To the Graduate Council:

I am submitting herewith a dissertation written by Stacy J. Prowell entitled "Sequence-Based Software Specification." I have examined the final copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Jesse Poore, Major Professor

We have read this dissertation
and recommend its acceptance.

Accepted for the Council:

Associate Vice Chancellor and
Dean of the Graduate School

SEQUENCE-BASED SOFTWARE SPECIFICATION

A Dissertation

Presented for the

Doctor of Philosophy Degree

The University of Tennessee, Knoxville

Stacy J. Prowell

May 1996

Copyright © 1996 by Stacy J. Prowell

All rights reserved

Dedication

To Darlene and Tom, the best of friends.

Acknowledgments

First I wish to thank my advisor, Professor Jesse Poore, for his excellent guidance, advice, and the many "learning opportunities" he has provided over the years.

I am grateful to Professors Michael Thomason, Robert Ward, and Robert McConnel for their service on my committee and for their careful review of this dissertation.

I wish to thank Dr. Carmen Trammell and Dr. Poore for the opportunity to work in the Software Quality Research Laboratory. A special thanks goes to Dr. Trammell for, well, everything.

Through the years I have worked in the Software Quality Research Laboratory, I have had the chance to interact with many talented people. I thank them all for their friendship, encouragement, and criticism. I especially wish to thank Ingrid Biery, Dr. Hailong Mao, Laura Prados, Dr. Gwen Walton, and Dr. Jim Whittaker.

Finally, I thank my best friend Darlene Lampman for love, patience, and encouragement.

Abstract

The primary result of this research is the sequence enumeration method of specification writing. Straightforward, systematic eumeration of all sequences to produce an arguably complete, consistent, and correct specification is made practical by a collection of techniques based on controlling the growth of an inherently combinatorial process.

By systematic investigation of stimulus histories, regular expression "rules" can be discovered and mapped to responses. These rules can be used to partition the set of all stimulus histories, providing an arguably complete, consistent, and correct black box specification. Computable predicates and other refinements are used where regular expressions are insufficient to capture behavior.

Using derivatives of regular expressions, a minimum-state recognizer can be created for each regular block of the black box. These recognizers can be combined by direct product into a sequential machine which is then minimized. The resulting sequential machine is a faithful representation of the original specification. The production of the sequential machine provides verification of the completeness and consistency of the original specification.

Through the introduction of state variables, the state machine's state space is partitioned until every state can be specified by giving values for each of the state variables. A language for state boxes is introduced, and a sequential machine whose state space has been properly partitioned can be automatically transformed into a state box. State migration issues are discussed as they arise during the distribution of the sequential machine's state space.

Contents

CHAPTER 1
Introduction 1

§1.1 Motivation 1

§1.2 Outline of Sequence-Based Box Structures 3

CHAPTER 2
Notation and Definitions 5

§2.1 Notation and Basic Concepts 5

§2.2 Regular Expressions 10

§2.3 Finite State Machines 13
2.3.1 Automata 13
2.3.2 Recognizers 15
2.3.3 Sequential Machines 19

§2.4 Reduction Systems 24

CHAPTER 3
The Black Box 27

§3.1 Stimuli and Responses 28

§3.2 The Black Box 31

§3.3 A Black Box Language 31
3.3.1 Refinement 34
3.3.2 Predicates 35

§3.4 Enumeration 38
3.4.1 The Interface and Natural Language Specifications 39
3.4.2 Sequence Enumeration 39
3.4.3 Implicit Rules 51

CHAPTER 4
The State Box 55

§4.1 Minimum State Derivation 56
4.1.1 Moore Machine Derivation 56
4.1.2 Derivation of the Minimum Mealy Machine 61

§4.2 The State Box 68
4.2.1 A State Box Language 69
4.2.2 State Partitioning 72
4.2.3 State Box Generation 76
4.2.4 Re-Introduction of Predicate Refinements 77

§4.3 State Box Composition 80
4.3.1 Series Composition 81
4.3.2 Parallel Composition 82

§4.4 Other State Box Issues 83
4.4.1 Data State and Control State 83
4.4.2 State Migration 86

CHAPTER 5
Advanced Specification Techniques 89

§5.1 Special Operators 89
5.1.1 Special Sequence Operator 90
5.1.2 Reset Operator 92
5.1.3 Concurrency Operators 93
5.1.4 Synchronous Composition Operator 95

§5.2 Abstraction Techniques 97
5.2.1 Abstraction 97
5.2.2 Stimulus Deletion Abstraction 103
5.2.3 Indexed Stimulus Deletion Abstraction 106

CHAPTER 6
Conclusions and Future Research 111

References 113

Appendices 117

APPENDIX A
Algorithms 118

§A.1 Black Box 118
A.1.1 Enumeration 118
A.1.2 Enumeration Mealy Machine Construction 119
A.1.3 Black Box Construction 121

