.

Therefore, for the sake of complexity of automata processing and operating, we sometimes need to make an automaton trim. From now on, all the automata in this paper are assumed to be trim. If an automaton, after operating, is not trim, we need to make it trim. Readers can refer to details of the trimming processing in [9, 37]. An example of an automaton before and after the trimming operation is shown in Figure 2.3.



(a) Untrimmed automaton (b) Trimmed automaton

Figure 2.3: An automaton accepting the language $L = a^*b$

untrimmed (left) and trimmed (right).

2.3 Cartesian product of two automata

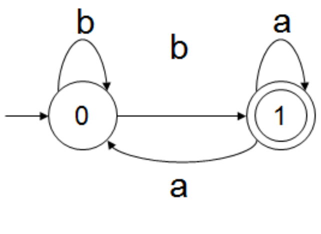
In automaton theory, the Cartesian product construction produces a new automaton out of two given automata. If the automaton A_1 accepts the language L_1 and the automaton A_2 accepts the language L_2, then we can construct a new automaton $A_3 = A_1 \cap A_2$ accepting the language $L_3 = L_1 \cap L_2$ as follows:

1. The states of the automaton A_3 are pairs of (p_1, p_2) of states where p_1 is a state in A_1 and p_2 is a state in A_2.
2. The start state in A_3 is the pair of start states from A_1 and A_2. For example if p_0 is the start state in A_1 and $p_0{}'$ is the start state in A_2, then we construct the pair of state $(p_0, p_0{}')$ as the start state in A_3.
3. The set of final states in A_3 consists of all states that are pairs of final states from A_1 and A_2. For example if f is a final state in A_1 and f' is a final state in A_2, then the pair (f, f') is a final state in A_3.
4. For every two transitions with the same label, (p_1, σ, q_1) from A_1 and

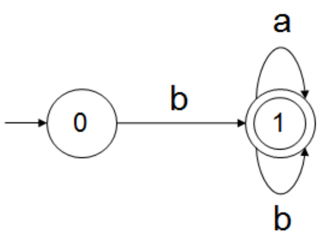
(p_2, σ, q_2) from A_2, we add a transition $((p_1, p_2), \sigma, (q_1, q_2))$ to A_3.

See [15, 22, 37, 47] for more information about the Cartesian product operation of two automata. Next, we give an example of this operation.

Example: *Let us consider two automata, A_1 in Figure 2.4a and A_2 in Figure 2.4b.*



(a) Automaton A_1



(b) Automaton A_2

Figure 2.4

The new automaton A_3 we constructed after Cartesian product operation of A_1 and A_2 is illustrated in Figure 2.5. For example, as $(0, b, 1)$ and $(1, b, 1)$ are transitions in A_1 and A_2, respectively, the tuple $((0,1), b, (1,1))$ is a transition in A_3.

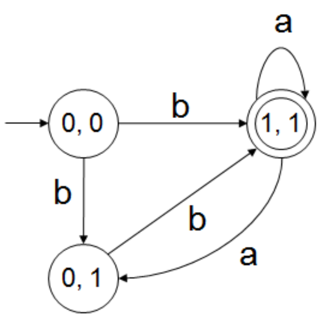
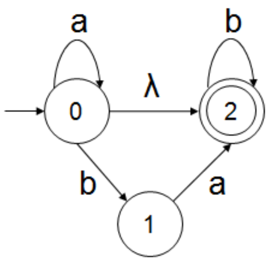


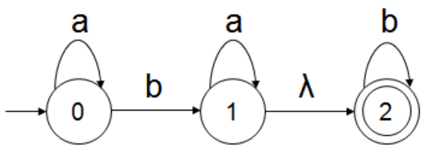
Figure 2.5: Cartesian product of A_1 and A_2 after trimming operation.

The automata A_1 and A_2 in Figure 2.4 we discussed above are two automata with no λ-transitions. However, if one or both have λ-transitions, we have to add self λ-transitions to every state in the two automata (starting from each state and ending at the state itself). Note that adding self λ-transitions to the automaton does not affect the accepted language. After adding self λ-transitions to both automata, we apply the Cartesian product operation in the same way we have discussed above in the case of automata with no λ-transitions, treating λ the same way as any other symbols in the alphabet.

Example: *Let us consider two automata A_1 in Figure 2.6a and A_2 in Figure 2.6b.*



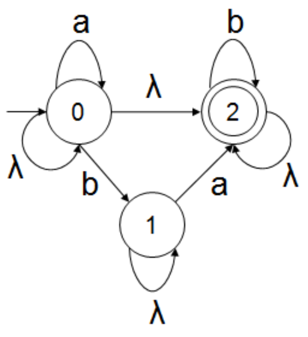
(a) Automaton A_1



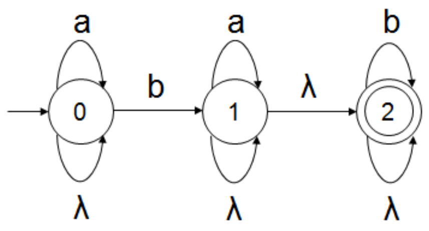
(b) Automaton A_2

Figure 2.6

Because both automata A_1 and A_2 have λ-transitions, we have to expand both A_1 and A_2 with self λ-transitions (Figure 2.7) in order to perform the Cartesian product operation of the automata A_1 and A_2.



(a) Automaton A_1 expanded with self λ-transitions



(b) Automaton A_2 expanded with self λ-*transitions*

Figure 2.7

2.4 Finite state transducers

2.4.1 Definition

A finite state transducer (FST) is a finite state machine that has two labels in each transition; the input label and the output label. See Figure 2.8 for an example of a transducer. The transducer contrasts with an ordinary finite state automaton, which has a single label in each transition. A transducer is said to translate its input to its output, by accepting its input word, as in the case of λ-NFA, and generating its output word. Formally, a finite transducer T in general form is a 6-tuple $T = (Q, \Sigma, \Gamma, q_0, F, E)$ such that:

- Q is the finite set of states.
- Σ is the input alphabet.
- Γ is the output alphabet.
- q_0 is the set of start states.
- F is the set of final states.

- E is the set of transitions, which are tuples of the form $(p, x/y, q)$ such that $p, q \in Q$, $x \in \Sigma^*$, and $y \in \Gamma^*$.

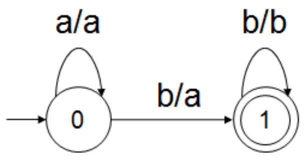


Figure 2.8: An example of a transducer.

A transducer may operate nondeterministically and produce one of many possible output words for a given input word. A transducer may also generate no output for a given input word, in which case it is said to reject the input word.

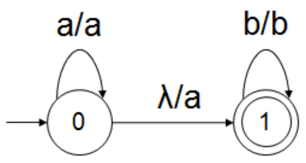
Similar to an automaton, a transducer processes the input words and decides whether or not to accept these input words and obtain the output words. In particular, the computation process is that given any input word $u = x_1 x_2 \cdots x_n$ at the start state q_0, the transducer reads each part $x_1, x_2, \cdots, x_n$ of the word u by following a sequence of the available transitions

$$(q_0, x_1/y_1, q_1),\ (q_1, x_2/y_2, q_2), \cdots, (q_{n-1}, x_n/y_n, q_n).$$

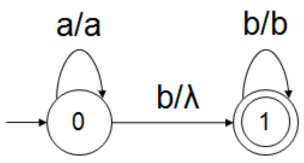
The input word $u = x_1 x_2 \cdots x_n$ is accepted and the word $v = y_1 y_2 \cdots y_n$ is outputted if the state q_n is in the set of final states F. In this case, we say that the pair of words (u, v) is realized, or accepted, by the transducer. In general, a transducer T realizes the binary relation consisting of all pairs (u, v) accepted by T.

Two transducers are called equivalent if they realize the same relation.

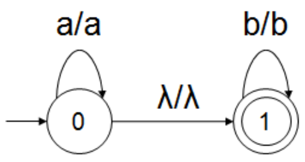
Similarly to λ-NFA, a transducer may have transitions which can involve λ as the input label, called λ-input transitions, or λ as the output label, called λ-output transitions, or both, called λ-transitions. Examples of transducer with transitions involving λ are illustrated in Figure 2.9.



(a) Transducer with λ-input transition.



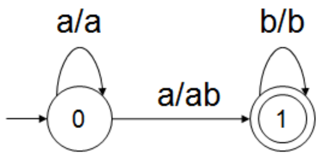
(b) Transducer with λ-output transition.



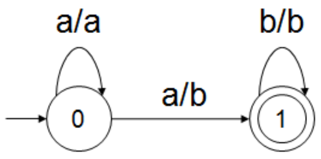
(c) Transducer with λ-transitions.

Figure 2.9 Transducers with transitions involving λ.

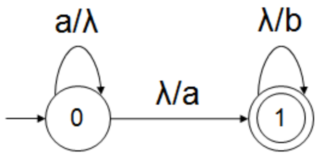
The transducer we defined above is in general form. A transducer is in standard form if, for every transition $(p, x/y, q)$, we have that $x \in \Sigma \cup \{\lambda\}$, and $y \in \Gamma \cup \{\lambda\}$. A transducer is in normal form if it is in standard form and for every transition $(p, x/y, q)$, at least one of x or y is λ. Examples of transducers in general form, standard form, and normal form are illustrated in Figure 2.10.



(a) A transducer in general form



(b) A transducer in standard form

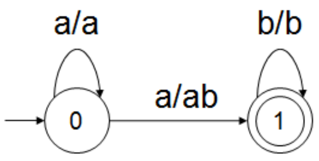


(c) A transducer in normal form

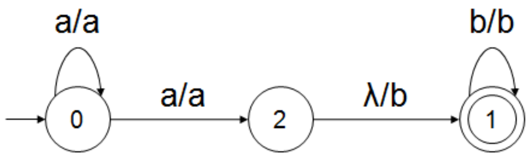
Figure 2.10

A transducer T is called restricted sequential if it is in standard form and, for every transition $(p, x/y, q)$ in T, we have that $x \in \Sigma$ and $y \in \Gamma$, that is, both x and y are not λ. The standard form transducer in Figure 2.10b is also a restricted sequential transducer.

For every transducer T in general form, we can translate T into an equivalent transducer in standard form. Also, for every transducer T' in standard form, we can translate T' into an equivalent transducer in normal form [42]. An example of translating a transducer T in general form to an equivalent transducer in standard form, and then going on to translate it into a normal form one is illustrated in Figure 2.11 and Figure 2.12.



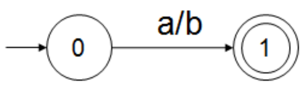
(a) A transducer T in general form



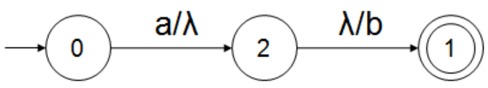
(b) The transducer T' in standard form equivalent to the one in Figure 2.11a

Figure 2.11

Example: *Let us consider the transducer T in <u>general form</u> in Figure 2.12a. In order to translate T into a transducer T' in <u>standard form</u>, we delete the transition $(0, a/ab, 1)$ that is not in accordance with the definition of standard form transducer, and we add a new state 2 and two new transitions $(0, a/a, 2)$ and $(2, \lambda/b, 1)$ to T.Using this method, we obtain the transducer T' in standard form (Figure 2.11b) equivalent to the transducer T in general form (Figure 2.11a).*



(a) A transducer T' in standard form



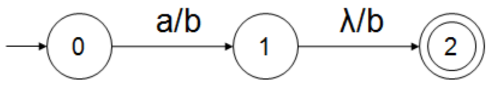
(b) The transducer T'' in normal form equivalent to the one in Figure 2.12a

Figure 2.12

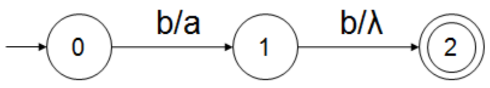
Example: *Let us consider the transducer T' in <u>standard form</u> in Figure 2.12a. In order to translate T' into a transducer T'' in <u>normal form</u>, we delete the transition $(0, a/b, 1)$ that is not in accordance with the definition of normal from transducer, and we add a new state 2 and two new transitions $(0, a/\lambda, 2)$ and $(2, \lambda/b, 1)$ to T''. Using this method we obtain the transducer T'' in normal form (Figure 2.12b) equivalent to the transducer T' in standard form (Figure 2.12a).*

Unless specified otherwise, all the transducers in this paper are always assumed to be in *<u>standard form</u>*.

The <u>inverse</u> of a transducer T, denoted by T^{-1}, is defined to be the transducer constructed by switching the input label with output label for every transition in the transducer T, without any changes to the states. See Figure 2.13 for example. Note that if transducer T realizes a binary relation R, then T^{-1} realizes the binary relation R^{-1}.



(a) An example transducer T



(b) The inverse transducer T^{-1} of transducer T in Figure 2.13a

Figure 2.13

2.4.2 Real-time transducer

A real-time transducer is an extension of a finite state transducer. A transducer T is said to be <u>real-time</u> [42], if for every transition, the input label is a letter in Σ and the output label is a regular expression over Γ, where Σ and Γ are the input and output alphabets of T. We write $REX(\Gamma)$ for the set of regular expressions over Γ.

Formally, a real-time transducer T is a 6-tuple $(Q, I, F, \Sigma, \Gamma, E)$ where:

- Q is the finite set of states.
- $I: Q \to REX(\Gamma)$ is the start states function; $I(q)$ is the regular expression describing the possible words to output before a computation starts at state $q \in Q$. If $I(q) = \emptyset$, then q is not a start state.
- $F: Q \to REX(\Gamma)$ is the final states function; $F(q)$ is the regular expression describing the possible words to output after a computation ends at state $q \in Q$. If $F(q) = \emptyset$, then q is not a final state.
- Σ and Γ are finite sets corresponding respectively to the input and output alphabets of the transducer.
- E is the set of transitions of the form $(p, x/e, q)$ where $p, q \in Q$, $x \in \Sigma$, and $e \in REX(\Gamma)$, that is, e is a regular expression over Γ.

***Example**: The Figure 2.14 shows an example of a real-time transducer. The start states are* 1, 2, 3, *and* 4, *and the final state are* 1 *and* 2. *We have that* $I(2) = (\mathrm{a} + \mathrm{b})$ *and* $F(1) = F(2) = \lambda$.

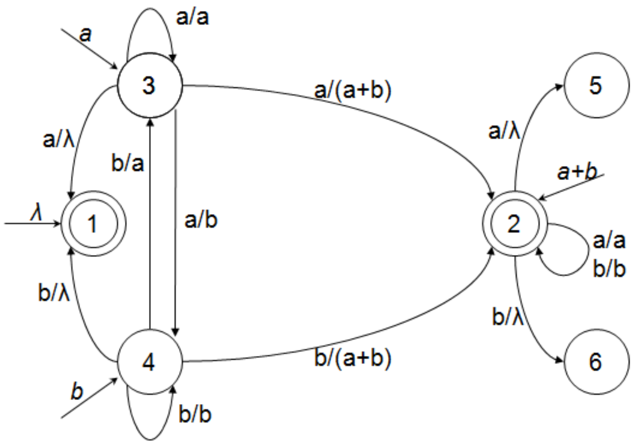


Figure 2.14: An example of a real-time transducer.

A real-time transducer processes the input words and decides whether or not to accept the input words and obtain the output words. The computation process is that given any input word $u = x_1x_2 \cdots x_n$ at the start state q_0, that is $I(q_0) \neq \emptyset$, the real-time transducer reads each part $x_1, x_2, \cdots, x_n$ of the word u by following a sequence of the available transitions

$$(q_0, x_1/e_1, q_1),\ (q_1, x_2/e_2, q_2), \cdots, (q_{n-1}, x_n/e_n, q_n).$$

The input word $u = x_1x_2 \cdots x_n$ is accepted and a word $v \in I(q_0)e_1e_2 \cdots e_nF(q_n)$ is outputted, if the state q_n is in the set of final states F, that is, $F(q_n) \neq \emptyset$. In this case, the pair of words (u, v) is realized or accepted by the real-time transducer. In general, a real-time transducer T realizes the binary relation consisting of the set of all pairs (u, v) such that there is a computation of T where the input word $u = x_1x_2 \cdots x_n$ is accepted and the word $v \in I(q_0)e_1e_2 \cdots e_nF(q_n)$ is outputted.

Given a transducer T in normal form, we can translate T into an equivalent real-time transducer. In Chapter 4, we present an algorithm to translate T into an equivalent real-time transducer, based on the mathematical methods of [42].

2.5 Combinatorial channels

In data communications, a binary message at the site of the sender is sent through a communication channel and arrives at the site of the receiver, and the channel possibly changes some of the bits in the message – called transmission errors. Transducers can be used to simulate communication channels [33], as we give an input word to the transducer and we get a word that may be different from the original given input word, a situation similar to communication channels.

A binary relation C is input-preserving, if for every word w in the domain of C, the pair of words $(w, w) \in C$. A (combinatorial) channel C is a binary relation that is input-preserving. This means that, given input w to the channel C, an output equal to w can be obtained. If $(w, w') \in C$ and $w \neq w'$, then we say that w' contains errors. The concept of channel is necessary when defining the language properties of error-detection and error-correction in Chapter 3.

There exist many different kinds of channels, such as SID channels and Homophonic channels. Definitions and constructions of these channels can be found in [10, 27, 30, 32, 38]. For example, the channel $Sub(m, \infty)$ consists of all pairs (u, v), such that

v results by substituting at most m symbols in u with m different symbols. More precisely, $u = x_1\sigma_1x_2 \cdots \sigma_kx_{k+1}$, with each σ_i being a symbol, and $v = x_1\sigma_1'x_2 \cdots \sigma_k'x_{k+1}$, with each σ_i' being a symbol other than σ_i, where $k \le m$.

Example: *Let us consider the transducer in Figure 2.15 realizing the channel* $Sub(1,\infty)$. *In Figure 2.15, symbols* σ *and* σ' *represent any symbol from the alphabet, but with* $\sigma \ne \sigma'$. *If* σ *represents* 0 *from the alphabet* $\Sigma = \{0,1\}$, *then* σ' *can only represent* 1. *For example, if the input word is* 1111, *then output of the channel can be* 1111 *(no error), or one word from* $\{1110, 1101, 1011, 0111\}$ *(one substitution error).*

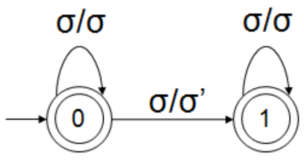


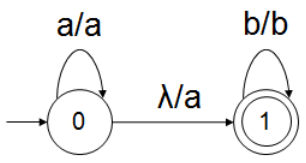
Figure 2.15: The channel $Sub(1,\infty)$.

2.6 Cartesian products of a transducer and an automaton

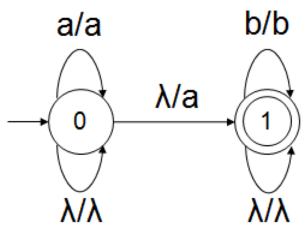
Given an automaton A accepting the regular language L and a transducer T in standard form realizing the binary relation R, we can build a new transducer T', denoted by $T \downarrow A$, realizing the binary relation $R \downarrow L$ by intersecting the input label in transitions of T with the label in transition of A (also called <u>input Cartesian product</u> of a transducer and an automation). Therefore, if a pair of words (u,v) is accepted by T', then (u,v) is accepted by T and the word u is accepted by A.

We construct this new transducer T' using an operation similar to the Cartesian product of two automata. If the automaton A contains λ-transitions or the transducer T contains λ-input transitions, then we have to add self λ-transitions to the automaton A or the transducer T by the following rules:

1. If T has λ-input transitions, then add self λ-transitions to A only. Examples can be found in Figure 2.6 and Figure 2.9.
2. If T doesn't have λ-input transitions and A has λ-transitions, then add self λ-transitions, which are in the form of (λ/λ), to T only. An example is illustrated in Figure 2.16b.



(a) A transducer with λ-input transition

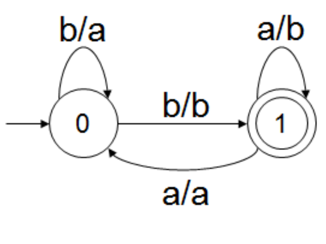


(b) The same transducer expanded with self λ-transition

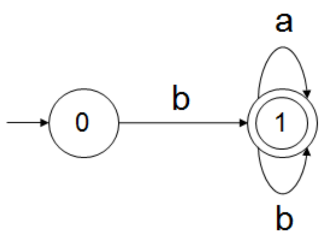
Figure 2.16: Example transducer expanded with self λ-transitions.

