Error-Correction and Finite Transductions

Christopher L. McAloney

March 2004
External Technical Report
ISSN-0836-0227-
2004-476

Department of Computing and Information Science
Queen's University
Kingston, Ontario, Canada K7L 3N6

Document prepared March 11, 2004

Abstract

Recently, there has been a renewed interest in the detection and the correction of errors which occur in data sent across noisy communication channels. We consider the problems of error-detection and error-correction from the perspective of formal languages, in which the transmitted data are strings of symbols from an arbitrary alphabet. We discuss in detail a particular class of error channels, the SID channels, in which the channel may cause up to k substitution, insertion or deletion errors in any l consecutive symbols of the input word.

In this thesis, we will examine the SID channels in some depth and, using tools from metric analysis, define a measure of distance which can be used formally to measure the number of errors between any pair of words. We then show, through a generalized finite transducer construction for SID channels that, provided the input language for the channel is regular, the union of the neighbourhoods of words in the language with respect to our distance measure is also regular and thus the output language from the channel is itself a regular language.

As motivation for our work, we discuss the Statecharts language in depth and define a written notation which can be used to formally discuss a restricted class of statecharts. Using this notation, we provide a statechart construction which can be used to accept the language output by a particular type or error channel.

Acknowledgements

I would like to thank my supervisor, Dr. Kai Salomaa, without whose guidance and support this thesis would not have been possible.

Many thanks to Amanda and my parents, Nancy and William McAloney, for their support and encouragement, and to all others who have influenced my thoughts over the course of the writing of this thesis, from conception to completion.

1 Introduction

Automatic error-correction techniques have been receiving a lot of attention in recent years. With the rapid overpopulation of the internet, much research has been devoted to finding new ways in which large amounts of information can be accurately transmitted between two points. As more offices look into converting their enormous quantities of paper records into a more manageable electronic format, error-correction methods have been incorporated into diagram recognition and optical character recognition fields [24]. In addition, new developments in networking research, most notably those in wireless communications [6], call for substantially stronger error-correction methods than were previously needed.

The problem of error-correction arises in many areas of computer science research, particularly those attempting to facilitate the interactions between humans and computers such as the various branches of diagram recognition or natural language recognition and generation. Note the distinction between error-detection and error-correction: the former merely detects that an error exists in the input[1] while the latter selects a word in the language in question to replace the word in which the error was detected.

Several error-detection and -correction methods, ranging from heuristic and probabilistic approaches to purely language theoretic ones, exist. Many of these heuristic approaches and probabilistic approaches carry the disadvantage that they require *a priori* knowledge of the language (either formal or natural) in question, and the error-correction techniques are based on analysis of trends which occur in the languages [24].

However, if this information isn't available, then these methods have little use and we are forced to rely on a more theoretical and general approach. The first and most common of these alternative methods is that of coding. The theory of codes contains very strong error-correction results, and has been extensively treated in the literature; see [2, 8, 26, 29] for some textbook references on codes in general, and error-correcting codes specifically. Coding usually takes the form of *block codes*, in which every word in the code is of the same length.

When dealing with errors in words transmitted across some error channel, there are three basic types of errors which can be introduced into the transmitted word: substitutions of one symbol in the relevant alphabet with another, insertion of new symbols into the transmitted word, and deletion of symbols which occur in the transmitted word. However, both the

[1] In many applications, such as the communication protocols used for communications over the internet today, this is sufficient, since the computer receiving the message can request that the sender re-transmit the word.

insertion and the deletion error-types change the length of the word in which these errors occur, so, if we assume that the length of the word does not change during transmission, substitution errors are the only errors which can be considered.

Since almost all of the research concerning codes has used block codes, most of the results in this field concern only substitution type errors with relatively few results for the deletion and insertion error types (see [25, 9, 14, 30, 15] for examples of the latter). In the more general case, however, when we are dealing with arbitrary data in which the words need not all be of the same length, we must also consider the insertion and deletion cases, which means that we can no longer rely on block codes which assume that the length of the word does not change. Thus, for our purposes, the vast majority of coding-theoretic results are not directly applicable, and we are forced to use a different approach.

This thesis will examine the problems of error-detection and error-correction from the perspective of formal languages and automata theory. We will use, in this discussion, tools from various diverse areas including coding theory [9, 25], metric analysis [7, 4, 3, 28] and a language theoretic view of error channels [17, 23, 21, 20] to develop a new and useful distance measure for use on words over an arbitrary alphabet and a means for which it can be applied to a certain class of error channels.

The major contributions of this thesis are:

- a formalization of the notion of SID channels found in the work of Konstantinidis and Jürgensen [23, 21, 20, 19, 22, 17] which results in the pseudometric definition found in §5.3.

- a generalized finite transducer construction (Theorem 5.2) which, for a given language, produces as output the set of all words which are within some error bound k of some word in the original language with respect to the pseudometric defined in §5.3. Similar results, with less complete proofs, have been independently obtained in [23].

- a written notation which can be used to formally discuss a restricted class of Statecharts, as defined by D. Harel in [13, 11, 10, 12] (§4.2 and §4.3); and

- a procedure to construct a statechart which, given a regular language used as input to the error channel, can recognize the language received as output from the error channel, for a certain type of error channel (Theorem 4.1).

1.1 Chapter Summary

The chapters of this thesis are organized as follows:

- in Chapter 2, we present some background and preliminary definitions which we will require in our language theoretic discussions, and present a survey of relevant results in the area of metric analysis as related to formal language theory.

- Chapter 3 contains a survey of definitions and results related to or required by this thesis, mostly drawn from the work of Jürgensen and Konstantinidis [23, 21, 20, 19, 22, 17]. In particular, we will formalize the notion of an error channel and discuss in detail a particular class of error channels, known as SID channels.

- in Chapter 4 we will define and discuss in some detail a subset of the Statecharts notation introduced by David Harel, and we will define a formal, written notation of a restricted class of the Statecharts language which can be used to formalize some properties of Statecharts. In addition, we will provide, as an extended example, a definition of a particular type of error channel, and define a method whereby a statechart can be constructed to accept the output language from the error channel, given that the input language is regular.

- Chapter 5 gives a definition of what we call a *pseudometric* which can be used to measure distances between words over an arbitrary alphabet which mimics the behaviour of the SID channels. As our main result, we provide a transducer construction which mimics the behaviour of some SID channels.

- Chapter 6 summarizes the results of the thesis and suggests several directions for possible future work.

2 Preliminaries

2.1 Set Theory and Formal Languages

In this thesis, we will make extensive use of notations and concepts from set theory, logic, and formal language theory. All of these are relatively standard notations in the literature of mathematics and computer science.

For a *set* A, we denote the cardinality of A by $|A|$. For an element a and a set A, we say $a \in A$ if a is a member of A. If A and B are sets, we write $A \subseteq B$ and call A a *subset* of B if every member of A is also a member of B. The *empty set*, denoted $\emptyset$, is the (unique) set with no members, and is trivially a subset of every set. We will denote $\{0, 1, 2, 3, \ldots\}$, the set of natural numbers, by $\mathbb{N}$; the set of integers, $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$, by $\mathbb{Z}$; and the set of real numbers $(-\infty, \infty)$ by $\mathbb{R}$.

We define sets, usually in terms of other sets, by the following notations:

- $\{x \mid \phi(x)\ \}$, where ϕ is some property which is either true or false, is the set of all elements x which satisfy ϕ;

- We use the standard notation for the basic set-theoretic operations of union ($\cup$), intersection ($\cap$) and set difference ($\setminus$).

- $A \times B$, the *Cartesian product* of A and B, is the set of all ordered pairs (a, b) such that $a \in A$ and $b \in B$. For a set S of ordered tuples of the form $(x_1, \ldots, x_n)$, we define the *projections* of S as $\pi_i(S) = \{x_i \mid (x_1, \ldots, x_i, \ldots, x_n) \in S\}$, for $1 \leq i \leq n$.

- $\mathcal{P}(A)$, called the *power set* of A for some set A, is the set of all subsets of A.

2.1 Set Theory and Formal Languages

An *alphabet*, Σ, is a finite non-empty set of symbols. A *word* over an alphabet Σ is a mapping $w : I_w \longrightarrow \Sigma$ such that Σ is the alphabet and $I_w \subseteq \mathbb{N}, I_w = \{0, \ldots, n-1\}$, for some $n \geq 0$, is a finite index set. Intuitively, then, a word is a sequence of alphabet symbols (which we will sometimes refer to as a *string*). Concatenation of words is defined in the natural way, and a word w is usually denoted by the juxtaposition of its alphabet symbols. If U and V are two sets of words over some alphabet Σ, we define the concatenation of U and V by:

$$UV = \{ \; uv \mid u \in U, \text{ and } v \in V \; \}$$

On many occasions, we will need to refer to the *length* of a word w, which is equivalent to the cardinality of its index set. We denote the length of w by $|w|$, where $|w| = |I_w|$. The *empty word* (denoted λ) is roughly analogous to the empty set — λ is the (unique) word such that $I_\lambda = \emptyset$. The set of all words of finite length over Σ is denoted by Σ^* and $\Sigma^+ = \Sigma^* \backslash \{\lambda\}$. A subset L of Σ^* is called a *language*.

Definition 2.1 *A* ***deterministic finite automaton*** *(often abbreviated DFA), M, is a five-tuple $M = (Q, \Sigma, \gamma, q_0, F)$ where*

$$\begin{aligned} Q &: \textit{ is the finite set of states;} \\ \Sigma &: \textit{ is the input alphabet;} \\ \gamma &: \textit{ is the transition function;} \\ q_0 \in Q &: \textit{ is the (unique) initial state; and} \\ F \subseteq Q &: \textit{ is the set of final states.} \end{aligned}$$

The transition function, $\gamma : Q \times \Sigma \longrightarrow Q$ maps a tuple consisting of a state and an alphabet symbol onto a new state. The name "transition function" indicates that the state of the DFA changes according to the alphabet symbols read.

The word "deterministic" in the name of the DFA indicates that for any ordered pair of a state and an alphabet symbol, the next state of the DFA is uniquely determined. This opens the door to an obvious generalization, namely that of the *nondeterministic finite automaton*.

Definition 2.2 *A* ***nondeterministic finite automaton (or NFA)*** *is a five-tuple $M = (Q, \Sigma, \gamma, q_0, F)$ with Q, Σ, q_0 and F defined as in a DFA. The transition function, γ, is defined as $\gamma : Q \times \Sigma \longrightarrow \mathcal{P}(Q)$. That is, instead of mapping a state and an input to one state, γ in an NFA is a mapping from a state and an input to a set of states.*

One further generalization of the NFA can now be made — that of the *NFA with lambda transitions* (or more commonly, λ-*NFA*). The λ-NFA is defined identically to the NFA with the exception that transitions on an empty input are now admitted; that is, $\gamma : Q \times (\Sigma \cup \{\lambda\}) \rightarrow \mathcal{P}(Q)$. Put another way, a λ-NFA is able to make a transition to a new state without reading an input symbol.

The advantage of using ($\lambda-$)NFAs over DFAs is primarily one of state complexity; the number of states required in a machine that accepts the given language. We will see in Thm 2.1 that every language which is accepted by a ($\lambda-$)NFA is also accepted by a DFA.

However, for a given NFA, M, with n states, the number of states required by a DFA accepting $L(M)$ could potentially be as many as 2^n.

Although the transition function for finite automata is defined on individual alphabet symbols, it can be extended in a natural way to be defined on words over the input alphabet, Σ. We define this extension recursively:

Definition 2.3 *The* ***extended transition function*** $\gamma^*(q_i, w)$*, for* $q_i \in Q$ *and* $w \in \Sigma^*$*, is defined as follows:*

- $\gamma^*(q_i, \lambda) = q_i \cup \gamma^*(q_j, \lambda)$*, for all* $q_j \in \gamma(q_i, \lambda)$
- *For* $w = au$*, where* $a \in \Sigma$ *and* $u \in \Sigma^*$*,* $\gamma^*(q_i, w) = \gamma^*(\ \gamma(q_i, a),\ u)\ \cup\ \gamma^*(q_i, \lambda)$

Finite automata are typically employed as acceptors of languages in the following sense: if, after reading an input word w, the machine M resides in a final state, then we say that w is in the *language accepted by* M, which we call $L(M)$. More formally, for a DFA, $w \in L(M)$ if, and only if, $\gamma^*(q_0, w) \in F$. This situation is slightly more complicated in the nondeterministic case, since for any given symbol of the input word, the next state is not necessarily unique — there are potentially many possible transitions. In the nondeterministic case, then, we say that w is accepted by a machine M if there exists a valid choice of state transitions which leaves M in a final state after reading w. More formally,

$$w \in L(M) \iff \gamma^*(q_0, w) \cap F \neq \emptyset.$$

Pictorially, DFAs or (λ-)NFAs are usually represented as labelled directed graphs with the nodes representing the states of the finite automaton, and an edge from q_i to q_j labelled with $a \in \Sigma$ if and only if $\gamma(q_i, a) = q_j$.

Theorem 2.1 *Let* Σ *be some finite alphabet, and let* $L \subseteq \Sigma^*$*. Then there exists a* $(\lambda-)$*NFA,* M*, accepting* L *if and only if there exists a DFA,* M'*, accepting* L*.*

A proof of the above theorem may be found in any introductory textbook on formal language theory (see, for example, [35, 32]). We will call a language $L \subseteq \Sigma^*$ *regular* if, and only if, there exists a DFA M with $L(M) = L$.

2.2 Finite Transductions

Finite automata are very useful tools in many aspects of theoretical computer science, but they have several restrictions. One of these is that they are only able to recognize a restricted class of the possible languages on a given alphabet and, although many alternate models exist to address this restriction, it is not the intent of this thesis to discuss these models. The usefulness of finite automata is due to the fact that their properties are very well known and are algorithmically decidable. However, another limitation of finite automata, their inability to provide output, restricts their usefulness for our purposes. Strictly speaking, the output of a DFA or an NFA is merely a "yes" or a "no" answer, depending on whether or not the machine resides in a final state after reading the input word. In many applications, the ability to provide more meaningful output is required. In this section, we will discuss an extension of the finite automaton model called a *finite state transducer*, or merely a *transducer*. Transducers, as we will see, have the important property that they preserve regularity.

Simply put, a transducer is a finite automaton with output. In addition to a transition function, a transducer also has an *output function* which maps a state and an input to a set of possible outputs and states.

2.2.1 Rational Transductions

We now present some preliminary definitions and provide a discussion of the theoretical importance of transductions. The following algebraic concepts are necessary for our development of a formal definition of a transduction.

A *semigroup* consists of a set M and an associative binary operation on M, which is usually denoted by multiplication. A *unit* or *identity element* is an element $1_M \in M$ such that, for any $m \in M$, $1_M m = m 1_M = m$.

