Computing Error-Detecting Capabilities of Regular Languages

By

Alifasi Daka

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 2011, Halifax, Nova Scotia

Approved : Dr. Stavros Konstantinidis
Supervisor

Approved : Dr. Jim Diamond
External Examiner

Approved : Dr. Sageev Oore
Supervisory Committee Member

Approved : Dr. Hai Wang
Supervisory Committee Member

Approved : Dr. Paul Muir
Graduate Studies Representative

Date : December 14, 2011

Your file Votre référence
ISBN: 978-0-494-82906-6
Our file Notre référence
ISBN: 978-0-494-82906-6

Abstract

Computing Error-Detecting Capabilities of Regular Languages

By

Alifasi Daka

Abstract: The property of error-detection ensures that when words of a language are communicated over a noisy channel, no word of the language can be transformed into another word of the language. The newer concept of maximal error-detecting capability seeks to find the noisiest channel a language is error-detecting for. We investigate, refine and implement algorithms related to error-detection for regular languages (words accepted by a finite automaton). As a result, we present some new ways of modeling channels using sequential machines. In addition, we adapt an existing transducer functionality algorithm to work with sequential machines, for the purpose of deciding the error-detection property. In the process we introduce, among others, the new concept of pseudo-sequential machines, and provide methods for converting them to sequential machines and vice-versa. We apply our new tools to the higher concept, and the result is a way of computing the maximal error-detecting capabilities of a regular language. We have implemented the new algorithms we developed, as well as some relevant existing theoretical algorithms. Finally, we have created a web interface for interacting with the tools we developed, and have made the source code of our implementation available to the research community.

December 14, 2011

Acknowledgements

I would like to thank my supervisor, Dr. Stavros Konstantinidis for his guidance, many suggestions, constant support and for taking me under his wings during this research. My thanks also go to my thesis committee members, Dr. Sageev Oore and Dr. Hai Wang, for all the support rendered to me during this research.

Many thanks go to my wife, Doria for being supportive and encouraging throughout this research work. Thank you sweetness. I would also like to thank my family and friends who have always believed in me and always encouraged me to pursue excellence. Special thanks go to my late mum who, despite having been denied an education, strongly believed in the benefits education brings to society, so much that she would cry as she urged her children to go to school.

I would also like to thank all my friends and colleagues in the Master of Applied Science program who made the program so much fun. Thanks for giving me a taste of different parts of the world and Canada without having to go there.

Many thanks go to the faculty of graduate studies for the graduate scholarship that greatly eased up life during this research. Thanks also for the many free pizzas!

Halifax, Nova Scotia — Alifasi Daka
December 2011

To Doria, Okondedwa and Odala.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Noisy Channels and Error-Detection

We consider a communication language, which is a set of words used to encode data. These words are communicated over a noisy channel. A physical channel is any transmission or storage medium which has a potential of deleting, substituting and inserting some symbols in the words passing through it [31]. We model a channel as a set of pairs of words (w_1, w_2) such that w_1 is the input message and w_2 is the received/output message. The received message w_2 can be different from w_1 when channel errors occur. In such a case, we can say that (w_1, w_2) is an error situation of the channel. The property of error-detection ensures that errors that alter an input message, as it passes through the channel, are detected and possibly corrected. To provide error-detection on channels, codes are designed and used. A code, as shown in [16], is a set of words that can be used to encode data, and any concatenation of

such words is uniquely decodable.

Codes are sometimes given as regular languages. A regular language is a set of words accepted by a finite state automaton [13]. These languages are usually designed to work for a particular channel with a known set of error situations. Although it can be decided whether a language is error detecting for a given channel, currently there are very few formal attempts [26] to compute the maximum amount of errors a language is capable of detecting under different error situations other than what it was designed for. This thesis looks at the set of algorithmic tools for deciding error-detection, as well as the new concept of computing maximal error-detecting capabilities of a communication language, that is, a maximal channel, from a certain class of channels, such that the language is error-detecting for that channel. We use the term *error model* for the class of channels under consideration.

1.2 Objectives and Scope

The objective of this research is to investigate existing and develop new algorithms related to error-detection, including maximal error-detecting capabilities of a communication language, in the direction introduced in [26]. We will work with regular languages, that is, languages accepted by finite automata. Our work will consider certain sets of combinatorial channels, such as SID channels, which specify the error type τ and number m of errors that the channel can introduce in a given length of

symbols being transmitted over it. For example, the channel $\tau(m, l)$ specifies that only up to m errors of type τ can occur in any l consecutive symbols passing through the channel.

The overall scope of this thesis research work is outlined as follows:

- Investigate, refine and implement relevant existing algorithms.
- Build on and add to the introductory work started in [26].
- Introduce and describe the modeling of channels using sequential machines.
- Adapt the transducer functionality and related algorithms to work with sequential machines, for the purpose of deciding the error-detection property.
- Make a web interface for interacting with the algorithms we implement.

We believe that the set of concrete algorithmic tools and source code developed in this thesis brings us a step closer to the realization of practical software for evaluating the error-detecting capabilities of languages.

1.3 Structure of Thesis

This thesis has seven chapters. In Chapter 2 we give some general definitions, notation and background information. This is followed by a literature review on error-detection and the use of automata in error-detection presented in Chapter 3. In

Chapter 4, we present some new methods for modeling channels using sequential machines. Chapter 5 contains some new and existing algorithms we have used for computing error-detecting capabilities. For example, it documents the new concept of pseudo-sequential machines. It also contains our adaptation of the transducer functionality algorithm for regular languages, in order to decide the error-detection property for a regular language. We also show how these algorithms can be used to compute maximal error-detecting capabilities of a regular language. A substantial part of this research has been the implementation and testing of new and relevant existing algorithms. Chapter 6 contains implementation details for this research work and provides some details on how to interact with the system we have implemented. We conclude the thesis in Chapter 7, as well as provide some remarks on possible future work.

Chapter 2

Definitions, Notation and Background Information

In this chapter we provide some basic definitions, notation and relevant background information.

2.1 Basic Definitions and Notation

We consider an *alphabet* to be a finite non-empty set of symbols. The size or cardinality of a set A is denoted by $|A|$. A word w over an alphabet A is a concatenation of some symbols from A. We use the notation $|w|$ for the length of the word w. For example, $|aba| = 3$. The empty word λ is the word with zero symbols. Hence, $|\lambda| = 0$. In a word w of the form xyz, the word y is called a factor of w. We use A^* to denote the set of all words over the alphabet A, including λ. On the other hand, A^+ denotes the set of all words over A excluding the empty word; that is $A^+ = A^* - \{\lambda\}$. A

language L over A is a subset of A^*. In notation we write this as $L \subseteq A^*$.

A *binary relation* $\mathbf{R}$ over two alphabets A and B, denoted by $\mathbf{R} : A^* \Rightarrow B^*$, is a subset of $A^* \times B^*$. The relation is functional, or single valued, if for each pair $(a, b) \in \mathbf{R}$ and $(a, b') \in \mathbf{R}$, it is the case that $b = b'$. The inverse $\mathbf{R}^{-1}$ of $\mathbf{R}$ is the relation $\{(b, a) \mid (a, b) \in \mathbf{R}\}$.

2.2 Theory of Automata

2.2.1 Automata

An automaton with empty transitions (λ-NFA) is a quintuple $M = (Q, A, q_0, E, F)$ where Q is a finite and non-empty set of *states*, A is a finite set called the *input alphabet*, $q_0 \in Q$ is the *start state*, $F \subseteq Q$ is the set of *final states* and E is the set of *transitions*. Each transition in E is a tuple of the form (q_1, a, q_2), also written as $q_1 a q_2$, such that q_1, $q_2 \in Q$ and $a \in A \bigcup \{\lambda\}$. A *computation* of M is a word ω of the form $q_0 a_1 q_1 \ldots a_n q_n$ such that each factor $q_{i-1} a_i q_i$ of ω is in E. This computation is called *loop-free* if all q_i's appearing in it are distinct. The computation ω is an *accepting computation* if q_0 is the start state and $q_n \in F$. The automaton M reads in a word w_n containing n symbols from A and makes appropriate transitions as defined in E. The word w_n is *accepted* by M, if after reading w_n, the automaton ends up in a state $q_n \in F$. The set of all words accepted by M is called the language of M, denoted as $L(M)$. For example, the automaton in Figure 2.1 accepts the string *bbab*, but not

bb. An accepting computation for $bbab$ in M is $1b1b1a2b3$. This computation is not loop-free.

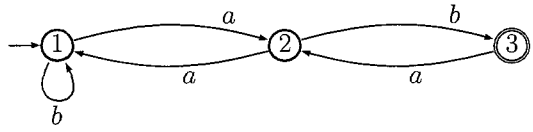

Figure 2.1: An example of an automaton

A *regular language* is a set of words that are accepted by some automaton. See [42, 13, 29] for more information on automata and regular languages.

The automaton M is said to be *accessible* if for every state $q \in Q$ there is a path from the start state q_0 to q. We say that M is *coaccessible* if for every state $q_i \in Q$ of M, there is a path from q_i to any $q_n \in F$. An automaton is *trim* if it is both accessible and coaccessible. M is a *non-deterministic finite automaton* (NFA) if for every transition q_1aq_2, a is not empty, that is, $a \in A$. It is called a *deterministic finite automaton* (DFA), if for every pair of transitions q_1aq_2 and q_1aq_3 it is the case that $q_2 = q_3$ and $a \in A$. The *diameter* of a deterministic finite automaton M, denoted as $diam(M)$, is the number of states in the longest loop-free accepting computation in M. For example, the diameter of the automaton in Figure 2.1 is three. We will use the terms *finite state machine* and *finite machine* to mean automaton. Furthermore, our research focuses on deterministic finite automata (DFA). Hence, unless explicitly specified otherwise, all the automata used for the regular languages in our research

are assumed to be deterministic.

2.2.2 Transducers and Sequential Machines

A finite state *transducer* (automaton with output) T is a sextuple (Q, A, B, q_0, E, F) where B is the *output alphabet*, and the definitions of Q, A, q_0 and F are the same as in the definition of an automaton with empty transitions. The only difference is that the transitions in E are of the form $q_1 a/bq_2$ with $q_1, q_2 \in Q$, $a \in A^*$ and $b \in B^*$. A computation of T is a word μ of the form $q_0 a_1/b_1 q_1 \ldots a_n/b_n q_n$ such that each factor $q_{i-1} a_i/b_i q_i$ of μ is in E. In this case, the word $a_1 \ldots a_n$ is the *input* part and the word $b_1 \ldots b_n$ is the *output* part of μ. The computation μ is an accepting computation if q_0 is the start state and $q_n \in F$. The transducer T realizes the relation $\mathbf{R}(T) : A^* \Rightarrow B^*$, such that a pair of words (w_1, w_2) is in $\mathbf{R}(T)$ if and only if there is an accepting computation of T that takes w_1 as input, gives w_2 as output and ends up in one of the final states of T. The definitions of accessible, coaccessible and trim are defined analogously to the definitions in an automaton. The relation recognized by T, also called a *rational relation*, is the set of all pairs of words $(w_1, w_2) \in \mathbf{R}(T)$. The transducer T is *functional* or *single-valued* if the relation $\mathbf{R}(T)$ is functional. The transducer T in Figure 2.2 has $1b/b1a/b2b/b2$ as an *accepting path*. Therefore, $(bab, bbb) \in \mathbf{R}(T)$. We next discuss some important types of transducers.

An *arbitrary* transducer reads in a word $x \in A^*$ and outputs a word $y \in B^*$.

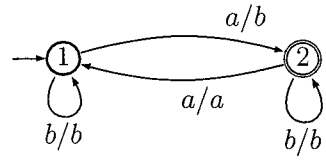

Figure 2.2: An example of a transducer T

A transducer is in *standard form* if for every transition $q_1 a/b q_2$ it is the case that $a \in A \cup \{\lambda\}$ and $b \in B \cup \{\lambda\}$. A *real-time* transducer reads in a symbol at a time and outputs a set of words for it. It has transitions of type $q_1 a/y q_2$, where $a \in A$ and y is a regular expression (a set of words $\in B^*$). A transducer is a *generalized sequential machine* (GSM) if it reads in one symbol per transition and outputs one word for it. Its transitions are of the form a/y where $a \in A$ and $y \in B^*$. Going forward, we use the term *sequential machine* to mean GSM. It should be noted that a sequential machine is a special type of a real-time transducer, whose set of output words per transition has one word. A transducer is called a *restricted sequential machine* if it can read and output strictly one symbol on each transition.

As we shall see later in Chapter 5, we need to introduce a special type of transducer, which we call *pseudo-sequential*. Specifically, let T be any transducer. A transition of T is called a *λ-input* transition if it is of the form $(p, \lambda/z, q)$, where λ is the empty word and z is the output word $\in B^*$. The transducer T is called *pseudo-sequential* if it is trim and in standard form and the *λ-input* transitions satisfy the following two conditions:

1. The start and final states have no outgoing λ-*input* transitions

2. If a state has an outgoing λ-*input* transition then that state has no other outgoing transitions.

Given a transducer T and a DFA M, we use $T \downarrow M$ to denote the transducer that results from intersecting the input part of T with M. More specifically, (u, v) is recognized by $T \downarrow M$ if (u, v) is recognized by T and u is accepted by M. Similarly, we use $T \uparrow M$ to denote the transducer that results from intersecting the output part of T with M; that is, (u, v) is recognized by $T \uparrow M$ if (u, v) is recognized by T and v is accepted by M. We refer the reader to Chapter 5 for details of the construction of these machines.

2.2.3 Size of Finite State Machines

Automata and transducers are given as input to various algorithms for error-detection. We use the term machine to refer to either of these two objects. The *size* of the machine is the number of states plus the sum of the sizes of the transitions in the machine. The size of a transition pxq is 1 plus the length of x. For example, the size of the transitions $1a/b2$ in Figure 2.2 is $1 + 3 = 4$, and the size of the machine in the figure is $2 + 4 * (1 + 3) = 18$. The size of a machine M is denoted as $|M|$. With this definition we can express statements about the complexity of an algorithm involving machines. Thus, if an algorithm takes as input a transducer T and we say

the algorithm is quadratic, we mean that the time complexity of the algorithm is $O(|T|^2)$.

2.3 Word Difference Measures

When a word w from a certain language L is communicated via a noisy channel, the presence of errors can change w to some word w' that is not in L. Correcting w' is the process of finding a word $z \in L$ that is as close to w' as possible—the aim being that $z = w$. This real-world scenario motivates the definition of distance between two words, which quantifies how different two words are from each other. Next we discuss two known distance measures.

2.3.1 Hamming Distance

The Hamming distance H is a measure of how different words of equal length are. Between two words, it is the number of positions at which the two words differ or, equivalently, the number of substitutions required to change one word to the other. For example, *bread* and *brood* differ in two positions. Hence, $H(bread, brood) = 2$.

With this in mind, a word is different from another word of equal length if and only if the Hamming distance of the two words is greater than zero [9, 20]. That is, if $H(w_1, w_2) > 0$ then $w_1 \neq w_2$.

Definition 2.3.1. Given a language (set of words) L, the Hamming distance of L is

the smallest distance between any two distinct words of L.

$$H(L) = \min\{H(w_i, w_j) \mid w_i \text{ and } w_j \in L, \text{ and } w_i \neq w_j\}$$

Remark 2.3.1. The Hamming distance based word difference measure, in its basic form, assumes that the words of the language are of equal length, such as in block codes. Hence, it can only work when only substitutions are permitted for changing words. On the other hand, a deletion would reduce the length of the word and would make our Hamming distance scheme fail. This problem has been solved by introducing variations to the Hamming distance. For example, a deletion can be seen as a substitution with an unknown symbol. In this case, the word length is considered to be the same. When the words differ in length, the shorter word is padded with some dummy symbols, at the end, to make the lengths of the two words equal [7]. After this the Hamming distance can be measured. The disadvantage of such a scheme is that very similar words of different lengths can end up having a very large Hamming distance between them, making them seem more different than they really are.

2.3.2 Levenshtein Distance

Perhaps the most comprehensive solution to the limitation of the Hamming distance word difference measure is the Levenshtein distance (Λ). Also referred to as the *edit distance*, it is a more sophisticated word difference measure and it covers substitutions, insertions and deletions [30]. It is defined as the minimum number of

possible substitutions, insertions or deletions that would transform one word into another. For example, the edit distance between *summer* and *stammer* is 2. That is $\Lambda(summer, stammer) = 2$. In this example, using all the possible error combinations, at least two errors are needed to change *summer* to *stammer* and vice-versa.

The shortest way to change *summer* to *stammer* is by

1. Inserting a t to get *stummer*

2. Substituting the u with an a to get *stammer*,

or do the same two steps in reverse. *There is no other way of doing it using one operation or error at a time.*

Definition 2.3.2. The edit distance Λ of a language L is the smallest edit distance between any two distinct words in L.

$$\Lambda(L) = \min\{\Lambda(w_i, w_j) \mid w_i \text{ and } w_j \in L, \text{ and } w_i \neq w_j\}$$

2.4 Channels

A *combinatorial channel* is a binary relation $\gamma \subset A^* \times A^*$ that is domain preserving; that is, the pair $(a, a) \in \gamma$ for all $a \in \{a' | (a', b) \in \gamma \text{ for some word } b\}$. Given a channel γ and a pair of words $(w_1, w_2) \in \gamma$, we say w_2 can be received as output from the channel on the input word w_1. If $w_1 \neq w_2$ we say that w_1 is received with errors. In this section we introduce some relevant channels.

2.4.1 SID Channels

Definition 2.4.1. An *SID* (Substitution,Insertion,Deletion) channel is a channel which is specified by the type τ and largest number m of errors the channel can introduce in a given length l, or any length ∞, of consecutive symbols passing through the channel; it is denoted as $\tau(m, l)$ or $\tau(m, \infty)$.

The error type τ can be one of the basic error types, substitution (σ), insertion(ι) and deletion(δ), or any combination of the three basic error types. We use $\odot$ as a connective when showing a combination of error types for a channel. For example, $\sigma \odot \delta$ means an error on the channel can be either a substitution or a deletion.