The construction of the input Cartesian product of the transducer T and automaton A is as follows: for every transition $(p_1, x/y, q_1)$ in T and (p_2, x, q_2) in A where the input label in $(p_1, x/y, q_1)$ is the same as the label in (p_2, x, q_2), we add a transition $((p_1, p_2), x/y, (q_1, q_2))$ to the new transducer T'; we construct the start state and the set of final states of T' in the same way we construct the Cartesian product of two automata as discussed in Section 2.3.

Example: *Let us consider the transducer T in Figure 2.17a and the automaton A in Figure 2.17b.*



(a) A transducer T



(b) An automaton A

Figure 2.17

In order to construct the input Cartesian product T' of T and A, we add a pair $(0,0)$ as the start state to T'. For the transitions $(0, b/a, 0)$ in T and $(0, b, 1)$ in A, we add the transition $((0,0), b/a, (0,1))$ to T'. Also, for the transition $(0, b/b, 1)$ in T and $(0, b, 1)$ in A, we add the transition $((0,0), b/b, (1,1))$ to T'. For the transition $(1, a/b, 1)$ in T and $(1, a, 1)$ in A, we add the transition $((1,1), a/b, (1,1))$ to T', and for transition $(0, b/b, 1)$ in T and $(1, b, 1)$ in A we add a transition $((0,1), b/b, (1,1))$ to T'. Finally we designate the pair $(1,1)$ as the final state in T'. The new transducer T' we constructed after performing the input Cartesian product operation of T and A and after the trimming operation is illustrated in Figure 2.18.

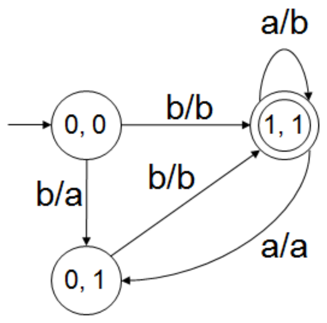
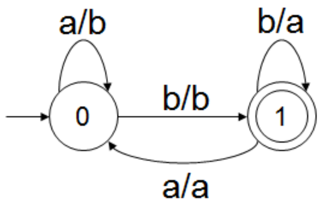


Figure 2.18: Input Cartesian product T' of T and A after trimming operation.

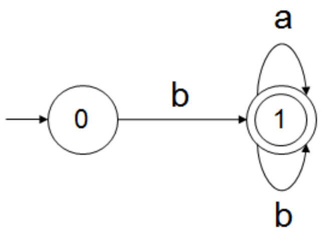
Similarly, given an automaton A accepting the regular language L and a transducer T in standard form realizing the binary relation R, we can build a new transducer T', denoted by $T \uparrow A$, realizing the binary relation $R \uparrow L$ by intersecting the output label in transitions of T with the label in transitions of A (also called output Cartesian product of a transducer and an automaton). In other words, if a pair of words (u, v) is accepted by T', then (u, v) is accepted by T and the word v is accepted by A.

The method of construction the output Cartesian product T' is similar to the method we mentioned above, except that for every transition $(p_1, x/y, q_1)$ in T and (p_2, y, q_2) in A where the output label in $(p_1, x/y, q_1)$ is the same as the label in (p_2, y, q_2), we add a transition $((p_1, p_2), x/y, (q_1, q_2))$ to the new transducer T'.

Example: *Let us consider the transducer T in Figure 2.19a and the automaton A in Figure 2.19b.*



(a) Transducer T



(b) Automaton A

Figure 2.19

In order to construct the output Cartesian product T' *of* T *and* A, *we add a pair* $(0,0)$ *as the start state to* T'. *For the transitions* $(0, a/b, 0)$ *in* T *and* $(0, b, 1)$ *in* A, *we add the transition* $((0,0), a/b, (0,1))$ *to* T'. *Also, for the transition* $(0, b/b, 1)$ *in* T *and* $(0, b, 1)$ *in* A, *we add the transition* $((0,0), b/b, (1,1))$ *to* T'. *For the transition* $(1, b/a, 1)$ *in* T *and* $(1, a, 1)$ *in* A, *we add a transition* $((1,1), b/a, (1,1))$ *to* T', *and for transition* $(0, b/b, 1)$ *in* T *and* $(1, b, 1)$ *in* A *we add a transition* $((0,1), b/b, (1,1))$ *to* T'. *Finally we designate the pair* $(1,1)$ *as the final state in* T'. *The new transducer* T' *we constructed from the output Cartesian product operation of* T *and* A, *and after applying the trimming operation is illustrated in Figure 2.20.*

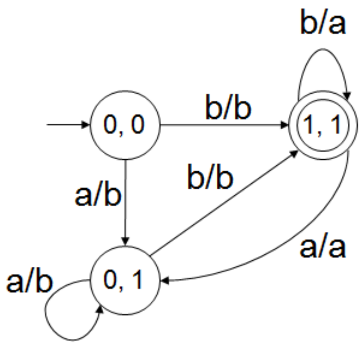


Figure 2.20: Output Cartesian product T' of T and A after trimming operation.

Chapter 3
Language properties

In this chapter we discuss some known language properties, as well as methods of describing language properties using input-altering transducer. Afterwards, we mainly focus on using input-preserving transducers to describe language properties. We present two language properties for a channel: the error-detection property and the error-correction property. Also we provide two propositions that are important to the question of whether a given language satisfies a given property.

3.1 Various language properties

We define a language property to be a set of languages. In practice, these languages posses a certain common feature of interest. If a given language belongs to the set of languages defining a particular language property, we say that this given language satisfies this particular property. Here we enumerate the code property and some classical code related properties used during our research, such as the well-investigated prefix code property, the suffix code property, the infix code property, and so on.

3.1.1 Code property

One of the most important and widely studied language properties is the code property (also called unique decodability) [5, 39]. Codes are useful for data compression.

However, a code would be useless if the code words cannot be identified in a unique way from the encoded message. A language L satisfies the code property if there is only one possible way to decompose the message $w_1 w_2 \cdots w_n$ into the words $w_1, w_2, \cdots, w_n \in L$.

Example: *Consider the language* $L = \{0, 10, 010, 101\}$. *A message such as* '0100101010' *can be decomposed over* L *in more than one ways. For example,* '0100101010' *can be interpreted in at least two ways:* '$(0)(10)(010)(101)(0)$' *and* '$(010)(0)(101)(010)$'. *Therefore, the language* $L = \{0, 10, 010, 101\}$ *does not satisfy the code property and therefore cannot be used for data compression.*

To our knowledge, although an algorithm for deciding whether a given regular language is a code is given in [21], no one has provided an open web interface for executing such an algorithm. One goal of this research is to implement the algorithm in [21] to decide whether a given regular language satisfies the code property and to provide a web interface which is hosted on the university server and available to unrestricted users. The algorithm will be presented in Chapter 4 and we implement this algorithm in our web interface in Chapter 6.

3.1.2 Code related properties

Code related properties are well investigated. Next, we provide examples of five different code related properties:

- A language satisfies the prefix code property if no word in this language is a prefix of any other word in the same language. For example, the language $L_1 = \{ab, aa, bab\}$ satisfies the prefix code property (also we say that L_1 is a prefix code). However, the language $L_2 = \{ab, aa, abaa\}$ is not a prefix code because ab is a prefix of $abaa$.
- A language satisfies the suffix code property if no word in this language is a suffix of any other word in the same language. For example, the language $L_1 = \{ab, aa, bbb\}$ satisfies the suffix code property (also we say that L_1 is a suffix code). However, the language $L_2 = \{ab, aa, bab\}$ is not a suffix code because ab is a suffix of bab.
- A language satisfies the infix code property if no word in this language is an infix of any other word in the same language. An infix of a word w is a word of the form u such that $w = xuy$ for some prefix x and suffix y of w. For example, the language $L_1 = \{ab, ba, aaa\}$ satisfies the infix code property (also we say that L_1 is an infix code). However, the language $L_2 = \{ba, bb, abbab\}$ is not an infix code because both ba ad bb are infixes of $abbab$.
- A language satisfies the outfix code property if no word in this language is an outfix of any other word in the same language. An outfix of a word w is a word of the form xy such that $w = xuy$ for some infix u of w. For example, the language $L_1 = \{ab, aa, babb\}$ satisfies the outfix code property (also we say that L_1 is an outfix code). However, the language

$L_2 = \{ab, aa, aabab\}$ is not an outfix code because both ab and aa are outfixes of $aabab$.

- A language satisfies the hypercode property if no word in the language is a scattered subword of another word in the same language. A word u is a scattered subword of a word w, if u is of the form $u_1u_2 \cdots u_n$ and w is of the form $x_1u_1x_2u_2 \cdots x_nu_nx_{n+1}$. For example, the language $L_1 = \{ab, aa, bbb\}$ satisfies the hypercode property (also we say that L_1 is a hypercode). However, the language $L_2 = \{aba, aa, aabab\}$ is not a hypercode because both aba and aa are subwords of $aabab$.

Other language properties, such as overlap-free language property, solid code property, and thin language property are also well investigated. Readers are referred to [7, 20, 26, 28, 41, 46] for more details about these code related properties.

3.2 Describing language properties using input-altering and input-preserving transducers

There are different ways to describe a language property. Usually, we use general mathematical methods [45] or formal methods to represent language properties. Three main formal methods for describing language properties are discussed in [15, 16]:

1. Implication conditions [25], which use first order logic formulas to describe language properties.
2. Regular Trajectories [13, 14], which describe language properties by regular

expressions.

3. Transducer methods, which describe language properties using various types of transducers.

In this thesis, we use transducers to describe language properties. As discussed in Chapter 2, a transducer can be used to realize a binary relation between words. Some language properties are defined by a relation between words within the language. Next, we discuss input-altering transducers that describe the code related properties we mentioned above.

3.2.1 Input-altering transducer

In [15, 16], the authors have used transducers to formally describe code related properties as in the examples in the previous section, and present methods for deciding whether a regular language satisfies a particular language property described by a transducer. However, the method in [15, 16] does have limitations. One of these limitations is that the transducer describing the language property is required to be an input-altering transducer.

A transducer T is called input-altering if for any given input word over the alphabet, this input word is not contained in the set of output words of T, that is, $\forall x \in \Sigma^*, x \notin T(x)$. Formally, the language property $\mathcal{P}_T$ described by an input-altering transducer T is defined as follows:

$$\mathcal{P}_T = \{L \subseteq \Sigma^* \mid T(L) \cap L = \emptyset\}. \tag{3.1}$$

In [15, 16], all of the transducers used to describe language properties are input-altering transducers, such as the transducers describing the prefix code, the suffix code, and the infix code property. For example, we can construct an input-altering transducer T_p such that, for any given input word, T_p can generate every possible proper prefix of this input word. Proper prefix of word means that the prefix of the word does not equal the original word itself. See Figure 3.1 for example.

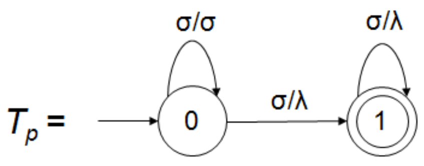


Figure 3.1: Input-altering transducer T_p describing the prefix code property.

Example: *Let us consider the input-altering transducer T_p in Figure 3.1. In T_p, symbol σ represents any symbol in a given alphabet. If a word $babb$ is sent to the transducer T_p, the possible outputs is the set of proper prefixes of $babb$:* $\{\lambda, b, ba, bab\}$.

Similar to transducer T_p describing the prefix code property, we can also construct an input-altering transducer T_s such that for any given input word, T_s can generate every possible proper suffix of this input word. Transducer T_s in Figure 3.2 describes the suffix code property.

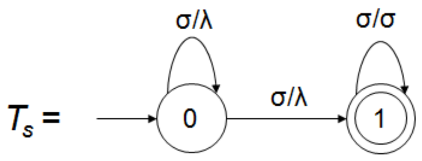


Figure 3.2: Input-altering transducer T_s describing the suffix code property.

Example: *Let us consider the input-altering transducer T_s in Figure 3.2. If a word $babb$ is sent to T_s, the possible outputs of T_s is the set of proper suffixes of $babb$:* $\{\lambda, b, bb, abb\}$.

We can also construct an input-altering transducer T_i such that for any given input word, T_i can generate every possible proper infix words of this input word. Transducer T_i in Figure 3.3 describes the infix code property.

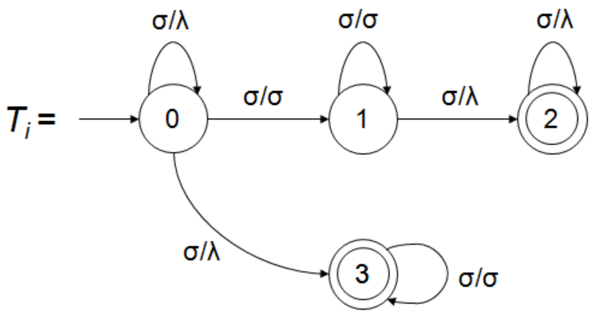


Figure 3.3: Input-altering transducer T_i describing the infix code property.

Example: *Let us consider the input-altering transducer T_i in Figure 3.3 for example. If a word $babb$ is sent to T_i, the possible outputs of T_i is the set of proper infixes of $babb$:* $\{\lambda, b, bb, abb, ba, bab, ab\}$.

We present the input-altering transducers describing the outfix code property and the hypercode property in Figure 3.4 and Figure 3.5.

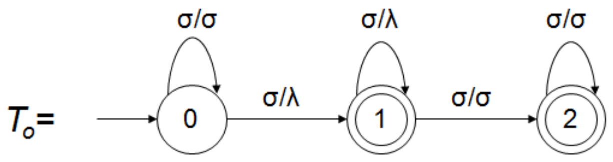


Figure 3.4: Input-altering transducer T_o describing the outfix code property

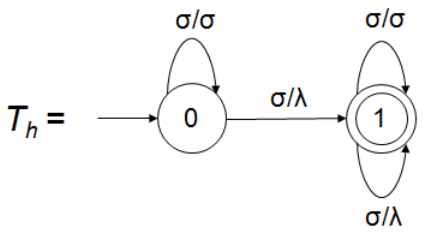


Figure 3.5: Input-altering transducer T_h describing the hypercode property

The question of deciding whether a language L satisfies a given language property $\mathcal{P}_T$ described by input-altering transducer T equals to testing the following condition:

$$T(L) \cap L = \emptyset.$$

This question is resolved by an algorithm mainly relying on the Cartesian product operation between two automata that we mentioned in Section 2.3. In order to use some of the functions in the web interface [36], we implemented algorithms to construct input-altering transducers describing these fixed code-related properties based on a given alphabet. We discuss these algorithms in Chapter 4.

3.2.2 Input-preserving transducer

Some language properties are described by input-preserving transducers, a method which is more general than that of input-altering transducers. Especially, we use input-preserving transducers to simulate combinatorial channels and decide the error-detection and error-correction properties. The question of deciding whether a language satisfies a given language property described by an input-preserving transducer is not addressed in [15]. It is addressed in [16], but without any details that would allow for time complexity estimates, and method of implementation.

Contrary to an input-altering transducer, a transducer T is called input-preserving, if for any given input word over the alphabet, the original input word is included in the set of output words of T, that is, $\forall x \in \Sigma^*, x \in T(x)$. Formally, the language property $\mathcal{P}_T{}'$ described by an input-preserving transducer T is defined as follows:

$$\mathcal{P}_T{}' = \{L \subseteq \Sigma^* \mid \forall x \in L, T(x) \cap (L - x) = \emptyset\}. \tag{3.2}$$

Example: *Let us consider the input-preserving transducer $T_p{}'$ describing the prefix code property in Figure 3.6. If a word $babb$ is sent to $T_p{}'$, the possible outputs of $T_p{}'$ is the set of prefixes of $babb$ including itself: $\{\lambda, b, ba, bab, babb\}$. T_p and $T_p{}'$ both describe the prefix code property. The difference between T_p and $T_p{}'$ is that in T_p, state 0 is not a final state while, in $T_p{}'$ state 0 is a final state, which makes $T_p{}'$ input-preserving. Similarly, the transducers describing the suffix code and infix code properties are illustrated in Figure 3.7.*

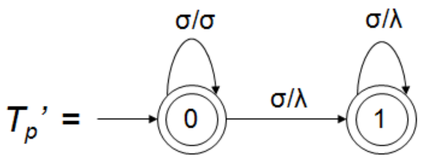
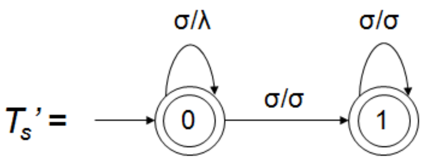
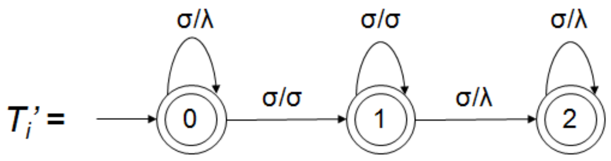


Figure 3.6: Input-preserving transducer $T_p{}'$ describing the prefix code property.



(a) Input-preserving transducer $T_s{}'$ describing the suffix code property.



(b) Input-preserving transducer $T_i{}'$ describing the infix code property.

Figure 3.7

One goal of this thesis is to present the details of an algorithm to answer the satisfaction question: whether a given regular language satisfies the language property described by given input-preserving transducer. We discuss this algorithm in Chapter 4 and implementation of this algorithm in our web interface in Chapter 6.

3.3 Error-detection and error-correction

3.3.1 Error-detection

A major objective in data communication systems is to process reliably a message that was transmitted via some communication channel C capable of introducing errors, such as substitution, insertion, and deletion errors. In [31, 33], if C is a channel, a language L satisfies the error-detection property for C (also we say that language L is error-detecting for C), if

$$\forall\, w', x : x \in L \ \wedge\ w' \in T(x)\ \wedge w' \in L\ \rightarrow x = w' \qquad (3.3)$$

which means, if a pair of words $(x, w') \in C$, and $x, w' \in L$, then $x = w'$. In other words, if a word w' is received via the channel C and $w' \in L$, then w' must be correct and equal to $x \in L$ that was sent to C. However, if the received word $w' \notin L$, then there must be a transmission error. An equivalent formulation is that the channel C cannot translate a word in the language L into another word that is also in L:

$$\text{L is error-detecting for C} \ \leftrightarrow\ \forall\, \text{x} \ \in \text{L}, \text{T(x)} \cap \left(\text{L-x}\right) = \emptyset\,. \qquad (3.4)$$

Therefore, from Equation 3.2 and 3.4, we conclude that the question of deciding whether a given language L satisfies the property $\mathcal{P}_T{}'$ described by an input-preserving transducer T is equivalent to decide whether L is error-detecting for the channel C realized by T.

***Example**: Let us consider the channel $SID(2,\infty)$. $SID(2,\infty)$ is the relation that consists of all pairs of words (u, v) such that v results by performing at most 2 substitutions/insertions/deletions/ in u. For example, the pair $(00000{,}0100) \in SID(2,\infty)$, while $(00000{,}111) \notin SID(2,\infty)$. In Figure 3.8, we show an input-preserving transducer realizing $SID(2,\infty)$. According to Equation 3.4, the transducer in Figure 3.8 describes the error-detection property for $SID(2,\infty)$*

channel.

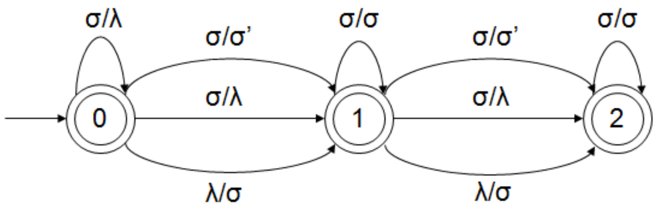


Figure 3.8: Input-preserving transducer realizing the $SID(2,\infty)$ channel.

3.3.2 Error-correction

In data communication systems, an error is detected exactly when a word w is received via a channel C and w is not in the language L (transmission error). In this situation, it is possible to find out which input word was transmitted, if L is error-correcting for C. Formally, a language L satisfies the error-correction property for C (we also say that L is error-correcting for C), if

$$x \in L, (x, w) \in C, and\ v \in L, (v, w) \in C \to x = v.$$

This means that, even if w has been received and contains errors, we can find out the unique input word $x = v \in L$, correcting thus the errors in w. In other words, if L is error-correcting for C, then for any different two words in the language L, the channel C cannot change these two words into the same output word.

Note that, if a language L is error-correcting for the channel C, then it is also error-detecting for channel C. Readers can refer to [31, 33] for details about the error-detection and error-correction property.