§A.2 State Box 122
A.2.1 Black Box Mealy Machine Construction Algorithm 122
A.2.2 Black Box Mealy Machine Minimization 124
A.2.3 State Box 126
A.2.4 State Box Specification 126
A.2.5 State Space Partitioning 127

APPENDIX B
Prime Number Switch 129

§B.1 Requirements 129
B.1.1 Tagged Requirements 129
B.1.2 Interface Specification 129

§B.2 Black Box Construction 130
B.2.1 Enumeration 130
B.2.2 Black Box Derivation 131

§B.3 State Box Construction 132
B.3.1 Minimum Recognizer Construction 132
B.3.2 Mealy Machine Construction 133
B.3.3 Mealy Machine Minimization 134
B.3.4 State Space Partitioning 134
B.3.5 State Box Specification 135

APPENDIX C
Plain Ordinary Telephone System 138

§C.1 Tagged Requirements 138
C.1.1 Introduction 138
C.1.2 POTS Requirements 139

§C.2 Interface Specification 144
C.2.1 LIC Interface 145
C.2.2 Operator Interface 146
C.2.3 Connector 147
C.2.4 Clock 147

§C.3 Natural Language Specification 148
C.3.1 Off-hook for phone *n* *148*
C.3.2 On-hook for phone *n* *148*
C.3.3 Digit *d* pressed on phone *n* *149*
C.3.4 Initialize the system 149
C.3.5 Add phone *n* to the system 149
C.3.6 Remove phone *n* from the system 150
C.3.7 Clock pulse 150

§C.4 Enumeration 150
C.4.1 First Enumeration 150
C.4.2 Second Enumeration (Deletion) 153
C.4.3 Third Enumeration (Indexed Deletion) 154
C.4.4 Abstraction 157
C.4.5 Conclusion 164

Vita 165

Chapter 1

Introduction

1.1 Motivation

Software development under the Box Structure Method (BSM) [15,17,18] begins with development of a *black box specification*, which is a specification solely in terms of the external stimuli and responses. The black box specification is therefore a mapping from sequences of stimuli, called *stimulus histories*, to responses. This mapping must be shown to be *complete*, meaning all stimulus histories are mapped, *consistent*, meaning each stimulus history uniquely determines the next response, and *correct*, meaning the response is correct relative to the requirements set by application domain experts.

Such a mapping is also called a *total function*. Black box specifications range in style from formal assertions in predicate logic to informal prose descriptions. It

can be difficult to demonstrate the completeness and consistency of some black box specifications, and this important step is often neglected in practice.

The black box specification is developed from the requirements, which are seldom either complete or consistent. The process of developing a black box specification from the requirements is highly inventive, and practitioners find little guidance in current formulations of BSM. This leads to two problems encountered in practice. First, there may be incompleteness or inconsistency in the specified behavior. Second, the specification may contain assumptions which resolve problems in the requirements, yet which are never stated; or worse, it may contain tacit information of which developers are unaware.

These two problems can be avoided by introducing a procedure for discovering the specification from the requirements in a controlled, documented fashion, such that the procedure itself assures completeness and consistency. Such controlled, documented discovery of a specification from requirements that can be verified for correctness by application domain experts is called *correctness traceability.* The specification technique discussed here, sequence partition specification, lends itself well to requirements traceability.

Once a black box specification has been developed, the next stage in BSM is the introduction of internal state to encapsulate the necessary portions of the stimulus history, creating a *state box specification.* The state box specification must have external behavior identical to the black box specification (it must specify the same complete, consistent, and correct mapping). The general nature of most existing

formulations of BSM gives practitioners little guidance in inventing the state box specification from the black box specification, so there is no guarantee that completeness and consistency will be preserved. Thus, there is a need for a well-defined procedure for inventing a state box specification from an existing black box specification. A procedure for converting a black box specification into a state box specification is presented which preserves completeness and consistency, yet allows practitioners to explore state-space alternatives.

The goal of this research is to provide developers with explicit methods for

1. constructing a black box specification from the requirements,
2. assuring completeness and consistency of the black box specification,
3. maintaining correctness traceability of the black box specification,
4. generating a state box specification from the black box specification,
5. assuring the correctness of the state box specification, and
6. specification composition.

1.2 Outline of Sequence-Based Box Structures

First, requirements and design ideas are collected into an *interface specification* and a *natural language specification*. These two documents form a bridge between the informal customer requirements and the formal black box specification.