Definition 2.4 *A semigroup which has an identity element is a* ***monoid****. For any set X, the* ***free monoid*** *X^*, with concatenation as the semigroup operation, is defined by*

$$X^* = \bigcup_{n \geq 0} X^n$$

where $X^0 = \{1_{X^*}\}$. Thus, the elements of a free monoid X^* are n-tuples of elements of X.

Example 2.1 *Let Σ be a finite alphabet. Then, Σ^*, with concatenation as the binary operation, and λ as the identity element, is a free monoid.*

Note that the identity element of a monoid is necessarily unique since, if 1 and $1'$ are both identity elements, then $1' = 11' = 1$. Strictly speaking, the elements of a free monoid X^* are n-tuples of the form $u = (x_1, x_2, \ldots, x_n)$. We will, however, write them by simply juxtaposing the elements of the n-tuple to make clear their relation to what we have been referring to as "words". Thus, u above, becomes $u = x_1 x_2 \ldots x_n$.

Theorem 2.2 [1] *Let X and Y be monoids. Then $X \times Y$ is a monoid with multiplication defined by $(x, y)(x', y') = (xx', yy')$.*

Definition 2.5 *Let M be a monoid. The family $Rat(M)$ of* ***rational subsets*** *of M is the least family $\mathcal{R}$ of subsets of M satisfying the following conditions:*

i) $\emptyset \in \mathcal{R}$; and $\{m\} \in \mathcal{R}$ for all $m \in M$;

ii) $A, B \in \mathcal{R} \Rightarrow A \cup B, AB \in \mathcal{R}$;

iii) $A \in \mathcal{R} \Rightarrow A^+ = \bigcup_{n \geq 1} A^n \in \mathcal{R}$.

If M is a free monoid, then the rational subsets of M correspond exactly to the regular subsets of M [1, 39]. In other words, if $M = \Sigma^*$, for some Σ, then a language $L \subseteq \Sigma^*$ is a rational subset of Σ^* iff there exists a DFA accepting L. This is not true in the more general case where M is an arbitrary monoid.

Definition 2.6 *Let X and Y be alphabets. A* ***rational relation*** *over X and Y is a rational subset of the monoid $X^* \times Y^*$.*

We are now able to describe relationships between two alphabets in terms of the notion of a rational set. However, our purposes, which involve the incorporation of these notions into a finite automaton, are more "dynamic" than this, so we require two further definitions to transform the static concept of a rational set into the dynamic concept of a rational transduction.

Definition 2.7 *A* ***transduction*** *from X^* into Y^* is a function from X^* into $\mathcal{P}(Y^*)$, the set of subsets of Y^*, which we will denote by $\tau : X^* \longrightarrow Y^*$ for the sake of clarity.*

We define the domain and the image of τ by

$$\begin{aligned} \mathrm{dom}(\tau) &= \{f \in X^* \mid \tau(f) \neq \emptyset\}; \\ \mathrm{im}(\tau) &= \{g \in Y^* \mid \exists f \in X^* : g \in \tau(f)\}. \end{aligned}$$

Definition 2.8 *Let $\tau : X^* \longrightarrow Y^*$ be a transduction, and let $R = \{(f, g) \in X^* \times Y^* \mid g \in \tau(f)\}$ be its* graph*. τ is a* ***rational transduction*** *iff R is a rational relation over X and Y.*

Using rational transductions, we now have a method to "translate" languages between different alphabets, or merely translating words over an alphabet to different words over the same alphabet, while preserving certain important structural features of the language being translated. In the context of this thesis, then, the introduction of various types of errors into an input word will be modelled by this sort of translation. We are now prepared to describe an extension of the finite automaton model which implements these ideas.

2.2.2 Finite-State Transducer

A finite transducer is essentially a string rewriter — a finite automaton with output. In addition to the automaton's input alphabet, the transducer also has an output alphabet. While some authors choose to separate the transition function and the output function in their definition of a finite transducer (see, for example, [5]), we will follow the definition of transducers as presented in [1], in which the state transitions and the outputs are described by a set of *edges*.

2.2 Finite Transductions

Definition 2.9 *A* finite transducer *is a 6-tuple* $(Q, \Sigma, \Delta, \sigma, q_0, F)$ *where*

$$\begin{array}{rcl} Q & : & \textit{is the finite set of states;} \\ \Sigma & : & \textit{is the input alphabet;} \\ \Delta & : & \textit{is the output alphabet;} \\ \sigma & : & \textit{is a finite subset of } Q \times \Sigma^* \times \Delta^* \times Q; \\ q_0 \in Q & : & \textit{is the distinguished start state; and} \\ F \subseteq Q & : & \textit{is the set of final states.} \end{array}$$

Note the format of σ. In the case of the transducer, the transitions are no longer represented by a function but are represented by 4-tuples consisting of a source state, an input word, an output word and a destination state. In general, finite transducers are nondeterministic, since there are no restrictions on the transition-and-output set, σ, which would guarantee determinism. In fact, the transducer construction given in §5.4 relies quite heavily on nondetermism. Finally, we note the increased generality in the definition of the transducer versus the definitions of finite automata: whereas a finite automaton may only read the input word one symbol at a time, the transducer definition allows state transitions based on longer substrings of the input word.

We now present several supplementary definitions for transducers. We will follow the definitions presented in [1]. Given a transducer, T, we define a *computation* of T to be a word ξ of the form

$$(p_1, x_1, y_1, q_1) \cdots (p_n, x_n, y_n, q_n), \text{ where } q_i = p_{i+1} \tag{1}$$

such that each factor (p_i, x_i, y_i, q_i) is in σ. Given a computation ξ of T as in (1), the *label* of ξ is the pair of words $|\xi| = (x, y)$ defined by $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_n$. For states p and q, we define $\Lambda(p, q)$ to be the set of all computations from p to q and

$$\Lambda(p, Q') = \bigcup_{q \in Q'} \Lambda(p, q), \text{ for } Q' \subseteq Q.$$

Finally, for $p, q \in Q$, we define $\mathbf{T}(p, q) = \{|\xi| \mid \xi \in \Lambda(p, q)\}$ and $\mathbf{T}(p, Q') = \{|\xi| \mid \xi \in \Lambda(p, Q')\}$. Thus $\mathbf{T}(p, q)$ is the set of all labels on the paths from p to q — that is, $\mathbf{T}(p, q)$ is the set of pairs of input and output words which form paths from p to q.

Definition 2.10 [1] *Let T be a finite transducer. The transduction $|T| : \Sigma^* \to \Delta^*$* ***realized*** *by T is defined by $|T|(x) = \{y \in \Delta^* \mid (x, y) \in \mathbf{T}(q_0, F)\}$. For a language $L \subseteq \Sigma^*$, where Σ is the input alphabet, we will define $|T|(L) = \{y \in \Delta^* \mid y \in |T|(x), \text{ for some } x \in L\}$.*

Example 2.2 *Figure 1 shows a simple example of the state transition diagram for a transducer $T = (\{q_0, q_1\}, \Sigma, \Sigma, \sigma, q_0, \{q_1\})$, where $\Sigma = \{a, b\}$ and*

$$\sigma = \{(q_0, a, b, q_0), (q_0, a, b, q_1), (q_1, a, a, q_1)\}.$$

The labels are written as u/v for input u and output v. For the input language $L = a^+$ then, $|T|(L) = b^+a^*$.

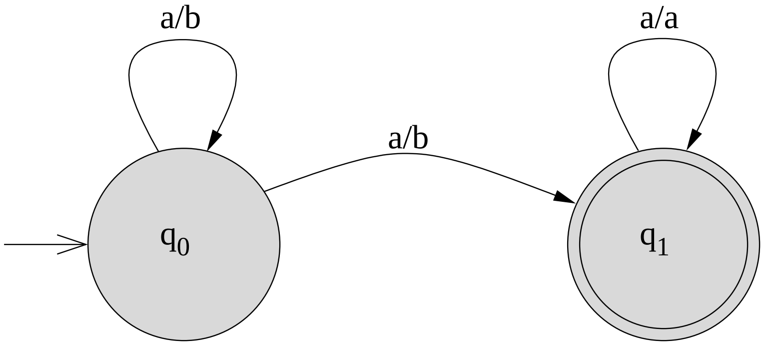

Figure 1: A simple transducer example.

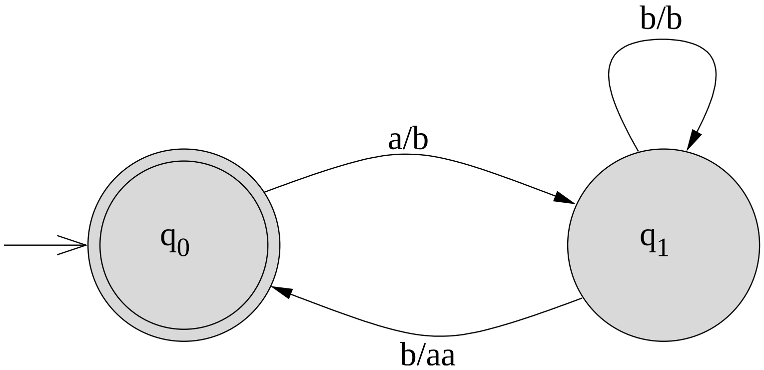

Figure 2: The state transition diagram for the transducer discussed in Ex. 2.3.

Example 2.3 *In Figure 2, we see a slightly more complex transducer example, for a transducer T'. For the input language $L' = (ab^+)^*$, we have that $|T'|(L') = (b^+aa)^*$.*

Theorem 2.3 [1] *A transduction $\tau : X^* \longrightarrow Y^*$ is rational iff τ is realized by a transducer.*

Theorem 2.4 [39] *The family of regular languages is closed under finite transduction.*

The significance of these results should be clear. Using a finite transducer T, we can guarantee that, as long as the input language, L, is regular, then T's output language, which we will denote by $|T|(L)$, is also regular. As was mentioned earlier, we will, in Chapter 5, be using transducers to model the behaviour of error-channels which allow various types of errors to be introduced into words which are transmitted across this channel. The above two results mean that, if an error channel can be modelled by a finite transducer, then for any regular language L a finite automaton can accept the set of words obtained from L by introducing all possible errors.

2.3 Metric Analysis and related results

Metric spaces and metric analysis have long been an integral part of different branches of topology. The techniques developed there have since been applied to various other fields, including the realm of formal language theory. These adaptations of metric analysis to formal languages formed the foundation of certain branches of information theory — coding theory in particular.

Intuitively, a metric, which is defined on a set, is a measure of the distance between various elements of the set. In the field of point-set topology, we can find a very rich theory derived in large part from the notion of a metric and from abstraction of the characteristics of metric spaces (see, for example, [7]).

2.3.1 Introduction and Definitions

We begin this section with some important definitions. For a more detailed discussion of metric spaces, see [7].

Definition 2.11 *Let S be a set. A function $\delta : S \times S \longrightarrow [0, \infty)$ is called a* ***metric*** *if, and only if, $\forall\, x, y, z \in S$, the following conditions hold:*

i) $\delta(x, y) = 0 \Leftrightarrow x = y$

ii) $\delta(x, y) = \delta(y, x)$

iii) $\delta(x, z) \leq \delta(x, y) + \delta(y, z)$

In the event that $\delta : S \times S \rightarrow [0, \infty)$ satisfies i) and ii) but fails to satisfy iii), we will call δ a ***pseudometric****.*

The first condition says that the distance between two elements of S, x and y can only be equal to zero in the event that $x = y$, and in fact *must* equal zero if $x = y$. This is exactly what we would expect, even relying only on the intuitive notion of "distance" since a distance of zero between two objects would imply that they were, in fact, in precisely the same location. The second condition is roughly analogous to the notion of symmetry in a binary relation — it simply states that the distance between x and y does not depend on the order in which we consider x and y. Finally, the third condition is more commonly known as the *triangle inequality*. It is called the "triangle inequality" because, given any three points, x, y, z, in the real plane $\mathbb{R} \times \mathbb{R}$, the distance between x and z can be at most the sum of the distances from x to y and from y to z. Since the Euclidean measures of distance (the "common" distance measures on the real number line and the real plane) form the basis for this definition of a metric, it is fairly obvious why the triangle inequality has been included in the definition of a metric.

In Chapter 5, however, we will require a term for a measure of distance which satisfies the first two requirements of a metric, but fails to meet the restriction of the triangle inequality. We will see that failing to satisfy the triangle inequality doesn't necessarily render a measure of distance useless, and in fact, we will define a very useful measure of distance which fails the third requirement of a metric.

Example 2.4 *Two examples of very commonly used metrics are the standard Euclidean metrics, used on the real number line* $\mathbb{R}$ *and on the real plane* $\mathbb{R} \times \mathbb{R}$.

The distance between two points, $x, y \in \mathbb{R}$, *is usually taken to be* $|x - y|$, *and the distance between two points in the plane,* $(x_1, y_1), (x_2, y_2) \in \mathbb{R}^2$ *is* $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

Next, we discuss the applications of these concepts to formal language theory. A distance measure satisfying the above definition would give us the ability to group related words (or languages) on some alphabet by their relative proximity according to the distance measure. In other words, given a distance which satisfies the above definition, and given some word $u \in \Sigma^*$, we can use this distance measure to obtain a language $U \subseteq \Sigma^*$ consisting of all words over Σ which are "close" (according to the distance measure) to u. In particular, we have the following definition.

Definition 2.12 *Let* S *be a set, and let* δ *be a (pseudo)metric on* S. *For* $K \subseteq S$ *and* $\epsilon \geq 0$, *we define* $E(K, \delta, \epsilon)$ *to be the* ***neighbourhood*** *(or* ***ϵ-neighbourhood****) of* K *of radius* ϵ *with respect to* δ *such that*

$$E(K, \delta, \epsilon) = \{s \in S \mid \exists\, k \in K,\ \delta(k, s) \leq \epsilon\}$$

In essence, this says that the neighbourhood of a set $K \subseteq S$ is the set of all elements of S that are within some error-bound ϵ of the elements of K. For the (not necessarily disjoint) partitioning scheme mentioned above, we can imagine setting some error-bound $\epsilon \geq 0$ and considering the ϵ-neighbourhoods $E(\{w\}, \delta, \epsilon)$ for all $w \in L$. When considering the neighbourhood of a singleton set (as in $E(\{w\}, \delta, \epsilon)$), we will drop the braces and denote the neighbourhood simply by $E(w, \delta, \epsilon)$.

2.3.2 Related Results in Metric Analysis

The adaptation of metric analysis techniques to the field of formal language theory has formed a large part of the foundational structure to the area of algebraic coding theory. In particular, many of the metrics used in formal language theory have been used extensively in measuring the distance between code words in coding theory. In this section, we will discuss in detail an extension of one of these metrics, the Hamming distance [9], and some of its related results.