Although there are algorithms proposed for deciding whether a given language L is error-detecting or error-correcting for a channel C realized by transducer T, no one has provided an open implementation of these algorithms, with the exception of [10] in which the error-detection property is implemented for channels realized by sequential transducers. In this paper, we also focus on answering the questions about whether a given language is error-detecting for a channel realized by an input-preserving transducer (equivalent to the question of whether a given language satisfies a given language property described by an input-preserving transducer) and whether a given language is error-correcting for the channel realized by a given transducer.

3.3.3 Two propositions

We present two propositions for answering these error-detection and error-correction questions next. Algorithms and examples for answering these questions will be discussed in Chapter 4.

Proposition 3.1: *Let C be a channel. A language L is error-detecting for C if and only if the relation $C \downarrow L \uparrow L$ is functional. Equivalently, let $\mathcal{P}_T{}'$ be a language property described by an input-preserving transducer T, A language L satisfies $\mathcal{P}_T{}'$ if and only if the transducer $T \downarrow A \uparrow A$ is functional.*

Proposition 2: *Let C be a channel. A language L is error-correcting for C if and* *only if the relation* $C^{-1} \uparrow L$ *is functional.*

In our research, we use these two propositions and the tools discussed in Chapter 2 to decide whether a language L accepted by an automaton A is error-detecting or error-correcting for the channel C realized by a given input-preserving transducer T. We will describe how to use these two propositions to address this error-detecting and error-correcting question in Chapter 4.

Chapter 4

Algorithmic tools and decision algorithms

In this chapter, we focus on presenting the algorithmic tools and decision algorithms to answer the following questions:

1. Given a transducer T, decide whether T is functional.
2. Given a transducer T realizing a channel C and an automaton A accepting a language L, construct transducers realizing the relations $C \downarrow L \uparrow L$ and $C^{-1} \uparrow L$.
3. Given a language property $\mathcal{P}_T{}'$ described by an input-preserving transducer T, and a language L accepted by an automaton A, decide whether L satisfies the property $\mathcal{P}_T{}'$. If not, we generate a counterexample to prove why L does not satisfy the property $\mathcal{P}_T{}'$.
4. Given a channel C realized by an input-preserving transducer T and a language L accepted by an automaton A, decide whether L is error-detecting for C. If not, we generate a counterexample to prove why L is not error-detecting for C.
5. Given a channel C realized by an input-preserving transducer T and a language L accepted by an automaton A, decide whether L is error-correcting for C. If not, we generate a counterexample to prove why L is not error-correcting for C.
6. Given a language L, decide whether L is a code. If not, we generate a counterexample to prove why L is not a code.
7. For certain fixed properties $\mathcal{P}$, given an alphabet Σ, construct an input-altering transducer T that describes $\mathcal{P}$, that is, $\mathcal{P} = \mathcal{P}_T$.

8. Given a transducer T in normal formal, translate T into an equivalent real-time transducer T'.

We follow the two propositions in Chapter 3 and apply the tools discussed in Chapter 2 to answer these questions. Also, we use plenty of examples to illustrate to readers the details of these methods.

4.1 Deciding functionality of a transducer

We first consider the question of whether a given transducer T is functional, as this question is fundamental to answering other questions: deciding whether a language L accepted by an automaton A is error-detecting or error-correcting for a channel C realized by an input-preserving transducer T, and whether a given language L is a code.

A transducer T realizes the relation R consisting of all pairs (u, v) accepted by T. The relation R is called functional, if $(u, v_1) \in R$ and $(u, v_2) \in R$ imply that $v_1 = v_2$, therefore deciding the functionality of T means to decide whether the relation R realized by T is functional, and vice versa.

Previous researchers have already proposed algorithms to decide the functionality of a transducer [3, 21, 23, 40], after it was realized in [44] that transducer functionality is decidable. Head & Weber [21] firstly brought forward an algorithm to decide the functionality of a restricted sequential transducer in quadratic time. Mohri [40] then proposed a more generalized algorithm to decide the functionality of a transducer in standard form. Also, Béal et al [3] introduced a very similar algorithm to Mohri's algorithm, in order to decide the functionality of a real-time transducer. We find that Mohri's algorithm and Béal et al's algorithms can be used to decide the functionality of a transducer either in standard form or in real-time. We present their algorithms that apply to different types of transducers: restricted sequential transducer, standard form transducer, and real-time transducer. In addition, when given a real-time transducer, we introduce a pre-functionality test to make a quick decision related to the functionality of a real-time transducer.

4.1.1 Functionality of a restricted sequential transducer

In [21], the following algorithm is brought forward to decide the functionality of a restricted sequential transducer in quadratic time (see Figure 4.1 for example):

Algorithm 4.1:

Let $T = (Q, \Sigma, \Gamma, q_0, F, E)$ be a restricted sequential transducer. We build an NFA $G' = (Q \times Q, (0,1), E', (q_0, q_0), F \times F)$ as follows:

1. Set the pair (q_0, q_0) as the start state of G', where q_0 is the start state of T.
2. We build the transition set E' as follows. Each pair of pairs (p, p') and (q, q') will be connected by at most one transition in G'. The pair will be connected by a transition $\big((p, p'), bit, (q, q')\big)$ in E' if and only if there is a symbol x in Σ for which there are transitions $(p, x/y, q)$ and $(p', x/$

$y', q')$ in E. If, for every such x in Σ, $y = y'$ then $bit = 0$, otherwise $bit = 1$.

3. Set all the pairs in $F \times F$ as the final states of G', where F is the set of final states in T.
4. Apply the trimming operation on G' to obtain the trimmed NFA G equivalent to G'.
5. T is functional if and only if symbol 1 does not appear as the label of any transition of G.

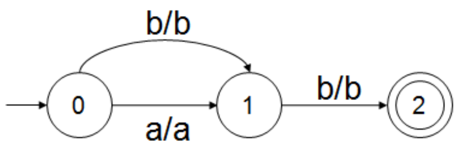
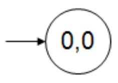


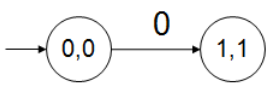
Figure 4.1: A restricted sequential transducer

Example: *Let us consider the restricted sequential transducer T in Figure 4.1. First of all we construct the start state $(0,0)$ of G' in Figure 4.2a. As, there is a transition $(0, a/a, 1)$ in T which matches the input label with itself, so we add a transition $\big((0,0), bit, (1,1)\big)$, where bit is determined later, and record the information $\{a, (a,a)\}$ which tells us that on the same input a, there are two transitions with outputs (a,a). Also for the transition $(0, b/b, 1)$ in T, we also add the transition $\big((0,0), bit, (1,1)\big)$ and record the information $\{b, (b,b)\}$. Then we conclude that $bit = 0$, as for a in the set of information, the output $a = a$ and for b in the set of information, $b = b$. So we change the transition $\big((0,0), bit, (1,1)\big)$ to*

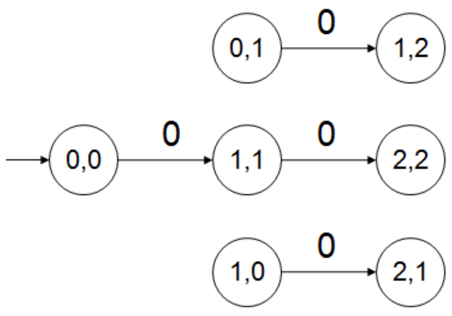
$\big((0,0),0,(1{,}1)\big)$ *in Figure 4.2b. Similarly, we add the transition* $\big((1,1),0,(2{,}2)\big)$ *in Figure 4.2c. For the transitions* $(0,b/b,1)$ *and* $(1,b/b,2)$, *we add two transitions* $\big((0,1),bit,(1{,}2)\big)$ *and* $\big((1,0),bit,(2{,}1)\big)$, *and record the information* $\{b,(b,b)\}$. *We can conclude that* $bit = 0$ *and change these transitions to* $\big((0,1),0,(1{,}2)\big)$ *and* $\big((1,0),0,(2{,}1)\big)$ *respectively as shown in Figure 4.2c. And finally, we designate state* $(2{,}2)$ *as the final state of* G' *in Figure 4.2d.*



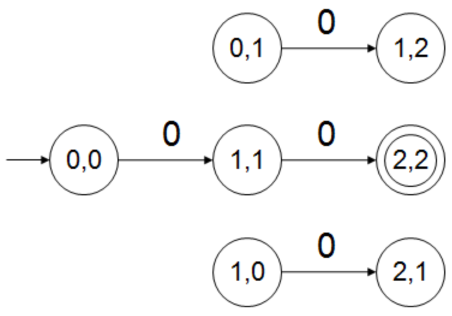
(a) Start state of G'



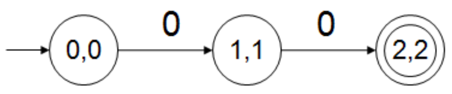
(b) Add transition $\big((0,0),0,(1{,}1)\big)$



(c) Add transition $\big((1,1),0,(2{,}2)\big)$, $\big((0,1),0,(1{,}2)\big)$, and $\big((1,0),0,(2{,}1)\big)$



(d) Deciding the final state of G'



(e) The trimmed NFA G equivalent to G'

Figure 4.2: Processing of constructing G' using **Algorithm 4.1**.

Then we perform the trimming operation on the NFA G' in Figure 4.2d. As the transitions $\big((0,1),0,(1,2)\big)$ and $\big((1,0),0,(2,1)\big)$ are not accessible from the start state $(0,0)$, we delete these two transitions and the four related states. The trimmed NFA G equivalent to G' is shown in Figure 4.2e.

For the NFA G in Figure 4.2e, all the labels in the transitions are 0. Therefore, the restricted sequential transducer T in Figure 4.1 is functional. Note that **Algorithm 4.1** can only decide the functionality of a restricted sequential transducer.

4.1.2 Functionality of a standard form transducer or a real-time transducer

As Mohri's [40] and Béal et al's [3] algorithms are very similar, we only give one description as follows:

Algorithm 4.2:

Given a transducer T in standard form (or real-time) with start state s, construct the product machine U as follows:

1. If $(p, a/x, q)$ and $(p', a/x', q')$ are transitions in T then add to U the transition $((p,p'),(x,x'),(q,q'))$.
2. The start state of U is (s,s).
3. The final states of U are all pairs (f,f') where both f and f' are final states in T.
4. Only keep states that can be reached from (s,s) and can reach a final state (f,f').

After constructing U, assign to each state of U a value, which is either $ZERO$, or a pair of words in $\{(\lambda, \lambda), (\lambda, u), (u, \lambda)\}$, where λ is the *empty* word and u is a nonempty word, as follows:

1. The start state gets the value (λ, λ).
2. If a state (p, p') has a value (y, y') and there is a transition $((p, p'), (x, x'), (q, q'))$ then (q, q') gets a value as follows:

   (a) If yx is a prefix of $y'x'$, so that $y'x' = yxu$, then $value = (\lambda, u)$.

   (b) If $y'x'$ is a prefix of yx, so that $yx = y'x'u$, then $value = (u, \lambda)$.

   (c) If yx equals $y'x'$, then $value = (\lambda, \lambda)$.

   (d) Else, $value = ZERO$.
3. Repeat until a state gets two different values, or the state value is $ZERO$, or every state gets one value.

If a state has two different values, or a state value is $ZERO$, or the value of a final state is not (λ, λ), then output <u>NO</u> (in other words, T is not functional). If every state has one value <u>AND</u> every final state of U has value (λ, λ) then output <u>YES</u> (in other words, T is functional).

Example: *Let us consider the standard form transducer T in Figure 4.1 again for constructing the product machine U. The start state of U is $(0,0)$ and the final state of U is $(2,2)$. As $(0, a/a, 1)$ is a transition in T which matches the input label with itself, we add the transition $\big((0,0), (a,a), (1,1)\big)$ to U. Also, as $(1, b/b, 2)$ is a transition in T, we add the transition $\big((1,1), (b,b), (2,2)\big)$ to U. For the transitions*

$(0, b/b, 1)$ *and* $(1, b/b, 2)$, *we add two transitions* $\big((0,1),(b,b),(1,2)\big)$ *and* $\big((1,0),(b,b),(2,1)\big)$ *to* U. *However, as the transitions* $\big((0,1),(b,b),(1,2)\big)$ *and* $\big((1,0),(b,b),(2,1)\big)$ *are not accessible from the start state* $(0,0)$, *we delete these two transitions and the four related states. The product machine* U *we constructed using* ***Algorithm 4.2*** *is presented in Figure 4.3.*

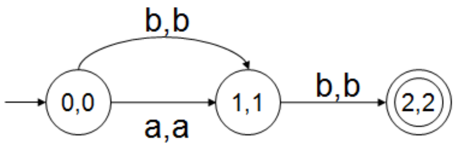
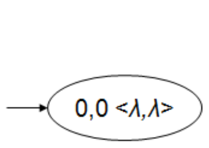
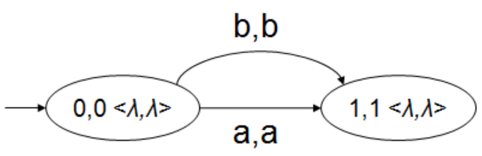

$$\begin{tikzcd}
\arrow[r] & (0,0) \arrow[r, "{a,a}"'] \arrow[r, bend left, "{b,b}"] & (1,1) \arrow[r, "{b,b}"] & \boxed{(2,2)}
\end{tikzcd}$$

Figure 4.3: The product machine U we constructed using **Algorithm 4.2**.

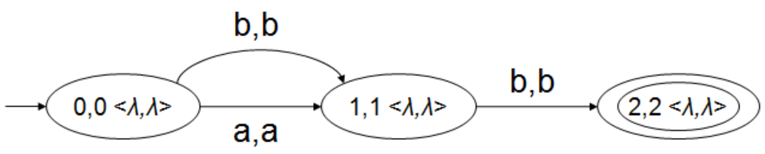
Next we add a value to every state in U *following the second part of* ***Algorithm 4.2****. The value of the start state* $(0,0)$ *is* (λ, λ). *Then, there is a transition* $\big((0,0),(a,a),(1,1)\big)$ *in* U, *so both* $y'x'$ *and* yx *are* $a\lambda$, *therefore the value of state* $(1,1)$ *is* (λ, λ). *Also, there is a transition* $\big((0,0),(b,b),(1,1)\big)$, *so both* $y'x'$ *and* yx *are* $b\lambda$, *so we add the value* (λ, λ) *to state* $(1,1)$. *As state* $(1,1)$ *has already got a value* (λ, λ), *then we don't have to add a new value to* $(1,1)$. *Similarly, the value for the final state* $(2,2)$ *is* (λ, λ). *The process of adding a value to each state in* U *is presented in Figure 4.4.*



(a) Give the start state a value (λ, λ)



(b) Give state $(1,1)$ a value (λ, λ)



(c) Give the final state $(2,2)$ a value (λ, λ)

Figure 4.4

As every state has one value and the value of the final state is (λ, λ), we conclude that the standard form transducer T in Figure 4.1 is functional.

As Figure 4.1 is a simple transducer, we now illustrate another transducer T in standard form in Figure 4.5 to provide a case where T is not functional.

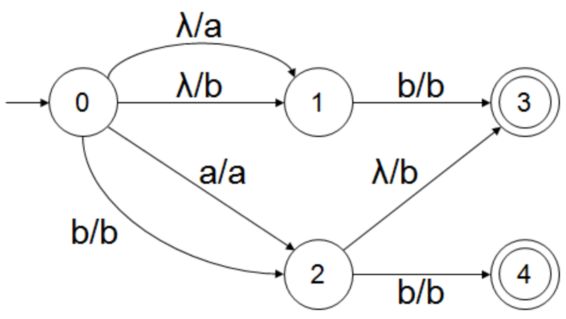


Figure 4.5: A standard form transducer T.

Example: *Let us consider the standard form transducer* T *in Figure 4.5 for constructing the product machine* U. *For the transitions* $(0, \lambda/a, 1)$ *and* $(0, \lambda/b, 1)$ *in* T, *we add four transitions* $\big((0,0),(a,a),(1,1)\big)$, $\big((0,0),(a,b),(1,1)\big)$, $\big((0,0),(b,a),(1,1)\big)$, *and* $\big((0,0),(b,b),(1,1)\big)$ *to* U. *Other transitions are constructed similarly to the previous example. The trimmed product machine* U *we constructed using* ***Algorithm 4.2*** *is presented in Figure 4.6. Note that for some transitions in* T, *such as* $(0, b/b, 2)$ *and* $(1, b/b, 3)$, *we can add the transitions* $\big((0,1),(b,b),(2,3)\big)$ *and* $\big((1,0),(b,b),(3,2)\big)$ *to* U. *However, as these transitions* are *not accessible from the start state* $(0,0)$, *they are not shown in Figure 4.6.*

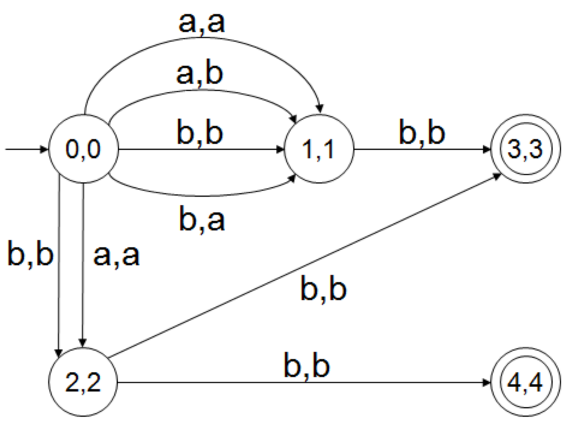


Figure 4.6: The product machine U we constructed using **Algorithm 4.2**.

Next we add a value to every state in U. *By the description of* ***Algorithm 4.2***, *the value of the start state* $(0,0)$ *is* (λ, λ). *As there is a transition* $\big((0,0),(a,b),(1,1)\big)$, *so* $yx = a\lambda$ *and* $y'x' = b\lambda$. *Note that* $y'x'$ *is not a prefix of* yx, *and* yx *is not a prefix of* $y'x'$, *and* $y'x' \neq yx$. *Therefore the value of state* $(1,1)$ *is* $ZERO$. *We then conclude that the transducer* T *in Figure 4.6 is not functional.*

4.1.3 Pre-functionality test of a real-time transducer

Recall that in Section 4.1.2, we can use **Algorithm 4.2** to decide the functionality of a real-time transducer. When given a real-time transducer, the output $I(q)$ of a start state and the output $F(q)$ of a final state are regular expressions. In addition, the output label of every transition $(p, x/e, q)$ in a real-time transducer is also a regular expression $e \in REX(\Gamma)$, that is, e is a regular expression over Γ. If one of the regular expressions, $I(q)$, $F(q)$, or $e \in REX(\Gamma)$, contains at least two words, we can get at least two different outputs through the computation of this real-time transducer based on the same given input, which means that the real-time transducer is not functional. The following <u>pre-functionality</u> test decides if a regular expression represents a language that contains at least two words:

1. First we use the following rules to simplify a regular expression RE:
   a) $\lambda \cdot RE = RE \cdot \lambda = RE$.
   b) $\emptyset \cdot RE = RE \cdot \emptyset = \emptyset$
   c) $\emptyset + RE = RE + \emptyset = RE$.
   d) $\lambda + RE = RE + \lambda = RE$, where $\lambda \in L(RE)$.
   e) If a regular expression unions the same regular expression to itself, the simplification result is the regular expression itself. For example, $abc + abc = abc$.
   f) $(RE^*)^* = RE^*$.
   g) $\emptyset^* = \lambda^* = \lambda$.
2. After simplification, if the symbol $+$ or $*$ is included in the regular

expression RE, the language L represented by RE contains two or more words, because the union symbol $+$ indicates that the left part of RE is different from the right part of RE and the Kleene start symbol $*$ indicates that RE contains infinitely many words. Either of the symbols $+$ or $*$ implies that $|L| \geq 2$, where L is the language represented by RE.

3. Examine all the regular expressions in the real-time transducer. If one regular expression represents a language containing at least two words, then the real-time transducer is <u>not functional</u>. If every regular expression contains only one word, then continue to use **Algorithm 4.2** to test the functionality of the real-time transducer.

The above simplification process in the pre-functionality test only takes linear time, where we allow that the given regular expression is represented as a tree. **Algorithm 4.2** takes quadratic time to decide whether a given real-time transducer is functional, therefore the pre-functionality test is faster.

4.2 Deciding whether a language satisfies the property described by an input-preserving transducer (deciding the error-detection property)

In this section, we focus on addressing the following two equivalent questions:

1. Given a language property $\mathcal{P}_T{}'$ described by an input-preserving transducer T and a language L accepted by an automaton A, decide whether L satisfies

the property $\mathcal{P}_T{}'$.