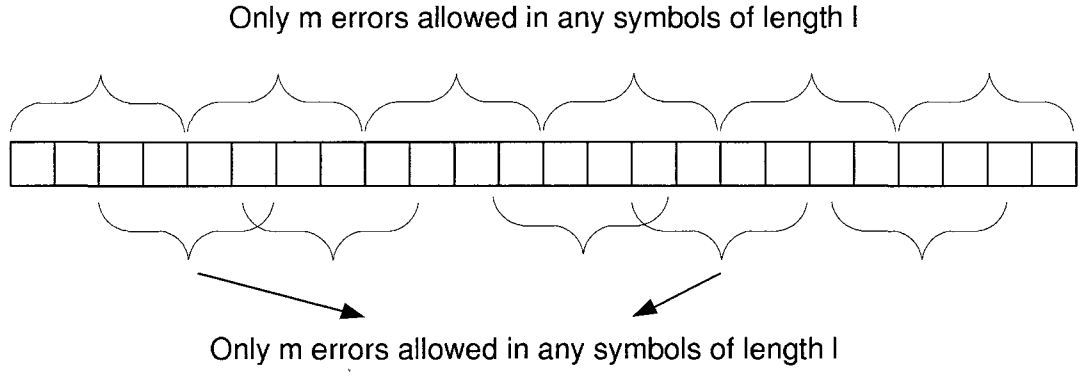

Figure 2.3: Error allowance on a combinatorial channel

For example, the pair $(aaaaaa, abaaba)$ is in the channel $\sigma(2, 4)$, while the pair $(aaaaaa, abbaba)$ is not, where σ represents the substitution error type, and, a and b are symbols in the alphabet.

Remark 2.4.1. Note that in combinatorial channels we do not provide an analysis or specification of the probability of certain errors occurring in the channel. Rather, it is assumed that any errors allowed by the channel have a high enough probability of occurring in a given length of symbols passing through the channel. These types of channels were first considered in [30] and then, more systematically, in [21, 22].

2.4.2 Homophonic Channels

Definition 2.4.2. A homophonic channel $\tau_h(m, l)$ is defined to be similar to the SID channel $\tau(m, l)$, with the only difference being that instead of allowing up to m errors in *any* block of symbols, it allows up to m errors in one fixed block of symbols regardless of what happened in the previous fixed block. Hence, $\tau_h(m, l)$ will allow no more than m errors in the first block of l symbols passing through the channel, then, regardless of what happened in the first block, it will allow a maximum of m errors in the next block of l symbols, and so on.

As can be seen in Figure 2.4, the block of l symbols that spans or overlaps two consecutive blocks of l symbols will not necessarily conform to the error restriction described above.

Remark 2.4.2. Homophonic channels can be useful in estimating the error-detecting capabilities of a regular language because they are less expensive to construct, compared to their corresponding SID channels, in terms of number of transitions. Refer

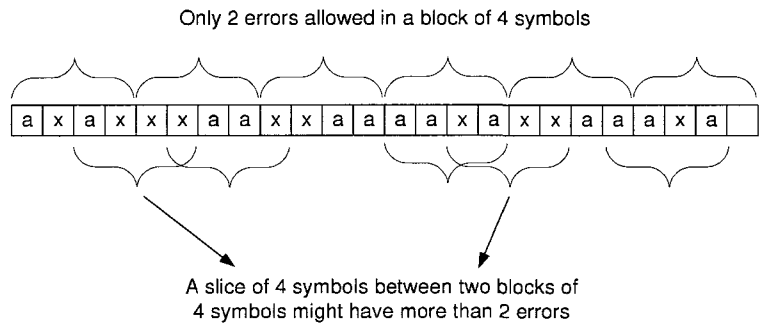

Figure 2.4: Error allowance on a sample homophonic channel $\tau_h(2,4)$

to [16, 2] for more information on homophonic channels.

2.5 Error-Detecting Languages

Definition 2.5.1. Given a language L, let $L_\lambda = L \cup \{\lambda\}$. L is *error-detecting* for a channel γ if γ cannot transform a word $w_1 \in L_\lambda$ to a different word $w_2 \in L_\lambda$; that is, if $(w_1, w_2) \in \gamma$ and $w_1, w_2 \in L_\lambda$ then $w_1 = w_2$.

Remark 2.5.1. This property enables us to tell if an error has occurred by simply checking if the received word from a channel belongs to the error-detecting language being used. The received word is in the language only if no error occurred. In this thesis we assume that the empty word λ is not significant in considering the error-detecting capabilities of the language, or that λ is already part of the language. In any case, we use the version of the above definition in which L is used in place of L_λ.

Chapter 3

Literature Review

The objectives of this thesis can be attained via a formal approach to defining channels, and a set of algorithmic tools for manipulating these objects. In this chapter we look at some relevant existing and useful concepts related to channels and error-detection.

3.1 Modeling Noisy Channels

Below we present some methods that have been previously used for modeling noisy channels.

3.1.1 Rational Channels

Transducers are automata that generate output for each input read. Their ability to read an input and produce an output is a feature that has been used to model noisy channels. The channels realized by transducers are called *rational channels* [24]. In

[17, 19, 24], the authors have used transducers to model channels. The disadvantage of rational channels is that they include some theoretical channels that can not be physically implemented or used [24]. Furthermore, the transducer realizing a channel can have a lot of states, even exponential in the number of errors the channel is generating [19, 24]. Nevertheless, transducers can be used to model most real world channels.

3.1.2 Edit Strings

Let A be an alphabet. An *edit string* is a concatenation of symbols called *edit operations* [17]. Edit operations are symbols of the type (a/a), (a/b), (a/λ), (λ/b) where a,$b \in A$. For each edit operation, the left symbol shows the input and the right symbol shows the output. An edit operation with a top symbol that differs from the bottom symbols is called an *error* [17, 19]. For example, (a/λ) is a deletion, (λ/a) is an insertion, (a/b) is a substitution and (a/a) is an error free edit operation. λ represents the empty word; that is $|\lambda| = 0$. An example of an edit string h is shown next.

$$h = (a/a)(a/b)(a/\lambda)(\lambda/b)$$

The input part of h is aaa and the output part is abb. Edit strings can be used to model a channel by representing the error situations that digital channels exhibit. They are also a convenient way of representing a channel where the three basic errors

can occur in varying combinations. The authors in [19, 20, 41] have used edit strings to define substitution, insertion and deletion (SID) channels. Moreover, for every SID channel there is a transducer realizing it [24]. In [2], the authors present an algorithm for encoding and decoding messages for multivalued codes, and this is conceptually similar to edit strings and SID channels because it is an attempt to deduce the source words of a channel's output words that can result from multiple possible error combinations (substitutions, insertions and deletions). In other words, the higher concept is the same as that of modeling a channel with edit strings.

3.2 Deciding Error-Detection Capabilities

In the previous chapter the error-detecting property of a language for a given channel was defined. In this section we look at some literature on how this property can be modelled, and how it can be decided for a given regular language.

3.2.1 Error-Detection via Transducer Functionality

In [23], the author shows that a language L is error-detecting for some channel γ if and only if the relationship $((\gamma \downarrow L_\lambda) \uparrow L_\lambda)$ is functional, where $L_\lambda = L \cup \{\lambda\}$, $\gamma \downarrow L_\lambda = \gamma \cap (L_\lambda \times \Sigma^*)$ and $\gamma \uparrow L_\lambda = \gamma \cap (\Sigma^* \times L_\lambda)$. With this method, one can use standard transducer constructions and decide error-detection by deciding whether a given transducer is functional. This problem is decidable [1, 8, 10]. See Section 5.7

for the details of the algorithms in [1], which is implemented in this work.

3.2.2 Language Inequations and Binary Operations

The authors in [18] present a way of defining language properties, including error-detection, using language inequations. This technique relies on modeling a channel using binary word operations. For instance, $\diamond$ can be used as a binary operation on two words or languages: $X \diamond Y$. In the paper, the authors show that, by varying $\diamond$ and Y, an inequation of the form $X \diamond Y \subseteq X^c$, where X^c is the complement of X, can be used in such a way that its solution, say L, for X indicates if L is error-detecting for a channel defined by $\diamond$ and Y.

3.3 Complexity Analyses and Implementation of Automata

Automaton theory comes with many possible operations on automata that are available in the theory of computation, such as catenation and Kleene star, which involve manipulating the states and transitions in the automata involved to achieve the desired results. The author in [44, 43] presents some state analyses for intersection, cross product and several other operations on regular languages. Others such as [11, 12, 14, 42] discuss state and time complexities for various operations on regular languages, rational languages, non-deterministic automata and formal languages in

general. These analyses are very important for solving problems using automata, and we have considered them in this research.

Other results such as deciding if a language is thin [38], deciding if a language is a code [32], shortest path algorithms [28, 33] and several algorithm design considerations [40] have also been used in this research.

In addition, software tools such as Grail [35, 36, 37] have been used to aid the study of these operations on automata. They are especially helpful when dealing with large automata that have a lot of states and transitions. We have used the Grail C++ library in our study and implementation of several algorithms.

3.4 Maximal Error-Detecting Capabilities

In [26] the authors introduce the concept of maximal error-detecting capabilities of formal languages. They discuss what possible research can be done in studying this new concept. Notably, they define an *error model* to be a set of channels. For example, the error model $\mathbb{C}_\tau[l] = \{\tau(m, l)$ for any m with $1 \leq m < l\}$ is a set of channels where l is fixed and the number m of errors allowed is varied within the range $1 \ldots (l-1)$. The error type, τ, can be a basic type or a combination of the three basic error types $\sigma, \iota. \delta$. Let $\mathbb{C}$ be an error model. A channel $\gamma \in \mathbb{C}$ is a $\mathbb{C}$-maximal error-detecting capability of a language L if

1. L is error-detecting for the channel γ.

2. L is not error-detecting for any channel $\gamma' \in \mathbb{C}$ which properly contains γ, that is, $\gamma' \supsetneq \gamma$

In other words, there is no channel in $\mathbb{C}$ larger than γ for which L is error-detecting. The definition of $\mathbb{C}$ might involve parameters such as τ, m and l. The authors of [26] consider several cases when the error type τ, the number of errors m and the segment length l are varied. Moreover, they present some preliminary results on computing maximal error-detecting capabilities for some cases involving substitutions, insertions and deletions, that is, $\tau = \sigma, \iota, \delta$. They also state that the channels in an error model can be realized using transducers, as indicated in [19]. As mentioned earlier, the number of states for each transducer realizing a SID channel could be exponential with respect to the number of errors m being permitted by the channel.

The next two subsections discuss how the maximal error-detection capability problem is related to computing the Hamming distance and edit distance of a language. Refer to Chapter 5 for further information on how we use these two computations.

3.4.1 The Hamming Distance of a Language

Given a channel $\sigma(m, \infty)$ as a transducer which will permit a maximum of m substitutions in any input message passing through it, one can decide in quadratic time if a given regular language L is error-detecting for the channel [26]. This is a special

case of the problem of computing maximal error-detecting capabilities of a given language, where the error model is the set of all channels $\sigma(m, \infty)$. In this case, there is a unique maximal channel, which is equal to $\sigma(M-1, \infty)$, where M is the Hamming distance of L.

Remark 3.4.1. In this research we have implemented the algorithm for computing the Hamming distance of a regular language presented in [20], by taking advantage of a modified approach to Dijkstra's shortest path algorithm [6]. See Chapter 5 for details.

3.4.2 The Edit Distance of a Language

Similarly, given an SID channel $\sigma \odot \iota \odot \delta(m, \infty)$ that will permit a maximum of m substitutions (σ), insertions (ι) and deletions (δ) anywhere in the input word, it can be decided if a regular language L is error-detecting for the channel using the results in [25, 19]. Note that the symbol $\odot$ is simply a connective which indicates that the channel can generate the three types of errors in any combination, up to the limit set by m. Again, this is a special case of the problem of computing maximal error-detecting capabilities of a given language, where the error model is the set of all channels $\sigma \odot \iota \odot \delta(m, \infty)$. In this case, there is a unique maximal channel.

Computing the edit distance involves constructing a transducer that realizes all the edit strings which model the effects of the channel on the language L and then

computing the distance of L in polynomial time [25]. Comparing this edit distance with the value of m reveals whether the language is error-detecting for the channel. Remember that a language can detect up to m SID errors if and only if its Levenshtein distance is greater than m [30]; that is to say, L is error-detecting for $\sigma \odot \iota \odot \delta(m, \infty)$ if and only if $\Lambda(L) > m$, where $\Lambda(L)$ is the Levenshtein distance of the language and m is the number of SID errors permitted on the channel in any length of consecutive symbols.

Remark 3.4.2. Using the edit distance to decide error-detecting capabilities of a regular language works well for the channels of the form $\tau(m, \infty)$. These are channels allowing a fixed maximum number m of errors in any input message passing through them.

Chapter 4

Algorithmic Tools I: Construction of Channels

As mentioned in the previous chapter, computing error-detecting capabilities of a regular language requires a means of modeling noisy channels. We showed a few existing models of these channels, which are rather at a conceptual level. In this chapter we considerably refined these models so as to bring them closer to the software implementation level. In this process, we also introduce a new way of modeling channels using sequential machines.

4.1 Sequential Machine vs General Transducer

In [17] the authors provide a construction of a channel using a general transducer. As shown in our literature review, transducer functionality is important for deciding error-detecting capabilities of regular languages. We note, once more, that deciding

functionality of a general transducer could be expensive in practice [24]. On the other hand, there are simpler algorithms provided in [1, 10] to decide transducer functionality if the given transducer is a sequential machine. We will investigate these in more detail in the next chapter. To take advantage of these algorithms, we introduce methods for constructing sequential machines that realize our channels of interest.

4.2 SID Channel Construction With Sequential Machines

Recall that SID channels are specified by the type τ and number m of errors that are permitted in any length l of symbols passing through the channel. In this chapter we always assume that $l \neq \infty$. Also, we use Σ to denote both the input and output alphabets. We agree that each channel error occurs at a certain symbol of the input word. This is clear for substitution and deletion errors. For example, $(abaaab, baaaa) \in \sigma \odot \delta(2, 6)$ because of a deletion at the first a of the input word, and substitution at the last b of the input word. When the SID channel permits insertions, namely τ contains ι, we assume that any insertion error occurs at the immediate left of some input symbol. For example, $(aaa, abaa) \in \iota(1, 3)$ because there is an insertion error at the second a of the input word. This convention is not unusual and in any case, one can define languages whose words end with a special marker \$,

so in effect any insertion to the left of \$ is an insertion at the end of the word.

4.2.1 Construction of the Channel $\iota(m, l)$

Proposition 4.2.1. *For every channel $\iota(m, l)$ there is effectively a sequential machine realizing it.*

Proof. The construction of the machine is shown below:

1. For each transition label x/y of the machine, $|x| = 1$ and $1 \leq |y| \leq (m + 1)$, where m is the number of errors permitted in any l input symbols.

2. Each state will remember what happened in the last $(l - 1)$ input symbols.

3. Each state will be represented by a string $q_c = i^{a_1}n \ldots i^{a_{(l-1)}}n$

   where i^{a_j} represents $0 \ldots m$ possible insertion errors that occurred on the j^{th} input symbol of the last $(l - 1)$ input symbols represented by the state and n represents the normal or error free input symbol. It can be observed that strings representing the different states of our machine will not necessarily be of the same length, but will have lengths in the range $(l - 1) \leq |q_c| \leq (l - 1 + m)$.

4. As all insertions occur to the left of the last read symbol, the states will always end in the following:

   n = normal error free transition

   in = one error occurred on the last read symbol

$i^{a_j}n = a_j$ errors occurred on the last read symbol

5. The start state will represent the situation where no errors occurred in the last $(l-1)$ symbols. Hence, the start state $q_0 = n^{l-1}$. For example, the start state for $\iota(2,4)$ is (nnn).

6. The total number of transitions originating from a state q will be

$$|\Sigma| + |\Sigma|^2 + \ldots + |\Sigma|^{(m-e_q+1)},$$

where e_q is the total number of insertion errors seen by the state q and Σ is the input alphabet. The transitions correspond to the possible events of the channel in response to reading the next input symbol; 0, or 1,..., or $(m-e_q)$ insertions occur to the left of that symbol.

For a given state p, a transition out of p would go to a state q such that the meaning of the states as described above is preserved. In particular, if p is of the form $i^k nx$ and the transition to q involves j insertion errors, then $q = xi^j n$. For example, the state $(nnin)$ represents the situation where 1 insertion error occurred on the third symbol of the last three input symbols. Transitions with 0 and 1 errors from $(nnin)$ would go to the states $(ninn)$ and $(ninin)$, respectively, and would be of the form $(nnin)a/a(ninn)$ and $(nnin)a/a'a(ninin)$. Figure 5.9 shows an example construction of $\iota(2,4)$, and Algorithm 4.2.1 shows

some pseudo code from our implementation to help visualize the above statements.

□

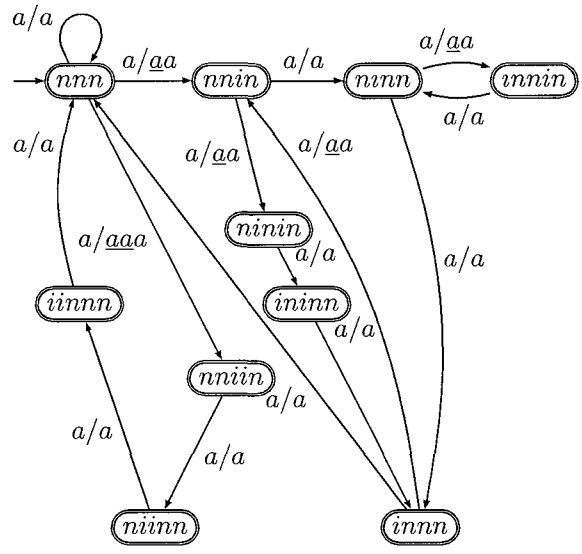

Figure 4.1: Sequential machine realizing $\iota(2,4)$

Remark 4.2.1. The figure above, and the rest of the figures in this section, only show the type of transitions and errors involved from one state to the other. This abstraction is necessary because it is infeasible to make a readable diagram showing all the transitions, even for the small values of m and l. For instance, there are actually $|\Sigma| + |\Sigma|^2 + \ldots + |\Sigma|^{(m-e_q+1)}$ transitions originating from each state q in the figure above.