Definition 2.13 *Let Σ be a finite alphabet. For $a, b \in \Sigma$, define*

$$\Delta(a, b) = \begin{cases} 1 & \text{if } a \neq b \\ 0 & \text{if } a = b \end{cases}$$

*In general, then, we define the **Hamming distance** to be*

$$\Delta_n(x_1x_2 \ldots x_n, y_1y_2 \ldots y_n) = \Sigma_{i=1}^n \Delta(x_i, y_i)$$

Simply put, then, the Hamming distance between two words is the number of positions or indices at which the words differ. It is easy to see that Δ_n satisfies the three conditions in Definition 2.11 and is therefore a metric. The Hamming distance alone, however, is not particularly useful from a formal languages point of view, as it requires that the two words being compared be of the same length. Two variations of the Hamming distance are defined in [4] — one of which will be defined in detail later. Another variation is known as the prefix-Hamming distance since it operates as the Hamming distance does on the two words for the first k letters, where the length of the shortest word is k, and then adds the length of the remaining suffix.

The more useful variant of the Hamming distance discussed here is known as the *shuffle-Hamming* distance.

Definition 2.14 *Let u, v be words in Σ^*. Then the **shuffle** of u and v, $\omega(u, v) \subseteq \Sigma^*$, is the set of all words $x_1y_1 \ldots x_my_m$ such that $u = x_1x_2 \ldots x_m$, $v = y_1y_2 \ldots y_m$, $x_i, y_i \in \Sigma^*$ and $i = 1, \ldots, m$ for $m > 0$.*

We first note that the x_i and y_i in the above definition are elements of Σ^* and they can thus be arbitrary-length or even empty substrings of x and y. Intuitively, then, the shuffle of two words is the set of all words that can be formed by "shuffling" the two words together. That is, for any word $w \in \omega(u, v)$ for some $u, v \in \Sigma^*$, there exist strictly increasing mappings from $I_u \longrightarrow I_w$ and from $I_v \longrightarrow I_w$.

Definition 2.15 *Let u, v be words in Σ^* and let $\#$ be a symbol not appearing in Σ. The **shuffle-Hamming distance** between u and v, denoted $\delta_H(u, v)$, is defined by*

$$\delta_H(u, v) = \min\{\Delta_k(x, y) \mid k \geq \max\{|u|, |v|\}, x \in \omega(u, \#^{k-|u|}), y \in \omega(v, \#^{k-|v|})\}$$

2.3 Metric Analysis and related results

For two words, u and v, the notion of "edit distance" is used in the literature to describe the minimum number of *edit operations* or *edit steps* (i.e. the *insertion* of a new symbol, the *deletion* of a symbol, or the *substitution* of one symbol for another) required to change u into v (see, for example, [30]), and it can be shown that the shuffle-Hamming distance $\delta_H(u, v)$ is the same as the edit distance between u and v [4, 27].

Example 2.5 *For example, consider the two words* abcba *and* bcbab*. The simplest way to convert* abcba *into* bcbab *would be to substitute each symbol in the first word with the symbol in the corresponding position in the second word, for a total of five edit operations. More efficiently, however, we could delete the first symbol from* abcba *and append* b *to the resulting string,* bcba*, for a total of two edit steps.*

So $\delta_H(abcba, bcbab) = 2$, which can be obtained by constructing the intermediate words abcba# *and* #bcbab *and computing their Δ_6 distance.*

Theorem 2.5 [4] *δ_H, as defined above, is a metric.*

Our primary interest in this development is to find a measure of distance which can be applied to words on some finite alphabet, and which is "closed" in the sense that the metric doesn't increase the complexity of the language in question. More precisely, we want a measure of distance, δ, for which we can guarantee that, for any regular language L and for any $\epsilon > 0$, $E(L, \delta, \epsilon)$ is also regular. We call such a distance measure *regularity preserving.* We will see in Chapter 5 that using a metric as a measure of distance is not a necessary condition to guarantee this preservation of regularity, and our next example demonstrates that this alone is also not a sufficient condition.

Example 2.6 *Let $\Sigma = \{a, b\}$. We define a distance δ as follows:*

$$\delta(u, v) = \begin{cases} 0, & \text{if } u = v; \\ \frac{1}{2}, & \text{if } u = a^n b a^n \text{ and } v = a^m b a^m \text{ for some } n, m \geq 0, n \neq m; \\ 1, & \text{otherwise.} \end{cases}$$

It is easy to see that δ satisfies the three conditions in Definition 2.11, so δ is a metric. However, $E\left(aba, \delta, \frac{1}{2}\right) = \{a^n b a^n \mid n \geq 0\}$, which is not a regular language.

Clearly, then, we need to impose additional constraints in order to ensure that a given metric is regularity preserving. We will see shortly that a distance measure which satisfies the conditions of Definition 2.11 as well as an additional property, known as *additivity*, is regularity preserving. The additivity property essentially states that the neighbourhoods induced by an additive metric are in some sense invariant under decompositions of the input word into smaller words. More formally, we have the following definition:

Definition 2.16 *Let δ be a metric. We say that δ is* ***additive*** *if, for $w = w_1 w_2, w_i \in \Sigma^*$, we have, for all $\epsilon \geq 0$,*

$$E(w, \delta, \epsilon) = \bigcup_{\beta_1 + \beta_2 = \epsilon} E(w_1, \delta, \beta_1)\, E(w_2, \delta, \beta_2)$$

Theorem 2.6 [4] *The shuffle-Hamming distance, δ_H, is additive.*

The notion of additivity, combined with the definition of a metric, provides us with the ability to ensure that a given distance preserves regularity.

Theorem 2.7 *Assume that δ is an additive distance on Σ^* and let $L \subseteq \Sigma^*$ be regular. Then $E(L, \delta, \epsilon)$ is regular for all $\epsilon \geq 0$* [4].

The implications of the above theorem are many. There is a large body of information concerning regular languages (see, for example, [39, 32, 35]), and the theorem states that neighbourhoods, with respect to any additive distance, preserve regularity. Perhaps more importantly, from the perspective of error-correction and error-detection, Theorem 2.7 ensures that, with a careful choice of a metric, the set of all words which are "close" to one of the original words is, at least in the sense that both of the languages are regular, no more complex than the original language.

3 SID Channels and Error Functions

3.1 Introduction

There has been a vast quantity of research concerning the properties of block codes — in which every code word over the alphabet Σ has the same length — when they are sent across some noisy error-inducing channel. However, the usage of block codes imposes a severe restriction on what sorts of errors the channel can induce. Specifically, block codes can only comfortably be used with channels that do not change the length of the code words, which rules out the basic error types of insertion and deletion, leaving only substitution errors. Unfortunately, real-world channels tend not to comply with such restrictions.

In order to model the behaviour of channels in which insertion and deletion errors may also occur, we need to drop the constraint that all words in the language be of the same length and consider the behaviour of channels which operate on words of unequal length. In this chapter, we consider a class of error channels called *SID channels* (for *substitution*, *insertion* and *deletion*), a formal model of error channel types which can add or remove symbols to or from a transmitted word as well as substitute symbols in the word for arbitrary symbols from the alphabet Σ. This chapter describes results and definitions concerning SID channels in general, and the specific channel on which we will focus in particular. More formal treatments of these error channels can be found in [21, 23, 17, 20].

3.2 Error Channels

So far, we have been using the word "channel" rather informally to refer to any communications medium through which data are transmitted. While this is essentially correct, we need, for our subsequent discussion, a more formal definition of this notion. In this section, we will formalize the idea of a channel, and discuss useful properties of the languages which are most suited to transmission across error inducing channels.

3.2 Error Channels

Definition 3.1 *Let Σ be a finite alphabet. A* ***(domain preserving) channel****, γ, is a binary relation over Σ^*, $\gamma \subseteq \Sigma^* \times \Sigma^*$ such that $(x, x) \in \gamma$ for all $x \in \{z \mid (z, y) \in \gamma$ for some word $y\}$. We define the sets $Input(\gamma) = \pi_1(\gamma)$ and $Output(\gamma) = \pi_2(\gamma)$, where $\pi_i(\gamma)$ is the ith projection of the relation.*

Thus, the statement that $(y, y') \in \gamma$ indicates that y' is a possible output of the channel γ on input y. For a given channel, γ, we define $\langle y \rangle_\gamma$ to be the set of all possible outputs of γ when y is given as input. Therefore, $\langle y \rangle_\gamma = \{(y, x) \mid x \in \Sigma^*$ and $(y, x) \in \gamma\}$.

It should be clear that a channel satisfying Definition 3.1 will likely require further constraints before it can be useful for the purposes of error correction or detection. A channel which is merely an arbitrary relation over Σ^* could have pairs $(y, y') \in \gamma$ where y' bears no obvious resemblance at all to y. We will consider a restricted class of channels, the *SID channels*, in which, for a given channel γ and input y, all words in $\langle y \rangle_\gamma$ can be constructed from y by some bounded number of substitution, insertion or deletion operations on symbols from y.

Definition 3.2 *Let γ be a channel, Σ a finite alphabet, and $L \subseteq \Sigma^*$ a language over Σ. We say that L is* ***error-detecting*** *for γ if, for all $w_1, w_2 \in L \cup \{\lambda\}$, $(w_1, w_2) \in \gamma$ implies that $w_1 = w_2$.*

Intuitively, then, a language is error-detecting for a given channel if no word in the language can be converted by the channel into another word in the language. As the name indicates, this allows us to determine if an error has occurred during transmission across the channel. If we assume that only words in $L \cup \{\lambda\}$ are input to γ, then the receipt of a word $w \in L \cup \{\lambda\}$ as output from γ guarantees that w is the word that was sent. If, on the other hand, a word w' *not* in $L \cup \{\lambda\}$ is received, then we are assured that an error has occurred during transmission [23]. However, the property of being error-detecting makes no guarantees that the errors which have occurred during transmission across the channel can be corrected. Our next definition provides necessary and sufficient conditions for this case.

Definition 3.3 *Let γ be a channel, Σ a finite alphabet, and let $L \subseteq \Sigma^*$ be a language over Σ. Then L is* ***error-correcting*** *for γ if $(w_1, z), (w_2, z) \in \gamma$ implies that $w_1 = w_2$ for any $z \in \Sigma^*$ and for all $w_1, w_2 \in L \cup \{\lambda\}$.*

There are obvious similarities between error-detecting and error-correcting languages — in fact, Definition 3.3 is a stronger form of Definition 3.2, which we can see simply by letting $z = w_1$ in Definition 3.3. Any error-correcting language, then, is also error-detecting, which is exactly what intuition would suggest, since the notion of error-correction is that of a strengthened form of error-detection. Essentially, the previous definition states that, if a language L is error-correcting for γ, and we assume that only words from $L \cup \{\lambda\}$ are used as input to γ, then for any word z received as output from γ, there is *exactly one* word in $L \cup \{\lambda\}$ which could have been used as input to γ [23].

We note in particular that, once a channel γ is fixed, error-detection and error-correction are properties of *languages* and not of γ. An error-detecting or -correcting language is "sparse" in the sense that, relative to the channel γ, the distance between the words in the language is large enough that the changes in the input word induced by γ are not sufficient to introduce ambiguities between words in the language. We will formalize this notion further in Chapter 5.

3.3 Error Functions and SID Channels

In this section, we will define a specific class of channels, the *SID channels*. An SID channel is a channel which can induce a finite number of substitution, insertion or deletion errors in the input word, where "substitution" is the replacement of a symbol in the input word with another; "insertion" is the addition of symbols between symbols of the input word; and "deletion" is the removal of symbols from the input word.

Definition 3.4 [21] *An* ***SID error type*** *has one of the following forms:*

i) it is a symbol in $\{\epsilon, \sigma, \iota, \delta\}$*; or*

ii) it is an expression $(\tau_1 \odot \tau_2)$*, where* τ_1 *and* τ_2 *are SID error types.*

We will denote the set of SID error types as $\mathfrak{T}_0$*. The symbol* $\odot$ *can be read as "or" in that the error type* $\tau_1 \odot \tau_2$ *represents errors of type* τ_1 *or* τ_2*.*

The symbols ϵ, σ, ι and δ correspond, respectively, to error-free, substitution, insertion and deletion errors. We are now prepared to define the *error functions* [21, 23], which have two distinct interpretations. The first, as the name would imply, is as functions which mimic the behaviour of the error types introduced above. The second interpretation is perhaps the more interesting for our purposes — error functions are themselves words over an alphabet of error function symbols, G.

We first note that any n-symbol word $w = a_0 a_1 \cdots a_{n-1}$ over an alphabet Σ can be equivalently written as a $2n+1$-symbol word over $\Sigma \cup \{\lambda\}$, $w = \lambda a_0 \lambda a_1 \cdots \lambda a_{n-1} \lambda$. Examining the word $w = a_0 a_1 \cdots a_{n-2} a_{n-1} = \lambda a_0 \lambda a_1 \lambda \cdots \lambda a_{n-1} \lambda$, we see that there are n possible symbols which could be deleted, and thus n positions of w at which a deletion could occur. Additionally, there are also n possible positions at which a substitution of symbols could occur and, since an insertion must occur between (or after) symbols of w, there are $n+1$ positions at which an insertion of new symbols into w could occur [21].

Definition 3.5 *Let* Σ *be a finite alphabet. Then an* ***error function symbol*** *is one of the following symbols:*

d: *which symbolizes the* deletion *function* $\mathbf{d} : \Sigma \longrightarrow \{\lambda\}$ *such that* $\mathbf{d}(a) = \lambda$ *for all* $a \in \Sigma$*.*

$\mathbf{i}_u$**:** *which symbolizes an* insertion *function* $\mathbf{i}_u : \{\lambda\} \longrightarrow \{u\}$ *for all* $u \in \Sigma^+$*.*

s: *which symbolizes a* substitution *function* $\mathbf{s} : \Sigma \longrightarrow \Sigma$ *such that* $\mathbf{s}(a) \neq a, \forall a \in \Sigma$*.*

e: *which symbolizes the* error-free *function* $\mathbf{e} : \Sigma \cup \{\lambda\} \longrightarrow \Sigma \cup \{\lambda\}$ *such that* $\mathbf{e}(a) = a, \forall a \in \Sigma \cup \{\lambda\}$.