2. Given a channel C realized by an input-preserving transducer T and a language L accepted by an automaton A, decide whether L is error-detecting for C.

Recall that, as discussed in Chapter 3, the above two questions are equivalent. We pick the first problem to demonstrate the algorithm. We have two steps for answering the first question following **Proposition 3.1**:

1. Construct a transducer T' that equals to transducer $T \downarrow A \uparrow A$.
2. Test whether T' is functional using **Algorithm 4.2**.

Next, we provide an example involving the input-preserving transducer T describing the suffix code property over the alphabet $\{a, b\}$ (Figure 4.7) and an automaton A accepting the language $L = \{ab, bb\}$ (Figure 4.8). For any given input word, the possible outputs of T is the set of suffixes of the input word, including the input word itself. In order to decide whether L satisfies the suffix code property, we construct the transducer $T' = T \downarrow A \uparrow A$, and then decide whether T' is functional.

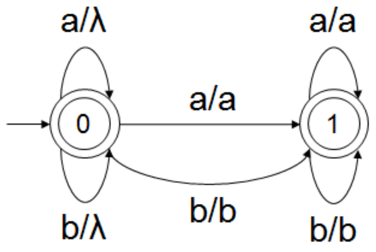


Figure 4.7: The input-preserving transducer T describing the suffix code property.

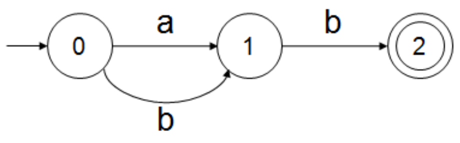


Figure 4.8: The automaton A accepting the language $L = \{ab, bb\}$.

4.2.1 Constructing the transducer $\boldsymbol{T \downarrow A \uparrow A}$

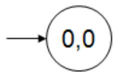
Recall that in **Proposition 3.1**, a language L satisfies a given language property described by an input-preserving transducer T if and only if the transducer $T \downarrow A \uparrow A$ is functional. We now show the steps of constructing $T \downarrow A \uparrow A$ based on the tools in Chapter 2. First we use the input Cartesian product operation of a transducer and an automaton (see Section 2.6) to construct the transducer $M = T \downarrow A$.

Then, the next step would be to construct another transducer $T' = M \uparrow A = T \downarrow A \uparrow A$. However, instead of using $M \uparrow A$ as a new operation, we use the following fact:

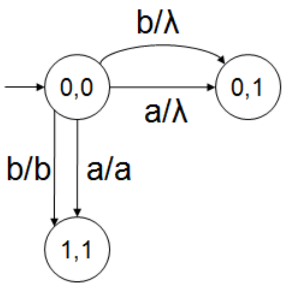
$$R \uparrow L = (\mathrm{R}^{-1} \downarrow L)^{-1}.$$

More specifically, we define $T' = (M^{-1} \downarrow A)^{-1} = M \uparrow A = T \downarrow A \uparrow A$. Therefore, after we constructed $M = T \downarrow A$, we build the inverse transducer M^{-1} of M and construct the input Cartesian product $M' = M^{-1} \downarrow A$ of the transducer M^{-1} and the automaton A. Finally, we further construct $T' = (M')^{-1} = (M^{-1} \downarrow A)^{-1}$, which is equal to the transducer $T \downarrow A \uparrow A$.

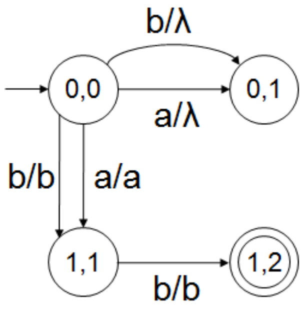
Example: *Let us consider the input-preserving transducer T describing the suffix code property (Figure 4.7) and the automaton A (Figure 4.8) representing the* *language $L = \{ab, bb\}$. Given the word ab to T, we can get the following set of possible outputs: $\{ab, b, \lambda\}$. In order to construct a new transducer $M = T \downarrow A$, we have to examine firstly whether T has λ-input transitions, or A has λ-transitions, and determine whether to add self λ-transitions to both T or A. Next, we construct the transducer $M = T \downarrow A$ (see Figure 4.9 for the process).*



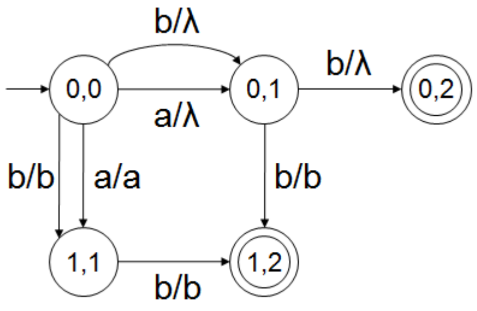
(a) Start state of the product transducer



(b) Transitions going out of the start state



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

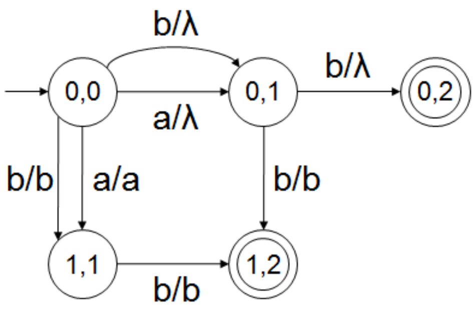


(d) Building new transitions going out of the state $(0,1)$ and it's the final product

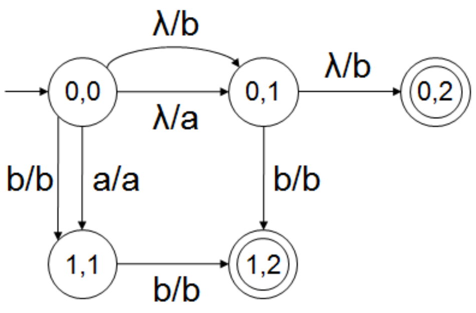
Figure 4.9: The process of constructing the input Cartesian product of T and A.

Now we build the inverse transducer M^{-1} of M and construct the input Cartesian product $M^{-1} \downarrow A$ of the transducer M^{-1} and the automaton A.

Example: *Let us consider the product transducer M in Figure 4.9d. To build the* *inverse transducer* M^{-1} of M*, we need to switch the input label with the output label for every transition in* M*, without any change to the states in* M*. The result of inverse operation of* M *is presented in Figure 4.10b.*



(a) The final product M in Figure 4.9d



(b) The inverse transducer M^{-1} of M

Figure 4.10

Afterwards, we construct the transducer $M' = M^{-1} \downarrow A$ *by applying the input Cartesian product operation again. Also, before applying the operation, we have to examine firstly whether* M^{-1} *has* λ*-input transitions or* A *has* λ*-transitions. In this case, as* M^{-1} *has* λ*-input transitions, we add self* λ*-transitions to the automaton* A *as illustrated in Figure 4.11.*

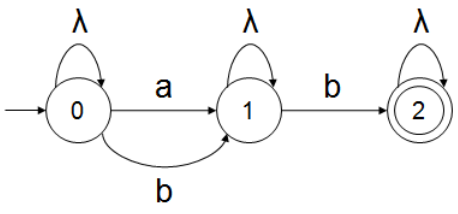
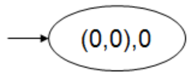
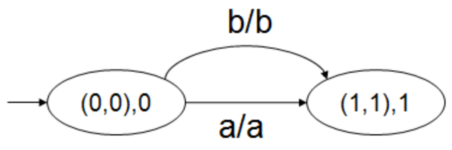


Figure 4.11: Automaton A expanded with λ-transitions, representing language $L = \{ba, bb\}$.

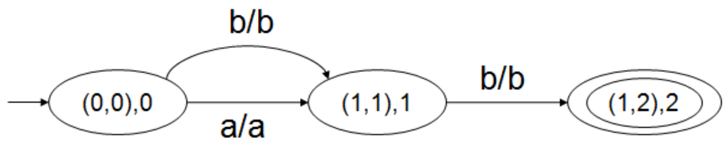
We apply the input Cartesian product operation between the transducer M^{-1} (Figure 4.10b) and the automaton A with self λ-transitions (Figure 4.11). The process of constructing $M' = M^{-1} \downarrow A$ is presented in Figure 4.12.



(a) Start state of the product transducer M'



(b) Transitions going out of the start state



(c) Building new transitions going out of the state $((1,1),1)$ and it is the final product

Figure 4.12: The process of constructing the input Cartesian product of M^{-1} and A

Finally, we further construct $T' = (M')^{-1} = (M^{-1} \downarrow A)^{-1}$, which is equal to $T \downarrow A \uparrow A$. We need to construct the inverse transducer $T' = (M')^{-1}$ of M' in Figure 4.12c. Note that in this case, the inverse transducer T' is the same as the transducer M' itself, as the input label is equal to the output label in every transition in transducer M'.

The transducer $T' = T \downarrow A \uparrow A$ in Figure 4.12c after the final operation is the one that we are going to use to test whether it is functional, in order to decide whether the language L satisfies the suffix code property.

4.2.2 Deciding the functionality of T'

Finally, we apply **Algorithm 4.2** to decide whether the transducer T' in Figure 4.12c is functional. Note that the transducer T' in Figure 4.12c is equivalent to the transducer in Figure 4.1, and we have already proved that the transducer in Figure 4.1 is functional. Then we conclude that the transducer $T' = T \downarrow A \uparrow A$ in Figure 4.12c is functional. According to **Proposition 3.1**, we conclude that the language $L = \{ab, bb\}$ accepted by the automaton A in Figure 4.8 satisfies the suffix code property described by the input-preserving transducer T in Figure 4.7.

The above example is reasonable for deciding whether a given language satisfies a language property. We won't show another example to demonstrate how to decide whether a given language is error-detecting for a channel, because the process for answering this question is the same as the process we discussed above.

4.3 Deciding the error-correction property

Now we consider the question of deciding whether a language L is error-correcting for a channel C, where L is accepted by an automaton A and C is realized by an input-preserving transducer T. We have two steps for answering this question following **Proposition 3.2**:

1. Construct a transducer T realizing the relation $C^{-1} \uparrow L$.
2. Test whether T is functional using **Algorithm 4.2**.

Similar to deciding whether a language L accepted by an automaton A is error-detecting for the channel C realized by a given input-preserving transducer T, to decide the error-correction property, we can construct the transducer M via the input Cartesian product operation such that:

$$M = T \downarrow A.$$

Again, as $R \uparrow L = (R^{-1} \downarrow L)^{-1}$, and $(R \downarrow L)^{-1} = (R^{-1} \uparrow L)$, we build the inverse transducer M^{-1} of M, such that:

$$M^{-1} = (T \downarrow A)^{-1} = T^{-1} \uparrow A.$$

Note that M^{-1} realizes the relation $C^{-1} \uparrow L$. Then, the problem of deciding whether L is error-correcting for C reduces to the question of deciding whether M^{-1} is functional.

***Example**: We provide an example involving the input-preserving transducer T realizing the $Del(1,\infty)$ channel C (Figure 4.13) and an automaton A accepting the language $L = \{ab, bb\}$ (Figure 4.8). For any given input word, the possible outputs of T is the set of words that are obtained by applying at most one deletion in the input word. In order to decide whether L is error-correcting for C, we construct the transducer M^{-1} realizing the relation $C^{-1} \uparrow L$, which is equivalent to the transducer in Figure 4.5.*

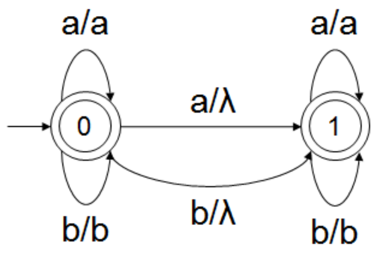


Figure 4.13: The input-preserving transducer T realizing the channel $Del(1,\infty)$.

Recall that in Section 4.1, we have already concluded that the transducer T in Figure 4.5 is not functional. According to **Proposition 3.2**, we conclude that the language $L=\{ab,bb\}$ accepted by the automaton A in Figure 4.8 is not error-correcting for the channel $Del(1,\infty)$ realized by the input-preserving transducer T in Figure 4.13.

4.4 Deciding the code property

Head & Weber [21] provided an algorithm to decide whether a language described by an NFA is a code. In this section we present this algorithm and give an example to illustrate how this algorithm works.

Algorithm 4.3:

Given a language L recognized by an NFA $G=\{Q,A,E,q_0,F\}$, decide whether L is a code as follows:

1. If the start state q_0 is in the set of final states F, then λ is accepted by G, which means that λ is in L. Therefore, L is not a code.
2. If the start state q_0 is not in the set of final states F, let T be the transducer $T=\{Q,A,\{0,1\},E',q_0,\{q_0\}\}$, in which the output label in every transition

can only be a symbol 0 or symbol 1, and state q_0 is the start state and the only final state.

3. The transition set E' is constructed as follows: for each edge (p, a, q) in E, where $p, q \in Q$ and $a \in A$, add $(p, a/0, q)$ in E' and, if q is in the set of final states F, then add $(p, a/1, q_0)$ in E' also.
4. Use **Algorithm 4.1** to decide whether T is functional. If T is functional, the language L is a code. If T is not functional, then L is not a code.

We provide an example to demonstrate the details of **Algorithm 4.3** in this section. By the definition of code, the language $L = \{0,01,110\}$ is a code. We will use **Algorithm 4.3** to decide that $L = \{0,01,110\}$ is a code.

Example: *Let us consider the language* $L = \{0,01,110\}$ *described by the NFA* G *in Figure 4.14. We now construct the transducer* T *according to the step 2 and step 3 in* ***Algorithm 4.3*** *as follows. The start state* 0 *of* G *is the only start state and final state of* T. *As shown in Figure 4.14, the NFA* G *has transitions:* $(0,0,2)$, $(0,1,3)$, *and* $(3,1,4)$, *in which the end state of each transition (state* 2, *state* 3, *and state* 4*) is not the final state of* G. *Therefore, we add three transitions to* T*:* $(0,0/0,2)$, $(0,1/0,3)$, *and* $(3,1/0,4)$, *as shown in Figure 4.15a. Also, the NFA G has three transitions* $(0,0,1)$, $(2,1,1)$, *and* $(4,0,1)$, *in which the end state of each transition is a final state in* G. *Therefore, we add three transitions to* T*:* $(0,0/1,0)$, $(2,1/1,0)$, *and* $(4,0/1,0)$, *as shown in Figure 4.15b. The transducer* T *in Figure 4.15b is the final result of this* *construction, and we test the functionality of T to decide whether L is a code.*

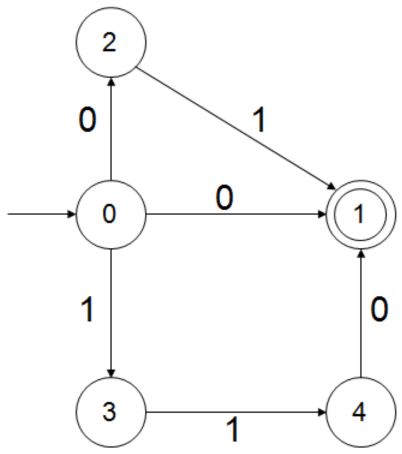
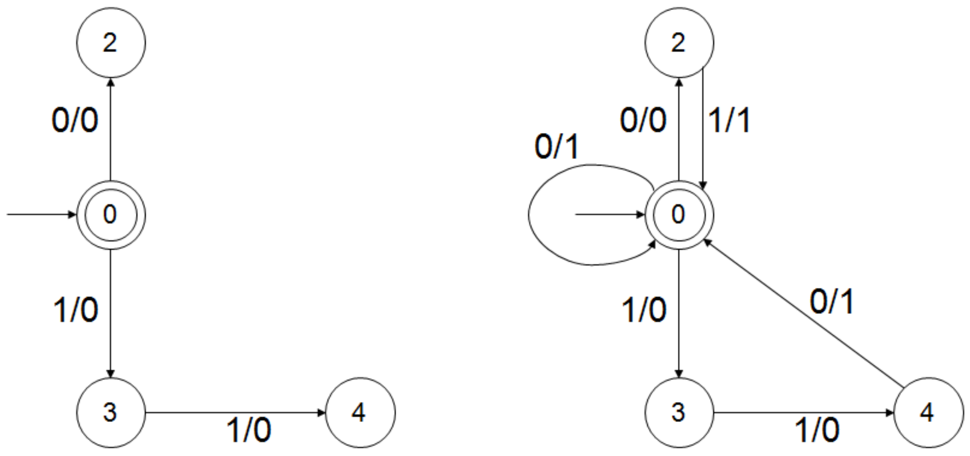


Figure 4.14: NFA G accepting the language $L = \{0,01,110\}$.



(a) (b)

Figure 4.15

The transducer T in Figure 4.15b tells us that for any pair $(u, v) \in T$, where the input word $u = u_1 u_2 \cdots u_n \in L^*$ and L is accepted by G, the output word v consists of 0s and 1s and $|u| = |v|$. The symbol 1s in v indicates that if $v_i = 1$, then after process u_i, the computation of T is currently in the position of the final state, and $u_{i-n} \cdots u_{i-2}u_{i-1}u_i \in L$, where $n \geq 0$, such that $v_{i-n}, \cdots, v_{i-2}, v_{i-1} = 0$ and $v_{i-n-1} = 1$. In other words, the output word v indicates how the input word u is decomposed. For example, giving $u = 11001 \in L^*$ to T, the output word $v = 00101$, and v indicates that the input word $u = 11001$ can be decomposed by $110 \in L$ and $01 \in L$.

As the transducer T in Figure 4.15b is a restricted sequential transducer, we use **Algorithm 4.1** to decide whether T is functional. Intermediate steps of applying **Algorithm 4.1** are not shown here. Readers are referred to Section 4.1 for details of this algorithm. We here give the final NFA G' constructed following **Algorithm 4.1** in Figure 4.16. In the NFA G', all the symbols in the transitions are 0. Hence, we conclude that the transducer T in Figure 4.15b is functional, which means that for any input word giving to T, we can only get one output word and this output word indicates the only way to decompose the input word. Hence, we conclude that the language $L = \{0,01,110\}$ is a code.

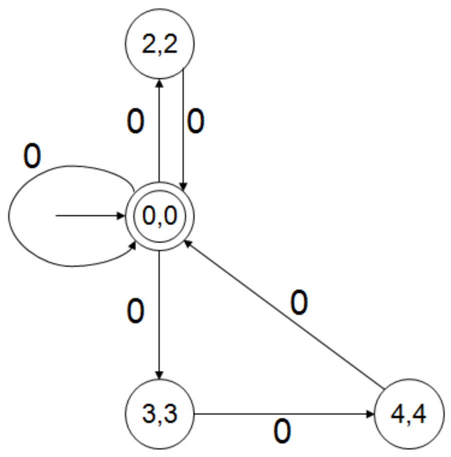


Figure 4.16: The NFA G' constructed by **Algorithm 4.2**.

4.5 Counterexample

In [15], in the context of deciding whether a given language L satisfies a given property $\mathcal{P}_T$ described by an input-altering transducer T, the author provides a method to give a counterexample in case where the answer is negative. In particular, the counterexample is a pair of different words $u, v \in L$ violating the property $\mathcal{P}_T$, that is, $v \in T(u)$. Here, we also design and implement the feature of providing counterexamples for the following three situations:

1. If a given language L is not error-detecting for a channel C realized by an input-preserving transducer T, which is equivalent to the situation that L does not satisfy the language property $\mathcal{P}_T{}'$ described by T.
2. If a given language L is not error-correcting for a channel C realized by an input-preserving transducer T.
3. If a given language L is not a code.

As discussed before, these three questions are eventually reduced to deciding functionality of the final product transducers, which are constructed following **Sections 4.2**, **4.3**, and **4.4**. For example, recall that in Section 4.2, the functionality of the transducer T' realizing the relation $C \downarrow L \uparrow L$ indicates whether or not the given language L is error-detecting for C. If it is not, then there must be at least one pair of words $(u, v) \in C$, where $u, v \in L$ and $u \neq v$. We refer to such a pair of words as a <u>counterexample</u>. This pair corresponds to a situation where T' is not functional, as $u \in T'(u)$, $v \in T'(u)$, and $u \neq v$. Therefore, providing a counterexample to prove that a given language L is not error-detecting for a channel C reduces to finding a pair of words that makes the transducer T' realizing the relation $C \downarrow L \uparrow L$ not functional.