Stimuli are identified from the interface specification, and stimulus histories are generated. Each of these histories is considered, and the appropriate response is obtained from the natural language specification. If the natural language specification does not specify a response, it is incomplete and must be updated. If the natural language specification does specify a response, but the response is not unique, then the natural language specification is inconsistent, and it must be revised. Whenever the natural language specification specifies a response, this should be documented for requirements traceability. This continues until a "rule" is discovered which specifies stimulus histories giving a common response. The language of regular expressions, extended with special refinements, is used to specify these rules. The investigation of stimulus histories and the creation of rules continues until the rules form a complete and consistent black box specification. The completeness and consistency of such a specification can be verified by the construction of a Mealy machine, though it will be readily apparent in a complete enumeration.

Next, developers transform the black box into a Mealy machine and partition the state space of the Mealy machine by introducing state variables on finite domains and specifying the values of each state variable. Once these state variables completely partition the state space (every block has no more than one member) then the Mealy machine is transformed into a state box specification. Algorithms are presented for every step of the process, and can be found in Appendix A. Appendix B contains a detailed example of the complete process.

Chapter 2

Notation and Definitions

2.1 Notation and Basic Concepts

Much of the notation used in this document is taken from Grätzer's *Universal Algebra* [10] and from Gécseg's *Products of Automata* [9], with minor departures indicated.

Sets are commonly denoted by upper-case Latin or Greek letters, while elements of sets are commonly denoted by lower-case Latin or Greek letters. The empty set is denoted $\varnothing$. If $B \subseteq A$, we say the complement of B with respect to A is the difference $A - B$. When the set A is understood from context, we write $\overline{B}$ for $A - B$. The number of distinct elements in a finite set is denoted $|A|$, and the power set of A is denoted $\mathcal{P}(A)$.

Definition 2.1 If $B \subseteq \mathcal{P}(A)$ then we say B is a *system* over A. We may index a system with an equipotent set $\mathbf{I}$ and write $(A_i | i \in \mathbf{I})$ for the collection of sets in the system. We call such an indexed system a *family.*

For convenience we will represent the union, intersection, and direct (or Cartesian) product of all sets in a family by $\bigcup (A_i | i \in \mathbf{I})$, $\bigcap (A_i | i \in \mathbf{I})$, and $\Pi (A_i | i \in \mathbf{I})$, respectively. For the product, $\mathbf{I}$ should be some linearly-ordered set, such as the natural numbers. We will extend the notion of a family to refer to indexed collections of any sort.

Definition 2.2 For $i \in \mathbf{I}$ let pr_i denote the mapping $\mathrm{pr}_i : \Pi (A_j | j \in \mathbf{I}) \to A_i$ which, for each element in the product, chooses the element contributed by the ith set of the family. We refer to the mapping pr_i as the ith *projection.*

If $\mathbf{I} = \{1, 2, \ldots, n\}$ for some positive natural number n, we may write $1 \le i \le n$ in place of $i \in \{1, 2, \ldots, n\}$ and $A_1 \times A_2 \times \cdots \times A_n$ in place of $\Pi (A_i | i \in \mathbf{I})$. In this case we represent elements of $\Pi (A_i | i \in \mathbf{I})$ as ordered tuples $\langle a_1, a_2, \ldots, a_n \rangle$, where each $a_i \in A_i$. Then $\mathrm{pr}_i \langle a_1, a_2, \ldots, a_n \rangle = a_i$ for all $1 \le i \le n$.

Definition 2.3 Let π_1 and π_2 be two partitions of a set A. Then π_1 is a *refinement* of π_2, written $\pi_1 \leq \pi_2$, if and only if every block of $c \in \pi_1$ is a subset of some block $d \in \pi_2$.

Let π_1 be a partition of A and $c \in \pi_1$ a block. If π_2 is a partition of c, then clearly $\pi_3 = (\pi_1 - \{c\}) \cup \pi_2$ is a partition of A with $\pi_3 \leq \pi_1$.

Let ρ be an equivalence relation on A. We write $a \equiv b\,(\rho)$ for $\langle a, b\rangle \in \rho$, and denote the equivalence class for $a \in A$ by $[a]\rho$. The set of all equivalence classes is the quotient set, denoted A/ρ. Note that we have $\rho_1 \subseteq \rho_2$ if and only if $A/\rho_1 \leq A/\rho_2$.

Definition 2.4 Let ρ be any equivalence relation. The *natural mapping* $\text{nat}\,\rho : A \to A/\rho$ is given by $\text{nat}\,\rho(a) = [a]\rho$.

Definition 2.5 Let $(p_i | 1 \leq i \leq n)$ be a collection of predicates. These predicates are *disjoint* on some set A if and only if the sets in the family $\left(\{a \in A | p_i(a)\} \,\middle|\, 1 \leq i \leq n \right)$ are disjoint. If the union of these sets is the orig-

inal set A, then the predicates *cover* the set A. If the predicates are disjoint on and cover A, then they *partition* A.