Algorithm 4.2.1. Construction of $\iota(m, l)$ as a sequential machine

INPUT : m, l and the alphabet Σ

```
queue<state> UNSEEN, SEEN = empty;//
startState=new state(nnn...n); // state with no errors
UNSEEN.push(startState);
while (!UNSEEN.empty()){
    currentState=UNSEEN.front();
    numOfTransitionsFromState = m - errorsSeen(currentState)+ 1;
    for (j=0;j < numOfTransitionsFromState;j++) {
        state destination = createNewState(currentState,j errors);
        for each symbol in alphabet
            addTransitions(currentState,j errors, destination,alphabet);
        if (!SEEN.contains(destination)) //add state to unseen if not seen
            UNSEEN.push(destination);
    }//end for loop
    SEEN.push(currentState);
    UNSEEN.pop(currentState);
}//end while loop
makeFinalStates(SEEN);
```

OUTPUT: Sequential machine realizing $\iota(m, l)$

Implementation Notes 4.2.1. Our construction creates states on the fly to make the $|\Sigma| + |\Sigma|^2 + \ldots + |\Sigma|^{(m-e_q+1)}$ transitions from each state q starting with the start state. It uses two queues to keep track of which states have already been seen so that we do not process a state more than once.

Notes on the above algorithm pseudo code:

- The function `errorsSeen(state)` returns the number of errors the current state has seen. For example, `errorsSeen(nnin) = 1` and `errorsSeen(nniin) = 2`

- The function `createNewState(currentState,j )` creates a new state with j errors from the current state. It does this by reading the current state string from left to right and erasing the leftmost occurrence of n and every symbol to its left. It then appends $i^j = j$ occurrences of the symbol i and ends with the symbol n to come up with a new state. For example, `createNewState(nnn,2)` first knocks off the leftmost n to get nn then appends iin to get $nniin$. Similarly, `createNewState(nniin,0)` knocks off the leftmost occurrence of n to get $niin$ then appends the symbol n only, since there are 0 errors, to get $niinn$. This operation can further be visualized as a FIFO keeping track of the symbol n in the string representing a state. When the symbol n is inserted on the right of the FIFO, the leftmost n with everything on its left is removed.

- The function `addTransitions(source,j errors,destination,alphabet)` adds $|\Sigma|^{(m-e_i+1)}$ transitions to our sequential machine with the specified source state, number of errors and destination state.

More details on this implementation can be found in Chapter 6.

4.2.2 Construction of the Channel $\sigma(m, l)$

Proposition 4.2.2. *For every channel $\sigma(m, l)$ there is a sequential machine realizing it.*

Proof. The construction of the machine is very similar to that in the proof in Proposition 4.2.1, with a few differences as shown below:

1. For each transition label x/y of our machine, $|x| = 1$ and $|y| = 1$

2. Each state will remember what happened in the last $(l-1)$ input symbols.

3. Each state will be represented by a string $q_c = a_1 \ldots a_{l-1}$

   where $a_j = n$ if an error did not occur at position j, and $a_j = s$ if a substitution error occurred at position j of the last $(l-1)$ input symbols represented by the state. This time the strings representing the different states of our machine will be of equal length. Specifically, $|q_c| = (l-1)$ for all states in our machine. Unlike the case of insertions, only one substitution can occur per transition.

4. The start state will represent the situation where no errors occurred in the last $(l-1)$ symbols. Hence, the start state $q_0 = n^{l-1}$. For example, the start state for $\sigma(2, 4)$ is (nnn).

5. Each state q will have $|\Sigma| \times |\Sigma|^{e_q}$ transitions originating from it, where e_q is the number of errors allowed on any transition from state q. Precisely, $e_q = 0$ if the

errors seen by the state $= m$ and $e_q = 1$ otherwise. Σ is the input alphabet. The transitions correspond to the possible events of the channel in response to reading the next input symbol. That is, 0 or 1 substitution errors will occur on the last input symbol. Hence, if $e_q = 0$ (when errors seen $= m$) we only add one error-free transition from state q to a state p, for all $a \in \Sigma$. If, however, $e_q = 1$ (errors seen $< m$) then we add $|\Sigma|$ transitions from state q to a state p, for all $a \in \Sigma$.

For a given state p, a transition out of p would go to a state q such that the meaning of the states as described above is preserved. In particular, if p is of the form nx and the transition goes to q, then $q = xa_j$, where $a_j = s$ if a substitution occurred, or $a_j = n$ if no error occurred. For example, the state $(nnss)$ represents the fact that 2 substitution errors occurred on the last 4 inputs read by the channel. Transitions with 0 and 1 errors from $(nnss)$ would go to the states $(nssn)$ and $(nsss)$, respectively, and would be of the form $(nnss)a/a(nssn)$ and $(nnss)a/a'(nsss)$.

Figure 4.2 shows an incomplete construction of $\sigma(2,5)$ to help visualize the above construction.

□

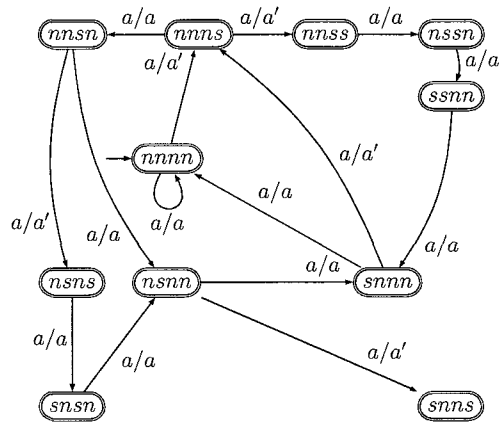

Figure 4.2: Partial depiction of sequential machine realizing $\sigma(2,5)$. In order to avoid clutter on the figure, the transition $(snns)a/a(nnsn)$ is not shown.

Implementation Notes 4.2.2. Our construction creates states on the fly to make the $|\Sigma|^{e_q+1}$ transitions from each state starting with the start state. It uses two queues to keep track of what states have already been seen so that we do not process a state more than once. The reader can see Chapter 6 for our implementation details.

4.2.3 Construction of the Channel $\delta(m,l)$

Proposition 4.2.3. *For every channel $\delta(m,l)$ there is a sequential machine realizing it.*

Proof. The construction of this machine will be exactly as in Proposition 4.2.2 for the channel $\sigma(m,l)$, with the only difference being that all substitutions will be with the empty symbol λ. We follow this approach because a deletion can be viewed as

a special substitution with an empty symbol. In addition, we replace the symbol s with the symbol d to represent a deletion whenever we show the machine states as strings.

Due to the similarity with the construction in Proposition 4.2.2 we have left out the construction rules. Nevertheless, Figure 4.3 shows a partial depiction of $\delta(2,5)$ to help visualize the construction. As in the the previous section, the transition $(dnnd)a/a(nndn)$ is not shown. □

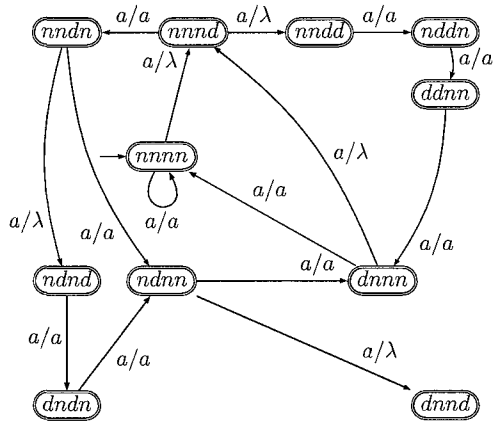

Figure 4.3: Sequential machine realizing $\delta(2,5)$

The next four channel constructions are derived from or use the last three constructions in some way, as will be made clear.

4.2.4 Construction of the Channel $\iota \odot \sigma(m, l)$

Proposition 4.2.4. *For every channel $\iota \odot \sigma(m, l)$ there is a sequential machine realizing it.*

Proof. The construction of the sequential machine is a combination of the construction rules in Proposition 4.2.1 for channel $\iota(m, l)$ and 4.2.2 for channel $\sigma(m, l)$, and is as follows:

1. Remember that the symbol $\odot$ is only used as a connective to show what types of errors are possible on the channel. Hence, $\iota \odot \sigma = \sigma \odot \iota$. This will be true in every case we use the symbol $\odot$.

2. Each state will remember what happened in the last $(l-1)$ input symbols of the channel.

3. Each state will be represented by a string $q_c = a_1 \ldots a_{l-1}$, where $a_j = n$ if no errors occurred, $a_j = i^k n$ if k insertion errors occurred, $a_j = s$ if only a substitution error occurred, and $a_j = i^k s$ if k insertion errors and a substitution error occurred on the j^{th} input symbol of the $(l-1)$ input symbols represented by the state.

4. The start state q_0 will represent the situation where no errors occurred in the last $(l-1)$ input symbols. Particularly, $q_0 = n^{(l-1)}$.

5. Only one substitution error and $0 \ldots m$ insertion errors can occur per transition.

6. The strings representing our states will not be of equal length but will have lengths in the range $(l-1) \leq |q_c| \leq (l-1+m)$.

7. Each state q containing e_q errors will have transitions as follows:

   (a) No substitution and $0, 1, \ldots, m - e_q$ insertions: The number of transitions from state q will be

   $$|\Sigma| \left(1 + |\Sigma| + |\Sigma|^2 + \ldots + |\Sigma|^{(m-e_q)}\right)$$

   (b) One substitution and $0, 1, \ldots, m - e_q - 1$ insertions: The number of transitions from state q will be

   $$|\Sigma| \left(|\Sigma| - 1\right) \left(1 + |\Sigma| + |\Sigma|^2 + \ldots + |\Sigma|^{(m-e_q-1)}\right)$$

   where e_q is the total number of errors seen by the state q.

For a given state p, a transition out of p would go to a state q such that the meaning of the states as described above is preserved. In particular, if p is of the form nx and the transition to q involves 1 or 0 substitution errors then $q = xa_j$, where $a_j = s$ if a substitution occurred or $a_j = n$ if no error occurred. On the other hand, if the transition from p to q involves j insertion errors, then $q = xi^j n$, where i^j represents the j insertion errors that occurred on the last input. For example, the state $(nnss)$ represents the situation where 2 substitution errors occurred on the last 4 inputs read by the channel. The state $(nnsin)$ represents

the situation where 1 substitution occurred on the third input and 1 insertion occurred on the fourth symbol of the last 4 input symbols represented by the state. The reader is referred to Figure 4.4 for a partial depiction of a sequential machine realizing $\iota \odot \sigma(2,4)$. The states $(nnis)$, $(nisn)$ and $(isnn)$, as well as a few transitions, have been left out for a clearer picture.

□

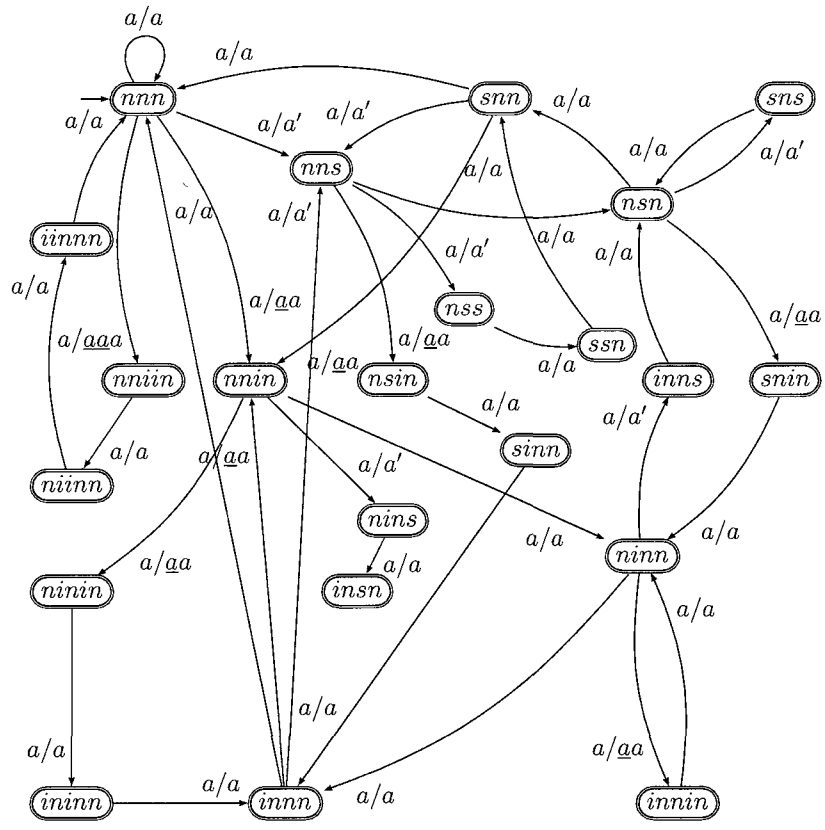

Figure 4.4: Partial depiction of sequential machine realizing $\iota \odot \sigma(2,4)$

4.2.5 Construction of the Channel $\iota \odot \delta(m, l)$

Proposition 4.2.5. *For every channel $\iota \odot \delta(m, l)$ there is a sequential machine realizing it.*

Proof. The construction of the machine is similar to the construction in Proposition 4.2.4 for the channel $\iota \odot \sigma(m, l)$, with the main difference being that all substitutions are with the empty symbol λ. We omit the rules for this construction and just provide a diagram showing a partial representation of a sequential machine realizing $\iota \odot \delta(2, 4)$. For a clearer figure, the states $(nnid)$, $(nidn)$ and $(idnn)$, as well as a few transitions, have been left out. □

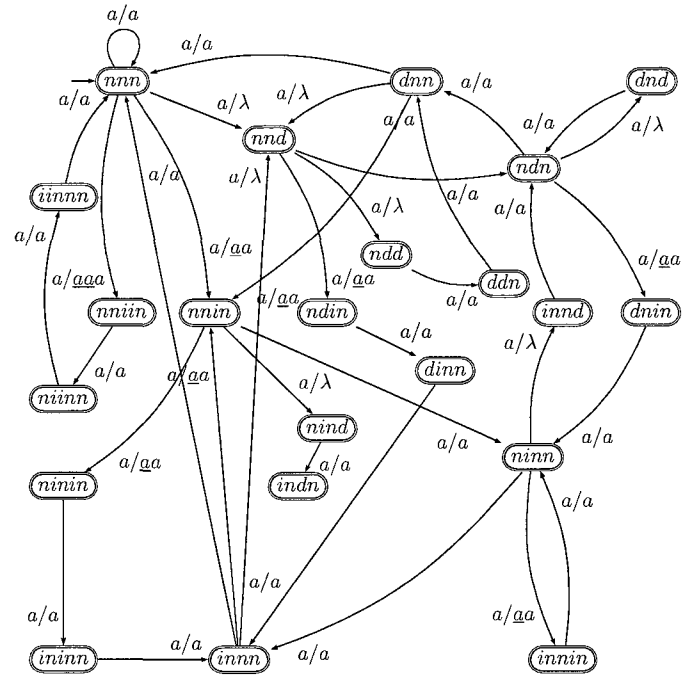

Figure 4.5: Part of sequential machine realizing $\iota \odot \delta(2,4)$

4.2.6 Construction of the Channel $\sigma \odot \delta(m,l)$

Proposition 4.2.6. *For every channel $\sigma \odot \delta(m,l)$ there is a sequential machine realizing it.*

Proof. There are two possible ways of constructing this machine. The first approach is to create states that track substitutions and deletions separately. This method would create unnecessary states since deletions can be viewed as special substitutions with the empty symbol. The method would only be useful if we strictly need to track deletions and substitutions separately.

Hence, we proceed with the second method which uses about half the states compared with the first method. This second approach simply treats a deletion as a special substitution and can be viewed as an extension of the channel $\sigma(m, l)$ in Proposition 4.2.2. The construction of the machine is as follows.

1. Construct the machine as in Proposition 4.2.2 with the following additional rules.

2. For each set of transitions from state p to q with the input a and involving a substitution error we add a transition of the kind $p(a/\lambda)q$, for all $a \in \Sigma$. This will add an extra $|\Sigma|$ transitions to each set of transitions involving a substitution error from a state p to a state q.

3. Hence, each state q will have $|\Sigma| \times (|\Sigma \cup \{\lambda\}|)^{e_q}$ transitions originating from it, where $e_q = 0$ if the errors (substitutions and deletions) seen by the state $= m$ and $e_q = 1$ if the errors seen by the state $< m$.

4. Recall that in Proposition 4.2.2 we used s to represent a substitution error. In this construction we replace the symbol s with the symbol e that represents either a substitution or a deletion error. Figure 4.6 shows an example of a sequential machine realizing $\sigma \odot \delta(2, 5)$. Note the similarity with Figure 4.2 with additional transitions representing deletions. For a clearer picture, we have left out the transition $(enne)a/a(nnen)$.

□

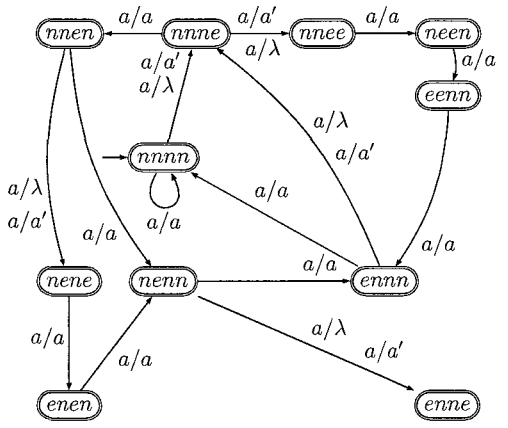

Figure 4.6: Sequential machine realizing $\sigma \odot \delta(2,5)$

4.2.7 Construction of the Channel $\sigma \odot \iota \odot \delta(m,l)$

Proposition 4.2.7. *For every channel $\sigma \odot \iota \odot \delta(m,l)$ there is a sequential machine realizing it.*

Proof. We construct the machine as follows:

1. We again treat deletions as special substitutions and utilize the rules in Proposition 4.2.6, for the channel $\sigma \odot \delta(m,l)$, to deal with substitutions and deletions.