Our algorithm to find the desired counterexample is a modification of **Algorithm 4.2** as follows:

Algorithm 4.4:

Given a transducer T with start state s, construct product the machine U as follows:

1. If $(p, x/y, q)$ and $(p', x/y', q')$ are transitions in T, then add to U the transition $((p, p'), (x, y, y'), (q, q'))$, where the label (x, y, y') is a tuple recording the input label x of $(p, x/y, q)$ and $(p', x/y', q')$, and the output labels y and y' of $(p, x/y, q)$ and $(p', x/y', q')$. We say that y is the <u>first output label</u> and y' is the <u>second output label</u>.
2. The start state of U is (s, s).

3. The final states of U are all pairs (f, f'), where both f and f' are final states in T.
4. Only keep states that can be reached from (s, s) and can reach a final state (f, f').

After constructing U, assign to each state (p, p') of U two values:

A. A <u>path value</u>, which is a tuple of the form $(x_1 \cdots x_n, y_1 \cdots y_n, {y_1}' \cdots {y_n}')$, where $x_1 \cdots x_n$ is the concatenation of the input labels in a path that begins from the start state (s, s) to state (p, p'), $y_1 \cdots y_n$ is the concatenation of the first output labels in every transition in this path, and ${y_1}' \cdots {y_n}'$ is the concatenation of the second output labels in every transition in this path.

B. A <u>state value</u>, which is either $ZERO$, or a pair of words in $\{(\lambda, \lambda), (\lambda, u), (u, \lambda)\}$, where λ is the *empty* word and u is a nonempty word.

The path value and state value are determined as follows:

1. The start state gets the path value $(\lambda, \lambda, \lambda)$ and the state value (λ, λ).
2. If a state (p, p') has a path value $(x_1, y_1, {y_1}')$ and a state value (y, y') and there is a transition $((p, p'), (x_2, y_2, {y_2}'), (q, q'))$, then (q, q') gets the path value $(x_1 x_2, y_1 y_2, {y_1}' {y_2}')$ and the state value as follows:

   a) If $y y_2$ is a prefix of $y' {y_2}'$, so that $y' {y_2}' = y y_2 u$, then the state value is (λ, u).

   b) If $y' {y_2}'$ is a prefix of $y y_2$, so that $y y_2 = y' {y_2}' u$, then the state value is (u, λ).

c) If yy_2 equals $y'y_2'$, then the state value is (λ, λ).

d) Else, the state value is $ZERO$.

3. Repeat until a state gets two different state values, or a state gets a state value $ZERO$, or every state gets one value. Given a state that already has a path value v_1, if the state gets a new path value v_1', then the path value for this state will now be v_1', independently of how the state value is updated.
4. If a state (p, p'), with a path value $(x_1x_2, y_1y_2, y_1'y_2')$, has two different state values, or (p, p') has a state value that is $ZERO$, where (p, p') is not a final state, then find any path from (p, p') to a final state (f, f') in U: $\big((p, p'), (x_3, y_3, y_3'), (q_1, q_1')\big)$, $\big((q_1, q_1'), (x_4, y_4, y_4'), (q_2, q_2')\big)$, $\cdots, \big((q_n, q_n'), (x_n, y_n, y_n'), (f, f')\big)$. The counterexample can be extracted from the path value that corresponds to the path from (s, s) to (f, f'): $(x_1x_2x_3x_4 \cdots x_n, y_1y_2y_3y_4 \cdots y_n, y_1'y_2'y_3'y_4' \cdots y_n')$.
5. If a final state (f, f') has two different state values, or (f, f') has a state value that is $ZERO$, or a state value that is not equal to (λ, λ), then the path value of (f, f') will be used to extract the counterexample.

When we get the path value:

$$(x_1x_2x_3x_4 \cdots x_n, y_1y_2y_3y_4 \cdots y_n, y_1'y_2'y_3'y_4' \cdots y_n'),$$

according to different situations, we further extract the different words from the path value, depending on the particular situations as follows:

1. In the case where L is not error-detecting for C, we extract the first word

$x = x_1x_2x_3x_4 \cdots x_n$ and z, where z is one of $y = y_1y_2y_3y_4 \cdots y_n$ or $y' = y_1'y_2'y_3'y_4' \cdots y_n'$ that is different from x, as our counterexample. The pair (x, z) is a valid counterexample, because it corresponds to a situation in which, on the same input $x \in L$, the transducer T can output $z \in L$ such that $x \neq z$.

2. In the case where L is not error-correcting for C, we extract the second word $y = y_1y_2y_3y_4 \cdots y_n$ and the third word $y' = y_1'y_2'y_3'y_4' \cdots y_n'$ as our counterexample. This pair (y, y') is a valid counterexample because it corresponds to a situation in which, on some input $x \in L$, the transducer T can output $y \in L$ and $y' \in L$ such that $y \neq y'$.
3. In the case where L is not a code, we simply extract the first word $x = x_1x_2x_3x_4 \cdots x_n$ as our counterexample. The word $x \in L^*$ and the path value tell us that x can be decomposed over L in two different ways as indicated in y and y'.

***Example**: Let us consider the suffix code property $\mathcal{P}_T{}'$ described by the input-preserving transducer T in Figure 4.7 and the language $L = \{ab, bab\}$ accepted by the automaton A in Figure 4.17. Through the steps discussed in Section 4.2, we conclude that L does not satisfy the suffix code property. Now we give a counterexample to prove that why L does not satisfies the suffix code property. We first construct a transducer $T' = T \downarrow A \uparrow A$ – see Section 4.2. The final transducer T' is illustrated in Figure 4.18.*

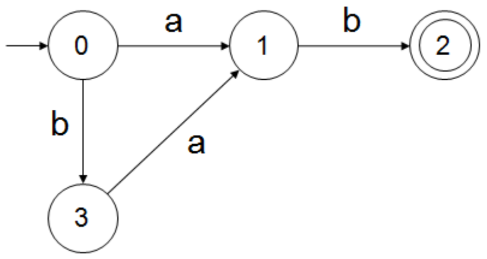


Figure 4.17: The automaton A accepting the language $L = \{ab, bab\}$.

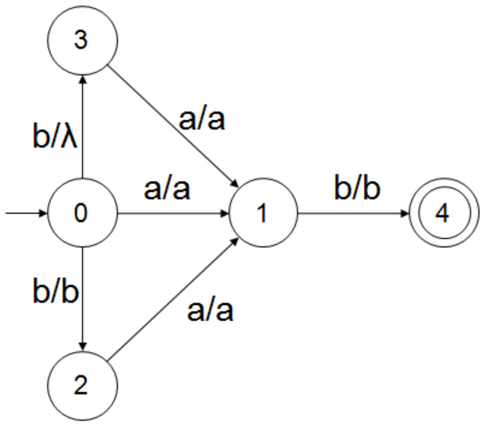


Figure 4.18: The transducer $T' = T \downarrow A \uparrow A$.

*Now we follow **Algorithm 4.4** to construct the product machine U in Figure 4.19.*

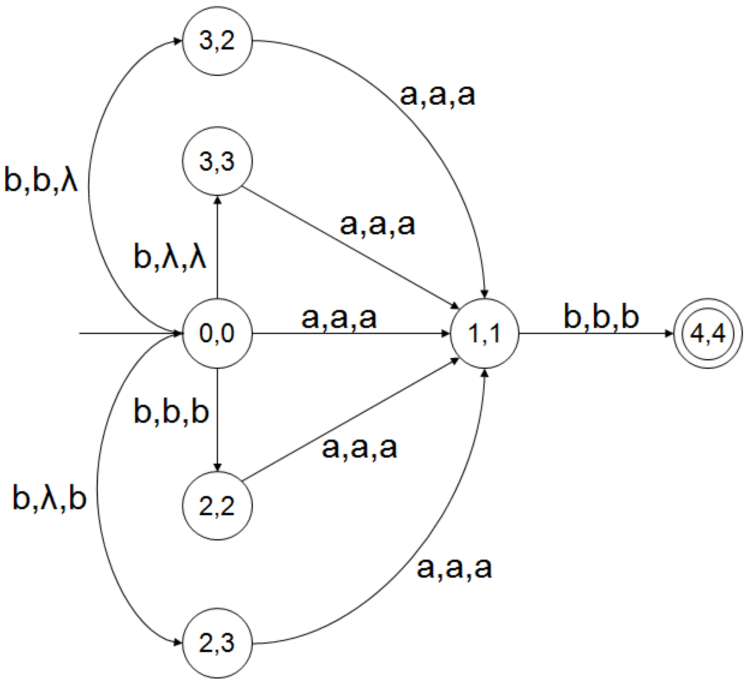


Figure 4.19: The product machine U.

Then we follow **Algorithm 4.4** *to add to every state a path value and a state value until we find a state gets two different state values, or a state gets a state value* $ZERO$*, or a final state* (f, f') *has a state value that not equals* (λ, λ)*. For the start state* $(0,0)$*, we add a path value* $(\lambda, \lambda, \lambda)$ *and a state value* (λ, λ)*. There is a transition* $((0,0), (b, b, \lambda), (3,2))$*, therefore, we add a path value* $(\lambda b, \lambda b, \lambda\lambda)$ *which is* (b, b, λ) *and a state value* (b, λ) *to state* $(3,2)$*. Then there is transition* $((3,2), (a, a, a), (1,1))$*, therefore we add a path value* $(ba, ba, \lambda a)$ *which is* (ba, ba, a) *and a state value* (b, λ) *to state* $(1,1)$*. We illustrate the current status of the product machine* U *in Figure 4.20.*

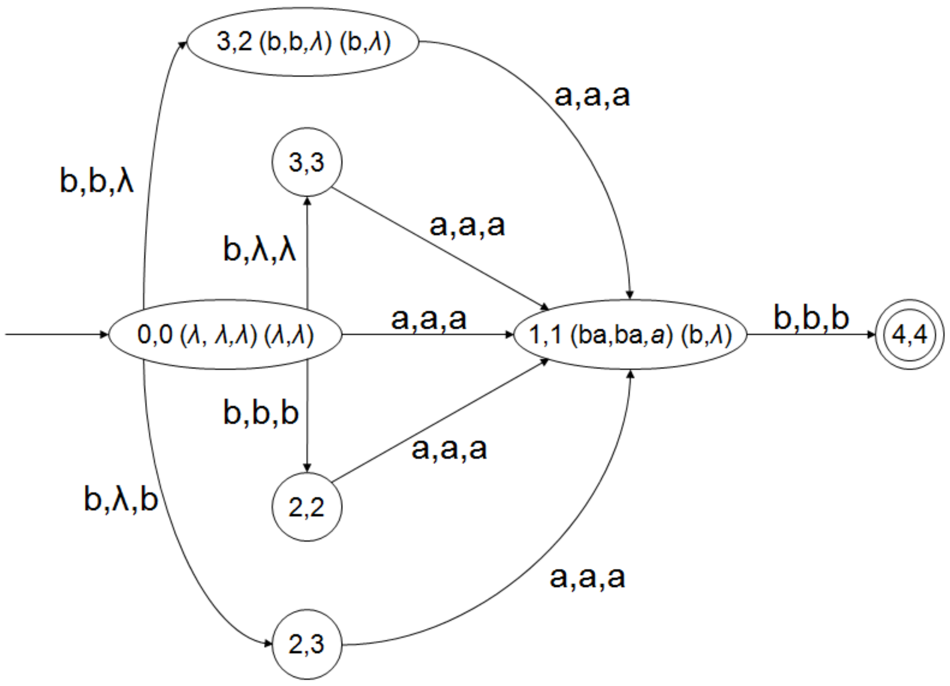


Figure 4.20: Current status of the product machine U.

As there is a transition $((0,0),(b,\lambda,b),(2,3))$, *therefore, we add a path value* $(\lambda b,\lambda\lambda,\lambda b)$ *which is* (b,λ,b) *and a state value* (λ,b) *to state* (2,3). *Also, as there is a transition* $((2,3),(a,a,a),(1,1))$, *therefore we add a path value* $(ba,\lambda a,ba)$, *which is* (ba,a,ba), *and a state value* (λ,b) *to state* $(1,1)$. *However, state* $(1,1)$ *has already got a state value* $(b,\lambda)\neq(\lambda,b)$, *so we finish adding values to the states. As state* $(1,1)$ *is not a final state, we have to find a path from state* $(1,1)$ *to a final state in* U *and the path value of the final state will be used to extract the counterexample. In this case, from state* $(1,1)$, *there is a path* $((1,1),(b,b,b),(4,4))$ *take us from state* $(1,1)$ *to the final state* $(4,4)$. *We then add a path value* (bab,ab,bab) *to state* $(4,4)$. *We conclude that* $L=\{ab,bab\}$ *accepted by the automaton in Figure 4.17 is does not satisfies the suffix code property* $\mathcal{P}_T{}'$ *described by the transducer* T *in Figure 4.5. We extract the first word* bab *and* *the second word ab, which is different from bab as our counterexample. The pair of words (bab, ab) is the counterexample to prove that L does not satisfy $\mathcal{P}_T{}'$, as ab is a suffix of bab, that is, $ab \in T(bab)$, and $bab, ab \in L$ such that $bab \neq ab$, which violates the definition of the suffix code property.*

***Example**: Let us consider the $Del(1, \infty)$ channel C realized by the input-preserving transducer T in Figure 4.13 and a language $L = \{ab, bb\}$ accepted by the automaton A in Figure 4.8 again. In Section 4.3, we have already concluded that L is not error-correcting for C. Now we use **Algorithm 4.4** to provide a counterexample to prove why L is not error-correcting for C. Note that we construct the transducer M^{-1} realizing the relation $C^{-1} \uparrow L$ which is equivalent to the transducer in Figure 4.5 and M^{-1} is not functional. We follow **Algorithm 4.4** to construct the product machine U in Figure 4.21.*

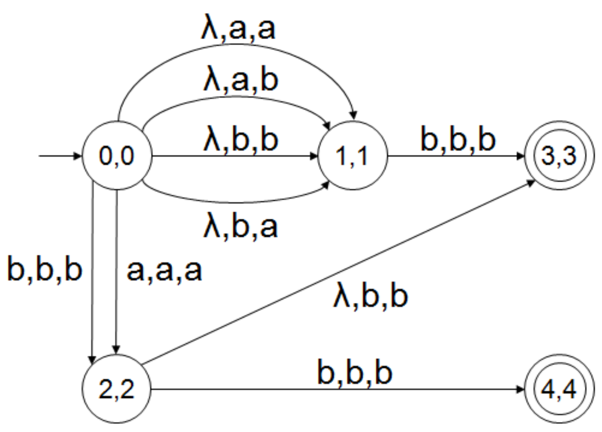


Figure 4.21: The product machine U constructed following ***Algorithm 4.4***.

*Then we follow **Algorithm 4.4** to add to every state a path value and a state value* *until we find a state gets two different state values, or a state gets a state value* $ZERO$, *or a final state* (f, f') *has a state value that not equals* (λ, λ). *For the start state* $(0,0)$, *we add a path value* $(\lambda, \lambda, \lambda)$ *and a state value* (λ, λ). *There is transition* $((0,0), (\lambda, a, b), (1,1))$, *therefore we add a path value* $(\lambda\lambda, \lambda a, \lambda b)$, *which is* (λ, a, b), *and a state value* $ZERO$ *to state* $(1,1)$, *and we finish adding values to the states. We illustrate the current status of the product machine* U *in Figure 4.22.*

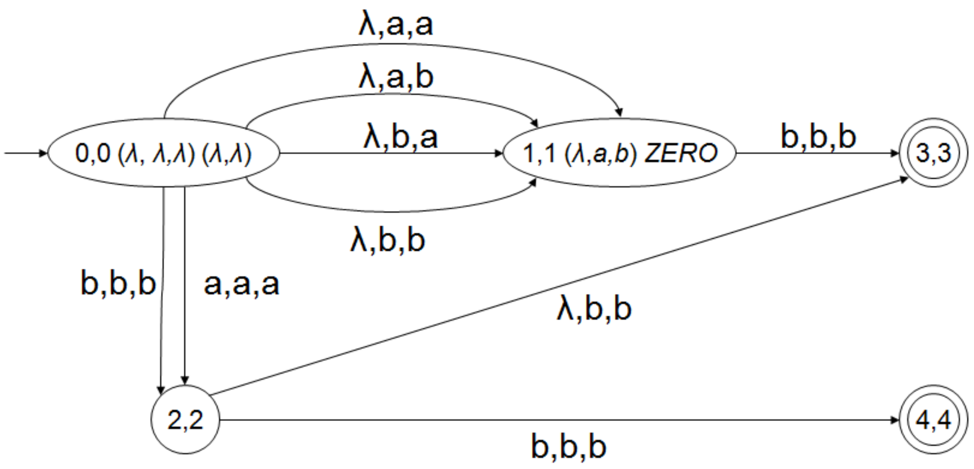


Figure 4.22: Current status of the product machine U.

As state $(1,1)$ *is not a final state, we have to find a path from state* $(1,1)$ *to a final state in* U *and the path value of the final state will be used to extract the counterexample. In this case, from state* $(1,1)$, *there is a path* $((1,1), (b, b, b), (4,4))$ *take us from state* $(1,1)$ *to the final state* $(4,4)$. *We then add a path value* (b, ab, bb) *to state* $(4,4)$. *We conclude that* $L = \{ab, bb\}$ *accepted by the automaton* A *in Figure 4.8 is not error-correcting for* C *described by the transducer* T *in Figure 4.13. We extract the second word* ab *and the third* *word* bb *from the path value* (b, ab, bb) *as our counterexample. The pair of words* (ab, bb) *is the counterexample to prove that* L *is not error-correcting for* C, *as* $(ab, b) \in C$, $(bb, b) \in C$, *and* $ab, bb \in L$ *such that* $ab \neq bb$, *which violates the definition of error-correction property..*

Example: *Let us consider the language* $L = \{0,01,10,11\}$ *described by the NFA* G *in Figure 4.23. Following the steps in Section 4.4, we conclude that* L *is not a code. Now we give a counterexample to prove why* L *is not a code. We construct the transducer* T *according to the step 2 and step 3 in* ***Algorithm 4.3*** *– see Section 4.4. The final transducer* T *we constructed following* ***Algorithm 4.3*** *is illustrated in Figure 4.24.*

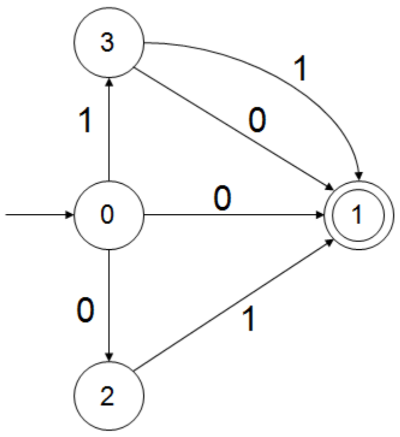


Figure 4.23: The NFA G accepting the language $L = \{0,01,10,11\}$.

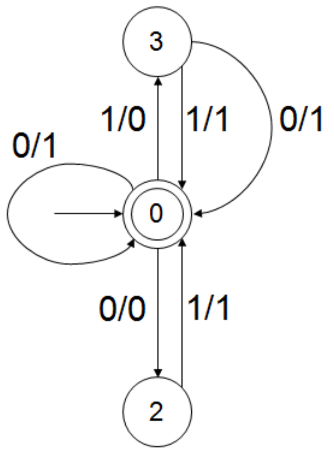


Figure 4.24: The final transducer T we constructed following **Algorithm 4.3**.

*In next step, we decide the functionality of T in order to decide whether L is not a code. We follow **Algorithm 4.4** to construct the product machine U in Figure 4.25.*

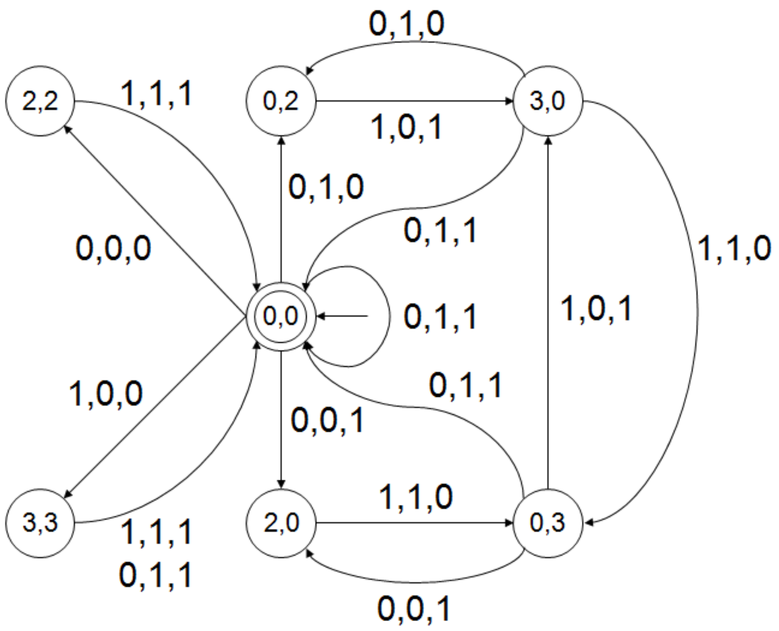


Figure 4.25: The product machine U constructed following ***Algorithm 4.4***.