We recall that these error function symbols are themselves functions which are applied to individual symbols of some word $w \in \Sigma^*$. The error-free function is included more as a placeholder to indicate an absence of errors at the particular symbol(s) of w to which it is applied. We must explicitly disallow arbitrary mappings from $\Sigma \rightarrow \Sigma$ as substitution functions, since mappings such as the identity mapping, which is the same as the error-free function, would be included in this set. We further note that the insertion functions allow insertions of arbitrary words into w, so that there are infinitely many functions of this type. We now define the sets $G_\epsilon, G_\sigma, G_\iota$ and G_δ as follows [21]:

- $G_\epsilon = \{\mathbf{e}\}$
- G_σ is the set of all substitution functions $\mathbf{s}$ from Σ into Σ.
- $G_\iota = \{\mathbf{i}_u \mid u \in \Sigma^+\}$
- $G_\delta = \{\mathbf{d}\}$

Finally, then, we define $G = G_\epsilon \cup G_\sigma \cup G_\iota \cup G_\delta$.

Definition 3.6 *An* ***error function*** *is a word* $\mathbf{h}$ *over* G *such that* $|\mathbf{h}|$ *is odd and, for all* $i \in I_{\mathbf{h}}$,

$$\mathbf{h}(i) \in \begin{cases} G_\epsilon \cup G_\iota, & \text{if } i \text{ is even;} \\ G_\epsilon \cup G_\sigma \cup G_\delta, & \text{if } i \text{ is odd.} \end{cases}$$

We denote the set of all error functions by $\mathcal{H}$.

Note the distinction between the insertion, deletion and substitution operations. Insertion of symbols can only occur *between* the symbols in the input word, while deletion and substitution may only occur at the symbol that they are meant to act upon. This is the reason for rewriting the word w in the form $\lambda a_0 \cdots a_{n-1}\lambda$.

Hence, an error-function, according to the definition above, effectively behaves as a function from Σ^* to Σ^* (through the intermediate translation of the word into its equivalent form as a word over $\Sigma \cup \{\lambda\}$ as mentioned above). The function is applied by individually applying each of the functions in the error-function to the corresponding symbol of the input word. We now define this more formally.

Definition 3.7 *Let* $w = a_0 \cdots a_{n-1}$ *be a word over* Σ^* *for some alphabet,* Σ. *We define* w*'s* ***extended word***, $\bar{w} = \lambda a_0 \lambda \cdots \lambda a_{n-1} \lambda$. *We will denote the ith symbol of* $\bar{w}$ *as* $\bar{w}[i]$ *for all* $i \in I_{\bar{w}}$. *Thus,* $\bar{w} = \bar{w}[0]\bar{w}[1] \cdots \bar{w}[2n]$.

Definition 3.8 *Let* $\bar{w}$ *be the extended word for* w *and let* $\mathbf{h} = \mathbf{h}(0) \cdots \mathbf{h}(2n)$ *be an error function. Then we define*

$$\mathbf{h}(\bar{w}) = \mathbf{h}(0)(\bar{w}[0])\mathbf{h}(1)(\bar{w}[1]) \cdots \mathbf{h}(2n)(\bar{w}[2n])$$

Henceforth, we will denote $\mathbf{h}(\bar{w})$ *by* $\mathbf{h}(w)$ *where there is no ambiguity.*

Example 3.1 *Let* $\Sigma = \{a, b\}$, $w = abba$ *and let* $\mathbf{h} = \mathbf{i}_b\mathbf{eedeseei}_b$. *Then*

$$\begin{aligned}
\mathbf{h}(w) &= \mathbf{h}(\bar{w}) \\
&= \mathbf{h}(\lambda\, a\, \lambda\, b\, \lambda\, b\, \lambda\, a\, \lambda) \\
&= \mathbf{i}_b(\lambda)\, \mathbf{e}(a)\, \mathbf{e}(\lambda)\, \mathbf{d}(b)\, \mathbf{e}(\lambda)\, \mathbf{s}(b)\, \mathbf{e}(\lambda)\, \mathbf{e}(a)\, \mathbf{i}_b(\lambda) \\
&= b\, a\, \lambda\, \lambda\, \lambda\, a\, \lambda\, a\, b \\
&= b\, a\, a\, a\, b
\end{aligned}$$

In this thesis, we will restrict our discussion to one specific type of SID channel, $\sigma \odot \iota \odot \delta(k, l)$, which allows up to k substitution, insertion or deletion errors in any l consecutive symbols of the input word. A full treatment of SID channels can be found in [21] or [17].

Definition 3.9 *Let* Σ *be a finite alphabet,* k *and* l *be integers,* $k, l \geq 0$ *and let* $S_{\sigma\odot\iota\odot\delta} = \{\mathbf{h} \in \mathcal{H} \mid \nu(\mathbf{h}) \leq k\}$, *where* $\nu(\mathbf{h})$ *is the maximum number of substitutions, deletions or insertions occurring in any 2l+1 consecutive symbols of* $\mathbf{h}$. *The SID channel* $\sigma \odot \iota \odot \delta(k, l)$ *is defined to be*

$$\{(x, y) \mid x \in \Sigma^*, y = \mathbf{h}(x), \text{ for some } \mathbf{h} \in S_{\sigma\odot\iota\odot\delta}\}.$$

We will describe the behaviour of this channel much more formally in Chapter 5.

4 An Application to Statecharts

4.1 Introduction and Basic Notation

The Statecharts language[2], first introduced by David Harel in 1984 [10] and since expanded considerably [11, 12, 13], was designed to circumvent some of the issues encountered when trying to model reactive or real-time systems with finite automata. The language has proven to be very helpful in the description, design and implementation of software in general, and has been used extensively in various areas of software engineering. Consequently, Statecharts have been almost fully incorporated into the Unified Modeling Language (UML) and into several software packages based on the UML (Statemate MAGNUM and Rational Rose, for example).

The Statecharts language, with the exception of final states, which will be discussed shortly, can be viewed as a superset of the standard notation for describing deterministic finite automata. With this in mind, we will describe the Statecharts notation with reference to finite automata, and in §4.2 we will define a written notation for Statecharts which is very similar to that used for finite automata.

Statecharts differ from deterministic finite automata in two major ways. The first is that a statechart may have a hierarchical structure — that is, states in a statechart may be nested within other states. The other primary extension is that Statecharts explicitly allow

[2]In this discussion, "the Statecharts language" or "Statecharts" are used to refer to the notation itself, while "statechart" is used to refer to a specific instance of the notation.

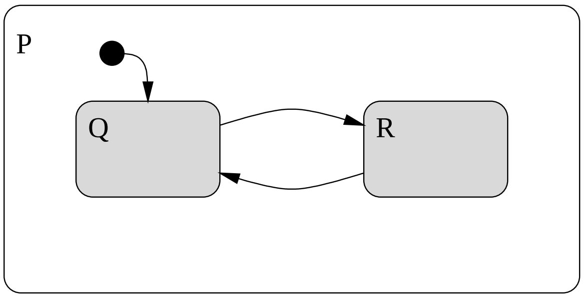

Figure 3: A simple example of an or-state.

orthogonality. In finite automata, the effects of orthogonality may be reproduced through the use of alternation (universal nondeterminism; see, for example, [39]). The Statecharts notation, however, is much more descriptive and intuitively understandable; both because of the inclusion of orthogonality and because of the ability to group related states together in a hierarchy of states.

Since Statecharts are intended to model real-time reactive systems, they do not contain any features analogous to final states in finite automata. In our discussion, however, we will be considering the Statecharts language in terms of the terminology and notation of finite automata, and we will discuss an application of Statecharts as acceptors of languages, so we will need to make use of the functionality of final states. We will, therefore, represent final states in the "traditional" way; that is, as a subset of the set of states in the statechart.

A statechart consists, at the lowest level, of two basic types of states: *or-states* and *and-states*. Or-states are analogous to the states of a finite automaton, with the exception that states can be nested within an or-state. For this reason, we will make a distinction between *primitive* or-states (or simply *primitive states*), those which contain no child states, and *complex* or-states (or *complex states*), those which have one or more substates. The name "or-state" comes from the behaviour of this type of state (in the exclusive-or sense); a statechart residing in a complex or-state, q, must reside in precisely one of the substates of q. An and-state is divided by dashed lines into *components*, each of which is itself an or-state. A statechart residing in an and-state is, as the name would imply, simultaneously residing in all of its components. Or-states, then, allow for a hierarchical organization of a statechart diagram, while and-states introduce orthogonality, or concurrent processing, to the notation.

Figure 3 shows a simple or-state example. The parent state P is a complex state, while its substates, Q and R, are primitive states. State transitions into (or out of) P may arrive at (or leave) any of P, Q or R. In the event that a transition ends on the outer surface of a complex state, the *default* transition, denoted by the filled circle with an arcing arrow (in the case of Fig. 3, the default transition is into Q), is followed. Similarly, a transition exiting a complex state is followed, regardless of the substates, whenever the event indicated by the transition occurs.

In Fig. 4, we see an example of an and-state (for the sake of simplicity, the names of most of the substates have been omitted). The name of an and-state is contained in a box

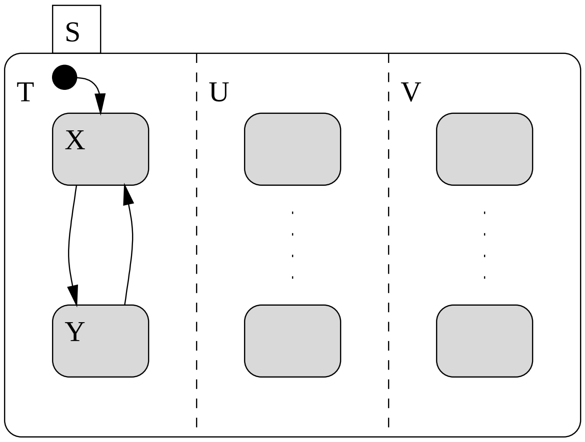

Figure 4: A simple and-state.

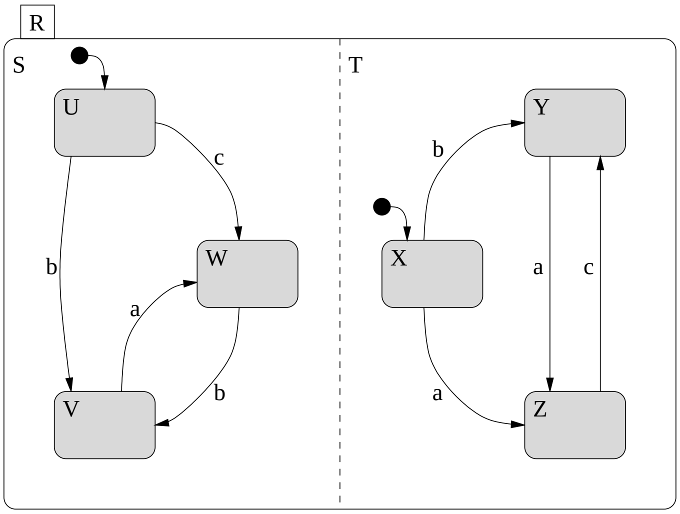

Figure 5: A less trivial statechart

attached to the upper-left corner of the state. As has already been mentioned, a statechart residing in state S resides simultaneously in all of T, U, and V. Since each of T, U, and V are themselves complex or-states, a statechart in state S simultaneously resides in exactly one substate from each of T, U, and V.

The orthogonal functionality introduced with and-states can have a significant impact on the state complexity of a statechart. Figures 5 and 6 show a simple orthogonal statechart and its equivalent DFA. The DFA in this example has only nine states, but it is easy to imagine a heavily orthogonal statechart requiring exponentially many states for an equivalent DFA representation. Even from this small example, we can see how the graphical depiction of orthogonal components in the Statecharts language allows for an easier and more intuitive understanding of the behaviour of a statechart as compared to an equivalent DFA.

We will discuss one more feature of Statecharts here; a feature which is tied rather closely

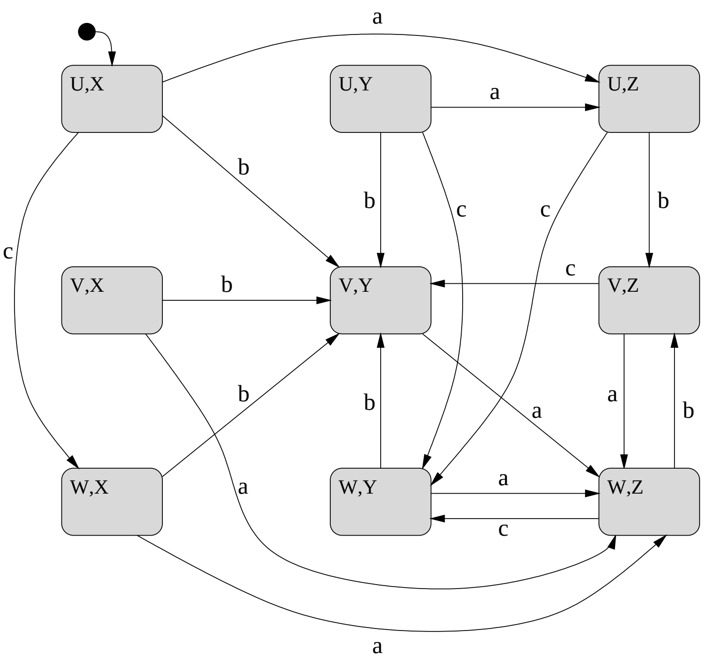

Figure 6: The DFA equivalent of Fig. 5

to the orthogonality discussed above. In addition to simply making transitions based on the input to the statechart, a transition to a new state may take place with arbitrary conditions (which allows transitions within an and-state to be followed, for example, only if another orthogonal component currently resides in a specific state, or is currently leaving a given state).

We have only covered here the most basic features of the Statecharts language — the notation itself is very robust, and contains many more features which are beyond the scope of the current discussion. The Statecharts notation contains the capability to generate outputs (called "actions" in [13]) which can, unlike the case of finite transducers, be treated as input to trigger transitions elsewhere in a statechart.

4.2 A Written Notation

In this section, we will develop a written notation which we will use to discuss statecharts more rigourously than we have thus far. We will define a notation which can be used to discuss a restricted class of statecharts containing all of the features we discussed in the previous section with the exception of outputs and conditional transitions. Several key issues need to be addressed in this discussion; namely the preservation of the hierarchical structure, and a proper handling of transitions within orthogonal components of and-states.