2. We then construct our machine as a special type of the construction in Proposition 4.2.4, for the channel $\iota \odot \sigma(m,l)$, where a substitution can be with another symbol in the alphabet or with the empty symbol.

3. Therefore, we replace the symbol s in Proposition 4.2.4 with the symbol e to represent either a deletion or a substitution error.

4. For each transition from state p to q, involving a substitution error in Proposition 4.2.4, we add a transition of the kind $p(a/\lambda)q$ for all $a \in \Sigma$. This will add an extra $|\Sigma|$ transitions to each set of transitions involving a substitution error.

5. Each state q will have a number of transitions as follows:

   (a) No substitution/deletion and $0, 1, \ldots, m - e_q$ insertions: The number of transitions from state q will be

   $$|\Sigma| \left(1 + |\Sigma| + |\Sigma|^2 + \ldots + |\Sigma|^{(m-e_q)}\right)$$

   (b) One substitution and $0, 1, \ldots, m - e_q - 1$ insertions: The number of transitions from state q will be

   $$|\Sigma| \left(|\Sigma| - 1\right) \left(1 + |\Sigma| + |\Sigma|^2 + \ldots + |\Sigma|^{(m-e_q-1)}\right)$$

   (c) One deletion and $0, 1, \ldots, m - e_q - 1$ insertions: The number of transitions from state q will be

   $$|\Sigma| \left(1 + |\Sigma| + |\Sigma|^2 + \ldots + |\Sigma|^{(m-e_q-1)}\right)$$

   where e_q is the total number of errors seen by the state q.

Figure 4.7 shows a partial depiction of a sequential machine that realizes the channel $\sigma \odot \iota \odot \delta(2,4)$. Similar to the preceding sections, we have left out the transition $(inen)a/a(enn)$.

□

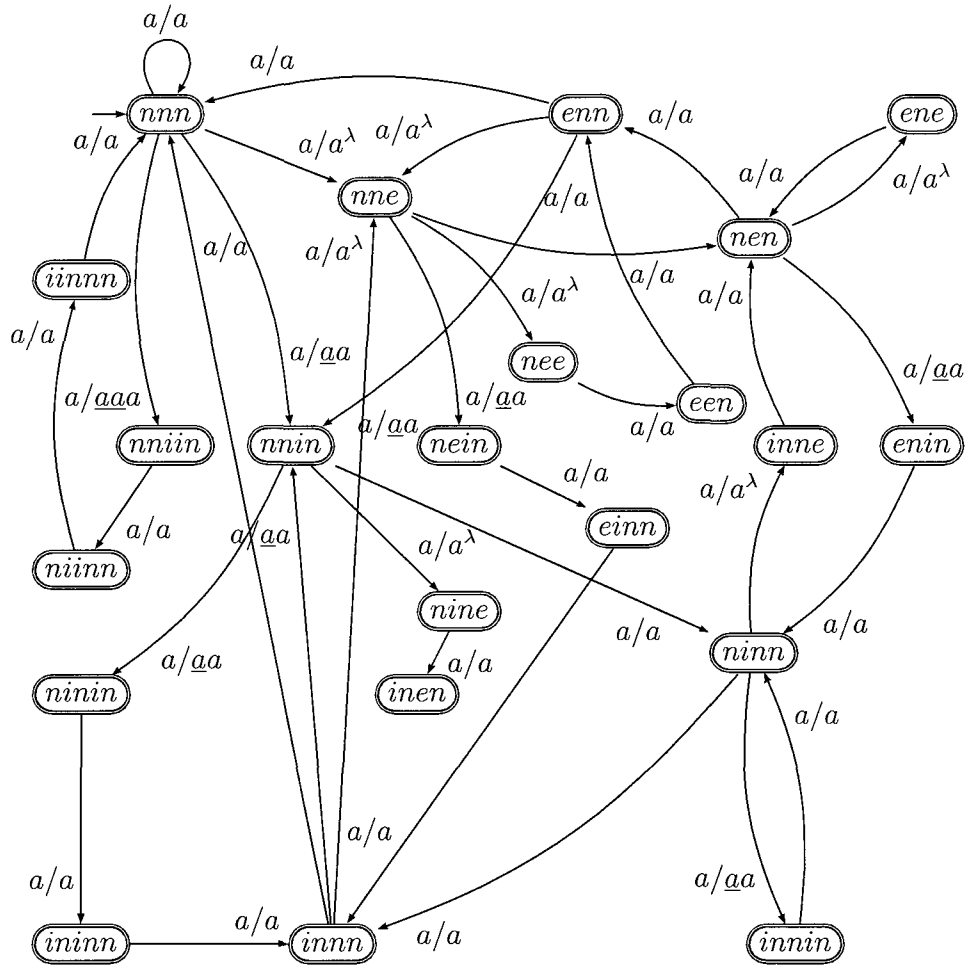

Figure 4.7: Partial depiction of sequential machine realizing $\sigma \odot \iota \odot \delta(2,4)$

Remark 4.2.2. In the figure above a label a/a^{λ} means that $a^{\lambda} \in \Sigma \cup \{\lambda\}$ with $a^{\lambda} \neq a$. The symbols $a, \underline{a} \in \Sigma$. The symbols of type $\underline{a^1 \ldots a^j}$ on a transition label from state p to state q represents all the transitions with j insertion errors. The actual number of symbol combinations for j error transitions from a state p to q would be $|\Sigma|^j$ for each input $a \in \Sigma$.

See Section 6.2 for implementation details of all the channels presented in this chapter.

4.3 Homophonic Channel Construction With Sequential Machines

The SID channels presented in the previous section would require transducer constructions with a large number of states, exponential in the number of errors allowed on a channel. This presents a computing time and complexity challenge, especially when we start increasing the values of m and l. Hence, we present methods for constructing homophonic channels using sequential machines. See Section 2.4.2 for the definition of homophonic channels. Proposition 4.3.1 shows how homophonic channels can be useful for determining error-detecting capabilities of a regular language.

Proposition 4.3.1. *If a language L is error-detecting for a homophonic channel $\tau_h(m, l)$ then it is also error-detecting for the SID channel $\tau(m, l)$.*

Proof. The proof is deduced from the definitions of the two types of channels in

question. A homophonic channel with the same m and l will allow more errors across the channel than its corresponding SID channel. More specifically, we have that $\tau(m,l) \subseteq \tau_h(m,l)$. Hence, if L is error-detecting for the case where more errors are allowed, it must be error-detecting for the case where less errors are allowed. □

Remark 4.3.1. Homophonic channels have the potential of helping in estimating the error-detecting capabilities of a regular language in lesser time because they require much smaller sequential machines for modeling the channel, compared to ordinary SID channels.

4.3.1 Construction of the Channel $\sigma_h(m,l)$

Proposition 4.3.2. *For every channel $\sigma_h(m,l)$ there is a sequential machine realizing it having a number of states $\leq 1 + (l-1) \times (m+1) - m$.*

Proof. The construction of the sequential machine is as follows:

1. For each transition label x/y of our machine, $|x| = 1$ and $|y| = 1$.

2. Since the requirement is to track what happens in blocks of l symbols, the machine will be reset to the start state after reading l symbols. Obviously, no more than m errors will be allowed in the l symbols read in.

3. Each state will be represented by a pair of integers (i,j), where i is the number of symbols read so far and j is the number of errors seen so far. For example,

the state $(2, 1)$ represents the situation where 2 symbols have been read in and 1 substitution has occurred.

4. The start state, indicated by $(0, 0)$, will represent the situation where no symbols have been read in and hence, no errors have occurred in the next block of l symbols.

5. Each state $q = (i, j)$ will have $|\Sigma| \times |\Sigma|^{e_q}$ transitions originating from it, where e_q is the number of errors allowed on any transition from the state. Precisely, $e_q = 0$ if $j = m$, and $e_q = 1$ if $j < m$, where Σ is the input alphabet, i is the number of symbols read so far and j is the number of errors seen so far. The transitions correspond to the possible events of the channel in response to reading the next input symbol. That is, 0 or 1 substitution errors will occur on the last input symbol. Hence, if $e_q = 0$ (when errors seen so far $j = m$) we only add one error free transition from a state p to a state q, for all $a \in \Sigma$. If, however, $e_q = 1$ (errors seen $j < m$) then we add $|\Sigma|$ transitions from a state p to a state q, for all $a \in \Sigma$.

   For a given state p, a transition out of p would go to a state q such that the meaning of the states as described above is preserved. In particular, if p is of the form (i, j) then $q = (i + 1, j)$ if no error occurred, and $q = (i + 1, j + 1)$ if a substitution occurred. For example, the state $(2, 0)$ represents the situation where 2 symbols have been read and no error has occurred. Transitions with 0

and 1 errors from $(2,0)$ would go to the states $(3,0)$ and $(3,1)$ respectively.

See Figure 4.8 for an example of such a construction.

□

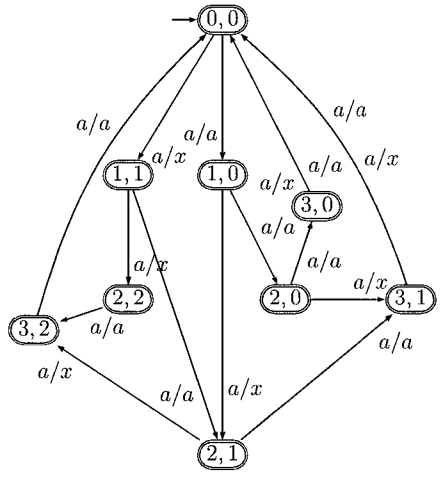

Figure 4.8: Sequential machine realizing $\sigma_h(2,4)$

Remark 4.3.2. In Figure 4.8, the states are represented by pairs (i,j), where i is the number of symbols read for the block and j is the number of errors seen. We skip showing the construction of the channels $(\iota \odot \sigma)_h(m,l)$, $(\iota \odot \delta)_h(m,l)$, $(\sigma \odot \delta)_h(m,l)$ and $(\sigma \odot \iota \odot \delta)_h(m,l)$ because they can be extended from the above construction and the methods used in the previous section.

Chapter 5

Algorithmic Tools II: Additional Relevant Algorithms

In the previous chapter we discussed methods for modeling noisy channels using sequential machines. In this chapter we move a step further and present some relevant algorithms used in computing error-detecting capabilities of regular languages.

5.1 Computing the Hamming Distance of a Regular Language

Recall that the Hamming distance of a language is the smallest number of *substitutions* that can change one word of the language into another word of the language [9, 20]. The next proposition explains how the Hamming distance of a language relates to error-detection.

Proposition 5.1.1. *[9] Given a language L and the channel $\gamma = \sigma(m, \infty)$ which allows a maximum of m substitution errors (σ) in any length of consecutive symbols across the channel, we have that L is error-detecting for γ if the Hamming distance of L is greater than the maximum number of errors the channel allows, that is, $H(L) > m$.*

In this section we discuss the algorithm for computing the Hamming distance of a regular language in quadratic time using the results in [20].

Theorem 5.1.2. *[20] Given an automaton A accepting a language L, the Hamming distance of L can be computed in quadratic time.*

The reader is referred to [20] for the proof. Nevertheless, we present a brief description of this computation, as well as a discussion of our implementation.

Given a NFA A accepting a language $L(A)$, we discuss how the Hamming distance of $L(A)$ is computed. Recall from Section 3.1.2 of Chapter 3 that we defined an edit string as a concatenation of edit operations from a given set of edit operations. The alphabet E_Σ of the basic edit operations (representing insertions, deletions and substitutions) is the set of all symbols x/y such that $x, y \in \Sigma \cup \{\lambda\}$ and at least one of x and y is in Σ. If E is any subset of E_Σ, the automaton $A \cap_E A$, [25], accepts all edit strings that can transform words in L to words in L using the edit operations in E. Note that the input and output parts of the edit strings accepted by $A \cap_E A$ would both be in L. More specifically, $A \cap_E A$ is defined as follows: $((p, q), x/y, (p', q'))$ is a

transition in $A \cap_E A$, if (p, x, p') and (q, y, q') are transitions in A and $x/y \in E$. Also, (s, s) is the start state of $A \cap_E A$, where s is the start state of A, and each pair (f, g) is a final state of $A \cap_E A$, if f and g are final states of A.

Now let $E_\sigma = \{x/y$, where $x, y \in \Sigma$ and $x \neq y\}$.

1. Construct the NFA $A_{\sigma'}$ over the alphabet E_σ which has two states s and g, where s is the start state and g is the only final state. $A_{\sigma'}$ has the transitions sx/xs, sx/yg, gx/xg, and gx/yg, for all symbols $x, y \in \Sigma$ with $x \neq y$. It is clear that $A_{\sigma'}$ accepts all edit strings in E_σ^* containing at least one substitution error. Furthermore, we assign a cost to each edit operation as follows:

$$f_{\sigma'}(x/y) = \begin{cases} 1, & \text{if } x = y \\ 0, & \text{if } x \neq y \end{cases}$$

2. Construct the automaton $A_\sigma = (A \cap_{E_\sigma} A) \cap A_{\sigma'}$ which accepts all edit strings that transform words of the language $L(A)$ into other words of $L(A)$ and have at least one substitution error. If $L(A_\sigma)$ is empty then the Hamming distance of $L(A)$ is infinite, and we ignore the next two steps. Otherwise, proceed to the next two steps.

3. Use Djikstra's algorithm to compute a shortest accepting path in A_σ.

4. The Hamming distance is the total number of errors or total cost on the shortest accepting path in A_σ.

Algorithm 5.1.1. Computing the Hamming distance of a regular language using shortest path algorithm

INPUT : NFA A

```
Define two queues Q0 and Q1
Initialize all entries of the boolean array Seen to false
Initialize all entries of the integer array Distance to 0
Q0.insert(startNode)
Seen[startNode]=true
length=0
while (Q0 is not empty)
    while (Q0 is not empty)
        a=Q0.front()
        for each edge (a,b) in Q0 with not Seen[b]
            Q0.insert(b), Seen[b] = true;
            Distance[b]=length
        end for
        Q0.delete(), Q1.insert(a)
    end while
    length = length + 1
    while (Q1 is not empty)
        a=Q1.front()
        for each edge (a,b) in G1 with not Seen[b]
            Q0.insert(b), Seen[b] = true;
            Distance[b]=length
        end for
        Q1.delete()
    end while
```

```
end while
```

OUTPUT: Hamming distance of $L(A)$

The reader can refer to Chapter 6 and [3] for details on our implementation of this.

5.2 Computing the Edit Distance of a Regular Language

Recall from Chapter 2 that the edit distance of a language is the smallest number of substitution, insertion and deletion errors that can change a word of a language into another word of the language. The definition given below shows how the edit distance is related to error-detection.

Theorem 5.2.1. *[30, 26] Given a language L and the channel $\gamma = \sigma \odot \iota \odot \delta(m, \infty)$ which allows a maximum of m substitution, insertion and deletion errors in any word, we have that L is error-detecting for γ if the edit distance, Λ, of L is greater than m, that is, $\Lambda(L) > m$.*

We use the algorithms in [27] to compute the edit distance of a regular language. Refer to it for a full understanding of our approach. Nonetheless, we present the main ideas here.

Given a language L, we would approach computing the edit distance as follows:

1. Let A be an automaton accepting L

2. Let B be the automaton $A \cap_E A$ which accepts all edit strings that can transform words in L to words in L using the edit operations in the alphabet of edit symbols E. Note that the input and output part of the edit strings accepted by B would both be in L.

3. If it was possible, we would construct an automaton T accepting all edit strings for which the input and output parts are distinct.

4. Then we would compute the transducer $B \cap T$ that accepts all edit strings for which the input and output parts are distinct and are in L.

5. Treating $B \cap T$ as a weighted graph where the weight of a transition is 1 when there is an error and 0 otherwise, the edit distance would simply be the weight of the shortest path.

Unfortunately, such a T does not exist [39]. In [27], the author presents an alternative approach that leads to the desired result using a transducer T_j described below.

T_j is an automaton that accepts all edit strings whose input and output parts differ at exactly position j for a given integer j. Refer to [27] for the complete rules for constructing T_j and proof of correctness. Of particular importance for our usage is the fact that it is enough to only consider all T_j for which $1 \leq j \leq q(A)$, where

$q(A)$ is the value $diam(A)$ if A is deterministic, or s^2 if it is non-deterministic, where s is the number of states in A.

Since we are only considering deterministic finite automata we use $diam(A)$ and not s^2. With this in mind and an understanding of T_j, we proceed with computing the edit distance of a regular language as follows:

1. As before, let A be an automaton accepting L.

2. Similarly, let B be the automaton $A \cap_E A$ which accepts all edit strings that can transform words in L to words in L using the edit operations in the alphabet of edit symbols E.

3. Initialize a distance tracking variable *dist* with the value of $diam(A)$. That is, $dist = diam(A)$.

4. For each j in the range $1...q(A)$

   (a) Construct T_j.

   (b) Construct $B \cap T_j$.

   (c) Find the shortest path in $B \cap T_j$ based on weight as described earlier.

   (d) If the total weight on this path is less than *dist* then update *dist* with this new value.

5. The edit distance of L is the final value of *dist*.

Below we present this algorithm in pseudo-code. See Section 6.2 for its full implementation details.

Algorithm 5.2.1. Computing the edit distance of a regular language

```
Input = some deterministic finite automaton A;
dist=diam(A);
```
$B = A \cap_E$ A;
```
m = diam(A);
for each j=1,...,m
begin
```
$G = B \cap T_j$;
```
weight=ShortestPathWeight(G);
if (weight <dist)
dist = weight
end
Output = dist;
```

5.3 Construction of A^λ from A

For a given deterministic finite automaton A there is an equivalent automaton A^λ that has λ transitions, but accepts the same language as A. We construct A^λ by simply adding λ self loops to every state in A. Figure 5.1 shows the addition of self loops to A in order to get A^λ. In this example, both A and A^λ accept the language a^*b.