*Then we follow **Algorithm 4.4** to add to every state a path value and a state value until we find the path value to extract the counterexample. For the start state* $(0,0)$, *we add a path value* $(\lambda, \lambda, \lambda)$ *and a state value* (λ, λ). *There is a transition* $((0,0),(0,1,0),(0,2))$, *therefore we should add a path value* $(\lambda 0, \lambda 1, \lambda 0)$ *which is* $(0,1,0)$ *and a state value* $ZERO$ *to state* $(0,2)$ *and we finishing adding values to the states.*

As state $(0,2)$ *is not a final state, we find a path from state* $(0,2)$ *to the final state* $(0,0)$ *in* U *and the path value of* $(0,0)$ *will be used to extract the counterexample. In this case, from state* $(0,2)$, *there is a path* $((0,2),(1,0,1),(3,0))$ *and* $((3,0),(0,1,1),(0,0))$ *take us from state* $(0,2)$ *to the final state* $(0,0)$. *We then add a path value* $(010,101,011)$ *to state* $(0,0)$. *We conclude that* $L = \{0,01,10,11\}$ *accepted by the automaton* G *in Figure 4.23 is not a code. We extract the first word* 010 *from the path value* $(010,101,011)$. 010 *is the counterexample to prove that* L *is not a code, as* 010 *could be interpreted in two ways:* $(0)(10)$ *and* $(01)(0)$, *which indicated by the two outputs* 101 *and* 011 *in the path value.*

4.6 Constructing input-altering transducers describing code related properties

Recall that in Chapter 3, we present five different code related properties: the prefix code property, the suffix code property, the infix code property, the outfix code property, and the hypercode property. We provide graphical presentations of input-altering transducers describing these properties. In order to use some of the functions [15, 36] in our web interface, we need to construct input-altering transducers based on a given alphabet to describe these properties. As these algorithms are similar, we choose to only present the algorithm to construct the input-altering transducer T_h describing the hypercode property in Figure 3.5.

Given an alphabet Σ, construct the input-altering transducer $T_h = (Q, A, \Gamma, q_0, F, E)$ describing the hypercode property as follows:

Algorithm 4.5:

1. Add a state 0 to Q, and set state 0 as the start state of T_h, that is, $q_0 = 0$.
2. For every symbol σ in the alphabet Σ, add a transition $(0, \sigma/\sigma, 0)$ to the set of transitions E.
3. Add a state 1 to Q, and set state 1 as the only final state of T_h, that is, $F = \{1\}$.
4. For every symbol σ in the alphabet Σ, add a transition $(0, \sigma/\lambda, 1)$ to the set of transitions E.
5. For every symbol σ in the alphabet Σ, add two transitions $(1, \sigma/\lambda, 1)$ and $(1, \sigma/\sigma, 1)$ to the set of transition E.
6. Set the input alphabet and the output alphabet the same as the given alphabet Σ, that is, $A = \Gamma = \Sigma$.

***Example**: Let us consider a given alphabet $\Sigma = \{a, b\}$. We follow **Algorithm 4.5** to construct an input-altering transducer T_h describing the hypercode property. First* *we add a state* 0, *and set state* 0 *as the start state of* T_h *in Figure 4.26a. Then for each symbol* a *or* b *in the alphabet* Σ, *we add two transitions* $(0, a/a, 0)$ *and* $(0, b/b, 0)$ *to* T_h *in Figure 4.26b. Afterwards, we add a state* 1 *and set state* 1 *as the final state of* T_h *in Figure 4.26c. Finally, we add two transitions* $(0, a/\lambda, 1)$ *and* $(0, b/\lambda, 1)$ *to* T_h, *and we add four transitions* $(1, a/\lambda, 1)$, $(1, b/\lambda, 1)$, $(1, a/a, 1)$, *and* $(1, a/a, 1)$ *to* T_h *in Figure 4.26d.*

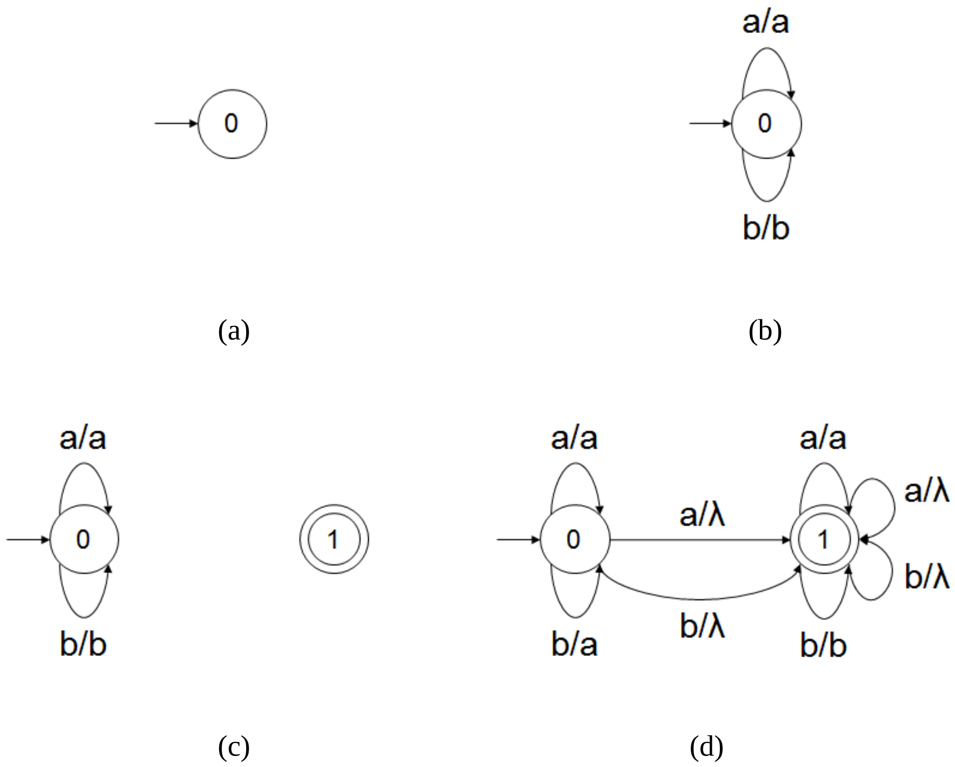


Figure 4.26: Constructing an input-altering transducer describing the hypercode property over the alphabet $\Sigma = \{a, b\}$.

Transducer T_h in Figure 4.26d is the input-altering transducer describing the hypercode property over the alphabet $\Sigma = \{a, b\}$. As algorithms for constructing other code related properties are similar to **Algorithm 4.5**, we won't provide these algorithms here. Readers are referred to Section 3.2.1 for details of using input-altering transducers to describe code related properties.

4.7 Translating a normal form transducer to an equivalent real-time transducer

In this section, we present an algorithm that translates a given transducer in normal form to an equivalent real-time transducer. Our algorithm is based on the method of [3], which uses matrix representation for transducers.

Given a transducer T in normal form, construct an equivalent real-time transducer T' as follows:

Algorithm 4.6:

1. The states of T are also the states of T'.
2. The start state function I of T' is determined as follows: for every state p in T, consider all paths from the start state s to p involving only λ-input transitions. If there exists no such paths, then $I(p) = \emptyset$, which means that p is not a start state in T'. Else, $I(p)$ is the language obtained by concatenating all output labels in those paths, that is, $I(p) \neq \emptyset$, which means that p is a start state in T'. For example, if $(s, \lambda/a, q)$ and $(q, \lambda/b, q')$ are transitions in T', then q and q' will be start states of T' with initial output is $I(q) = a$ and $I(q') = ab$, respectively.

3. For every state p, every state q, and every state r in T, consider any transition of the form $(p, \sigma/\lambda, r)$, with $\sigma \in \Sigma$, and all the paths from state r to state q involving only λ-input transitions. Then, add to T' the transition $(p, \sigma/e, q)$, where e is the regular expression representing the language obtained by concatenating all output labels in those paths from the state r to state q involving only λ-input transitions in T. In other words, e represents the language accepted by the automaton, whose start state is r, the only final state is q, and has transitions (i, x, j) for all transitions $(i, \lambda/x, j)$ in T. For example, if $(p, a/\lambda, r)$, $(r, \lambda/a, q)$, and $(r, \lambda/b, q)$ are transitions in T, then $(p, a/(a+b), q)$ is a transition in T', where $(a+b)$ is a regular expression.
4. The final state function F of T' is as follows: $F(q) = \lambda$, if q is a final state in T, and $F(q) = \emptyset$, if q is not a final state in T.

We provide an example of translating a normal form transducer in Figure 4.27 to an equivalent real-time transducer by using the **Algorithm 4.6**.

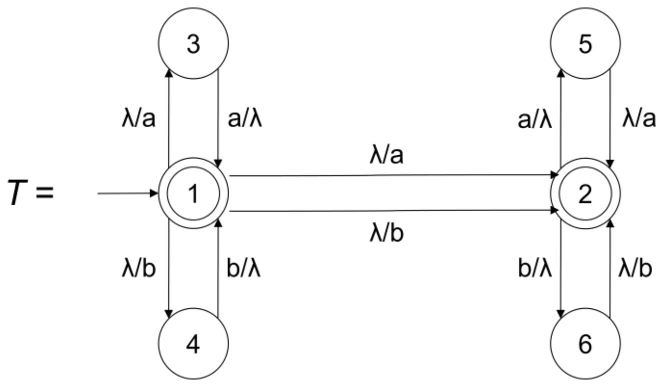


Figure 4.27: A normal form transducer T.

Example: *Let us consider the transducer* T *in normal form (Figure 4.27). In the first step in* ***Algorithm 4.6****, we construct the states of the real-time transducer* T'*, which are the same as the states in* T *(Figure 4.28). Then for step 2, we consider all paths from the start state* s *to state* p *involving only* λ*-input transitions. For example, there is a path from state* 1 *to state* 3 *with the transition* $(1, \lambda/a, 3)$*, which means from the start state* 1*, we can reach state* 3 *without consuming any input symbol and we can get the output symbol* a*. Therefore, state* 3 *is defined to be one of the start states of* T'*, that is,* $I(3) = a$*. Also, let us consider the path* $(1, \lambda/a, 2)$ *and* $(1, \lambda/b, 2)$ *from start state* 1 *to state* 2*, which means that we can reach state* 2 *from start state* 1 *without consuming any input symbol and we can get the output symbol* a *or* b*. Therefore, state* 2 *is also defined to be one of the start states in* T'*, that is, that is,* $I(2) = a + b$*, which is a regular expression. The real-time transducer* T' *we constructed after step 2 in* ***Algorithm 4.6*** *is presented in Figure 4.28. Note that there are arrows pointing to some of the states in the real-time transducer* T'*. This means that these states are the start states of* T'*, and the label or regular expression in the arrows represents the output of the state, respectively.*

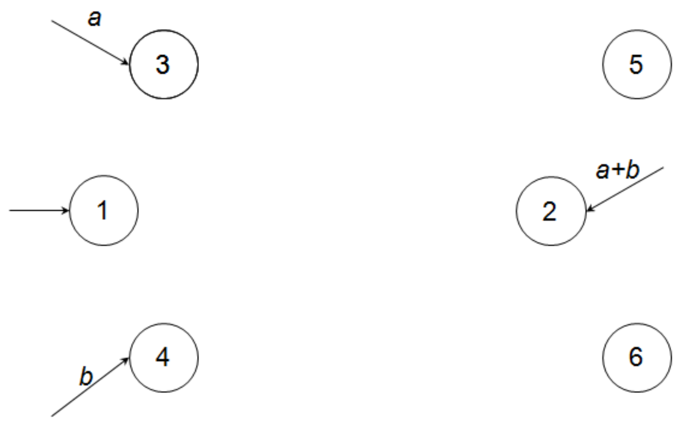


Figure 4.28

Then we add transitions to T'. *Let us consider the path* $(3, a/\lambda, 1)$, $(1, \lambda/a, 3)$ *from state* 3 *to state* 1 *and from state* 1 *to state* 3 *itself. According to step 3 in* ***Algorithm 4.6****, state* 3 *will be the state* p, *and state* 1 *will be the state* r, *and state* 3 *will be the state* q. *For* $(3, a/\lambda, 1)$, *the input label is* $a \in \Sigma$, *and the path from state* r *to state* q *involving only* λ*-input transitions, that is,* $(1, \lambda/a, 3)$. *Therefore, we add the transition* $(3, a/a, 3)$ *to* T', *in which the input label is the same to the input label in transition* $(3, a/\lambda, 1)$ *and the output label is the language of concatenating all output labels in those paths from the state* 1 *to state* 3 *involving only* λ*-input transitions in* T, *which is only one symbol word* a. *Also, let us consider the path* $(3, a/\lambda, 1)$, $(1, \lambda/a, 2)$ *or* $(3, a/\lambda, 1)$ $(1, \lambda/b, 2)$. *We apply the same process to add two transitions* $(3, a/a, 2)$ *and* $(3, a/b, 2)$ *from state* 3 *to state* 2. *Or more precisely, we add one transition* $(3, a/(a+b), 2)$ *from state* 3 *to state* 2. *The real-time transducer* T' *we constructed after step 3 in* ***Algorithm 4.6*** *is presented in Figure 4.29.*

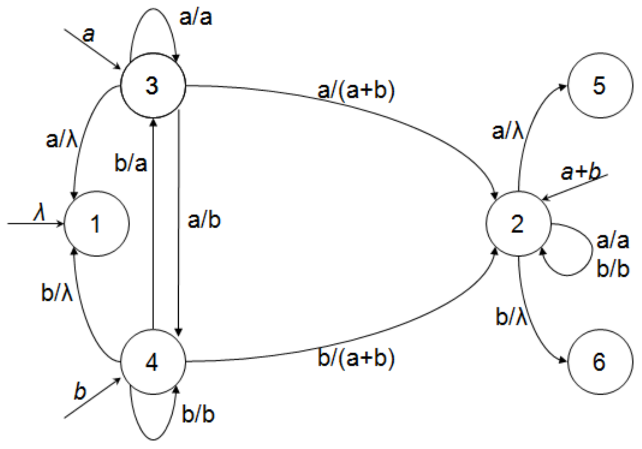


Figure 4.29

Finally we designate state 1 *and state* 2, *which are the two final states in* T, *as the final states of* T'. *The final result of* T', *which is equivalent to* T, *is presented in Figure 4.30.*

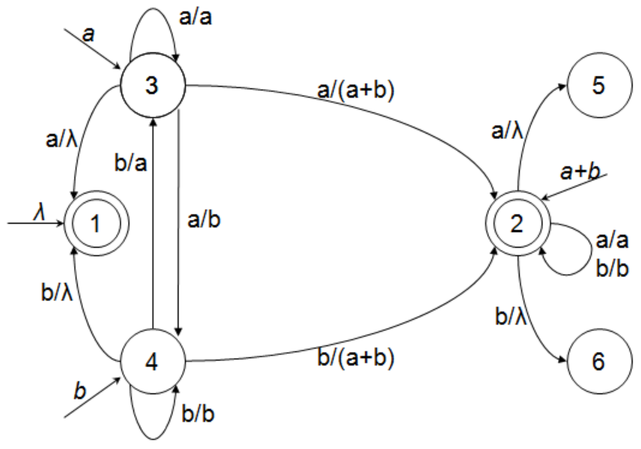


Figure 4.30: The real-time transducer T' equivalent to T in Figure 4.27.

Chapter 5
Computing the edit distance of a regular language

In this chapter, we look at the problem of computing the edit distance of a given regular language. We discuss a known method that uses the error-detection property, and we introduce a new method that uses the error-correction property to estimate the edit distance of a regular language. Our method is inexact, as it produces two possible values for the desired edit distance, but it is faster than the existing method. We use the algorithmic tools in Chapter 2 and Chapter 4 to estimate the edit distance, and we present examples to illustrate how these algorithmic tools work. Also, we provide a few performance tests for the existing method and our method.

5.1 Edit distance

The concept of edit distance [38] is important in various information processing applications, such as speech processing and bioinformatics [43]. The edit distance (also called Levenshtein Distance) of two words a and b, denoted by $D(a,b)$, is the minimum number of possible edits (substitutions, insertions, and deletions) required to transform a into b. The edit distance of a regular language L, denoted by $D(L)$, is the smallest edit distance between any two distinct words in L, that is,

$$D(L) = \min\{D(u,v) | u,v \in L, u \neq v\}.$$

***Exmaple**: The edit distance between* 010011 *and* 1110011 *is*

$D(010011, 1110011) = 2$. *Although, there are many ways to transform* 010011 *to* 1110011 *and vice versa, the shortest way to transform* 010011 *to* 1110011 *is by inserting a* 1 *at the beginning of* 010011 *to get* 1010011, *and then substituting the first* 0 *with* 1 *in* 1010011 *to get* 1110011. *Therefore, 2 is the minimum number of possible edits required to transform* 010011 *to* 1110011 *, that is,* $D(010011, 1110011) = 2$. *The edit distance of the regular language*

$L = \{10110, 01100,101,01110\}$ *is* $D(L) = 1$, *as the edit distance between* 01100 *and* 01110 *is* 1, *so* $D(01100, 01110) = 1$ *is the smallest edit distance between any two distinct words in* L.

Given a DFA A accepting a regular language L, we have the following facts:

1. The edit distance $D(L)$ of a language L is less than, or equal to, the diameter of A, that is, $D(L) \leq diam(A)$ [29].
2. Given a language L and integer $m \geq 0$, we have that $D(L) > m$ if and only if L is error-detecting for $SID(m, \infty)$ [35, 38], where $SID(m, \infty)$ is the channel which allows a maximum of m substitutions, insertions, and deletions errors in any given input word, we have that.
3. Given a language L and integer $m \geq 0$, we have that $D(L) > 2m$ if and only if L is error-correcting for $SID(m, \infty)$ [38].

In the next section, we discuss how to use the error-detection property to compute the edit distance of a regular language.

5.2 Computing the edit distance using the error-detection property

Recall that in Section 5.1, given a DFA A accepting a regular language L and an integer $m \geq 0$, we have that $D(L) > m$ if and only if L is error-detecting for $SID(m, \infty)$. Therefore, the question of computing the edit distance of a given regular language accepted by A is to find the largest integer m in $[0, diam(A)]$, such that L is error-detecting for $SID(m-1, \infty)$ but not error-detecting for $SID(m, \infty)$ [35]. If such m is found, then $D(L) = m$.

Given a regular language L accepted by a DFA A and a channel C realized by an input-preserving transducer T, the time complexity of the Cartesian product operation $T \downarrow A$ is $O(|T||A|)$, where $|T|$ is the size of the transducer T and $|A|$ is the size of the automaton A. The size of an automaton or a transducer is the sum of the number of states and transitions. Therefore, the time complexity of constructing the transducer T' realizing the relation $C \downarrow L \uparrow L$ is $O(|T||A|^2)$. As the time complexity of deciding whether the transducer T' is functional is $O(|T'|^2)$, the time complexity of deciding whether L is error-detecting for C is $O(|T'|^2) = O(|T||A|^2)^2 = O(|T|^2|A|^4)$.

In order to find the largest number m in $[0, diam(A)]$, such that L is error-detecting for $SID(m-1, \infty)$ but not error-detecting for $SID(m, \infty)$, we have to perform the algorithm to decide whether L is error-detecting for the SID channel for each m in $[0, diam(A)]$. However, if we use binary search to find the desired m in $[0, diam(A)]$, the error-detection algorithm will be used $O(log\, d)$ times, where $d = diam(A)$. Given an integer m, the size of the transducer T realizing $SID(m, \infty)$ is determined by m, that is, $|T| = O(m) = O(diam(A))$, where $0 \leq m \leq diam(A)$. Therefore, the time complexity of computing the edit distance of a given regular language L accepted by a DFA A using the error-detection property is $O(|A|^4\, |T|^2 log\, d) = O(|A|^4\, d^2 log\, d)$, where $d = diam(A)$.

In the next section, we introduce an algorithm to estimate the edit distance of a given regular language using the error-correction property. Our algorithm is inexact, as it produces two possible values, but it is faster than the above method.