For any pair of states in a statechart, q_i and q_j, if q_j is a substate of q_i, we will write

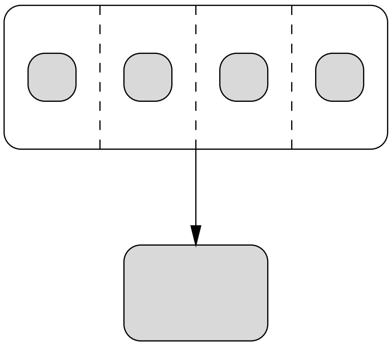

Figure 7: Exiting an and-state regardless of any of the substates

$q_j \prec q_i$. We introduce a *default state mapping* $D : Q \longrightarrow Q$ defined as follows:

$$D(q_i) = \begin{cases} q_i, & \text{if } q_i \text{ is a primitive state; and} \\ q_j, & \text{if } q_j \prec q_i \text{ is the default state for } q_i. \end{cases}$$

Since a statechart currently residing in an and-state is necessarily in a list of states simultaneously, we will refer to an and-state's *configuration*, the list of states within the and-state in which the statechart currently resides, and represent an and-state's configuration with a set of states. Additionally, we will differentiate the names of and-states from those of or-states by using boldface for the and-state's name ($\mathbf{q}_i$, for example). For a given and-state $\mathbf{q}$ with orthogonal components, each of which is itself an or-state, $q_1, \ldots, q_i$, we will make no distinction between the name of the and-state and the set of names of its components; we will consider the set of component states $\{q_1, \ldots, q_i\}$ a synonym for $\mathbf{q}$. Finally, we will incorporate the hierarchical nature of Statecharts into our notation by including for every state, in the definition of the state set, a list of all its parent states up to the highest-level state. Thus, if $q_i \prec q_j$, we may refer to q_i as $q_j \cdot q_i$. In our discussion, however, we will, for the sake of brevity, usually drop as much of the state's name as is allowable while preventing ambiguity.

One of the more important issues is that of entering or exiting orthogonal and-states. For a given and-state, $\mathbf{q}$, the Statecharts language allows two primary means of entering $\mathbf{q}$, and three basic means of leaving $\mathbf{q}$. We could enter $\mathbf{q}$ either through a transition to $\mathbf{q}$ using only the default transitions, or through a transition which indicates specific states in some or all of $\mathbf{q}$'s components, using default transitions for the unspecified components. Similarly, we could exit $\mathbf{q}$ using transitions directly from $\mathbf{q}$, transitions from a specific configuration of states in the components of $\mathbf{q}$, or transitions from states in some components of $\mathbf{q}$ and regardless of the state of the other components. For the entry conditions, it will suffice to use the default state mapping, D, but we need to introduce a *variable state*, x, for the third exit condition depicted in Figure 9. Figures 7, 8 and 9 show the three exit situations.

We need only define the transition function for Statecharts. A formal and generalized written notation for the transition function is beyond the scope of this thesis, so we will, in the next section, define a transition function for a specific statechart example.

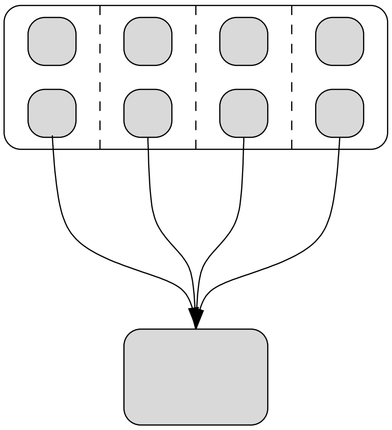

Figure 8: Exiting from a specific and-state configuration

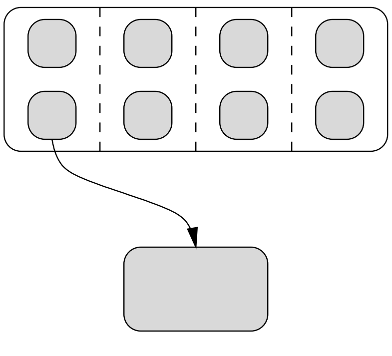

Figure 9: Exiting an and-state regardless of the state of some components

4.3 Error-Correction with Statecharts

In this section, we will construct a statechart which solves a certain class of error-correction problems. We consider an error-channel which changes the input in such a way that it is apparent that an error has occurred during transmission. To formalize this notion, we let Σ be the set of symbols which are sent through the error-channel, and let $\Sigma \cup \{\flat\}$ be the set of output symbols from the error-channel, where $\flat$, a symbol not occurring in Σ, is used to indicate that an error has occurred during transmission. We will refer to $\flat$ as a "blank" symbol. We now formalize the definition of this language.

Definition 4.1 *Let Σ be a finite alphabet, $L \subseteq \Sigma^*$ be a language, and let $k \geq 0$ be an integer representing the maximum tolerable number of errors. Then we define*

$$\begin{aligned} L_k = \{w = u_0\flat u_1 \ldots \flat u_l \quad | \quad & 0 \leq l \leq k;\ u_i \in \Sigma^* \text{ for all } i \geq 0; \text{ and} \\ & \exists\, w' = u_0 a_1 u_1 a_2 \ldots a_l u_l \text{ such that } a_i \in \Sigma, \text{ and } w' \in L\}. \end{aligned}$$

Note in particular that $L_0 = L$ and for any language L, $L \subseteq L_k$ for all $k \geq 0$.

Essentially, L_k is the set of all words containing at most k blanks such that there is at least one assignment of symbols from Σ to the blank symbols which will result in a word in L. Note that this definition is not sufficiently strong to guarantee that there is *at most* one such word; such a guarantee can, however, be obtained if the language, L is error-correcting for the channel we are discussing. Using the notation introduced in Chapter 3, the error channel of Definition 4.1 is defined by error functions

$$\mathbf{h} \in \bigcup_{0 \leq i \leq k} (G_\epsilon^+ G_\sigma)^i G_\epsilon^+ \qquad (2)$$

where the substitution functions are defined by $\mathbf{s} : \Sigma \to \{\flat\}$, for $\flat \notin \Sigma$.

We now formalize the meaning we intend for any subsequent usage of the term "statechart". This definition is not a general definition intended to formalize the entire Statecharts notation, but one which will allow us to discuss a restricted class of the Statecharts language more formally:

Definition 4.2 *A **restricted statechart** (henceforth, merely **statechart**) is a 5-tuple, $S = (Q, \Sigma, \Delta, D, F)$ where*

$$\begin{aligned} Q &: \text{ is the finite set of states;} \\ \Sigma &: \text{ is the input alphabet;} \\ F \subseteq Q &: \text{ is the set of final states.} \end{aligned}$$

$\Delta : \mathcal{P}(Q) \times \Sigma \longrightarrow \mathcal{P}(Q)$ *is the **state transition mapping**, mapping a subset of Q and an input symbol into another subset of Q, and $D : Q \to Q$, the **default state mapping**, maps a complex state q_i into its default state (which must be an immediate substate), and maps every primitive state q_j into itself. We will call a statechart's computation **accepting** if, after reading the input word, at least one component of the statechart resides in a final state. That is, if a statechart resides in the state configuration $C = \{q_1, \ldots, q_j\}$, then the statechart will accept the input word if, and only if, $F \cap C \neq \emptyset$.*

4.3 Error-Correction with Statecharts

The statechart construction given in this section to accept the language L_k, for regular languages L, relies heavily on the definition of a DFA which accepts L; in particular, we will refer frequently to the transition function for that DFA. However, since the state transition mapping for statecharts operates on *sets* of states instead of on individual states, we define the following extension to the transition function for a DFA.

Definition 4.3 *Given a DFA $M = (Q, \Sigma, \delta, s, F)$, $L \subseteq \Sigma^*$, and a set $S \subseteq Q$, $S = \{s_1, \ldots, s_n\}$, we define the **set transition**, $\delta(S, a)$, as:*

$$\delta(S, a) = \{\delta(s_1, a), \ldots, \delta(s_n, a)\}, \textit{ for all } a \in \Sigma.$$

*In a similar vein, we define the **language transition**, $\delta^*(q_i, L)$, as:*

$$\delta^*(q_i, L) = \{\delta^*(q_i, w_1), \delta^*(q_i, w_2), \ldots, \delta^*(q_i, w_m)\} \textit{ where } L = \{w_1, \ldots, w_m\}.$$

The hierarchical nature of the Statecharts language allows for recursive statechart constructions — that is, using identical portions of the statechart in multiple places. Since each state in a statechart must have a unique name, we have introduced the "dot-notation" used in §4.2. However, it will be convenient in the current discussion to refer to states by their lowest-level names only, with the implicit assumption that the names of all of a given state's superstates are prepended to the given state name. We now construct a statechart which will accept L_k whenever the original language, L, is regular.

Theorem 4.1 *Let $\Sigma = \{a_1, \ldots, a_n\}$ be a finite alphabet and let $k \in \mathbb{N}$ be some fixed error-bound. Then for every regular language $L \subseteq \Sigma^*$, there exists a statechart accepting L_k.*

Proof:
Let $M = (Q_M, \Sigma, \delta_M, s, F_M)$ be a DFA accepting L. We construct a statechart, $S = (Q_S, \Sigma \cup \{\flat\}, \Delta_S, D, F_S)$. We will slightly modify the state transition mapping given above, and will define our transition mapping as $\Delta : \mathcal{P}(Q_M) \times \{0, \ldots, k\} \times \Sigma \longrightarrow \mathcal{P}(Q_M) \times \{0, \ldots, k\}$.

The general idea for the statechart construction is that, whenever a $\flat$ symbol is read by the statechart, it transitions to an orthogonal state in which it simultaneously continues a simulation of the DFA M with each orthogonal component interpreting the $\flat$ symbol as a different symbol from the alphabet Σ. In this way, we simulate M by following the transitions that M would follow whenever the current input symbol is in Σ and then branching into an orthogonal state to simulate M on all symbols in Σ if the current input symbol is a blank.

We have slightly altered the transition mapping from that given in Definition 4.2. This change was made to emphasize the importance of the hierarchical structure in our construction. The level of the state hierarchy in which the statechart currently resides corresponds to the number of blank symbols which have so far been read. Alternately, the level in which the statechart currently resides corresponds precisely to the number of errors which have so far occurred in the input. We use the error counters in the transition mapping, then, to indicate whether the statechart is to continue computation in the current level, or to transition into the next, deeper level.

We will define our statechart S in a hierarchical way, and construct the sets Q_S, F_S and the transition mapping as a "modified union" (with the translation of all state names used into their long state names in the dot-notation introduced in §4.2) of the corresponding sets and mappings Q_q, F_q and Δ_q for all substates $q \prec S$.

We begin with the highest level state in S, $S_0 = (Q_0, \Sigma \cup \{\flat\}, \Delta_0, D, F_0)$, defined as follows:

$$\begin{array}{rcl} Q_0 & = & Q_M \cup \{\mathbf{S}_1\} \\ \Delta_0 & : & \mathcal{P}(Q_M) \times \{0\} \times \{\Sigma \cup \{\flat\}\} \to \mathcal{P}(Q_M) \times \{0,1\} \\ D(q) & = & \left\{ \begin{array}{ll} s, & \text{if } q = S_0\text{; and} \\ q & \text{otherwise} \end{array} \right. \\ F_0 & = & F_M \end{array}$$

Δ_0, the highest level state transition mapping, is very similar to δ_M. Every transition in δ_M is present, and we add transitions to the next level for blank input. Note that as long as the input symbols are in Σ, the behaviour of S_0 is identical to that of M. We define Δ_0 as follows:

$$\Delta_0(q, 0, a) = \left\{ \begin{array}{ll} (\delta_M(q,a), 0), & \text{if } a \in \Sigma\text{; and} \\ (\{\delta_M(q,a_1), \ldots, \delta_M(q,a_n)\}, 1), & \text{if } a = \flat. \end{array} \right.$$

For all $i \in \{1, \ldots, k\}$, define an and-state $\mathbf{S}_i = \{S_{i,a_j} \mid a_j \in \Sigma\}$. We now define the lowest level of our recursive construction, $\mathbf{S}_k = \{S_{k,a_j} \mid a_j \in \Sigma\}$. For all $a_j \in \Sigma$, define $S_{k,a_j} = (Q_{k,a_j}, \Sigma \cup \{\flat\}, \Delta_{k,a_j}, D, F_{k,a_j})$ as follows:

$$\begin{array}{rcl} Q_{k,a_j} & = & Q_M \cup \{\emptyset\} \\ \Delta_{k,a_j} & : & \mathcal{P}(Q_M) \times \{k\} \times \{\Sigma \cup \{\flat\}\} \to \mathcal{P}(Q_M) \cup \{\emptyset\} \times \{k\} \\ D(q) & = & q, \text{ for all } q \in Q_{k,a_j}. \\ F_{k,a_j} & = & F_M \end{array}$$

where Δ_{k,a_j} is defined as

$$\Delta_{k,a_j}(R, k, a) = \left\{ \begin{array}{ll} (\delta_M(R,a), k), & \text{if } a \in \Sigma\text{; and} \\ (\emptyset, k), & \text{if } a = \flat. \end{array} \right.$$

In the lowest level of the statechart's hierarchy (see Fig. 10), the DFA M is simulated in all orthogonal components. If a $\flat$ is read in this level, it signifies that there are more errors in the input than can be introduced by the error channel — that is, the input is not in L_k. Thus, every state in the lowest level of the statechart transitions to a "dead state", $\emptyset$, when a $\flat$ is read.

For all $i \in \{1, \ldots, k-1\}$ and $\forall a_j \in \Sigma$, define $S_{i,a_j} = (Q_{i,a_j}, \Sigma \cup \{\flat\}, \Delta_{i,a_j}, D, F_{i,a_j})$ as follows:

$$\begin{array}{rcl} Q_{i,a_j} & = & Q_M \cup \{\mathbf{S}_{i+1}\} \\ & = & Q_M \cup \{S_{i+1,a_1}, \ldots, S_{i+1,a_n}\} \\ \Delta_{i,a_j} & : & \mathcal{P}(Q_M) \times \{i\} \times \{\Sigma \cup \{\flat\}\} \to \mathcal{P}(Q_M) \times \{i, i+1\} \\ D(q) & = & \left\{ \begin{array}{ll} s, & \text{if } q = S_{i,a_j}\text{; and} \\ q & \text{otherwise.} \end{array} \right. \\ F_{i,a_j} & = & F_M \end{array}$$

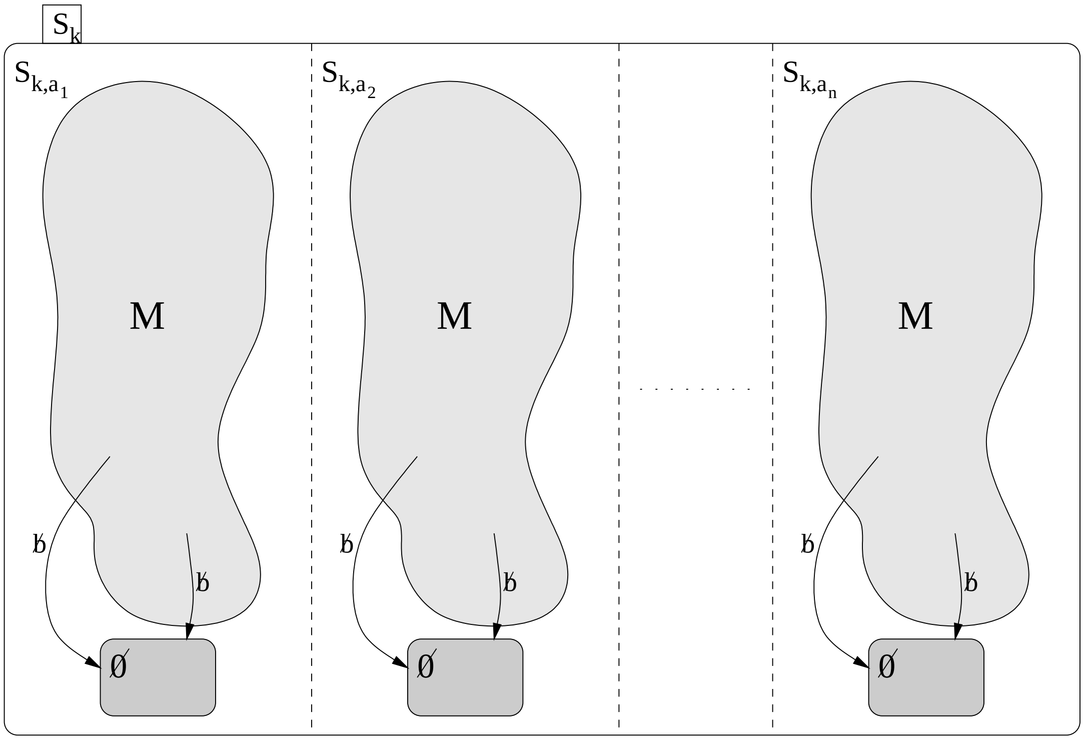

Figure 10: The bottom level, $\mathbf{S}_k$, of the statechart construction.