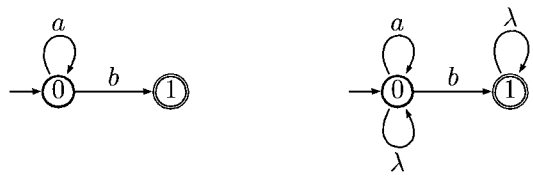

Figure 5.1: An example automaton A and its equivalent A^{λ}

5.4 Construction of $T \downarrow A$ using a Special Cross Product

As mentioned earlier, for a given transducer T and a deterministic finite automaton A, the transducer $T \downarrow A$ is the intersection of the input part of T with A. We achieve this construction using a special cross product construction as follows:

1. Start with an empty machine $T \downarrow A$

2. For each transition (p, x, q) in A^{λ}, get all transitions $(p', x/y, q')$ in T that have x as the input label, and for each such transition create the transition $(p'p, x/y, q'q)$ in $T \downarrow A$.

3. A state in $T \downarrow A$, such as $p'p$, is the start state only if p' and p are start states in their respective automata.

4. Similarly, a state in $T \downarrow A$, such as $q'q$, is a final state only if both q' and q are final states in their respective automata.

The resulting transducer accepts pairs (u, v) of words such that $(u, v) \in \mathbf{R}(T)$ and $u \in L(A)$. In the following figures we show an example transducer T and the resulting transducer $T \downarrow A$ using A^λ from Figure 5.1.

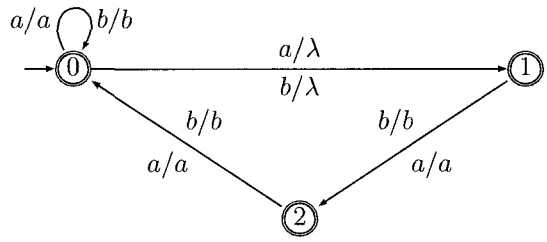

Figure 5.2: An example transducer T

Below we have Table 5.1 showing the transitions in the T and A^λ in the examples above, and Table 5.2 showing the matching on the input that gets used in the cross product construction of $T \downarrow A$.

Transitions in T	Transitions in A^λ
$(0, a/a, 0)$	$(0, a, 0)$
$(0, b/b, 0)$	$(0, \lambda, 0)$
$(0, a/\lambda, 1)$	$(0, b, 1)$
$(0, b/\lambda, 1)$	$(1, \lambda, 1)$
$(1, a/a, 2)$	
$(1, b/b, 2)$	
$(2, a/a, 0)$	
$(2, b/b, 0)$	

Table 5.1: Transitions in T and A^λ

The following table shows the matching on the input of transitions in T with transitions in A^{λ}.

Transitions in A^{λ}	Transitions in T with matching input
$(0, a, 0)$	$(0, a/a, 0)$ $(0, a/\lambda, 1)$ $(1, a/a, 2)$ $(2, a/a, 0)$
$(0, \lambda, 0)$	
$(0, b, 1)$	$(0, b/b, 0)$ $(0, b/\lambda, 1)$ $(1, b/b, 2)$ $(2, b/b, 0)$
$(1, \lambda, 1)$	

Table 5.2: Matching transitions in A^{λ} with the input part of T

$T \downarrow A$ is then constructed by a cross product using the matched states in Table 5.2. The resulting transducer is shown in the figure below:

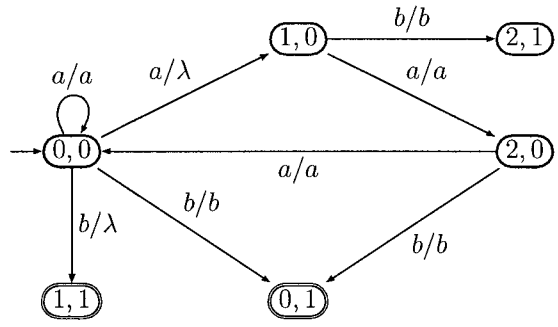

Figure 5.3: The transducer $T \downarrow A$

5.5 Construction of $T \uparrow A$ using a Special Cross Product

Similarly, $T \uparrow A$ is the intersection of the output part of T with A. We achieve this using a product construction as follows:

1. For each transition (p, y, q) in A, get all transitions $(p', x/y, q')$ in T that have y as the output label, and for each such transition create the transition $(p'p, x/y, q'q)$

2. A state such as $p'p$ is the start state only if p' and p are start states.

3. Similarly, a state such as $q'q$ is a final state only if both q' and q are final states.

The resulting transducer outputs words that are both outputs of T and also accepted by A. For a better understanding, next we provide an example of this construction using $T \downarrow A$ from the previous section in place of T. The table below shows transitions in $T \downarrow A$ and A^{λ} that are also used in the cross product construction of $(T \downarrow A) \uparrow A$.

Transitions in $T \downarrow A$	Transitions in A^{λ}
$([0,0], a/a, [0,0])$	$(0, a, 0)$
$([0,0], b/b, [0,1])$	$(0, \lambda, 0)$
$([0,0], b/\lambda, [1,1])$	$(0, b, 1)$
$([0,0], a/\lambda, [1,0])$	$(1, \lambda, 1)$
$([1,0], a/a, [2,0])$	
$([1,0], b/b, [2,1])$	
$([2,0], a/a, [0,0])$	
$([2,0], b/b, [0,1])$	

Table 5.3: Transitions in $T \downarrow A$ and A^{λ}

The following table shows the matching on the output of transitions in $T \downarrow A^{\lambda}$ with transitions in A^{λ}.

Transitions in A	Transitions in $T \downarrow A$ with matching output
$(0, a, 0)$	$([0,0], a/a, [0,0])$ $([1,0], a/a, [2,0])$ $([2,0], a/a, [0,0])$
$(0, \lambda, 0)$	$([0,0], b/\lambda, [1,1])$ $([0,0], a/\lambda, [1,0])$
$(0, b, 0)$	$([0,0], b/b, [0,1])$ $([1,0], b/b, [2,1])$ $([2,0], b/b, [0,1])$
$(1, \lambda, 1)$	$([0,0], b/\lambda, [1,1])$ $([0,0], a/\lambda, [1,0])$

Table 5.4: Matching transitions in A^{λ} with the output part of $T \downarrow A$

Similarly, $(T \downarrow A) \uparrow A$ is then constructed by cross product using the matched states in Table 5.4. The resulting transducer is show in the figure below:

As a side note, the input words recognized by $(T \downarrow A) \uparrow A$, and their corresponding output words, are both accepted by the automaton A. In other words, the transducer accepts pairs of words whose input and output parts are both in $L(A)$. We will utilize these tools later in this chapter.

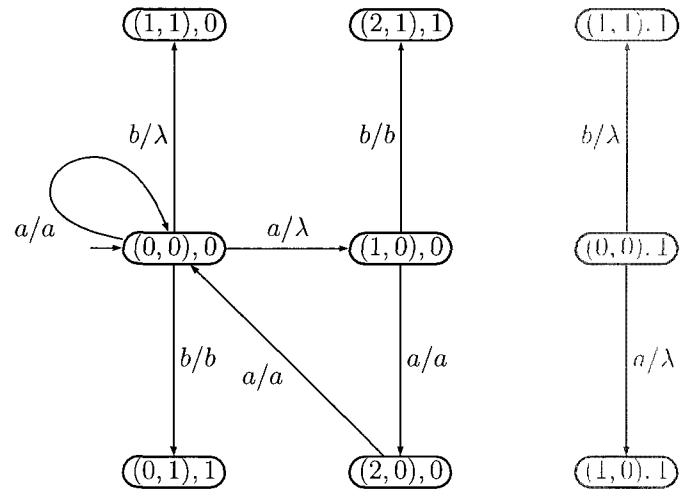

Figure 5.4: The transducer $(T \downarrow A) \uparrow A$ with unreachable states in gray

5.6 Introducing Pseudo-Sequential Machines

Recall from Chapter 2 that in a pseudo-sequential machine start and final states have no outgoing λ-*input* transitions, and if a state has an outgoing λ-*input* transition then it has no other outgoing transitions. As a build-up to the next section, we now present some useful concepts related to pseudo-sequential machine T'. First note that if $(p, \lambda/y, q)$ is a λ-*input* transition of T', then there is a unique path of λ-*input* transitions from q to some state t such that t has no outgoing λ-*input* transitions. This follows from the constraints on the λ-*input* transitions of a pseudo-sequential transducer T', and the fact that T' must be trim.

Theorem 5.6.1. *If T is a sequential machine then there is an equivalent pseudo-sequential machine T'.*

Proof. T' is constructed by expanding some transitions as follows:

1. Make a copy of T

2. Replace each transition $(p, x/y, q)$ for which $|y| > 1$, with the transitions $(p, x/y_1, r_1)$, $(r_1, \lambda/y_2, r_2)$, ..., $(r_{k-1}, \lambda/y_k, q)$, where $y_1...y_k = y$ and r_1, r_2,...,r_k are the new states added for the sake of expanding the original transition. Figure 5.5 shows how the states get replaced in this operation.

□

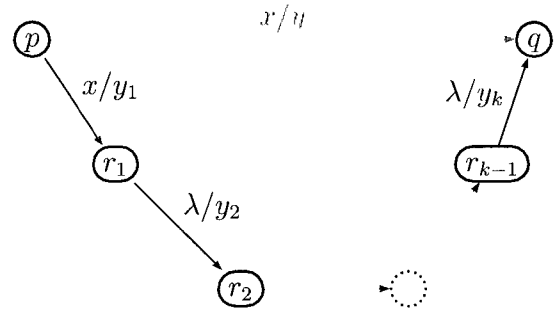

Figure 5.5: Converting from sequential to pseudo-sequential

Theorem 5.6.2. *If T' is a pseudo-sequential machine then there is an equivalent sequential transducer T''.*

Proof. Construction of T'' from T'

1. Make a copy of T'.

2. For every transition $(p, x/y, q)$, with $x \neq \lambda$, for which there is $(q, \lambda/z_1, r)$, replace $(p, x/y, q)$ with $(p, x/yw, t)$, where t is the unique state with no outgoing λ-*input* transitions that can be reached from q, and w is the word formed by the output labels in the unique path from q to t.

□

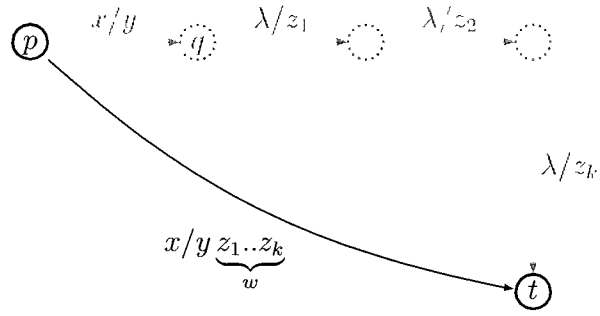

Figure 5.6: Converting from pseudo-sequential to sequential

Theorem 5.6.3. *If T' is a pseudo-sequential machine and M is a DFA then $T' \uparrow M$ is also pseudo-sequential.*

The proof of this can be found in [5].

5.7 Deciding Transducer Functionality

Utilizing the edit and Hamming distances for computing error-detecting capabilities of a regular language works well for channels of type $\tau(m, l)$ with $l = \infty$. The methods generally break down when we have $l < \infty$. To address this limitation we utilize transducer functionality to decide error-detecting capabilities of regular languages. The work in [24] introduces the method of using transducers to decide error-detecting capabilities.

Theorem 5.7.1. *[10, 1] Given a sequential machine, it is decidable in quadratic time whether it is functional .*

The reader is referred to [10, 1] for the proof. The construction in [10] works when the transducer being considered strictly reads one symbol and outputs one symbol for each transition (restricted sequential machine). Hence, it does not work when we consider our sequential machines realizing channels with insertion errors. Obviously, with insertion errors, the output might be more than one symbol per transition. Therefore, we concentrate on the construction provided in [1] as it works for real-time transducers (transducers that can output a set of words for a transition). Recall that our sequential machines are simply a special type of real-time transducers. The reader can refer to [1] for the complete proof of correctness. Nevertheless, for completeness we present the steps of this algorithm. More details can be seen in our implementation notes in Chapter 6.

For a sequential transducer T with a start state s, construct a 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 (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.

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 some value V and there is a transition $((p, p'), (x, x'), (q, q'))$ then (q, q') gets a value as follows:

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

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

(c) Else, {value = ZERO; return NO }

3. Repeat until a state gets two values or every transition has been seen.

4. If every state has one value AND every final state of U has value (λ, λ) then output YES (T is functional) Else output NO.

The figures below give an example of how we utilize this algorithm.

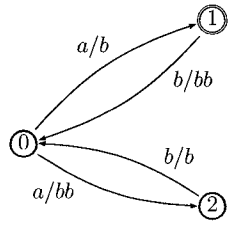

Figure 5.7: An example transducer T

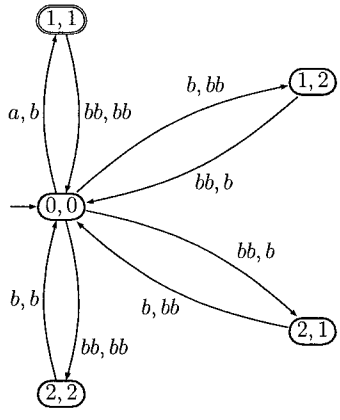

Figure 5.8: U is a cross product construction from T in Figure 5.7

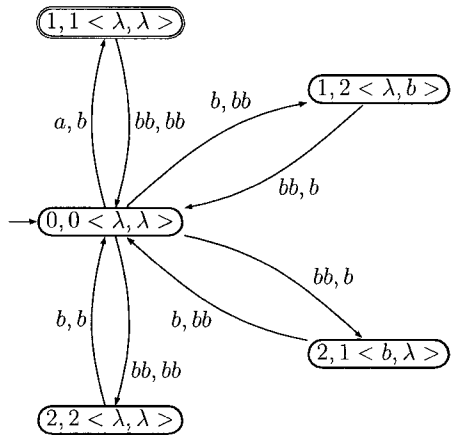

Figure 5.9: Assigning values to the trim part of U

5.8 Deciding Error-Detection Using Transducer Functionality

In this section we discuss the use of transducer functionality for deciding if a language is error-detecting for a channel.

Theorem 5.8.1. *[24] Given an automaton A accepting language $L(A)$ and a transducer T realizing a given channel, it is decidable if L is error-detecting for the channel realized by T .*

Proof. We refer the reader to [24] for the complete proof. Nonetheless, for completeness we have adopted the steps as follows.

We assume that the channel transducer T is given in sequential form and that the automaton A is deterministic.

1. Construct transducer $B = T \downarrow A$, which results by intersecting the input part of T with A. Hence, B realizes all pairs (w, z) such that $(w, z) \in T$ and $w \in L(A)$. Note that B is sequential, as it has the same transition labels with T.

2. Make B pseudo-sequential using Theorem 5.6.1.

3. Construct $C = B \uparrow A$, which results by intersecting the output part of B with A. Therefore, C realizes all pairs (w, z) such that $(w, z) \in T$ and $w, z \in L(A)$. Note that, as B is pseudo-sequential, Theorem 5.6.3 implies that C is also pseudo-sequential as well.

4. Convert C to a sequential machine using Theorem 5.6.2.

5. Decide if C is functional.

6. If C is functional then $L(A)$ is error-detecting for the channel realized by T; Otherwise it is not.

□

The algorithm operates in time $(|T||A|^2)^2 = |T|^2|A|^4$. This is because the construction of B takes time $|T||A|$, the construction of C takes time $|T||A|^2$, and the decision of whether C is functional takes quadratic time. The reader can refer to Section 6.2.7 for details on our implementation of this algorithm.

5.9 Computing Maximal Error-Detecting Capabilities

In this section we present algorithms for the computation of maximal error-detecting capabilities of regular languages for various error models. We will utilize the tools developed in the preceding sections and chapters to achieve this goal.

5.9.1 Overview of our Approach

Recall that by definition an error model is simply a set of channels. Using the methods from Chapter 4, we can construct channels belonging to an error model. We then use the methods in the earlier sections of this chapter to decide if a regular language is error-detecting for a given channel. For a given error model, we systematically apply all the tools we described earlier to compute the maximal channel for which the regular language is error-detecting.

Recall that for an error model $\mathbb{C}$, a channel $\gamma \in \mathbb{C}$ is a $\mathbb{C}$-*maximal error-detecting capability* of a regular language L, if L is error-detecting for γ and L is not error-detecting for another channel γ' that properly contains γ.

5.9.2 The Error Model $\mathbb{C}_{\tau}^{0}[\infty] = \{\tau(m, \infty)$: for all $m \geq 1\}$

This error model represents channels that allow a maximum of m errors of type τ in any given length of symbols passing through the channel. That is,

$\mathbb{C}^0_\tau[\infty] = \{\tau(1,\infty), \tau(2,\infty), \tau(3,\infty), \ldots\}$. Below we present a couple of scenarios for this error model and, in each case, show how to compute the maximal error-detecting capability.

Substitutions - $\mathbb{C}^0_\sigma[\infty] = \{\sigma(m,\infty)$: **for all** $m \geq 1\}$

This variation of the error model contains channels that only allow substitution errors. Hence, for a given $m \geq 1$, only m substitutions are allowed in any length of symbols passing through, that is,

$$\mathbb{C}^0_\sigma[\infty] = \{\sigma(1,\infty), \sigma(2,\infty), \sigma(3,\infty), \ldots\}$$

For this instance of the error model, the problem of computing maximal error-detecting capabilities of a regular language reduces to computing the Hamming distance of the language. Hence, given a regular language L we compute the Hamming distance $H(L)$ of the language using Algorithm 5.1.1.