5.3 Estimating the edit distance using the error-correction property

Recall that in Section 5.1, given a language L and an integer $m \geq 0$, we have that $D(L) > 2m$ if and only if L is error-correcting for $SID(m, \infty)$. Therefore, the question of computing the edit distance of a given regular language accepted by A is to find the largest positive integer number m in $[0, \lceil diam(A)/2 \rceil]$, such that L is error-correcting for $SID(m-1, \infty)$ but not error-correcting for $SID(m, \infty)$. If such m is found, then $D(L) > 2(m-1)$, and $D(L) \leq 2m$. Therefore the edit distance of L is $D(L) = 2m - 1$ or $D(L) = 2m$.

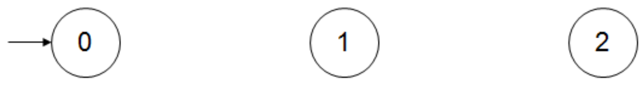
In order to use the error-correction property to estimate the edit distance of a regular language, we have to construct an input-preserving transducer realizing the $SID(m,\infty)$ channel based on a given positive integer number m and a given alphabet Σ. The algorithm to construct such transducer $T=(Q,\Sigma,\Gamma,q_0,F,E)$ is as follows:

Algorithm 5.1:

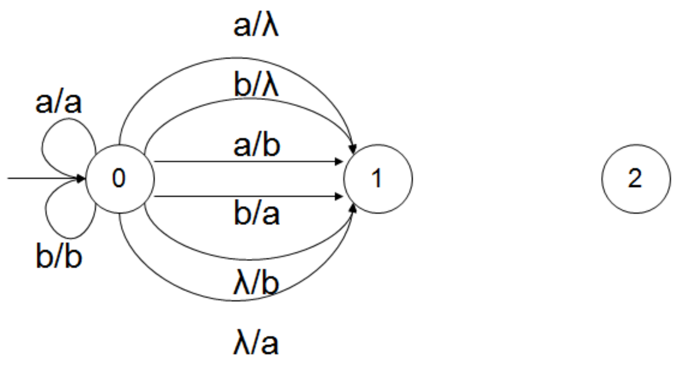
1. Set the state set $Q=\{0,1,2,\cdots,m\}$. The total number of states in T is $m+1$.
2. Set state 0 as the start state of T, that is, $q_0=0$.
3. For every state $p\in Q=\{0,1,2,\cdots,m-1\}$, add the following transitions to E:
   - $(p,\sigma/\sigma,p)$ for all $\sigma\in\Sigma$
   - $(p,\sigma/\lambda,p+1)$ for all $\sigma\in\Sigma$
   - $(p,\lambda/\sigma,p+1)$ for all $\sigma\in\Sigma$
   - $(p,\sigma/\sigma',p+1)$ for all $\sigma\in\Sigma$ and $\sigma'\in\Sigma-\{\sigma\}$
4. For state $p=m$, add the following transitions to E:
   - $(p,\sigma/\sigma,p)$ for all $\sigma\in\Sigma$
5. Set all states $p\in Q=\{0,1,2,\cdots,m\}$ as final states of T, that is, $F=\{0,1,2,\cdots,m\}$.

***Example**: Let us construct an input-preserving transducer T realizing the $SID(2,\infty)$ channel based on a given positive integer number $m=2$ and a given alphabet*

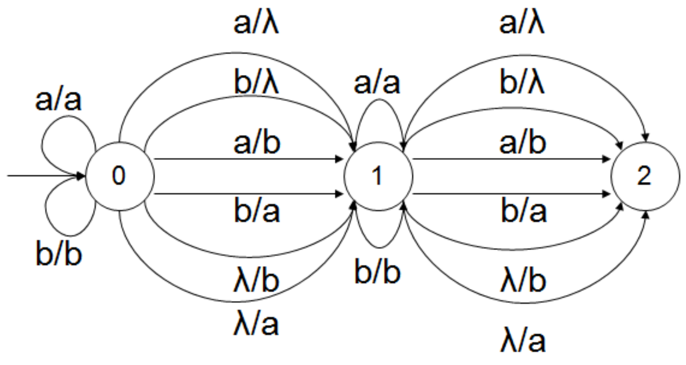
$\Sigma = \{a, b\}$. *Following* ***Algorithm 5.1****, we define the state set* $Q = \{0,1,2\}$, *and set the state* 0 *as the start state of* T *in Figure 5.1a. Then for state* 0 *and for every symbol in* $\Sigma = \{a, b\}$, *we add the following transitions to* T*:* $(0, a/a, 0)$, $(0, b/b, 0)$, $(0, a/\lambda, 1)$, $(0, b/\lambda, 1)$, $(0, \lambda/a, 1)$, $(0, \lambda/b, 1)$, $(0, a/b, 1)$, *and* $(0, b/a, 1)$ *in Figure 5.1b. Also, for state* 1 *and for every symbol in* $\Sigma = \{a, b\}$, *we add the following transitions to* T*:* $(1, a/a, 1)$, $(1, b/b, 1)$, $(1, a/\lambda, 2)$, $(1, b/\lambda, 2)$, $(1, \lambda/a, 2)$, $(1, \lambda/b, 2)$, $(1, a/b, 2)$, *and* $(1, b/a, 2)$ *in Figure 5.1c. Finally we add the following transitions to* T*:* $(2, a/a, 2)$, $(2, b/b, 2)$, *and we set all the states* $Q = \{0,1,2\}$ *as the final states of* T *in Figure 5.1d, and the transducer* T *in Figure 5.1d realizes the channel* $SID(2, \infty)$.



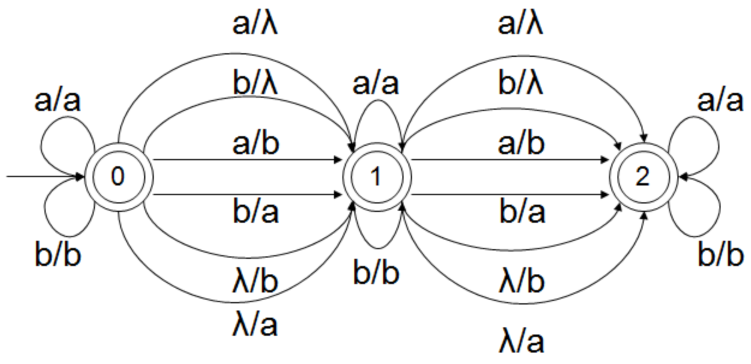
(a)



(b)



(c)



(d)

Figure 5.1: The input-preserving transducer T realizing the channel $SID(2,\infty)$.

Now, we can find the largest positive integer number m in $[0,\lceil diam(A)/2\rceil]$, such that L is error-correcting for $SID(m-1,\infty)$ but not error-correcting for $SID(m,\infty)$. We here only give examples to show how to find the integer m. Readers are referred to Chapter 2 and Chapter 4 for the question of deciding whether a language L is error-correcting for a channel C.

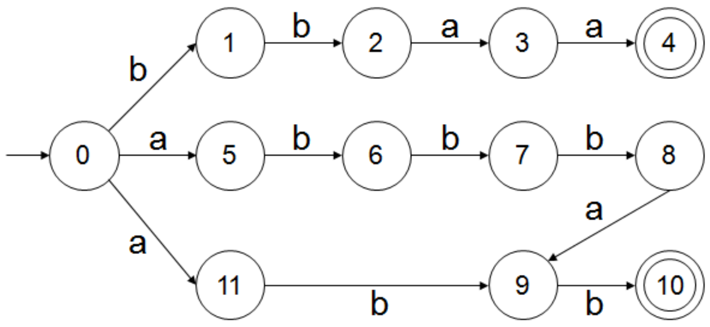


Figure 5.2: Automaton accepting the language $L = \{bbaa, abb, abbbab\}$.

***Example**: Let us consider the regular language $L = \{bbaa, abb, abbbab\}$ in Figure 5.2. Following the interpretation in Section 4.3, it can be decided that the language L is error-correcting for $SID(1, \infty)$. We do not show the details of how this decision is made. However, it is also decided that the language L is not error-correcting for $SID(2, \infty)$. The counterexample abb and $bbaa$ can be found following **Algorithm 4.4**. It is easy to understand why $L = \{bbaa, abb, abbbab\}$ is not error-correcting for $SID(2, \infty)$, as both abb and $bbaa$ can be changed into bba. Therefore, we conclude that the edit distance of L is* $D(L) = 2m - 1 = 3$ or $D(L) = 2m = 4$.

Given a regular language L accepted by a DFA A and a channel C realized by an input-preserving transducer T, the time complexity of constructing the transducer T' realizing the relation $C^{-1} \uparrow L$ is $O(|T||A|)$. The time complexity of deciding whether the transducer T' is functional is $O(|T'|^2)$. Therefore, the time complexity of deciding whether L is error-correcting for C is $O(|T'|^2) = O(|T||A|)^2 = O(|T|^2|A|^2)$.

In order to find the largest positive integer number m in $[0, \lceil diam(A)/2 \rceil]$, such that L is error-correcting for $SID(m-1, \infty)$ but not error-correcting for $SID(m, \infty)$, we have to perform the algorithm to decide whether L is error-correcting for the SID channel for each m in $[0, \lceil diam(A)/2 \rceil]$. Again, if we use binary search algorithm to find the desired m in $[0, \lceil diam(A)/2 \rceil]$, the error-correction algorithm will be used $O(\log d)$ times, where $\mathrm{d} = \lceil diam(A)/2 \rceil$. Therefore, the time complexity of computing the edit distance of a given regular language L accepted by a DFA A using the error-correction property is $O(|T|^2 |A|^2 \log d) = O(|A|^2 (d)^2 \log d)$, where $\mathrm{d} = \lceil diam(A)/2 \rceil$.

Obviously, estimating the edit distance of a regular language L using the error-correction property is faster than computing it using the error-detection property, but the new algorithm produces two values for $D(L)$: $2m-1$ or $2m$, where m is the integer such that L is error-correcting for $SID(m-1, \infty)$ but not error-correcting for $SID(m, \infty)$. In order to compute exactly the edit distance of L, we can test whether L is error-detecting for $SID(2m-1, \infty)$. If L is error-detecting for $SID(2m-1, \infty)$, then $D(L) = 2m$, if not, $D(L) = 2m-1$. We conclude that computing the exact edit distance of a regular language L using the combination of the error-correction property and error-detection property is still faster than computing the edit distance of a regular language L using only the error-detection property.

***Example**: Let us consider again the regular language* $L = \{bbaa, abb, abbbab\}$ *in*

Figure 5.2. We have already concluded that the edit distance of L is $D(L) = 2m - 1 = 3$ or *$D(L) = 2m = 4$. In order to decide the exact edit distance of L, we decide whether L is error-detecting for $SID(3, \infty)$. Following the interpretation in Section 4.2, we conclude that L is not error-detecting for $SID(3, \infty)$, and the counterexample is abb and $bbaa$, as abb could be changed into $bbaa$ that is also in L. Therefore we conclude that the edit distance of $L = \{bbaa, abb, abbbab\}$ is $D(L) = 3$.*

5.4 Performance tests

We provide two performance tests in terms of time elapse for the existing method of computing the edit distance of a regular language using only the error-detection property, and for our new method of estimating the edit distance using the error-correction property and computing the exact edit distance using the combination of the error-correction property, and error-detection property. The performance tests are executed on the compute resource named *Mahone* in *ACEnet* [1], which is located at Saint Mary's University, Halifax, Nova Scotia, Canada.

The first performance test involves the regular language $L_n = (a^n)^*b$, for $n \geq 2$. The automaton A_n accepting L_n is shown in Figure 5.3. The edit distance is $D(L_n) = n$. In this sequence of automata, the number of states, the edit distance, and the diameter grow with n. In particular, the number of states is n, and the diameter is n, so this presents a worst case input to the algorithm when it comes to the number of times that the language is tested for the error-detection property and the error-correction property.

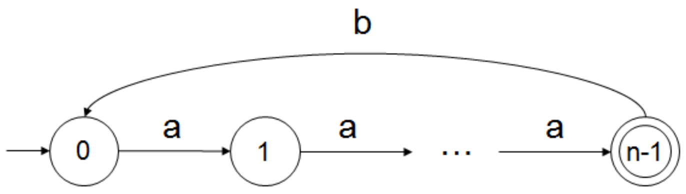


Figure 5.3: The automaton A_n accepting the language $L_n = (a^n)^*b$, for $n \geq 2$.

The result of the first performance test is shown in Table 5.1. The time elapse for each automaton is an approximate time, which is depending on different computers.

Algorithm / Automaton	Error-detection only	Error-correction only	Error-correction + one error-detection
A_4	0.4s	0.04s	0.11s
A_6	11.6s	0.30s	2.75s
A_8	195.7s	1.8s	44.45s
A_{10}	2151.3s	8.7s	559.08s
A_{12}	13553.5s	32.3s	4388.38s

Table 5.1

The result of Table 5.1 shows that our method of estimating the edit distance of a regular language is much faster than the existing method using only the error-detection property. We can also see that computing the exact edit distance of a regular language using the combination of the error-correction property and error-detection property is still faster than computing the edit distance via only the error-detection property.

We provide a more detailed result for our method in Table 5.2.

Algorithm / Automaton	Error-correction only
A_4	0.04s
A_6	0.30s
A_8	1.84s
A_{10}	8.71s
A_{12}	32.31s
A_{14}	93.28s
A_{16}	340.74s
A_{18}	1049.76s
A_{20}	2764.68s

Table 5.2

The second performance test involves the well known codes

$$L(n) = \left\{ b_1 b_2 \cdots b_n | \left(\sum_{i=1}^{n} i \cdot b_i \right) \equiv 0 \big(mod\ (n+1)\big) \right\}$$

for $n \geq 2$ of Levenshtein [38]. The edit distance of $L(n)$ is $D(L(n))$ = 2, for all $n \geq 2$. In the sequence of automata accepting $L(n)$, the number of states is $O(n^2)$ and the diameter is equal to $n+1$, but the edit distance is fixed for all $n \geq 2$ – unlike the edit distance in the first performance test.

The result of the second performance test is shown in Table 5.3.

Algorithm / Automaton	Error-detection only	Error-correction only	Error-correction + one error-detection
B_2	0.05s	0.02s	0.01s
B_4	1.46s	0.38s	0.40s
B_6	385.48s	3.96s	4.03s
B_8	63868.80s	265.60s	270.45s
B_{10}	N/A	2086.56s	2249.09s

Table 5.3

Also, the result of Table 5.3 shows that our method of estimating the edit distance of a regular language is much faster than the existing method. Note that there is no such obvious difference of time elapse between the method of using error-correction only and the method of combination of error-correction and error-detection. This is because for every $n \geq 2$, the edit distance of $L(n)$ is always $D(L(n))$ = 2. Therefore, the time elapse difference between these two methods is the processing time of one test for deciding if $L(n)$ is error-detecting for $SID(1,\infty)$, whose time complexity is only $O(|A|^4|T|^2) = O(|A|^4 d^2)$, where $d = 1$.

Chapter 6
Implementation

In addition to our theoretical research, we develop an implementation of the algorithms and a web interface. An existing web interface named LaSer (Language Server) [36] is already established in [15], and another one in [10]. As our research is a continuation of these works, we enhance the capabilities of LaSer and we deliver a new web interface named I-LaSer [24], consisting of a web interface and implementations of the algorithmic tools and methods discussed in the previous chapters. In doing so, we also provided an implementation of transducer classes in our copy of FAdo libraries.

In [15], the software consists of two main elements: implementations of the algorithms and a web interface. The algorithms are implemented in the C++ language with Boost libraries [6]. The web interface of [15] is developed using Python language with Django web framework [12]. In our research, we do not make any changes to the architecture of the web interface. We do not use C++ to implement our methods, instead we implement our algorithms in Python language with FAdo libraries [2, 18], as FAdo libraries are available during our research and are powerful libraries contain most of basic implementations of concepts in automaton theory. This combination allowed us to take advantage of both the convenience of the existing FAdo libraries and the convenience of Django as a rapid web application development solution.

The architecture of our implementation is illustrated in Figure 6.1.

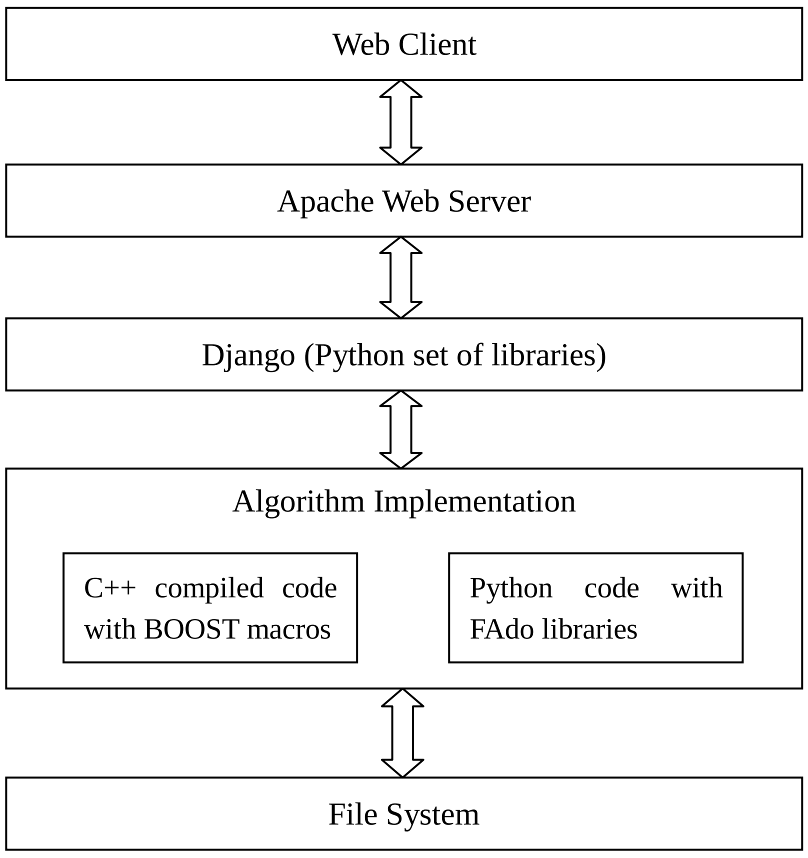

Web Client
Apache Web Server
Django (Python set of libraries)
Algorithm Implementation
C++ compiled code with BOOST macros
Python code with FAdo libraries
File System

Figure 6.1 System Architecture.

6.1 Implementation of the algorithms

The back end functionality and data structures in the implementations of our algorithms are encapsulated in three main classes:

- FA – the basic class for finite automata, encapsulating the logic of single automaton and all the operations performed on automata. The classes of DFA and NFA are derived from the FA class. These were already available in FAdo libraries.

- Transducer – encapsulate the logic of single transducer and all the operations performed on transducer. This class is an outcome of this research.
- Real-time transducer – encapsulate the logic of real-time transducer and all the operations performed on real-time transducer. This class is an outcome of this research.

The FA class encapsulates the structure of an automaton, as a set of start states, a set of final states, a list of states, a set of alphabet, and a dictionary of transitions. Note that the data structure of dictionary in Python is similar to the data structure of hash table in C++ and Java. The FA class also encapsulates the operations performed on automata.

Example: *The automaton in Figure 2.1 would be logically represented as follows:*

- *List of states:* $['0','1','2']$, *where every element in this list is the string of the name of the state.*
- *Set of start states:* $set([0])$, *where* 0 *is the index to find the start state in the state list. For example, in this case, the start state is* '0'.
- *Set of final states:* $set([2])$, *where 2 is the index to find the final state in the state list.*
- *Alphabet:* $set(['a','b'])$.
- *Dictionary of transitions:* $\{0\colon \{'a'\colon set([1]),'a'\colon set([0])\}, 1\colon \{'b'\colon set([2])\}\}$, *where the keys of this dictionary are the origin states in the transitions, and*

the values are another dictionary in which the keys are the label in the transitions and values are the end states.

Following are some important methods in the public interface under the FA class and the derived class DFA and NFA:

- **trim()**: This method removes states that do not reach a final state, or, inclusively, cannot be accessed from the start state. Only useful states remain.
- **regexpSE()**: This method generates a regular expression obtained by state elimination whose language is accepted by the automaton.
- **epsilonClosure()**: This method returns the set of states connected only by λ-transitions from the given state or set of states.
- **__and__()**: This method performs the Cartesian product operation of two automata discussed in Chapter 2.
- **addEpsilonTransition()**: This method adds self λ-transitions to every state in the automaton.
- **CodeP()**: This is a method introduced in this research under the NFA class. The method decides whether a language accepted by an NFA is a code using **Algorithm 4.3**.

The Transducer class encapsulates the structure of a single transducer as a set of start states, a set of final states, a list of states, a set of input alphabet, a set of output alphabet, and a dictionary of transitions. The Transducer class also encapsulates the operations performed on transducers.