Thus, the default state of a primitive state is, of course, itself, and the default state for each of the S_{i,a_j} is the start state for its nested copy of M. We note once more that, although the names of the states in M are being used directly in this construction, in order to uniquely refer to a given state within the statechart S, the names of all of a given state's parent states must be prepended to the simplified name we use.

For all states S_{i,a_j} where $i \in \{1, \ldots, k-1\}$, and for all subsets $R \subseteq Q_M$, we define the state transition function, Δ_{i,a_j} as follows:

$$\Delta_{i,a_j}(R, i, a) = \begin{cases} \left(\bigcup_{a_j \in \Sigma} \delta_M(R, a_j), i+1\right), & \text{if } a = \flat\text{; and} \\ (\delta_M(R, a), i), & \text{if } a \in \Sigma. \end{cases}$$

Thus, when a blank is read while the statechart is currently in $\mathbf{S}_i$, each component of $\mathbf{S}_i$ transitions into a specific set of states in the components of $\mathbf{S}_{i+1}$. Figure 11 shows an arbitrary component of the and-state, $\mathbf{S}_{j-1}$, where $j \leq k-1$. The states $q_{i_1}, \ldots, q_{i_n}$ represent $\delta_M(q_i, a_1)$ through $\delta_M(q_i, a_n)$, respectively. If a symbol from Σ is read, the computation proceeds exactly as in M.

We construct the statechart, then, by nesting nearly identical copies of S_0 (the S_{i,a_j} states) inside orthogonal components of the and-state $\mathbf{S}_i$. The nested states are not completely identical to their parent, since they differ in the number of nested substates before the construction bottoms out in the $\mathbf{S}_k$ states. Every component in every and-state contains a copy of the DFA M with added transitions leaving every state in this copy which specify

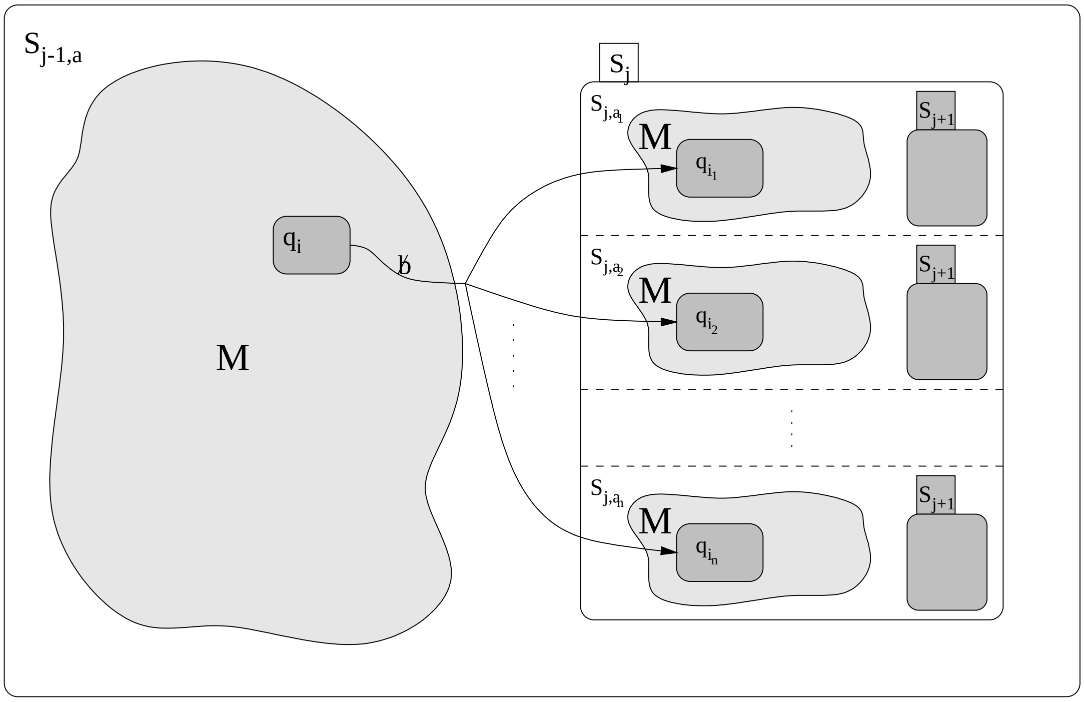

Figure 11: An arbitrary component of $\mathbf{S}_{j-1}$ for $j \leq k-1$.

the transition to be taken if a blank symbol is read. With the exception of the bottom level states, every component in each and-state also contains a nearly identical copy of itself with the error counter incremented — hence, for example, S_{i,a_j} contains a copy of S_{i+1,a_j}.

Finally, define $S = (Q_S, \Sigma \cup \{\flat\}, \Delta_S, D, F_S)$ as follows:

$$\begin{aligned} Q_S &= \bigcup_{i \in \{0,\ldots,k\}, a_j \in \Sigma} S_{i,a_j} \cup S_0 \\ \Delta_S &= \bigcup_{i \in \{0,\ldots,k\}, a_j \in \Sigma} \Delta_{i,a_j} \cup \Delta_0 \\ F_S &= \bigcup_{i \in \{0,\ldots,k\}, a_j \in \Sigma} F_{i,a_j} \cup F_0 \end{aligned}$$

We must now prove that S accepts the language L_k. For this, we will need the following lemma.

Lemma 4.2 *For any $w = u_0 \flat u_1 \flat \ldots \flat u_l$ where $l \leq k$ and $u_i \in \Sigma^*$ for all $0 \leq i \leq k$, the statechart S simulates the operation of the DFA M on all possible words in $H = \{u_0 a_1 u_1 \ldots a_l u_l \mid a_i \in \Sigma$, for all $i\}$. That is, $\Delta_S(D(S_0), 0, w) = (\delta_M(s, H), l)$.*

Proof:
Fix $w = u_0 \flat u_1 \flat \ldots \flat u_l$ with $l \leq k$ and $u_i \in \Sigma^*$, for all $0 \leq i \leq k$. If $l = 0$, then $w = u_0$ so, since the transitions in S_0 mimic those of M for input symbols from Σ, and since $u_0 \in \Sigma^*$, S simulates M on u_0.

Next, suppose that, for some $0 \leq i \leq l$, S simulates M simultaneously on all input words $\{u_0 a_1 u_1 \ldots a_i u_i \mid a_j \in \Sigma$ for all $j.\}$. We describe the behaviour of S on the input

$u_0\flat \ldots \flat u_i \flat u_{i+1}$, which is a prefix of w. By our assumption, after reading $u_0\flat \ldots \flat u_i$, the configuration of S is such that it is simulating M simultaneously on all possible words in $T_i = \{u_0 a_1 \ldots a_i u_i \mid a_j \in \Sigma\}$ — that is, the configuration of S is $(\delta_M(s, T_i), i)$. Since S has read i blank symbols at this point, S currently resides on the ith level in some configuration, call it C, of states $C = \{q_{i_1}, \ldots q_{i_m}\}$, with each state q_{i_j} in this configuration representing the state in which M would reside after reading at least one of the words in the set T_i.

However, upon reading the $\flat$ which follows u_i in w, S follows the following transition:

$$\begin{aligned}
\Delta_S(C, i, \flat) &= \left(\bigcup_{a_j \in \Sigma} \delta_M(C, a_j), i+1 \right) \\
&= (\delta_M(C, a_1) \cup \ldots \cup \delta_M(C, a_n), i+1) \\
&= (\{\delta_M(q_{i_1}, a_1), \ldots, \delta_M(q_{i_m}, a_1), \ldots, \delta_M(q_{i_1}, a_n) \ldots, \delta_M(q_{i_m}, a_n)\}, i+1) \\
&= (\{\delta_M(q_{i_1}, a_1), \ldots, \delta_M(q_{i_1}, a_n), \ldots, \delta_M(q_{i_m}, a_1) \ldots, \delta_M(q_{i_m}, a_n)\}, i+1) \\
&= (C', i+1)
\end{aligned}$$

Since each state $q_{i_j} \in C$ represents the state in which M would reside after reading some word in T_i and since the state $\delta_M(q_{i_j}, a_r)$ is in the new configuration, C' for every $a_r \in \Sigma$, the statechart S simulates M on all possible words in $T_{i+1} = \{u_0 a_1 u_1 \ldots a_{i+1} u_{i+1} \mid a_j \in \Sigma\}$.

$\square$

We now show that $L(S) = S_k$. Since by Lemma 4.2, for any given word $w = u_0\flat u_1 \flat \ldots \flat u_l$, S simulates M on all possible words made by replacing the blank symbols with symbols from Σ, and since no word with more than k blanks in it will be accepted, we have that $L(S) \subseteq S_k$.

Conversely, we let $w \in L_k$. Then $w = u_0\flat u_1 \flat \ldots \flat u_l$ for some $l \leq k$. Since $w \in L_k$, there exists a word $w' = u_0 a_1 u_1 \ldots a_l u_l$ such that $a_i \in \Sigma$ and $w' \in L$, the original language. Then, also by Lemma 4.2, since S simulates M on all words in $T_l = \{u_0 a_1 u_1 \ldots a_l u_l \mid a_i \in \Sigma\}$, then $w' \in T_l$. Since $w' \in L$, then, the simulation of M on w' will terminate in an accepting state, and therefore, $w' \in L(S)$. So $L_k \subseteq L(S)$, and therefore, since $L(S) \subseteq L_k$, $L(S) = L_k$.

$\square$

5 Transductions on SID Channels

5.1 Introduction

The shuffle-Hamming distance is a very useful metric for comparing words over some alphabet Σ. However, it is somewhat limited by the fact that the size of the words, or the relative frequency of errors in the words, is not taken into account. In order to build a language recognizer, such as a DFA, based on the shuffle-Hamming distance, we would need to fix an upper-bound for the maximum number of errors which can be recognized by our DFA, regardless of the relative sizes of the words. That is, for an input word w in some regular language L and a DFA constructed to accept the neighbourhood of L with respect to δ_H to within some error-bound k, at most k errors are tolerated in w, regardless of the length of w. Real-world error channels, however, are typically *very* dependent on the length of the input word; they will typically cause more errors in longer words than shorter words, thus crippling the effectiveness of our model.

Following this line of reasoning, then, we are led to consider other seemingly natural metrics in which the distance between two words varies according to the length of those words. In this chapter we consider, from a language- and automata-theoretic perspective, a seemingly natural variant of the shuffle-Hamming distance which is modified to work around this problem, and a pseudometric derived from the $\sigma \odot \iota \odot \delta(k, l)$ SID channel discussed in Chapter 3.

5.2 The importance of regularity-preservation

It was shown in Chapter 2 that the shuffle-Hamming distance preserves regularity — a very important property for our purposes since we would like to find a measure of distance for which the neighbourhood of a language L can be accepted by the same type of machine as L itself. However, as we discussed above, the number of errors tolerated by the shuffle-Hamming distance is independent of the lengths of the words. We are thus led to consider other measures of distance between words and their properties.

The obvious candidate for such a distance measure would be one in which the number of errors between two words is divided by the average length of the two words. For w_1 and $w_2 \in \Sigma^*$, let

$$\delta(w_1, w_2) = \frac{\delta_H(w_1, w_2)}{\left(\frac{|w_1|+|w_2|}{2}\right)}$$

Unfortunately, we have the following result.

Proposition 5.1 *δ, as defined above, does not preserve regularity.*

Proof:
Let $\Sigma = \{a, b\}$ and let $L = a^*$. We show that $E(a^*, \delta, \frac{1}{2})$, the set of all words $w' \in \Sigma^*$ such that $\delta(w, w') \leq \frac{1}{2}$ for some $w \in a^*$, is not regular. Let $L' = E(a^*, \delta, \frac{1}{2})$. Thus, any word in L' will either also be in L or will have some finite number of b's in it, say m. Since the positioning of the errors within the word is irrelevant, we consider words in which there is a string of m b's in the middle of the word, $w' = a^k b^m a^{n-k}$. The word in a^* which is closest to w', with respect to δ, is a^n, for $n \geq 0$. Therefore,

$$\delta(a^n, a^k b^m a^{n-k}) = \frac{m}{\left(\frac{n+n+m}{2}\right)} \leq \frac{1}{2}$$

since we're considering only values of m for which the distance is at most $\frac{1}{2}$.

So we have, then

$$\delta(a^n, a^k b^m a^{n-k}) \leq \frac{1}{2}$$

$$\Leftrightarrow \quad \frac{m}{\binom{2n+m}{2}} \leq \frac{1}{2}$$
$$\Leftrightarrow \quad 3m \leq 2n$$

which means that $E(a^*, \delta, \frac{1}{2}) = \{w \in \Sigma^* \mid 3|w|_b \leq 2|w|_a\}$, which is not regular.

□

We have, then, a very natural-seeming distance measure which, despite the fact that it is derived from the shuffle-Hamming distance, doesn't even preserve regularity.

The problem with δ seems to be the opposite of the problem that we saw with the shuffle-Hamming distance. The shuffle-Hamming distance imposed too harsh a restriction on the number of errors which can occur in a word, while δ imposes no restriction at all on either the number or the distribution of errors in a word. In a way, this problem can be summarized by the fact that δ allows all of the errors in a word to be clustered together — it imposes no restrictions on how many errors can occur within some fixed number of symbols.