The $\mathbb{C}^0_\sigma[\infty]$-maximal error-detecting capability of L is $\sigma(H(L)-1,\infty)$ provided that $H(L)-1 > 1$. Otherwise, L does not have error-detecting capabilities for the error model.

The reader can look at Sections 6.2.3 and 6.3 for details of how we implemented this.

SID - $\mathbb{C}^0_{\sigma\odot\iota\odot\delta}[\infty]$= $\{\sigma \odot \iota \odot \delta(m, \infty)$: **for all** $m \geq 1\}$

This SID variation of the error model allows substitutions, insertions and deletions. Therefore, for a given $m \geq 1$, only m errors, that can consist of substitutions, insertions and deletions, are allowed in any length of symbols passing through, that is,

$$\mathbb{C}^0_{\sigma\odot\iota\odot\delta}[\infty]=\{\sigma \odot \iota \odot \delta(1, \infty), \sigma \odot \iota \odot \delta(2, \infty), \sigma \odot \iota \odot \delta(3, \infty), \ldots\}$$

In this variation of the error model, the problem of computing maximal error-detecting capabilities of a language is solved by computing the edit distance Λ of the language. Hence, given this error model and a language L, we compute the edit distance of L using Algorithm 5.2.1.

The $\mathbb{C}^0_{SID}[\infty]$-maximal error-detecting capability of L is $\sigma \odot \iota \odot \delta(\Lambda(L) - 1, \infty)$ provided $\Lambda(L) - 1 > 1$. Otherwise, the language does not have any error-detecting capability for the error model.

We have described our implementation of this in Section 6.2.4 and Section 6.3.

5.9.3 The Error Model $\mathbb{C}^1_\tau[l]$= $\{\tau(m, l)$: **for any** m **with** $m < l\}$

In the model, $\mathbb{C}^1_\tau[l]$= $\{\tau(m, l)$: for any m with $m < l\}$, we have the error type τ and length l of symbols fixed. As can be observed, we only have a finite number of channels to consider, that is,

$$\mathbb{C}^1_\tau[l]=\{\tau(1, l), \tau(2, l), \ldots, \tau(l - 1, l)\}.$$

For a given language L, error types τ and a value for l, we start assigning values to m from the range $1 \ldots (l-1)$. For each m, we use the algorithms in Chapter 4 to construct a sequential machine realizing the channel. We then use the algorithm in Section 5.8 to decide if L is error-detecting for the channel. The process continues until the first value of m is found for which L is not error-detecting for the channel. Ideally, the process can be sped up by using binary search to go through the values of m. However, because the size of the sequential machines generated grows exponentially with respect to m, from an implementation point of view, the binary search does not help and might hurt performance. The $\mathbb{C}_\tau^1[l]$-maximal error-detecting capability of L is $\tau(m-1, l)$, provided $m > 1$. Otherwise, L does not have any error-detecting capabilities for the error model.

Details on how we implemented this have been included in Sections 6.3, 6.2.7 and 6.2.5.

5.9.4 The Error Model $\mathbb{C}_\tau^2[m] = \{\tau(m, l)$: for any l with $l > m\}$

In this error model, we have the error type τ and the number m of errors fixed. Our goal is to find the smallest l for which a language is error-detecting for the channel $\tau(m, l)$.

Theorem 5.9.1. *[26]*

For a given automaton A, accepting at least two words and a value of m, the

language L(A) is error-detecting for the channel $\tau(m, l)$, for some $l > m$, if and only if it's error-detecting for $\tau(m, 2ms^2)$, where s is the number of states in A.

The reader is referred to [26] for the proof and more details. Nonetheless, Theorem 5.9.1 provides us with an upper bound on the values of l we need to consider. As can be observed, this reduces the number of channels to a finite set:

$$\{\tau(m, m+1), \tau(m, m+2), \ldots, \tau(m, 2ms^2)\}$$

Therefore, for a given language L and m, we start assigning values to l from the range $m + 1 \ldots 2ms^2$, where s is the number of states of the automaton A whose language is L. The process stops when we find the smallest l for which L is error-detecting for $\tau(m, l)$.

The $\mathbb{C}_\tau^2[m]$-maximal error-detecting capability of L is $\tau(m, MIN(l))$, where $MIN(l)$ is the smallest $l > m$ for which L is error-detecting for $\tau(m, l)$.

The reader can look at Section 6.3 for our implementation details for this.

Chapter 6

Implementation Details, Testing and Interacting with the System

We have implemented all the relevant algorithms in this thesis using C++ and Grail [36]. In addition, we have created a web interface for interacting with the tools we have developed. This chapter gives some details on our implementation and shows how to interact with the system.

6.1 Overall System Architecture

The system has been implemented and tested on Linux and Windows platforms. It has three tiers: web tier, logical tier and data tier. Figure 6.1 shows the overall architecture of our implementation. The heart of our implementation is in the logical tier, which has been implemented in a modular fashion using C++. Hence, most classes we have written are fairly decoupled for added flexibility.

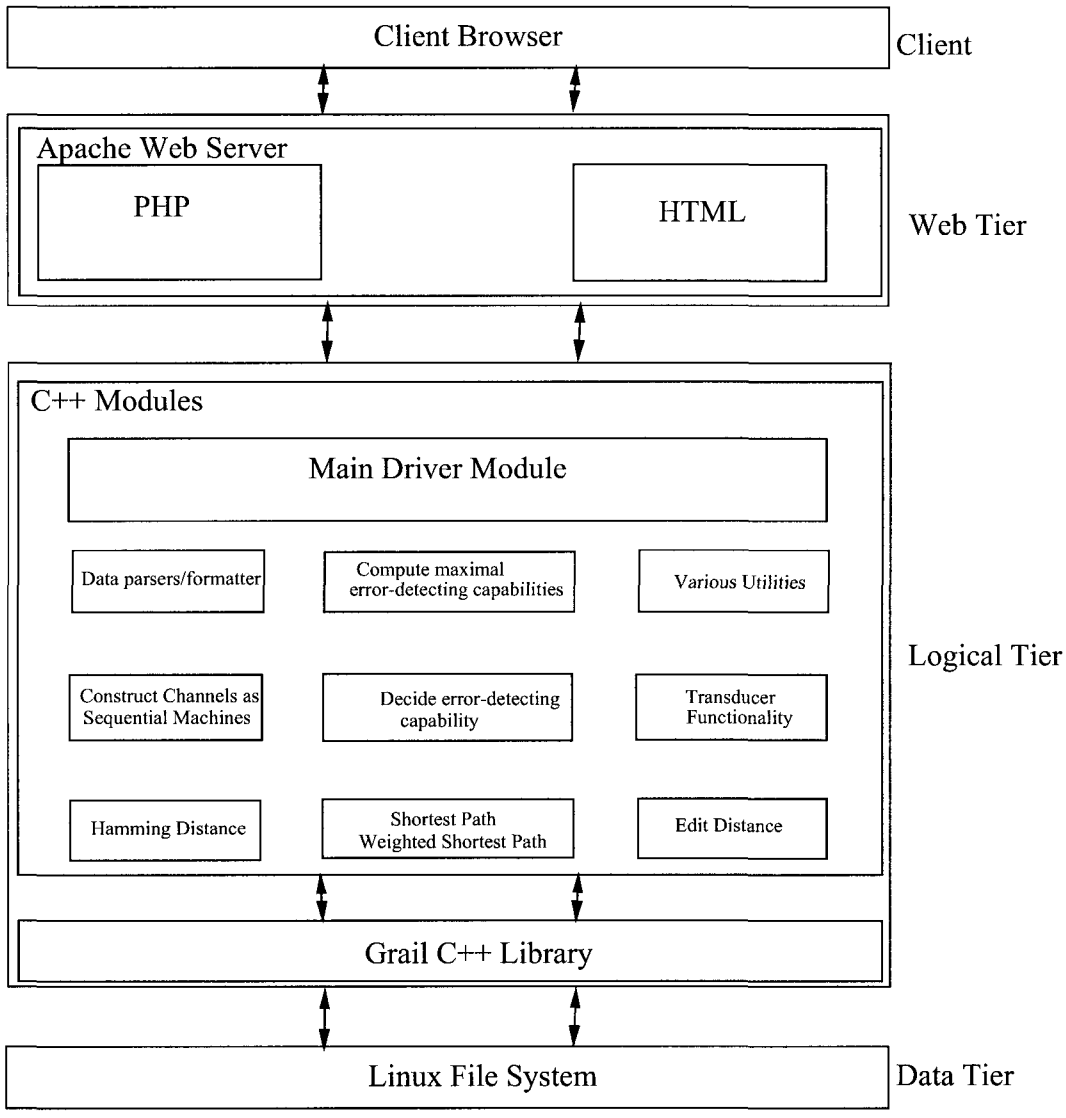

Figure 6.1: Overall system architecture

6.1.1 Web/Presentation Tier

The presentation tier is composed of a web application written using PHP [34] and jQuery [15]. We have deployed this application on an Apache web server. It gets input from a user, submits it to the logical tier for processing, and renders the results from the logical tier back to the user. We have followed a Model-View-Controller pattern to develop the web application. The controller component of the application is implemented in `submitJobs.php` and `functions.php`. It gets all the user input and action requested, passes it to the model for processing, and hands over control to the view component. The model is composed of the C++ components that process the requested job and make the result available to the view component. The view component is implemented in `displayJobResults.php`. It queries the model using a job id to retrieve the results and displays them to the user. Figure 6.2 shows the main web interface for interacting with the algorithms we have implemented. This web interface can be accessed using [4]. We have made available the source code for our implementation, including the PHP web application, via [3].

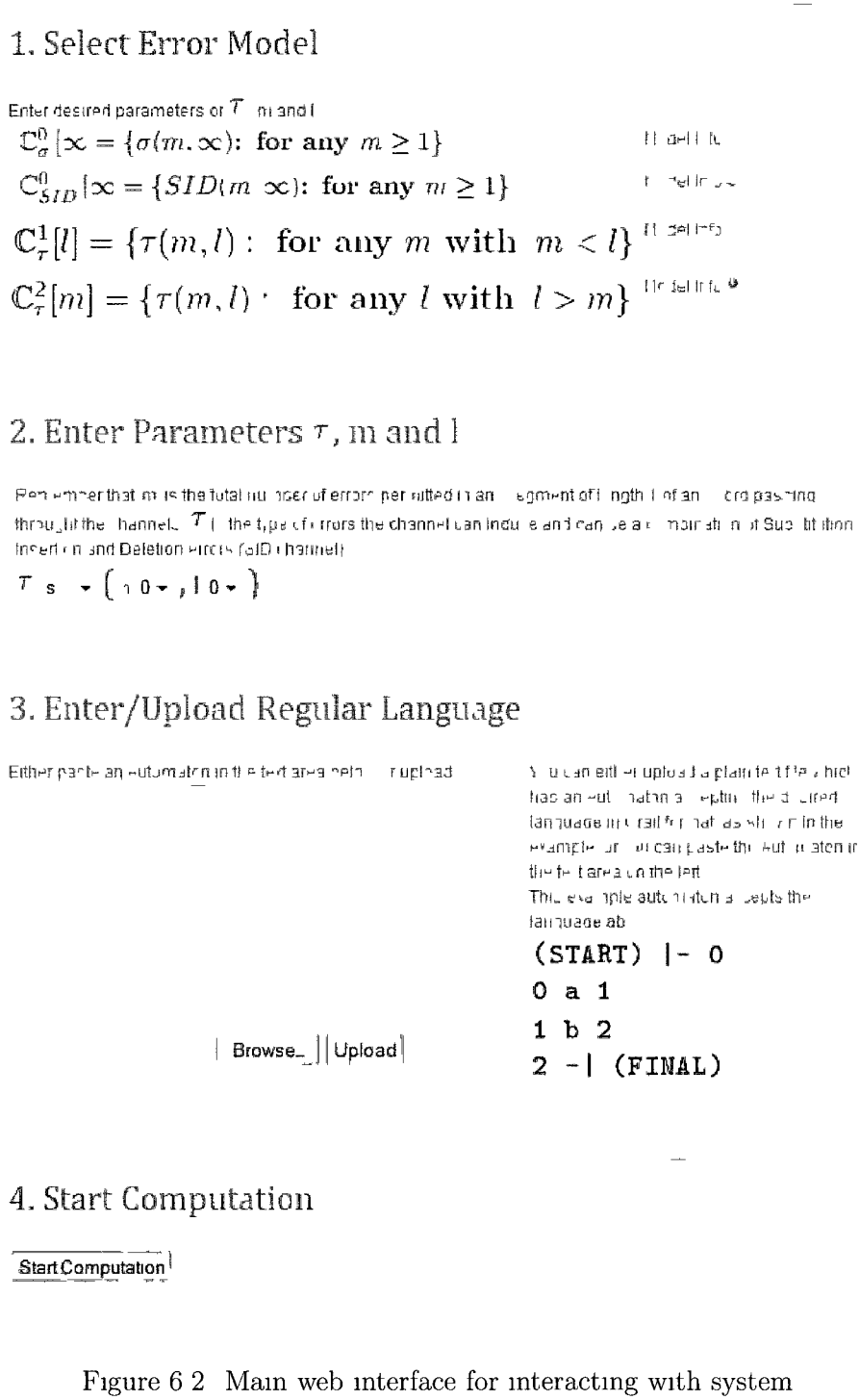

Figure 6.2 Main web interface for interacting with system

6.1.2 Logical Tier

The logical tier contains the heart of our system. It consists of C++ modules that implement all the algorithms in this research. Some notable modules in the system deal with:

1. Weighted shortest path
2. Hamming distance of a regular language
3. Edit distance of a regular language
4. Construction of channels with sequential machines
5. Converting from sequential to pseudo-sequential
6. Converting from pseudo-sequential to sequential
7. Transducer functionality
8. Deciding error-detection
9. Maximal error-detecting capabilities
10. Several helper utilities

All the modules we have developed are built either on top of or alongside the Grail C++ Library. We present more details on these modules later in this chapter.

6.1.3 Data Tier

The data tier utilizes the filesystem to store useful data such as input, output and configuration details. Grail was originally designed to work with file based input and output. Hence, we have built our implementation to also use file input and output. This has performance implications when large data sets are used. A possible future improvement would be to include adapters for Grail that can utilize a database instead of the filesystem.

6.2 Logical Tier I—A Detailed Look at Core Tools

As mentioned earlier in the chapter, the logical tier is the biggest and most important part of our implementation. In this section we provide a closer look at this core of our implementation. We present some useful information on relevant classes and methods that work together to create our logical tier.

6.2.1 Relevant Grail Classes

Here we outline the major and relevant Grail C++ library classes we use in our implementation. Refer to [37] for more details on Grail. Most classes in this library extensively use C++ templates. For brevity, we sometimes omit the template details in our implementation descriptions shown later in this chapter.

The class **fm** is used to represent a finite machine. Recall that the basic components of any finite state machine are states, transitions, transition labels from an alphabet, a start state and a set of final states. The class **fm** utilizes, among other support classes, **bit**, **array**, **list**, **set**, **string**, **state** and **inst**, to represent the components of a finite state machine. Notably, **state** implements the states while **inst** implements the transitions of a finite state machine respectively.

As part of our implementation, we extended the functionality of some base classes to meet the requirements for our usage. Our general approach has been to extend a base class and make required additions to achieve our goals. For example, Grail only stores states as integers. Hence, it can not be used out-of-the-box for many of our automata because they require a state to store its meaning. Therefore, to work around this limitation we extended the base class **fm** and added a way for us to track the meaning of a state even though it is still stored as an integer. Since most of the classes in our implementation descend from **fm**, we have used Code Snippet 6.2.1 to show some relevant parts of the class. This should help in visualizing how the other classes in our implementation fit into the picture.

Furthermore, the out-of-the-box Grail functionality only supports **char** and **int** alphabets. On the other hand, some of our automata implementation, such as channels handling insertion errors, required an extension to Grail to use either an array of **chars** or **string** for alphabet symbols. We chose to implement the support for

string alphabets.

Code Snippet 6.2.1. *A partial view of the* ***fm*** *class*

```
class fm
{
set<inst<Label> >arcs; // set of instructions
set<state>start_states; // set of start states
set<state>final_states; // set of final states
public:
  fm() { ; } // default constructor
void add_instruction(const inst<Label>& i);
void set_finals(const set<state>& s);
void set_starts(const set<state>& s);
set<state>& sources(set<state>&) const;
set<state>& sinks(set<state>&) const;
set<state>& starts(set<state>& r) const
int is_deterministic() const;
int number_of_final_states() const;
int number_of_states() const;
.
.
.
~fm() { ; } // destructor
};
```

The significant classes we have added for our purposes are: **MaximalFm**, **Transducer**, **TransducerK**, **DistanceFm**, **ErrorFm** and several other supporting classes. The class **MaximalFm** extends Grail's base finite machine class **fm**, and is the base class for most classes in our implementation. As noted earlier, Grail only stores states as integers. However, most of the finite machines in our work, such as all the constructions in Chapter 4, need to store the meaning of each state. Hence, to fill this gap, we created **MaximalFm**, a descendant of **fm**, that adds our desired functionality. Specifically, **MaximalFm** adds on a map data structure called `stringToStateLookUpMap`, implemented as a `hashmap`. This map, working together with associated methods, provides a way for us to use Grail's underlying states while keeping track of the meaning of states as desired. In most cases, we only need to keep track of the meaning of a state when constructing our automata. After we construction, we generally manipulate the automata like any other Grail constructed object. Code Snippet 6.2.2 shows a part of the class **MaximalFm** extending from **fm**.

Code Snippet 6.2.2. *A partial view of the class* ***MaximalFm*** *extended from* ***fm***

```
template< class Label >
class MaximalFm:public fm <Label>{
hash_map<string,state> stringToStateLookUpMap;
public:
MaximalFm();
    bool addEmptySelfLoops();
    int getDiameter();
    state getStateFromMap(string strState);
                .
                .
                .
};
```

In addition, we provide an example in Table 6.1 showing how the lookup map, `stringToStateLookUpMap`, would look for the sequential machine in Figure 4.2.

stringToStateLookUpMap	
String State	Grail State
nnnn	0
nnsn	1
nnns	2
nsns	3
snsn	4
nsnn	5
nnss	6
snnn	7
nssn	8
ssnn	9
snns	10

Table 6.1: Mapping of MaximalFm states to Grail's states

In the next sections, we provide details for the other classes and their methods, as we describe the functionality that utilizes them.