Example: *The transducer in Figure 4.7 would be logically represented as follows:*

- *List of states:* $['0','1']$
- *Set of start states:* $set([0])$
- *Set of final states:* $set([0,1])$, *where* 0 *and* 1 *indicates that in this case the transducer has two final states.*
- *Input alphabet:* $set(['a','b'])$
- *Output alphabet:* $set(['a','b'])$
- *Dictionary of transitions:* $\{1: \{'a': [['a', set([1])]],\ 'b': [['b', set([1])]]\},$ $0: \{'a': [['@epsilon', set([0])],\ ['a', set([1])]],$ 'b':$[['@epsilon', set([0])],$ $['b', set([1])]]\}\}$

Following are some important functions in the public interface under the Transducer class:

- **toRealTimeREType()**: This method translates a transducer in normal form to an equivalent real-time transducer using **Algorithm 4.6**. The output labels of this type of the real-time transducer are represented as regular expressions.
- **toRealTimeAutomatonType()**: This method translates a transducer in normal form to an equivalent real-time transducer **Algorithm 4.6**. The output labels of this type of the real-time transducer are represented as automata. This method will export a file containing all the descriptions of the automata.

- **__and__()**: These two methods perform the input Cartesian product operation $T \downarrow A$ of a transducer T and an automaton A – see Section 2.6.
- **outputIntersect()**: This method performs the output Cartesian product operation $T \uparrow A$ of a transducer T and an automaton A – see Section 2.6.
- **inverse():** This method generates the inverse transducer T^{-1} of a given transducer T by switching the input label with the output label in every transition. No start state or final states will be changed.
- **epsilon()**: This method tests whether a given transducer has λ-input transitions.
- **addEpsilonTransition()**: This method adds self (λ/λ) transitions to every state in the transducer.
- **standardToNormalForm()**: This method translates a given transducer in standard form to an equivalent transducer in normal form.
- **generalToStandardForm()**: This method translates a given transducer in general form to an equivalent transducer in standard form.
- **functionalP()**: This method decide whether a given transducer in standard form is functional using **Algorithm 4.2**. In addition, if the transducer is not functional, it will generate a counterexample to show why it is not functional – see **Algorithm 4.4**.
- **crossProductConstruction()**: This method performs the cross product construction between a standard form transducer and itself to construct the product machine U **Algorithm 4.2**.

The Real-time transducer class encapsulates the structure of a real-time transducer as a dictionary of start states, a set of final states, a list of states, a set of input alphabet, a set of output alphabet, and a dictionary of transitions. The Real-time transducer class also encapsulates the operations performed on real-time transducers.

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

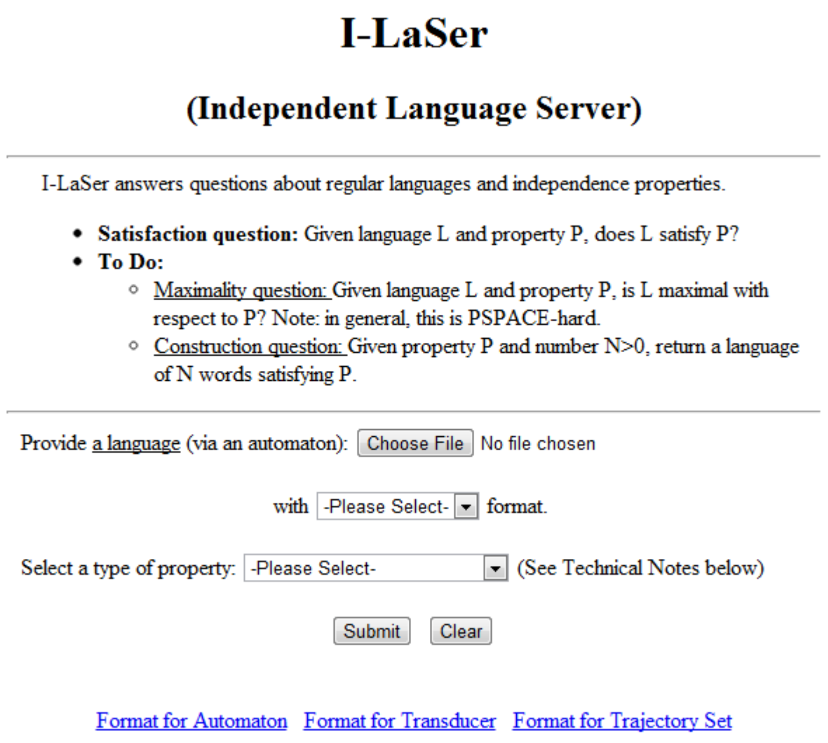
- **simpleFunctionalP()**: This method decides whether a restricted sequential transducer is functional using **Algorithm 4.1**.
- **functionalP()**:This method decides whether a given real-time transducer is functional using **Algorithm 4.2**. In addition, if the transducer is not functional, it will generate a counterexample to show why it is not functional – see **Algorithm 4.4**.

6.2 User interface

Our software is accessible via a web interface, which is called I-LaSer (Independent Language Server) under following URL:

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

The outlook of I-LaSer is illustrated in Figure 6.2.


I-LaSer

(Independent Language Server)

I-LaSer answers questions about regular languages and independence properties.

- **Satisfaction question:** Given language L and property P, does L satisfy P?
- **To Do:**
  - Maximality question: Given language L and property P, is L maximal with respect to P? Note: in general, this is PSPACE-hard.
  - Construction question: Given property P and number N>0, return a language of N words satisfying P.

Provide a language (via an automaton): Choose File No file chosen

with -Please Select- format.

Select a type of property: -Please Select- (See Technical Notes below)

Submit Clear

Format for Automaton Format for Transducer Format for Trajectory Set

Figure 6.2: The outlook of I-LaSer.

I-LaSer integrates the functions in LaSer and the algorithms in our research. For now, I-LaSer is currently capable of answering the satisfaction question: given the description of a regular language and the description of an independence property, decide whether the language satisfies the property. Readers are referred to [27] for the details about the general concept of independent language properties. In our research we restrict our attention to 3-independence properties, or equivalently, properties defined by binary relations [45].

Comparing to LaSer, I-LaSer has the following functional improvements:

1. Provide functions to answer the questions of whether a given regular language satisfies a fixed property (Prefix, Suffix, Infix, Outfix, Hypercode,

Code).

2. Allow users to describe trajectories properties via regular expressions.
3. Provide the function to answer the question of whether a given regular language satisfies a given language property described by an input-preserving transducer (equivalently, whether a given regular language is error-detecting for a channel realized by the same input-preserving transducer).
4. Provide the function to answer the question of whether a given regular language is error-correcting for a channel realized by an input-preserving transducer.
5. Allow users to upload files to describe the automaton and transducer in either Grail or FAdo format.
6. Implement the feature of providing a counterexample when a given language does not satisfies a given language property.

We allow user to describe the independence properties in the following three methods:

1. Via sets of trajectories [13]. Trajectory is a formal method for describing an independence property via a regular expression e over $\{0,1\}$, such that a language L satisfies the property if

   $$L \cap (L \amalg_e \Sigma^+) = \emptyset, \tag{1}$$

   Where $\amalg_e$ is the shuffle operation on the trajectory set e. For example, 0^*1^* describes the prefix code property and $0^*1^*0^*$ describes the infix code property.

2. Via input-altering transducers [15, 16]. In particular, a property is defined via an input-altering transducer T such that a language L satisfies the property if

$$L \cap T(L) = \emptyset. \qquad (2)$$

3. Via input-preserving transducers [16]. A property is defined via an input-preserving transducer T such that a language L satisfies the property if, for all $x \in L$,

$$(L - x) \cap T(x) = \emptyset \qquad (3)$$

In order to decide the satisfaction questions of a regular language and a property described by one of the three decision methods we mentioned above, users are required to specify a file containing the description of the automaton accepting the regular language and a file containing the description of the language property via trajectory or transducer. In each case, I-LaSer tests the corresponding condition (1), (2), or (3) shown above, and returns the computation result displayed in the web interface.

In addition, I-LaSer provides six fixed code related language properties for user to choose. Decision of the "Code" property is implemented in Python with FAdo libraries in our research – see Section 4.4. For the other five code related properties, we inherit the implementations in LaSer [15]. As these code-related language properties are fixed in the selection box, users do not have to specify any file containing the description of these properties. I-LaSer will construct the corresponding description via transducer based on user's selection in the backend using **Algorithm 4.5**.

Our interface inherited functions from LaSer, where the files must be in Grail format [19]. Rules for describing automaton and transducer in Grail format can be found in [15, 36]. We also allow users to use files in Grail format in I-LaSer. In addition, we allow users to provide the files in FAdo format.

An automaton in FAdo format is described in a file as follows:

- `@DFA` or `@NFA` begins a new automata (and determines its type) and must be followed by the list of the final states separated by blanks.
- The origin state of the first transition is the start state.
- A line of the form `p σ q` describes a single transition, where `p` is the origin state of the transition, `q` is the destination state, and `σ` is the label of such transition.

States have to be represented by non-negative integers. Labels can consist of a sequence of alphanumerical symbols. However, in the transition, fields should be separated by a blank (e.g. transition `1 a2 3` means that `a2` is a label in the alphabet of the language accepted by the automaton described in such file). Note that the label can be described using any character sequence including `@epsilon` which is reserved for representing the empty word λ. An automaton can have multiple final states and only one start state.

Example: *Automaton accepting* $ab + bba$ *in FAdo format:*

```
@NFA 3
1 a 2
2 b 3
1 b 4
4 b 5
5 a 3
```

A transducer is described in a file in FAdo format as follows:

- `@Transducer` begins a new transducer and must be followed by the list of the final states separated by blanks.
- The origin state of the first transition is the start state.
- A line of the form `p x y q` describes a single transition, where `p` is the origin state of the transition, `q` is the destination state, and `x` is the input label and `y` is the output label.

Similarly to an automaton, states in transducer have to be described by non-negative integers. Both the input and output labels can be described using any character sequence including `@epsilon` which is reserved for representing the empty word λ. A transducer can have multiple final states and only one start state.

Example*: An example file that describes a transducer describing* $SID(1, \infty)$ *channel* *in FAdo format:*

```
@Transducer 0 1
0 a a 0
0 b b 0
0 a b 1
0 b a 1
0 a @epsilon 1
0 b @epsilon 1
0 @epsilon a 1
0 @epsilon b 1
1 a a 1
1 b b 1
```

For the sake of convenience for the users who may be familiar with one type of file format, we provide a small Python script to carry out the translation from Grail format to FAdo format and vice versa. Therefore in I-LaSer, users can use files either in Grail format or FAdo format.

When using I-Laser, if users choose “Fixed” property to decide whether a language L satisfies some code related language properties, for example the prefix code property, users need to upload a file describing an automaton accepting L and click the Submit button. Afterwards, our application will compute the answer. If the language L satisfies the prefix code property, user will simply get a confirmation of the fact, e.g. **“Yes, the language satisfies the prefix property”**. Otherwise, they will get the negative fact, e.g. **“No, the language does not satisfy the prefix property”** followed by the counterexample. For other properties besides “Fixed”, after users submit a file describing an automaton accepting L and a file of a transducer describing the language property and click the Submit button, users will simply get a confirmation of that fact, such as “**Yes, the language satisfies the property**”, or “**No, the language does not satisfy for the property**” followed by the counterexample.

Chapter 7
Conclusion and future work

7.1 Conclusion

In this thesis, we introduce our algorithmic tools and decision methods to answer the satisfaction question for regular languages and language properties. These questions include: decide whether a given language L satisfies a given language property described by an input-preserving transducer T, which is equivalent to the question of whether L is error-detecting for a channel realized by T; decide whether a given language is error-correcting for a channel; decide whether a regular language is a code. These algorithmic tools involve automaton tools and transducer tools. In addition, we provide a new method to estimate the edit distance of a regular language using our tools. Finally, we implement our algorithms and methods through establishing a web interface, which strengthens the capabilities of an existing web interface.

In Chapter 4, we present algorithms to construct machine products and to decide functionality of three types of transducer: restricted sequential transducer, standard form transducer, and real-time transducer. We use the functionality decision algorithms to decide the above satisfaction questions following the interpretation of two propositions discussed in Chapter 3 and the algorithm in [21]. We introduce our algorithms to provide counterexamples in cases where answers of these satisfaction questions are negative. Moreover, we present an algorithm to translate a transducer in standard form into an equivalent real-time transducer based on the mathematical method in [42]. In Chapter 5, we discuss our method to estimate the edit distance of a regular language using the error-correction property. We provide an algorithm to construct an input-preserving transducer to realize SID channels. We also provide two performance tests, where the result shows that our new method is much faster.

Our final research goal is to implement all of the algorithms involved and make them accessible via a web application. We use the Python programming language and FAdo libraries to implement our methods. Furthermore, we integrate the existing web interface LaSer based on C++ and BOOST libraries to our new web application, and establish a new web interface. Improvements are implemented when integrating two web interfaces together. In Chapter 6, we include a brief tutorial on how to access and use our system, as well as some examples of input files.

7.2 Future work

Combined with the existing methods to decide whether given languages satisfy properties described by input-altering transducers, our research only answer the general satisfaction questions for regular languages and independence properties: given a language L and a language property P, does L satisfy P?

In our opinion, there are two main directions for future work: the maximality question and the construction question. The maximality question is about deciding whether a given language is maximal with respect to a given language property. Note that in general, maximality question is PSPACE-hard. The construction question is to generate a language that satisfies some given properties. The construction question can be developed for finite languages, or if possible for infinite languages, where the output would be an automaton recognizing the generated language.

For any software application, there must be places for improvement. Our web interface also needs to be improved in the future. Besides implementing the above maximality and construction question and adding these functions in our new web interface, we hope our web interface would be more integrated to use one particular programming language, either C++ or Python. In addition, time complexity and space complexity are other important aspects for future improvement.

In conclusion, we hope that the technical ideas and methods in our research will shed some light on other broader research directions for us to understand the world of automaton and information theory comprehensively.

Bibliography

1. ACEnet, August 2012. http://www.ace-net.ca/wiki/Mahone.
2. A. Almeida, M. Almeida, J. Alves, N. Moreira and R. Reis. *FAdo and GUItar: Tools for automata manipulation and visualization*. In: CIAA 2009. LNCS 5642, Springer-Verlag, Berlin, 65 - 74, 2009.
3. M. P. Béal, O. Carton, C. Prieur and J. Sakarovitch. *Squaring transducers: an efficient procedure for deciding functionality and sequentiality*. Theoretical Computer Science, 292(1): 45 - 63, 2003.
4. J. Berstel. *Transductions and Context-free Languages*, Teubner, Leipzig, 1979.
5. J. Berstel and D. Perrin. *Theory of codes*. Academic Press Orlando, 1985.
6. Boost C++ libraries, August 2012. http://www.boost.org/.
7. A. Carpi. *Overlap-free words and finite automata*. Theoretical Computer Science, 115: 243 - 260, 1993.
8. N. Chomsky. *Three models for the description of language*. IRE Transactions on Information Theory, 2:113 - 124, 1956.
9. B. Courcelle, D. Niwinski, and A. Podelski. *A Geometrical View of the Determinization and Minimization of Finite State Automata*. Math Systems Theory, 24: 117 - 146, 1991.
10. A. Daka. *Computing Error-Detecting Capabilities of Regular Languages*. MSc thesis, Department of Mathematical and Computing Science, Saint Mary's University, Canada, 2011.
11. A. Daka and S. Konstantinidis. *Refinement and Implementation of Algorithmic*

Tools for Deciding the Error-detection Property. Technical Report 002, Department of Mathematics and Computing Science, Saint Mary's University, Canada, 2011.

12. Django web framework, August 2012. https://www.djangoproject.com/.
13. M. Domaratzki. *Trajectory-based codes*. Acta Information, 40(6-7): 491 - 527, 2004.
14. M. Domaratzki and K. Salomaa. *Codes defined by multiple sets of trajectories*. Theoretical Computer Science, 366(3): 182 - 193, 2006.
15. K. Dudzinski. *A system for describing and deciding properties of regular languages using input altering transducers*. MSc thesis, Department of Mathematical and Computing Science, Saint Mary's University, Canada, 2011.
16. K. Dudzinski, and S. Konstantinidis. *Formal descriptions of code properties: decidability, complexity, implementation*. International Journal of Foundations of Computer Science, 23(1): 67 - 85, 2012.
17. S. Eilenberg, *Automata, Languages and Machines*, Vol. A, Academic Press, New York, 1974.
18. FAdo, August 2012, http://fado.dcc.fc.up.pt/.
19. Grail++, August 2012. http://www.csd.uwo.ca/Research/grail/.
20. Y. S. Han and D. 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.
21. 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.

22. J. E. Hopcroft, R. Motwani, and J. Ullman. *Introduction to automata theory, languages, and computation (3rd edition)*. Addison-Wesley, MIT, 2006.
23. E.M. Gurari and O.H. Ibarra. *A Note on Finite-Valued and Finitely Ambiguous Transducers*. Mathematical Systems Theory, 16: 61 - 66, 1983.
24. I-LaSer, August 2012. http://laser.cs.smu.ca/independence/.
25. H. Jürgensen. *Syntactic monoids of codes*. Acta Cybernetica, 14(1): 117 - 134, 1999
26. H. Jürgensen, M. Katsura, and S. Konstantinidis. *Maximal solid codes*. Journal of Automata, Languages and Combinatorics, 6(1): 25 - 50, 2001.
27. H. Jürgensen and S. Konstantinidis, *Codes*, Handbook of formal languages, vol. 1, Springer-Verlag, New York, 1997
28. H. Jürgensen and S. S. Yu. *Solid codes*. Journal of Information Processing and Cybernetics, 26(10): 563 - 574,1990.
29. S. Konstantinidis. *Computing the edit distance of a regular language*. Information and Computation, 205: 1307 - 1316, 2007.
30. S. Konstantinidis. *An algebra of discrete channels that involve combinations of three basic error types*, Information and Computation, 167 (2): 120 - 131, 2001.
31. S.Konstantinidis. *Classical-theoretical foundations of computing (a concise textbook)*. Department of Mathematics and Computing Science, Saint Mary's University, Canada, 2010

32. S. Konstantinidis. *Structural analysis of error-correcting codes for discrete channels that involve combinations of three basic error types*, IEEE Transactions on Information Theory, 45 (1): 60 - 77, 1999.
33. S. Konstantinidis. *Transducers and the Properties of Error-Detection, Error-Correction and Finite-Delay Decodability*. Journal of Universal Computer Science, 8: 278 - 291, 2002.
34. S. Konstantinidis and A. O'Hearn. *Error-Detecting Properties of Languages*. Theoretical Computer Science, 276: 355 - 375, 2002.
35. S. Konstantinidis and P. V. Silva. *Maximal error-detecting capabilities of a formal language*. Journal of Automata, Languages and Combinatorics, 13(1): 55 - 71, 2008.
36. LaSer, August 2012. http://laser.cs.smu.ca/transducer/.
37. M. V. Lawson, *Finite automata*, CRC Press, 2003.
38. V.I. Levenshtein. *Binary codes capable of correcting deletions, insertions, and reversals*. Soviet Physics Doklady, 10(8): 707 - 710, 1966.
39. Al. A. Markov. *Nonrecurrent coding*. Problemy Kibernetiki, 8: 169 - 186, 1961.
40. C. Allauzen and M. Mohri. *Efficient algorithms for testing the twins property*. Journal of Automata, Languages and Combinatorics, 8(2): 117 - 144, 2003.
41. G. Paun and A. Salomaa. *Thin and slender languages*. Discrete Applied Mathematics, 61(3): 257 - 270, 1995.
42. J. Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, Berlin, 2009.

43. D. Sankoff, J. Kruskal. *Time Warps, String Edits, and Macromolecules: The theory and practice of sequence comparison*. CSLI Publications, 1999.

44. M.P. Schützenberger. *Sur les Relations Rationnelles Entre Monoides Libres*. Theoretical Computer Science, 3: 243 - 259, 1976.

45. H. Shyr and G. Thierrin. *Codes and binary relations*. SÉMINAIRE D'ALGÈBRE PAUL DUBREIL PARIS 1975 - 1976 (29ÈME ANNÉE). Lecture Notes in Mathematics, 586: 180 - 188, 1977.

46. H. Shyr and S. Yu. *Solid codes and disjunctive domains*. Semigroup Forum, 41: 23 - 37, 1990.

47. S. Yu, *Regular languages*, Handbook of Formal Languages, Vol. 1(Chapter 2): 41 - 110, Springer Verlag, 1997.