5.3 A pseudometric definition for SID channels

We now describe a measure of distance taking these ideas into account. The distance measure, or pseudometric, is derived from the SID channel $\sigma \odot \iota \odot \delta(k, l)$ defined in Chapter 3. The pseudometric, which we will call δ_l, measures the number of substitution, insertion or deletion errors in any l consecutive symbols of the input word, for $l \in \mathbb{N}$.

As we saw in the last section, then, we need a method of measuring the distance between two words of variable length which allows some amount of freedom in the number of errors that can occur in a word while simultaneously not allowing "too much" freedom — a restriction that was justified by the example of δ in §5.2. The SID channels offer just such allowances and restrictions.

We now formalize this notion with a pseudometric definition. The definition is based on the error functions defined in §3.3. The idea behind this definition is that, given a pair of words $u, v \in \Sigma^*$ for some alphabet Σ, there is some set of error functions, $\mathcal{H} = \{\mathbf{h} \mid v = \mathbf{h}(u)\}$, which describe the differences between u and v. Given some $\mathbf{h} \in \mathcal{H}$, then, we can determine the number of errors between u and v by examining the symbols of $\mathbf{h}$.

Definition 5.1 below, then, simply quantifies the number of errors represented by each error function symbol in $\mathbf{h}$.

Definition 5.1 *For $a \in G$, a basic error function symbol, define*

$$d(a) = \begin{cases} 0, & \text{if } a = \mathbf{e}; \\ 1, & \text{if } a = \mathbf{d} \text{ or } a = \mathbf{s}; \\ j, & \text{if } a = \mathbf{i}_u, \text{ where } j = |u|. \end{cases}$$

Definition 5.2 *For $\mathbf{w} \in G^+$, define*

$$D_l(\mathbf{w}) = \max\left\{ \sum_{i=0}^{2l} d(\mathbf{w}(i)), \sum_{i=2}^{2l+2} d(\mathbf{w}(i)), \ldots, \sum_{i=n-2l-1}^{n-1} d(\mathbf{w}(i)) \right\}$$

where $|\mathbf{w}| = n$.

$D_l(\mathbf{w})$, then, is the maximum number of errors which occur in any l consecutive symbols of the error function $\mathbf{w} \in G^+$. However, since for any words u and $v \in \Sigma^*$ there could be many error functions $\mathbf{h}$ such that $\mathbf{h}(u) = v$, and since these error functions will typically contain very different numbers of errors in any l consecutive symbols, we need to ensure that we use the optimal error function to determine the value of the pseudometric, δ_l. Hence, we define δ_l as follows:

Definition 5.3 *For two words,* w_1 *and* w_2*, over an alphabet* Σ*, define*

$$\delta_l(w_1, w_2) = \min \{\, D_l(\mathbf{h}(x)) \mid \mathbf{h}(x) \text{ is an error-function and } \mathbf{h}(w_1) = w_2 \,\}$$

We have been referring to δ_l as a pseudometric, and it is clear that it satisfies the first two conditions of Definition 2.11. Unfortunately, δ_l does not satisfy the triangle-inequality and therefore does not fulfill the definition of a metric. For instance, consider the following words, for $l = 2$: let $x = aa$, $y = baab$, and $z = cbacabc$. Now, $x = aa = \lambda a \lambda a \lambda$, so the error-function $\mathbf{h}_{xy} = \mathbf{i}_b\mathbf{eee}\mathbf{i}_b$ satisfies $y = \mathbf{h}_{xy}(x)$ with $\delta_l(x, y) = 2$. Also, $\mathbf{h}_{yz} = \mathbf{i}_c\mathbf{eee}\mathbf{i}_c\mathbf{eee}\mathbf{i}_c$ is the smallest error-function satisfying $\mathbf{h}_{yz}(y) = z$, and $\delta_l(y, z) = 2$. But $\mathbf{h}_{xz} = \mathbf{i}_{cb}\mathbf{e}\mathbf{i}_c\mathbf{e}\mathbf{i}_{bc}$ is the smallest error-function satisfying $\mathbf{h}_{xz}(x) = z$, and $\delta_l(x, z) = 5 > \delta_l(x, y) + \delta_l(y, z)$.

Restrictions on the definition of δ_l fare no better for satisfaction of the requirements for a metric. We may, observing that the problem exhibited above seems to be related to insertion errors, disallow insertions, but then we would have a problem in which the error-channel is able to shorten a word, but is unable to lengthen it again. This would make the relation antisymmetric (since if $|v| < |w|$ for two words v and w, then $\delta_l(w, v) = n$ for some n, but $\delta_l(v, w)$ would be undefined). If we then add the restriction that an error-channel cannot delete *or* insert symbols, then we no longer have any particular reason to choose δ_l over more conventional metrics such as the Hamming distance.

5.4 A transducer construction for δ_l

Finite transducers can, as we mentioned, be viewed as string rewriters. However, it can also be useful to consider channels in the same way — as rewriters of strings. In the case of a general channel, it behaves very much like a "black box" in that the input to the channel is modified in unknown ways to produce the word which is received as output from the channel. For some types of channels, however, it is possible to model them with finite transducers. In [23] we see that the SID channels are such channels. In this section, we present a transducer construction for the δ_l pseudometric defined in the previous section.

Theorem 5.2 *Let* Σ *be some finite alphabet, let* δ_l *be the pseudometric defined in Definition 5.3, and fix* $l, k \in \mathbb{N}$*. There exists a finite transducer,* T_k*, such that, for any* $L \subseteq \Sigma^*$*,* $T_k(L) = E(L, \delta_l, k)$*.*

Proof:
We define a finite transducer $T = (Q, \Sigma, \Sigma, \sigma_k, s, F_k)$ as follows:

- $Q \subseteq \{0,1\} \times \{0,\ldots,k\}^{2l+1} \cup \{\emptyset\}$ is the set of states of the transducer. We specifically include a "dead state", which we denote by $\emptyset$. Each other state is a tuple consisting of a 0 or a 1, the purpose of which will be described later, and a $(2l+1)$-tuple, which we call the *error-vector*, representing the number of variations between the input and the output which have occurred in the last $2l+1$ transitions. This tuple is of length $2l+1$ in order for the transducer to mimic the operation of the δ_l distance described earlier. We denote the i'th element of the error-vector by e_i, where $e_i \in \{0,\ldots,k\}$. Q is defined as follows: A tuple $(j, [e_1, \ldots, e_{2l+1}]), j \in \{0,1\}$ is in Q whenever $\sum_{i=1}^{2l+1} e_i \leq k$.

  Note that every state in Q, with the exception of the dead state, has an error-vector sum less than or equal to k. We use this fact in the construction of the edge set to indicate that the transducer has introduced more than k errors in the most recent $2l+1$ symbols of the input word and hence the δ_l distance between the input word and the transducer's output would be greater than k.

- $s = (0, [0, \ldots, 0])$

- $\sigma_k \subseteq Q \times \{\Sigma \cup \{\lambda\}\} \times \Sigma^* \times Q$ is the transition-and-output function (or, more precisely, σ_k is a set of directed edges), consisting of a state, an input symbol (possibly λ), an output word, and another state. σ_k is defined by the following edge schemata:

  i) $(\ (0, [e_1, \ldots, e_{2l+1}]), \lambda, \lambda, (1, [e_2, \ldots, e_{2l+1}, 0])\ ) \in \sigma_k$;

  ii) $(\ (1, [e_1, \ldots, e_{2l+1}]), a, a, (0, [e_2, \ldots, e_{2l+1}, 0])\ ) \in \sigma_k,\ \forall a \in \Sigma$;

  iii) $(\ (1, [e_1, \ldots, e_{2l+1}]), a, b, (0, [e_2, \ldots, e_{2l+1}, 1])\ ) \in \sigma_k,\ \forall a \neq b \in \Sigma$;

  iv) $(\ (1, [e_1, \ldots, e_{2l+1}]), a, \lambda, (0, [e_2, \ldots, e_{2l+1}, 1])\ ) \in \sigma_k,\ \forall a \in \Sigma$;

  v) $(\ (0, [e_1, \ldots, e_{2l+1}]), \lambda, u, (1, [e_2, \ldots, e_{2l+1}, j])\ ) \in \sigma_k,\ \forall u \in \Sigma^+$ where $j = |u| \leq k$;

  Note that these schemata will produce invalid transitions in the event that the destination state signified by a particular schema is not in the state set Q. This will happen precisely when the sum of the error-vector elements in the indicated destination state is greater than k, and in these cases, we add transitions into the dead state instead.

- $F_k = Q \setminus \{\emptyset\}$

The structure represented by these edge schemata is important for our subsequent discussion. Each edge schema represents a particular type of error between the input and the output that is introduced by the transducer. The first two schemata correspond to the error-free condition, and the next three correspond to substitution, deletion and insertion, respectively.

The first symbol in the name of the states in Q is a sort of "parity" indicator. Recall that the various types of errors have specific locations in an extended word at which they can occur. More specifically, insertion can *only* occur between input symbols (i.e. on empty input), and substitution and deletion *cannot* occur between symbols. The error-free condition can occur either between symbols or on symbols, so there are two edge schemata (types i) and

ii)) corresponding to error-free transitions. The parity indicator is required because, in order to model the behaviour of the δ_l distance accurately, we must allow for error-free transitions between input symbols (which indicate that nothing is inserted), which means that we must allow error-free transitions with empty input and empty output. But since the transition is error-free, we must add a 0 to the end of the error-vector, and if no constraints were placed on the number of such transitions (i.e. with empty input and empty output), then it would be possible to erase the error-vector by following $2l + 1$ of these empty transitions. The parity indicator is then added to the construction to ensure that two consecutive transitions on empty input cannot occur, which prevents the problem described above.

We now claim that $T_k(L)$, the language produced by T_k on input L, is equal to L_k. Since $L_k = E(L, \delta_l, k) = \bigcup_{w \in L} E(w, \delta_l, k)$, it will be sufficient to show that, for any $w \in L$, $T_k(w) = E(w, \delta_l, k)$.

We first show that $E(w, \delta_l, k) \subseteq T_k(w)$. Let $u \in E(w, \delta_l, k)$, for some $w \in L$. Then there exists an error-function $\mathbf{h}$ of length $2|w| + 1$ such that $\mathbf{h}(w) = u$ and $D_l(\mathbf{h}) \leq k$. We follow transitions in T_k which correspond to the symbols in $\mathbf{h}$ in order to convert w into u. We start from the initial state $s = (0, [0, \ldots, 0])$ and we consider the error function $\mathbf{h}$ on a symbol-by-symbol basis, with each symbol in $\mathbf{h}$ corresponding to exactly one transition in the computation of T_k on w. Let $\mathbf{h} = h_1, h_2, \ldots, h_{2l+1}$. Then for each h_i, we follow transitions according to what symbol is in position i of $\mathbf{h}$ as indicated below:

$$
\begin{array}{rcl}
\mathbf{e} & : & (\ (0, [e_1, \ldots, e_{2l+1}], \lambda, \lambda, (1, [e_2, \ldots, e_{2l+1}, 0])\ )\text{ if } i \text{ is odd;} \\
 & & (\ (1, [e_1, \ldots, e_{2l+1}], a, a, (0, [e_2, \ldots, e_{2l}, 0])\ )\text{ otherwise} \\
\mathbf{i_u} & : & (\ (0, [e_1, \ldots, e_{2l+1}], \lambda, u, (1, [e_2, \ldots, e_{2l+1}, j])\ ),\text{ where } j = |u|; \\
\mathbf{d} & : & (\ (1, [e_1, \ldots, e_{2l+1}], a, \lambda, (0, [e_2, \ldots, e_{2l+1}, 1])\ ),\text{ where } w(\frac{i}{2}) = a; \\
\mathbf{s_b} & : & (\ (1, [e_1, \ldots, e_{2l+1}], a, b, (0, [e_2, \ldots, e_{2l+1}, 1])\ ),\text{ where } w(\frac{i}{2}) = a.
\end{array}
$$

In this way, then, we derive a sequence of computation steps through which T_k can produce u on input w. Since $D_l(\mathbf{h}) \leq k$, we are assured that, for any state in this computation sequence, $\sum_{i=1}^{2l+1} e_i \leq k$ so all of the transitions described above are valid. We note, however, that the transducer T_k is highly nondeterministic and thus on input w there are many computations which output u (or some prefix of u) and which may terminate in the dead state of T_k after the error-vector exceeds the bound k. However, for our purposes, it is sufficient that $\mathbf{h}$ with $D_l(\mathbf{h}) \leq k$ guarantees the existence of at least one successful computation of T_k as described above. Thus, $u \in T_k(w)$, and $E(w, \delta_l, k) \subseteq T_k(w)$.

Conversely, suppose that $u \in T_k(w)$, for some $w \in L$. Then, when w is input to T_k, there is an accepting sequence of transitions followed by T_k to produce the output word u. We can, then, apply the transformation described above in reverse to convert this sequence of transitions into an error function $\mathbf{h}$ such that $\mathbf{h}(w) = u$ since every edge schema in the construction of T_k corresponds to a basic error-function symbol (and at the same time enforces the restriction that the error function symbol $\mathbf{i}_u$ can only occur at odd-numbered indices of $\mathbf{h}$ while $\mathbf{d}$ and $\mathbf{s}$ can only occur at even-numbered indices). However, since $u \in T_k(w)$, every state appearing in this sequence of transitions must have an error-vector sum of at most k, which means that, in the conversion from w to u, there are never more than k

errors introduced in any $2l+1$ consecutive transitions. This, then, implies that $D_l(\mathbf{h}) \leq k$, and so $\delta_l(w,u) \leq k$. Therefore, $u \in E(w, \delta_l, k)$ and $T_k(w) \subseteq E(w, \delta_l, k)$.

Finally, since $E(w, \delta_l, k) \subseteq T_k(w)$ and $T_k(w) \subseteq E(w, \delta_l, k)$, $E(w, \gamma_l, k) = T_k(w)$, for any $w \in L$. Therefore, $T_k(L) = L_k$. □

Corollary 5.3 *The language induced by the pseudometric δ_l preserves regularity.*

Proof:
This follows immediately from the existence of the transducer T_k and from Theorem 2.4. □

We have shown that, in the case of the SID channel $\sigma \odot \iota \odot \delta(k,l)$, for any l and $k \in \mathbb{N}$, there exists a transducer realizing the channel. This, when combined with earlier results concerning transducers, proves that the pseudometric δ_l preserves regularity, meaning that if the input language for the transducer T_k is regular, then the output language must be regular as well. This implies that, for any $\sigma \odot \iota \odot \delta(k,l)$ channel and a regular language L, there exists a deterministic finite automaton accepting the k-neighbourhood of L with respect to δ_l. If, in addition, we add the constraint that L be error-correcting for $\sigma \odot \iota \odot \delta(k,l)$ then we are guaranteed that, for every channel output word that the DFA accepts, there is *exactly one* possible input which could have become the output word.