6.2.2 Implementation of Weighted Shortest Path Computation

We have utilized the weighted shortest path algorithm as part of computing the Hamming distance and the edit distance of a regular language. To this end, we have implemented Dijkstra's shortest path algorithm [6, 28], in the class **Graph**. Our approach for utilizing this algorithm has been to to create a weighted graph based on the transitions and labels of the finite state machine to which we wish to apply the algorithm. We then apply Dijkstra's shortest path algorithm on the created graph. Code Snippet 6.2.3 shows some parts of the class **Graph**.

We added the suitable constructor `Graph( const fm<Label>& a )` that constructs a weighted graph based on the transitions of the input finite machine. We then utilize Dijkstra's algorithm to get the constructed graph's weighted shortest path and its associated weight.

We also note that our implementation of this algorithm extends from that used in [19].

Code Snippet 6.2.3. *A partial view of the class* ***Graph***

```
class Graph{
    public:
        Graph(int size);
        Graph( const fm<Label>& a );
        ~Graph();
        void AddEdge( int from, const Label&, int to );
        int GetSize();
        int DeleteEdge( int from, int to );
        void ShortestPaths_Dijkstra( const set<int>& sources,
                                     double distances[],
                                     int previousNode[],
                                     Label previousLabel[],
                                     double (*Distance)( const Label& ) );
    private:
        GraphListNode<Label>** AdgacencyList;
        int MaxNode;
        friend class DistanceFm<Label>;
}
```

6.2.3 Implementation of Hamming Distance Computation

Our implementation for computing the Hamming distance of a regular language uses the classes **Graph**, **ErrorFm** and **DistanceFm**. **ErrorFm** realizes, among other things, a finite machine that transforms words of the input regular language to other words of the same language using substitution errors. Recall that a description of

such a finite machine is given in Section 5.1. **DistanceFm** is utilized for creating a **Graph** object based on an input **ErrorFm** object, and subsequently for applying Dijkstra's shortest path algorithm to return the Hamming distance.

Note that both **ErrorFm** and **DistanceFm** are also used in the process of computing the edit distance, as we will show shortly. The next code snippets show partial details of the two classes, and an example of how the two classes are used when computing the Hamming distance.

Code Snippet 6.2.4. *Partial view of the class* ***ErrorFm***

```
class ErrorFm:public Transducer<Label>{
public:
ErrorFm(){;};
ErrorFm(const fm A1,const fm A2,EMPTY_TRANSITION);
ErrorFm(const MaximalFm& A1,const MaximalFm& A2,EMPTY_TRANSITION,
    const string distanceType );
ErrorFm(const ErrorFm<Label> A2, const ErrorFm<Label> B);
ErrorFm(const fm<Label>& A1);
ErrorFm(const TransducerK<Label>& Tk);
ErrorFm(const set<Label> alpha);
};
```

There are several **ErrorFm** constructors depending on the need. For example, `ErrorFm(A1,A2,'~','Hamming')` would create an automaton accepting all edit strings over $\{x/y : x, y \in \Sigma\}$, for which the input part is accepted by automaton $A1$ and the output part is accepted by $A2$— see Section 5.1.

Code Snippet 6.2.5. *Partial view of the class* ***DistanceFm***

```
class DistanceFm : protected Transducer <Label>
{
    public:
        DistanceFm(...);
String< Pair<Label> >& GetMinWord(...);
    private:
}
```

Code Snippet 6.2.6. *Partial view of driver code of computing the Hamming distance*

```
MaximalFm<string> myFm1 = MaximalFm<string>(inputFm);
MaximalFm<string> myFm2 = MaximalFm<string>(inputFm);
ErrorFm<string> fmA( myFm1, myFm2,"~","Hamming");
ErrorFm<string> fmB(myFm1.getAlphabet());
ErrorFm<string> errorFm;
errorFm.cross_product(fmA,fmB);
DistanceFm<string> hamDistFm(errorFm);
//Use Djikstra's algorithm
hamDistFm.GetMinWord(editHammingStr,"~", resultHammingDistance);
pair<int,String<Pair<string>>> result(resultHammingDistance,editHammingStr);
```

6.2.4 Implementation of Edit Distance Computation

Computing the edit distance utilizes the already described classes **ErrorFm**, **DistanceFm** and **Graph**, in conjunction with the class **TransducerK**. **TransducerK** implements T_j and **ErrorFm** realizes B, as both specified in Algorithm 5.2.1. Code Snippet 6.2.7 shows a part of **TransducerK** and Code Snippet 6.2.8 shows a part of the driver code that computes the edit distance using the mentioned classes.

Code Snippet 6.2.7. *The partial view of the class* ***TransduerK*** *showing methods for generating each of the 19 types of transitions described in [27]*

```
class TransducerK:public Transducer<Label>{
public:
TransducerK(){;}
TransducerK(set<Label>& alphabet,int k,const Label &EMPTY_TRANSITION);
state encodeStateToGrailFormat(...);
void make_i_j_E_a_i_j_p_1(int i, int j);// #1 (if j<k-1)
void make_i_j_E_a_i_ak(int i, int j,int k); // #2 (if j=k-1
void make_i_j_a_E_i_p_1_j(int i, int j);// #3 if i < k-1
void make_i_j_a_E_ak_j(int i,int j, int k);// #4 if i=k-1
void make_i_j_to_i_p_1_j_p_1(int i,int j);// #5
void make_i_j_a_b_to_ak_j_p_1(int i,int j,int k);// #6
void make_i_j_a_b_to_i_p_1_bk(int i,int j,int k);// #7
void make_i_j_to_k_k(int i,int j);// #8 if i=j=k-1
void make_kk_x_y_kk(int i, int j);// #9
void make_ak_j_b_E_ak_j(int j,int k);// #10
void make_ak_j_b_c_ak_j_p_1(int j,int k);// #11
```

```
void make_ak_j_b_c_k_k(int j,int k);// #12
void make_ak_j_E_b_ak_j_p_1(int j,int k);// #13
void make_ak_j_E_b_k_k(int j,int k);// #14
void make_i_ak_E_b_i_ak(int i, int k);// #15
void make_i_ak_b_c_i_p_1_ak(int i,int k);// #16
void make_i_ak_b_c_k_k(int i, int k); // #17
void make_i_ak_b_E_i_p_1_ak(int i, int k);// #18
void make_i_ak_b_E_k_k(int i, int k);// #19
}
```

Code Snippet 6.2.8. *A partial view of the driver code for computing the edit distance*

```
//Create transducer whose input and output words belong to the input fm
ErrorFm<string> fmB( myFm1, myFm2,"~","Edit");
ErrorFm<string> errorFm;

for(int i=1;i<diameter;i++) {
TransducerK<string> Tk(myFm1.getAlphabet(),i,"~");
ErrorFm<string> fmTk(Tk);
errorFm.cross_product(fmB,fmTk);
DistanceFm<string> editDistFm(errorFm);
editDistFm.GetMinWord(tempEditString,"~", tempEditDistance);
if(tempEditDistance < resultEditDistance) {
editDistance=tempEditDistance;
editString=tempEditString;
}
}
pair<int,String<Pair<string>>> result(editDistance,editString);
```

6.2.5 Implementation of Channel Construction Using Sequential Machines

To implement all the channels described in Chapter 4, and the other specialized transducers in our research, we have created the class **Transducer** which extends **MaximalFm**. Instead of having separate classes for each type of channel, we utilize the same class and simply leverage the different constructors and methods to construct the sequential machine that realizes a desired channel. This class also relies on the ability to track the meaning of each state that is implemented in **MaximalFm**. The code snippet below shows a part of **TransducerFm**.

Code Snippet 6.2.9. *A partial view of the class* ***Transducer***

```
class Transducer:public MaximalFm < Pair<Label> >{
set<Label> currentAlphabet;
bool channelConstructed;
string currentErrorType;
map<int,Pair<string> > mapOfUValues;
public:
Transducer(){;};
Transducer(const fm& A1,const fm& A2 );
Transducer(Transducer T,const fm& A1,string whichSide);
Transducer(const MaximalFm& A1,const MaximalFm& A2 );
Transducer(const string errorType,int m,int l,set& alphabet);
Transducer(const MaximalFm& A1);
Transducer(const fm<Pair> sourceFm);
```

```
bool createSubChannel(int m,int l,set alphabet);
bool createInsChannel(int m,int l,set alphabet);
bool createDelChannel(int m,int l, const set alphabet);
bool createSubInsChannel(int m,int l,set alphabet);
bool createSubDelChannel(const int m,int l,set alphabet);
bool createInsDelChannel(const int m,int l,set alphabet);
bool createSubInsDelChannel(const int m,int l,set alphabet);
bool intersectWithInput(Transducer inputTransducer,fm inputFm);
bool intersectWithOutput(Transducer inputTransducer,fm inputFm);
bool isTransducerFunctional(Transducer T);
bool convertFromSequentialToPseudo();
bool convertFromPseudoToSequential();
                    .
                    .
                    .
};
```

In addition, below is a code snippet showing one of the constructors and how it constructs the desired channel based on the input error type.

Code Snippet 6.2.10. *A constructor for the class* ***SidFm***

```
SidFm(const string errorType,int m, int l, set<Label> alphabet)
{
currentErrorType = errorType;
currentAlphabet = alphabet;
 if (S.compare(errorType) == 0) {
 channelConstructed = createSubChannel(m,l,alphabet);
 } else if (I.compare(errorType) == 0) {
```

```
 channelConstructed = createInsChannel(m,l,alphabet);
 } else if (D.compare(errorType) == 0) {
 channelConstructed = createDelChannel(m,l,alphabet);
 } else if (SI.compare(errorModel) == 0) {
channelConstructed = createSubInsChannel(m,l,alphabet);
 } else if (SD.compare(errorModel) == 0) {
channelConstructed = createSubDelChannel(m,l,alphabet);
 } else if (ID.compare(errorModel) == 0) {
 channelConstructed = createInsDelChannel(m,l,alphabet);
 } else if (SID.compare(errorModel) == 0) {
 channelConstructed = createSubInsDelChannel(m,l,alphabet);
 }
}
```

In a typical scenario utilizing this class, we are given the number of errors m, the type τ, the length l being considered and an alphabet that is either derived from a finite machine or is entered separately. We construct the sequential machine realizing the channel by creating an instance of **Transducer** using a suitable constructor in the class, such as

`Transducer(const string errorType,int m, int l, set alphabet)`.

Based on the error type, the constructor then calls a suitable method, such as `createSubChannel(m,l,alphabet)`, to create all the states and transitions of the desired sequential machine.

Another typical scenario is when users enter their own channel transducer in a

Grail-like format. In this case, the default constructor `Transducer()` is used, followed by a call to a utility method to load the transducer from the user-provided input.

6.2.6 Implementation of Transducer Functionality Algorithm

We use the class **Transducer**, described above, to decide if a transducer is functional. Specifically, we use the method `isTransducerFunctional()` to implement the rules and constructs in Section 5.7, for deciding if the given sequential machine is functional. We have made this generic so that it can be applied on a sequential machine that has either been generated or has been entered through user input.

6.2.7 Implementation of Error-Detecting Capability Algorithm

Deciding error-detecting capabilities of an input regular language for a channel is an application of the modules described in the preceding sections. Specifically, to decide if a language is error-detecting for a channel, we use the class **Transducer** to either load a user-entered sequential machine realizing the channel or generate a new sequential machine realizing the channel using user-entered parameters of m, l and τ. We trust that having the option for a user-entered sequential machine realizing some channel opens our tools up for others who might want to test channels that we have not covered. We then call **Transducer**'s methods `intersectWithInput()`

and `intersectWithOutput()` to generate a sequential machine based on the channel, but having the input and output words that belong to the input regular language. Note that **Transducer** has the methods `convertFromSequentialToPseudo()` and `convertFromPseudoToSequential()` for converting from sequential to pseudo-sequential and vice-versa, when necessary. These methods are useful when dealing with channels that involve insertions as the resulting transducer after intersecting might not be sequential. We then call the method `isTransducerFunctional()` to decide the functionality of the resulting transducer. If the transducer is functional then the input regular language is error-detecting for the channel. Otherwise, it is not.

6.3 Logical Tier II—Implementation of Maximal Error-Detecting Capabilities

This module applies the tools described in previous sections to compute maximal error-detecting capabilities of a regular language for various error models.

Recall that an error model $\mathbb{C}$ is a set of channels. A channel γ is a $\mathbb{C}$-*maximal error-detecting capability* of a regular language L if the following three conditions are are true:

1. γ is in $\mathbb{C}$

2. L is error-detecting for γ

3. L is not error-detecting for another channel γ' in $\mathbb{C}$ that properly contains γ.

We use the method `getMaximal()`, directly callable from the main method in `maximal.cpp`, to compute the maximal error-detecting capabilities of a language L for a given error model $\mathbb{C}$. Currently, we only consider the error models described in Section 5.9.

Next we show some more implementation details for the error models we have considered.

6.3.1 Implementation for the Error Model $\mathbb{C}_\sigma^0[\infty]$

As noted in Section 5.9, computing maximal error-detecting capabilities for this error model involves computing the Hamming distance of the input regular language. The code snippet below shows a part of the method `getMaximal()` showing how the computation proceeds for several error models including this one. Note that this error model is shown as `SIGMA_INFTY_ER_MODEL()` in the code.

Code Snippet 6.3.1. *The method* `computeUsingHamming()` *of the class* ***MaximalErrorDetectingCapability***

```
if (SID_INFTY_ER_MODEL().compare(errorModel) == 0) {
getEditDistance(fmInputFilename,outputFilename,"maximal");
} else if (SIGMA_INFTY_ER_MODEL().compare(errorModel) == 0) {
        getHammingDistance(fmInputFilename,outputFilename,"maximal");
} else if (ANY_L_GT_M_ER_MODEL().compare(errorModel) == 0) {
        int numStates=inputFm.number_of_states();
```

```
int upperBound= 2 * m * numStates *numStates;
if(isErrorDetecting(errorType,m,upperBound,inputFm.getAlphabet()) {
for(int loopL = m+1; loopL <upperBound + 1; loopL++) {
//Create channel
Transducer myChannel(errorType, m, loopL,inputFm.getAlphabet());
//Intersect with input language on both input and output
Transducer rTransducer = Transducer(myChannel,myFm,"InputOutput");
//Test if the resulting transducer is functional
Transducer<string> U=Transducer();
isFunctional =U.isTransducerFunctional(rTransducer);
if (isFunctional) {
//Get the smallest l it is error-detecting for
maximalL=loopL;
} else {
//stop as soon as we get an l it is not error-detecting for.
break;
}
}
}
} else if (ANY_M_LT_L_ER_MODEL().compare(errorModel) == 0) {
for (int loopM=1; loopM < l; loopM++) {
Transducer myChannel(errorType, loopM, l,inputFm.getAlphabet());
Transducer rTransducer = Transducer(myChannel,myFm,"InputOutput");
Transducer<string> U=Transducer<string>();
isFunctional =U.isTransducerFunctional(rTransducer);
if (isFunctional) {
//Get the largest m it is error-detecting for
```

```
maximalM=loopM;
} else {
//stop as soon as we get an m it is not error-detecting for.
break;
}
}
}
```

6.3.2 Implementations for the Error Model $\mathbb{C}^0_{SID}[\infty]$

As before, this error model uses the edit distance for computing the maximal error-detecting capabilities of a language.

Code Snippet 6.3.1 includes the code for this error model. The error model is shown as `SID_INFTY_ER_MODEL()` in the code snippet.

6.3.3 Implementations for the Error Model $\mathbb{C}^1_\tau[l]$

The error model $\mathbb{C}^1_\tau[l] = \{\tau(m,l)$: for any m with $m < l\}$, described in Section 5.9.3 is shown as `ANY_M_LT_L_ER_MODEL()` in Code Snippet 6.3.1. We compute the $\mathbb{C}^1_\tau[l]$-maximal error-detecting capabilities of L by finding the largest $m < l$ for which the language is error-detecting.

6.3.4 Implementations for the Error Model $\mathbb{C}^2_\tau[m]$

For this error model, $\mathbb{C}^2_\tau[m] = \{\tau(m,l)$: for any l with $l > m\}$, Code Snippet 6.3.1 shows it as `ANY_L_GT_M_ER_MODEL()`.

Recall that the goal in this error model is to find the smallest l for which a language L is error-detecting for the channel $\tau(m, l)$. We start by testing if L is error-detecting for the upper bound for l, as presented in Theorem 5.9.1. That is, create the channel $\tau(m, 2ms^2)$, where s^2 is the number of states in the input finite machine, and test if L is error-detecting for it. If it is, then start a loop for the values of l in the range $(m+1)\ldots 2ms^2$. For each l the process proceeds by testing if L is error-detecting for the current channel. The process stops when the first l is found for which L is not error-detecting for the channel $\tau(m, l)$, and the channel $\tau(m, l-1)$ is the maximal error-detecting capability of L for the error model.

6.4 Testing the System

We tested all the algorithms in our implementation for correctness. Some of the test data we used can be accessed via [4]. In addition, we tested the main algorithms on three sequences $E_b(n), M_b(n), L_0(n)$ of automata of increasing size, that is, $|E_b(n)| < |E_b(n+1)|$, $|M_b(n)| < |M_b(n+1)|$, and $|L_0(n)| < |L_0(n+1)|$.

The main algorithms in this thesis are deciding error-detection (DED), computing Hamming distance (CHD), computing edit distance (CED) and computing maximal error-detection (CMED). Next we describe the test automata. After that we present our observed results, in terms of processing times, in Section 6.4.2.