5.4.1 An equivalent NFA construction

Although the above construction, coupled with the closure results concerning regular languages presented, for example, in [39], guarantees the existence of an NFA accepting the error-bound around the language L, the construction in Theorem 5.2 doesn't immediately provide that NFA. The proof of Theorem 2.4 is effective, so we know that an NFA can be algorithmically constructed. A close examination of the transducer construction, however, will suggest a method whereby the NFA can be constructed. For the sake of completeness, then, we provide that NFA construction as well.

Theorem 5.4 *Given a DFA $M = (Q, \Sigma, \gamma, q_0, F)$ accepting L and natural numbers k, l such that $k \leq l$, we can construct a λ-NFA $M_k = (Q_k, \Sigma, \gamma_k, s, F_k)$ accepting $E(L, \delta_l, k)$.*

Proof:
The construction of the λ-NFA $M_k = (Q_k, \Sigma, \gamma_k, s, F_k)$ will mimic quite closely the construction of the finite transducer given earlier in this section. We construct the λ-NFA accepting $E(L, \delta_l, k)$ as follows:

- The set of states is similar to the state set of the transducer T_k, with the exception that, for each state in which the error-vector sum is at most k, we must also include the state in which the original DFA M would reside. We define the set of states, Q_k, as follows:
$$Q_k \subseteq Q \times \{0,1\} \times \{0,\ldots,k\}^{2l+1} \cup \{\emptyset\}.$$

Then, as in the transducer construction, we explicitly include the state $\emptyset$, to which M_k will transition whenever too many errors have occurred, and we define the rest of the states in Q_k to be the following set:

$$Q_k = \left\{ (q, i, [e_1, \ldots, e_{2l+1}]) \;\middle|\; q \in Q, i \in \{0,1\}, \text{ and } \sum_{j=1}^{2l+1} e_j \leq k \right\}$$

- $s = (q_0, 0, [0, \ldots, 0])$
- The set of final states, F_k is defined to be the set of all states in Q_k such that the DFA state it represents is also final:

$$F_k = \left\{ (q, i, [e_1, \ldots, e_{2l+1}]) \;\middle|\; i \in \{0,1\}, \sum_{j=1}^{2l+1} e_j \leq k \text{ and } q \in F \right\}$$

- The transition function for M_k is essentially a direct translation of the transition-and-output mapping for the transducer — we add transitions fitting any of the following transition schemata, for all $(q_i, j, [e_1, \ldots, e_{2l+1}]) \in Q_k$:

$$\begin{array}{rcl}
(q_i, 1, [e_2, \ldots, e_{2l+1}, 0]) & \in & \gamma_k(\ (q_i, 0, [e_1, \ldots, e_{2l+1}]), \lambda); \\
(\gamma_M(q_i, a), 0, [e_2, \ldots, e_{2l+1}, 0]) & \in & \gamma_k(\ (q_i, 1, [e_1, \ldots, e_{2l+1}]), a),\ \forall a \in \Sigma; \\
(\gamma_M(q_i, a), 0, [e_2, \ldots, e_{2l+1}, 1]) & \in & \gamma_k(\ (q_i, 1, [e_1, \ldots, e_{2l+1}]), b),\ \forall a \neq b \in \Sigma; \\
(\gamma_M(q_i, a), 0, [e_2, \ldots, e_{2l+1}, 1]) & \in & \gamma_k(\ (q_i, 1, [e_1, \ldots, e_{2l+1}]), \lambda),\ \forall a \in \Sigma;\ \text{and} \\
(q_i, 1, [e_2, \ldots, e_{2l+1}, j]) & \in & \gamma_k^*(\ (q_i, 0, [e_1, \ldots, e_{2l+1}]), u), \forall u \in \Sigma^+ \text{ where} \\
 & & j = |u| \leq k.
\end{array}$$

  We note that, as was the case in Theorem 5.2, these transition schemata correspond to the error types which can be caused by the SID channel. That is, schemata *i*) and *ii*) correspond to the error-free condition occurring "between" input symbols and at input symbols, respectively. Schemata *iii*), *iv*) and *v*) correspond to substitution, deletion and insertion type errors, respectively, with the "parity" indicator enforcing the restriction that each error-type can only occur at particular points of the input word. Note that, although the definition of insertion allows insertion of arbitrary words, we can (and, in the case of a nondeterministic finite automaton, we must) restrict the length of the inserted words. Thus insertions of strings of length greater than k are explicitly disallowed in schema *v*) since their insertion would exceed the error bound, and since the inclusion of such insertions into our specification of the λ-NFA would make our specification infinite.

It now remains to demonstrate that the language accepted by this λ-NFA is actually $E(L, \delta_l, k)$. Since the λ-NFA is constructed directly from the transducer construction given in Theorem 5.2, we refer the reader to that theorem for a complete proof of correctness and completeness. We will only discuss the basic behaviour of the construction here.

We note that, as was the case with Thm. 5.2, the transitions given above may produce invalid states as their destination state in the case where some number of errors was added to the current error-vector which resulted in an error-vector sum greater than k. In these cases, we add transitions to the dead state, $\emptyset$, instead.

Essentially, the nondeterminism in this construction is in guessing the "intended" input and counting the number of these guesses. From any given state with parity indicator 1, $(q, 1, [e_1, \ldots, e_{2l+1}]) \in Q_k$, there are error-free transitions from this state for all symbols in Σ representing the actual behaviour of the DFA M. In addition to these transitions, there are also transitions representing substitutions of the actual input symbol (transition type *iii*)) and transitions representing deletion of the input symbol (transition type *iv*)).

Also, for any given state with parity indicator 0, $(q, 0, [e_1, \ldots, e_{2l+1}]) \in Q_k$, we again have a transition for the error-free condition which, in this case, takes the form of a λ-transition to a state with parity indicator 1, a 0 pushed into the end of the error-vector and the same DFA state q. Finally, we also include the transitions for insertion which can potentially insert any word of length at most k, for our upper-bound k. We then add λ-transitions for every word u in Σ^+ with length at most k with the destination state indicated by the behaviour of the DFA on u and by the size of u. $\square$

We saw earlier in this section that the SID channel $\sigma \odot \iota \odot \delta(k, l)$ can be realized by a transducer which, if the input language is regular, guarantees the existence of a DFA which accepts the set of all possible outputs of this channel when the input is a word in the original regular language. We have given, in this subsection, a λ-NFA construction derived from the transducer construction which can accept the output language of $\sigma \odot \iota \odot \delta(k, l)$. By using an error-correcting regular language for this channel, then, we are assured that the output language can be accepted with perfect error-correction by a DFA.

6 Discussion

In this thesis, we have discussed several types of error channels, and their properties, from the viewpoint of formal language theory and automata theory. In Chapter 2, for example, we saw that, while a distance measure on words δ having the property of being a metric did not guarantee that its neighbourhoods preserved the regularity of the language to which it was applied, the additional property of δ being additive guaranteed a preservation of regularity. Further, in Chapter 5 we showed, by presenting a distance measure on words that was not even a metric but still preserved regularity, that being a metric is not even a necessary condition for the preservation of regularity.

From an automata-theoretic perspective, the preservation of regularity is a very desirable property for distance measures. Deterministic and nondeterministic finite automata have been very exhaustively studied, and their closure properties and state-complexity results are well known. Deterministic finite automata have the additional advantage that, when used as acceptors of languages, they can determine if a given word is in the language in precisely as many computation steps as there are alphabet symbols in the word. The SID channels are well studied and they provide a theoretical foundation for a very natural type of real-world error channel. With this in mind, the knowledge that these error channels preserve the regularity of the input language is a very powerful tool when attempting to correct the errors induced by the channels.

6.1 Future Work

- We have seen that the use of additive metrics on words over an arbitrary alphabet is sufficient to guarantee the preservation of regularity. However, we have also shown that a metric alone is neither sufficient (as exemplified in Chapter 2) nor necessary (which we saw in Chapter 5 in the fact that the pseudometric δ_l preserves regularity). We obtained one characterization of regularity preservation using finite transductions in Chapter 5. However, it would be useful to find regularity preserving distance measures that are as general as possible, if not to find a measure of distance which is both necessary and sufficient to guarantee that its neighbourhoods are regular.

- In Chapter 4, we defined a partial written notation, similar to that used for finite automata, to be used with the Statecharts language, but the notation defined was not complete or robust enough to be used to describe arbitrary statecharts. This notation could be extended to cover the rest of the Statecharts language, permitting formal arguments similar to those used with finite automata to be used with Statecharts as well.

- We defined a statechart construction, relying heavily on the features of orthogonality incorporated into the Statecharts language, which would accept the language output by a particular error channel in Chapter 4. However, the construction of statecharts for more general cases, such as the class of SID channels, has yet to be explored. The benefits of Statecharts, both in their ease of comprehension as compared to the standard finite automaton formalism and in the benefits that their orthogonal structure bring, make the behaviour of Statecharts in error correction situations worthy of further research.

References

[1] J. Berstel, *Transductions and context-free languages*, Teubner Stuttgart, 1979.

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

[3] C. S. Calude and E. Calude, *On some discrete metrics*, Bull. Math. Soc. Sci. Math. R. S. Roumanie (N.S.) **75** (1983), no. 27, 213–216.

[4] C. S. Calude, K. Salomaa, and S. Yu, *Metric lexical analysis*, Lecture Notes in Computer Science **2214** (2001), 48–59.

[5] J. Carroll and D. Long, *Theory of finite automata with an introduction to formal languages*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1989.

[6] D. A. Eckhardt and P. Steenkiste, *A trace-based evaluation of adaptive error correction for a wireless local area network*, Mobile Networks and Applications **4** (1999), 273–287.

[7] M. C. Gemignani, *Elementary topology*, 2 ed., Addison-Wesley, Reading, MA, 1972.

[8] S. Guiaşu, *Information theory with applications*, McGraw-Hill, London, 1977.

[9] R. W. Hamming, *Error detecting and error correcting codes*, Bell Systems Technical Journal **29** (1950), 147–160.

[10] D. Harel, *Statecharts: A visual approach to complex systems*, Tech. Report CS84-05, Department of Applied Mathematics, The Weizmann Institute of Science, 1984.

[11] ———, *Statecharts: A visual formalism for complex systems*, Science of Computer Programming **8** (1987), 231–274.

[12] D. Harel and E. Gery, *Executable object modeling with statecharts*, Proceedings of the 18th International Conference on Software Engineering, IEEE Computer Society Press, 1996, pp. 246–257.

[13] D. Harel and M. Politi, *Modeling reactive systems with statecharts*, McGraw-Hill, 1998.

[14] M. Ito, L. Kari, and G. Thierrin, *Insertion and deletion closure of languages*, Theoretical Computer Science **183** (1997), 3–19.

[15] ———, *Shuffle and scattered deletion closure of languages*, Theoretical Computer Science **245** (2000), 115–133.

[16] J. Jędrzejowicz and A. Szepietowski, *Shuffle languages are in $\mathcal{P}$*, Theoretical Computer Science **250** (2001), 31–53.

[17] H. Jürgensen and S. Konstantinidis, *Codes*, in Rozenberg and Salomaa [31], pp. 511–607.

[18] S. C. Kleene, *Representation of events in nerve nets and finite automata*, in Shannon and McCarthy [34], pp. 3–40.

[19] 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.

[20] ———, *Error-detecting properties of language*, Tech. Report 004, Saint Mary's University, Halifax, Nova Scotia, Canada, 2000.

[21] ———, *An algebra of discrete channels that involve combinations of three basic error types*, Information and Computation **167** (2001), 120–131.

[22] ______, *On the decidability of the error-detection property*, Tech. Report 003, Saint Mary's University, Halifax, Nova Scotia, Canada, 2001.

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

[24] K. Kukich, *Techniques for automatically correcting words in text*, ACM Computing Surveys **24** (1992), no. 4, 379–439.

[25] V. I. Levenshtein, *Binary codes capable of correcting deletions, insertions and reversals*, Soviet Physics-Doklady **10** (1966), 707–710.

[26] F. J. MacWilliams and N. J. A. Sloane, *The theory of error-correcting codes*, North-Holland, Amsterdam, 1977.

[27] U. Manber, *Introduction to algorithms — a creative approach*, Addison-Wesley, Reading, MA, 1989.

[28] A. Mateescu, A. Salomaa, K. Salomaa, and S. Yu, *Lexical analysis with a simple finite-fuzzy-automaton model*, Journal of Universal Computer Science **1** (1995), no. 5, 292–311.

[29] W. W. Peterson and Jr. E. J. Weldon, *Error-correcting codes*, 2 ed., MIT Press, Cambridge, MA, 1972.

[30] G. Pighizzini, *How hard is computing the edit distance?*, Information and Computation **165** (2001), 1–13.

[31] G. Rozenberg and A. Salomaa (eds.), *Handbook of formal languages*, vol. 1, Springer-Verlag, Berlin, 1997.

[32] A. Salomaa, *Formal languages*, Academic Press, Orlando, Florida, 1973.

[33] K. Salomaa, S. Yu, and Q. Zhuang, *The state complexities of some basic operations on regular languages*, Theoretical Computer Science **125** (1994), 315–328.

[34] C. E. Shannon and J. McCarthy (eds.), *Automata studies*, Princeton University Press, Princeton, NJ, 1956.

[35] T. A. Sudkamp, *Languages and machines*, 2 ed., Addison-Wesley, Reading, Massachusetts, 1997.

[36] J. D. Ullman, *The role of theory today*, ACM Computing Surveys **27** (1995), no. 1, 43–44.

[37] A. Weber, *Transforming a single-valued transducer into a Mealy machine*, Journal of Computer and System Sciences **56** (1998), 46–59.

[38] A. Weber and R. Klemm, *Economy of description for single-valued transducers*, Information and Computation **118** (1995), 327–340.

[39] S. Yu, *Regular languages*, in Rozenberg and Salomaa [31], pp. 41–110.

[40] ______, *State complexity of regular languages*, Proceedings of Descriptional Complexity of Automata, Grammars and Related Structures, 1999, pp. 77–88.

Vita

Name	Christopher Lee McAloney
Place and Year of birth	Bridgewater, NS, 1975
Education	Mount Allison University, 1993-2000 B.Sc. (Honours, Computer Science) 2000
	Queen's University, 2000-02
Experience	Research Assistant, Environment Canada, Summer 2000
	Research Assistant, Department of Computing and Information Science, Queen's University, 2001-02
	Teaching Assistant, Department of Computing and Information Science, Queen's University, 2001-02
Awards	Queen's Graduate Award, 2000
	Queen's Graduate Fellowship, 2000