6.4.1 Description of Test Input

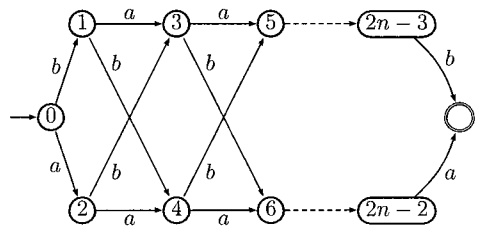

Figure 6.3: Depiction of $E_b(n)$

The DFA in Figure 6.3 accepts the even parity code of length n. Even numbered states mean that the number of b's seen is even. Odd numbered states mean that the number of b's seen is odd. Every accepted word has an even number of b's. This is enforced in the last two transitions to the final state. This is the most well known code for detecting 1 substitution error per word (channel $\sigma(1, \infty)$).

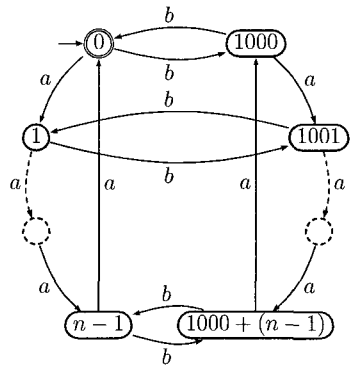

Figure 6.4: Depiction of $M_b(n)$

For the DFA in Figure 6.4, every accepted word has an even number of b's and a number of a's equal to 0(mod n). States $r = 0, \ldots, n-1$ mean that the machine has seen an even number of b's and a number of a's equal to r(mod n). States $1000 + r$ mean that the machine has seen an odd number of b's and a number of a's equal to r(mod n). As an example, Figure 6.5 shows a depiction of $M_b(3)$.

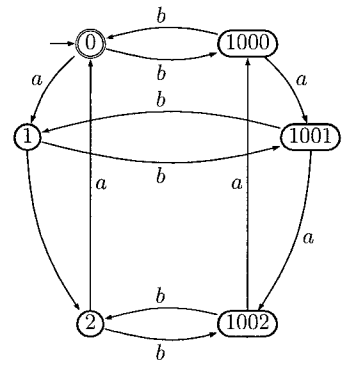

Figure 6.5: Depiction of $M_b(3)$

The third sequence $L_0(n)$ represents Levenshtein's codes [30] that are error-detecting for $(\iota \odot \delta)(2, \infty)$. These are binary codes of length n defined as follows:

$$L_0(n) = \{b_1 \ldots b_n \mid \left(\sum_{i=1}^{n} i b_i\right) \equiv 0(\text{mod } n+1) \}$$

If we use a for 0, and b for 1, we can define a sequence of automata accepting $L_0(n)$ as follows. For each $n \leq 3$:

- The start state is $(0, 0)$

- The final state is $(n, 0)$
- All other states are (i, s) with $1 \leq i \leq n-1$ and $0 \leq s \leq n$. (s is the sum so far modulo $(n+1)$).

Figure 6.6 below shows a depiction of $L_0(n)$ and as an example, Figure 6.7 shows a depiction of $L_0(4)$.

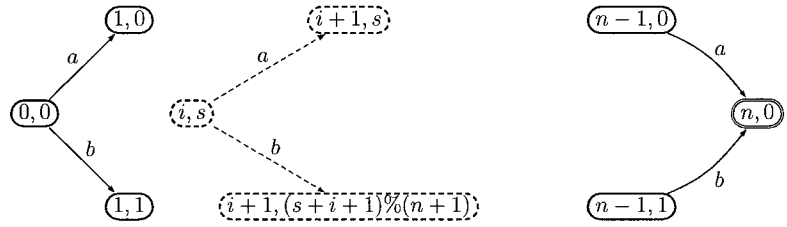

Figure 6.6: Depiction of $L_0(n)$

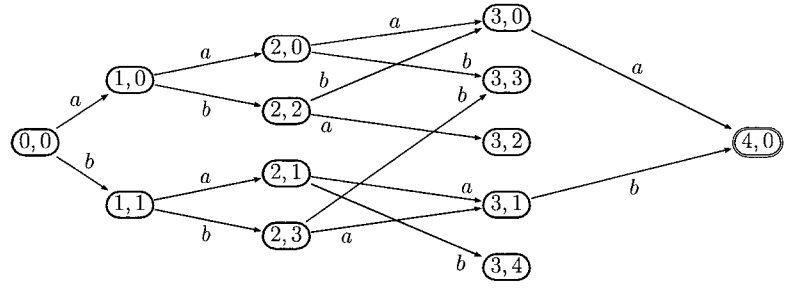

Figure 6.7: Depiction of $L_0(4)$

Figure 6.8 shows two equivalent channels that can both introduce at most one transposition error. Transpose 1a uses substitutions and Transpose 1b uses a combination of a deletion and a special form of insertion.

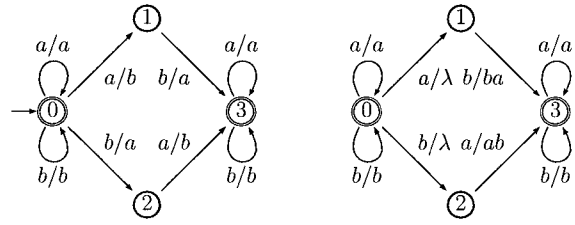

Figure 6.8: Depiction of Transpose 1a and Transpose 1b

6.4.2 Some Empirical Test Results

Below are the results from our testing. The tests were run on a 1.73 GHz processor with 8Gb of RAM. A table cell with no number in it means that we did not run that particular test or the processing time was so high that we stopped the test.

DED				
Channel	n	Processing Times by Sequence		
		$E_b(n)$	$M_b(n)$	$L_\theta(n)$
$\sigma(1,\infty)$	5	1 s	2 s	11 s
	10	21 s	33 s	6060 s
	15	122 s	169 s	
	20	508 s	648 s	
	25	1142 s	1770 s	
$(\iota \odot \delta)(1,\infty)$	5	40 s	59 s	181 s
	10	929 s	759 s	
	15	4775 s	5873 s	
	20	16635 s	20313 s	
	25	36392 s	37124 s	
Transpose 1a	5	11 s	84 s	11 s
	10	603 s	1284 s	9212 s
	15	4697 s	7528 s	
	20	15438 s	25103 s	
	25	32920 s	54954 s	
Transpose 1b	5	11s	24 s	44 s
	10	209 s	364 s	37715 s
	15	2841 s	3998 s	
	20	8040 s	10992 s	
	25	20164 s	54979 s	

Table 6.2: Processing times for Deciding Error-Detection

6.4.3 Testing Summary

We tested the main algorithms using sequences of automata that accept three classes of codes, $E_b(n), M_b(n), L_0(n)$. The tests helped to validate the correctness of our implementation. In addition, they provided us with some insights on performance. The processing times increased with the size of the input automata. As the running times of these tests are quite high, we did not do further tests on larger automata, in particular for the Maximal Error-Detection Capability problem.

CMED				
Error Model/Parameters	n	Processing Times by Sequence		
		$E_b(n)$	$M_b(n)$	$L_\theta(n)$
$\mathbb{C}^0_\sigma[\infty]$ (Uses CHD)	5	0 s	13 s	1 s
	10	0 s	19 s	48 s
	15	1 s	174 s	794 s
$\mathbb{C}^0_{\sigma\odot\iota\odot\delta}[\infty]$ (Uses CED)	5	14 s		30 s
	10	206 s		4368 s
$\mathbb{C}^1_\sigma[l] = \sigma(m,l)$, with l=3	5	23 s		29 s
	10	4379 s		
$\mathbb{C}^2_\sigma[m] = \{\sigma(m,l)$, with m=2	5	388 s		232 s
	10			

Table 6.3: Processing times for Computing Maximal Error-Detecting Capability

As the channel Transpose 1a involves no insertions and deletions, the transducer functionality algorithm does not do the pseudo-sequential conversions—see Section 5.8—for this type of error-detection, whereas in the case of the channel Transpose 1b, these conversions are necessary. Despite this, the running times for the tests involving Transpose 1a are higher than those where Transpose 1b is involved. By looking at the execution logs of these tests we found that the product machines U constructed in the tests involving Transpose 1a were much larger than the corresponding product machines in the tests for Transpose 1b. In addition, the execution logs revealed that most of the processing time was spent on intersections and deciding functionality.

We have made 120 of the tests we ran available via [3]. These tests can be downloaded and run locally.

6.5 Interacting with the System

As mentioned earlier, we have created a web interface accessible via [4] for interacting with the tools we developed. Some of the computations such as computing the edit distance, intersecting transducers on input and output, and transducer functionality are computationally expensive and tend to take longer computation time as you increase the input size. Hence, for larger input we advise downloading the source code via [3] and running the code from the console. When the input from the web interface is large and the computation takes time, the browser request might be timed out and the computation stopped by the web server. On the other hand, running the program from the console ensures that it runs to the end even for extended computation times. The source code root folder has a `readme.txt` file with instructions on how to compile and run the program from the console for both Linux and Windows platforms. In addition, the source code root has a folder called `testData` with some test input data.

Chapter 7

Conclusions and Future Work

In this thesis, we investigated and implemented several existing and new concepts and algorithms related to the computation of error-detecting capabilities of regular languages. In this chapter, we conclude our findings and share our thoughts for possible future work.

7.1 Conclusions

As part of this research, we investigated the existing literature for related concepts and algorithms. We found some interesting concepts that helped us, and are worth studying on their own. For example, we found the algorithms for deciding transducer functionality really intriguing, and adapted one to work with sequential machines.

We modeled SID channels using sequential machines in Chapter 4, and to our knowledge, this has never been done before. Furthermore, we extended the channel constructions using sequential machines to homophonic channels which are less

expensive to construct compared to SID channels.

In Chapter 5, we adapted some existing algorithms and applied them to regular languages. We used the existing and our new algorithms for the purpose of computing error-detecting capabilities. Some of the algorithms included computing the Hamming distance and edit distance, intersecting a transducer on the input and output, and deciding transducer functionality with sequential machines. We introduced a new type of sequential machine called pseudo-sequential and showed how to use it as part of deciding the error-detection property for a regular language. In addition, in Section 5.9 we systematically and collectively applied all the existing and new algorithms, and showed how to implement the algorithms that compute the maximal error-detecting capabilities of a regular language for a several error models.

A large part of our research was implementing, interacting and testing the algorithms we were dealing with. Our implementation was substantial and could be completely documented in this thesis without dramatically increasing the size. Nonetheless, we used Chapter 6 to provide an insight into our implementation. We learnt that implementing these algorithms is not a trivial matter, especially if the algorithm is given at a very high or theoretical level. We utilized the Grail C++ library for automata. Furthermore, we created a web interface for interacting with the system we implemented. Finally, we made the source code from our implementation available to the research community. It can be downloaded via the web interface we created.

7.2 Future Work

This research work taught us a number of things and gave us some ideas on the direction we can take after this.

Working with Grail showed us its strengths and weaknesses. Improvements that can be made to Grail include updating the code to newer versions and making the library work with several input and output mechanisms such as relational databases. In addition, the library could be updated to newer C++ standards so that it is easily portable to different platforms.

The algorithms we implemented were expensive in terms of computational resources. Hence, this limited the size of machines they could reasonably work with. In the future we could rewrite the implementation to be run on high performance computing platforms such as ACEnet. This would let us work with larger automata and would let us submit a job which might need to run for an extended period of time.

We applied the new concept of computing maximal error-detecting capabilities of languages to regular languages. In addition, we limited our focus to deterministic finite automata, and looked at a few error models. In the future we could expand the scope further to include non-deterministic automata, and other error models.

We showed how to construct sequential machines realizing Homophonic channels, but did not implement them. These are theoretically less expensive to construct

and can prove useful in estimating error-detecting capabilities of regular languages. Future work can target implementing and investigating these further.

As with any software implementation, there is always room for improvement. Obvious future improvements to our system would be increasing its efficiency, performance, robustness, usability and improving the processing times.

In closing, we wish to express our belief that the technical contributions of this thesis bring us one step closer to the realization of a practical software capable of evaluating the quality of real-world languages in terms of their error-detecting capabilities.

Bibliography

[1] Marie-Pierre Béal, Olivier Carton, Christophe Prieur, and Jacques Sakarovitch, *Squaring transducers: an efficient procedure for deciding functionality and sequentiality*, Theor. Comput. Sci. **292** (2003), no. 1, 45–63.

[2] R. M. Capocelli, L. Gargano, and U. Vaccaro, *Decoders with initial state invariance for multivalued encodings*, Theoretical Computer Science **86** (1991), no. 2, 365–375.

[3] A. Daka, *Implementation source code*, `http://research1.cs.smu.ca/~a_daka/downloads.php`, 2011.

[4] ———, *Thesis companion website*, `http://research1.cs.smu.ca/~a_daka/`, 2011.

[5] A. Daka and S. Konstantinidis, *Refinement and implementation of algorithms for deciding the error-detection property*, Technical Report 002, Mathematics and Computing Science, Saint Mary's University, Canada (2011).

[6] E. W. Dijkstra, *A note on two problems in connection with graphs*, Numerische Mathematik **1** (1959), no. 60, 269–27.

[7] W. B. Frakes and C. J. Fox, *Strength and similarity of affix removal stemming algorithms*, ACM SIGIR Forum **37** (2003), no. 1, 26–30.

[8] E. M. Gurari and O. H. Ibarra, *A note on finite-valued and finitely ambiguous transducers*, Mathematical Systems Theory **16** (1983), 61–66.

[9] R. W. Hamming, *Error detecting and correcting codes*, The Bell System Technical Journal **XXVI** (1950), no. 2, 147–160.

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

[11] M. Holzer and M. Kutrib, *State complexity of basic operations on non deterministic finite automata*, Implementation and Application of Automata, 7th International Conference, CIAA (2002), 148157.

[12] ———, *Non deterministic descriptional complexity of regular languages.*, International Journal of Foundations of Computer Science **14** (2003), no. 6, 1087–1102.

[13] J. E. Hopcroft and J. D. Ullman, *Introduction to automata theory, languages, and computation*, Addison-Wesley, Reading MA, 1979.

[14] G. Jirskov, *State complexity of some operations on binary regular languages*, Theoretical Computer Science **330** (2005), no. 2, 287–298.

[15] jQuery, `http://www.jQuery.com`, 2011.

[16] H. Jrgensen and S. Konstantinidis, *Codes*, Handbook of formal languages, vol. 1, Springer-Verlag, New York, 1997.

[17] L. Kari and S. Konstantinidis, *Descriptional complexity of error/edit systems*, Pre-Proceedings of Descriptional Complexity of Formal Systems (2002), 133–147.

[18] ———, *Language equations, maximality and error detection*, Journal of Computer and System Sciences **70** (2005), no. 1, 157–178.

[19] L. Kari, S. Konstantinidis, S. Perron, G. Wozniak, and J. Xu, *Finite-state error/edit-systems and difference-measures for languages and words*, Technical Report 001, Department of Math and Computing Science, Saint Mary's University (2003).

[20] ———, *Computing the Hamming distance of a regular language in quadratic time*, WSEAS Transactions on Information Science and Applications **1** (2004), 445449.

[21] 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** (1999), no. 1, 60–77.

[22] ———, *An algebra of discrete channels that involve combinations of three basic error types*, Information and Computation **167** (2001), no. 2, 120–131.

[23] ———, *On the decidability of the error-detection property*, Technical Report 003, Saint Mary's University (2001).

[24] ———, *Transducers and the properties of error-detection, error-correction, and finite-delay decodability*, Journal of Universal Computer Science **8** (2002), no. 2, 278–291.

[25] ———, *Computing the Levenshtein distance of a regular language*, Information Theory Workshop, IEEE (2005), 108–111.

[26] S. Konstantinidis and P. V. Silva, *Maximal error-detecting capabilities of a formal language*, Journal of Automata, Languages and Combinatorics **13** (2008), no. 1.

[27] Stavros Konstantinidis, *Computing the edit distance of a regular language*, Inf. Comput. **205** (2007), no. 9, 1307–1316.

[28] J. B. Kruskal, *An overview of sequence comparison: Time warps, string edits, and macromolecules*, SIAM Review **25** (1983), no. 2, 201–237.

[29] M. V. Lawson, *Finite automata*, CRC Press, 2003.

[30] V. I. Levenshtein, *Binary codes capable of correcting deletions, insertions and reversals*, Soviet Physics Doklady **10** (1966), no. 8, 707710.

[31] B. H. Marcus, R. M. Roth, and P. H. Siegel, *An introduction to coding for constrained systems*, Lecture Notes (2001).

[32] R. McCloskey, *An $O(n^2)$ time algorithm for deciding whether a regular language is a code*, Journal of Computing and Information **2** (1996), 7989.

[33] E. W. Myers, *An overview of sequence comparison algorithms in molecular biology*, University of Arizona, Dept. of Computer Science, Arizona, 1991.

[34] PHP, `http://www.php.net`, 2011.

[35] D. Raymond, *Grail: A C++ library for finite-state machines and regular expressions*, Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research (1994).

[36] D. R. Raymond and D. Wood, *Grail: A C++ library for automata and expressions*, JSC **17** (1994), no. 4, 341–350.

[37] ———, *The Grail papers: version 2.3*, (1995).

[38] G. Rozenberg and A. Salomaa, *eds. Handbook of Formal Languages*, vol. 1, Springer-Verlag, Berlin, 1997.

[39] S. Yu S. Konstantinidis, N. Santean, *On implementing recognizable transductions*, International Journal of Computer Mathematics **87** (2010), 260–277.

[40] S. S. Skiena, *The algorithm design manual*, Springer, 1998.

[41] J. Xu, *Formalization of error models with application to spelling correction*, Masters thesis, Department of Math and Computing Science, Saint Mary's University (2004).

[42] S. Yu, *Regular languages*, Handbook of Formal Languages, vol. 1, ch. 2, pp. 41–110, Springer Verlag, 1997.

[43] ———, *State complexity of regular languages*, Journal of Automata, Languages and Combinatorics **6** (2001), no. 2, 221.

[44] S. Yu and Q. Zhuang, *On thè state complexity of intersection of regular languages*, ACM SIGACT News **22** (1991), no. 3, 52–54.