Definition 2.6 Let $f:A \to B$ be a mapping, and $C \subseteq A$. Then the *restriction of* f *to* C, denoted $f|C$, is the mapping $f \cap (C \times B)$.

Definition 2.7 The *kernel* of a mapping f, denoted $\ker f$, is the equivalence relation given by $a \equiv b \,(\ker f)$ if and only if $f(a) = f(b)$.

Definition 2.8 Let $A \subseteq B$. The mapping $f:A \to B$ such that $f(a) = a$ for all $a \in A$ is known as the *inclusion mapping*. If $A = B$ the inclusion mapping is also the *identity mapping*.

The product of mappings will be represented as in Grätzer [10], except that we will use an intervening dot to avoid ambiguity. So, the product of two mappings $\phi:A \to B$ and $\psi:B \to C$ is the mapping $\phi \bullet \psi:A \to C$.

We will make use of the basic decomposition theorem for mappings, which is proven as Theorem 1 in §0.2 of Grätzer [10], and stated below.

Theorem 2.1 Let $f:A \to B$ be a mapping, and put $\rho = \ker f$. Then ρ is an equivalence relation on A, $f(A)$ is a subset of B, and there is a decomposition of f,

f = nat $\rho \bullet f' \bullet \mu$, where nat $\rho : A \to A/\rho$ is a surjection, $f' : A/\rho \to f(A)$ is a bijection, and $\mu : f(A) \to B$ is the inclusion mapping, an injection. □

This situation is illustrated by the commutative diagram below (Fig. 2.1). If f is a surjection as well, then μ is the identity mapping and we can discard it, giving the decomposition as f = nat $\rho \bullet f'$ where f' is a bijection.

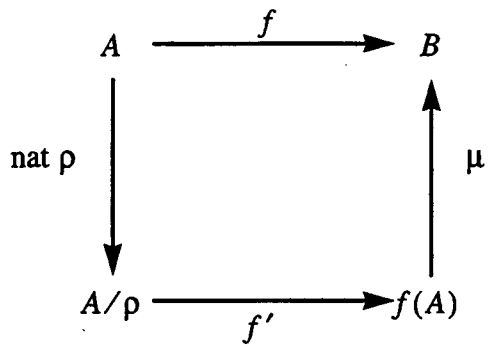


Figure 2.1: Basic Decomposition Theorem for Mappings

We can relax our requirements and choose $\rho \subseteq \ker f$. This is sufficient to uniquely determine a decomposition f = nat $\rho \bullet f'$ (theorem 3.3 of Cohn [4]), except that f' is now a surjection. From the preceding we can uniquely determine a function $f : A \to B$ by first creating a partition π of A and then constructing a surjection $f' : \pi \to B$ which maps blocks onto the selected range.

2.2 Regular Expressions

Definition 2.9 An *alphabet* is a finite, non-empty set of *symbols*.

Definition 2.10 An *sequence* over an alphabet X (also called an *X-sequence*) is a finite list of symbols from X, juxtaposed.

Definition 2.11 The number of occurrences of symbols in a sequence u is the *length* of the sequence, denoted $|u|$.

We denote the special empty sequence (the sequence of no symbols, or whose length is zero) by λ. X^* denotes the set of all sequences which may be formed from symbols in X, including λ. Note that X^* is closed under concatenation by juxtaposition, and that $\lambda u = u\lambda = u$ for any sequence u. Thus, X^* is a free monoid. We will make no distinction between a single symbol from X and the sequence composed of only one occurrence of that symbol. We can thus view X as the set of all sequences of length one. Though X^* is a countably-infinite set, every sequence is of finite length.

Let u and v be sequences over X. We denote the concatenation of v onto the right end of u by $u \cdot v$, or sometimes just uv. We extend the concatenation opera-

tion to sets of sequences as follows. Let A and B be sets of sequences over X. Then $A \cdot B = \{uv | (u \in A) \text{ and } (v \in B)\}$. Concatenation, both of sequences and sets of sequences, is associative.

Definition 2.12 Let A be a set of sequences over X.

$A^0 = \{\lambda\}$, and

$A^n = A \cdot A^{n-1}$ for any positive integer n.

Definition 2.13 The *iterate* A^* is the set $A^* = \bigcup_{i=0}^{\infty} A^i$.

Thus, X^n is the set of all sequences over X of length n and X^* is the set of all sequences over X. We use the notation X^+ to denote $X^* - \{\lambda\}$, which is the free semigroup generated by X.

Consider the Boolean algebra defined on $\mathcal{P}(X^*)$, where the universal set is X^*. Let A and B be elements of the Boolean algebra. The following Boolean functions are used in this work.

Definition 2.14 Let $\langle \mathcal{P}(X^*), \cap, \cup, \bar{}, \varnothing, X^* \rangle$ be a Boolean algebra with elements A and B. The *union* of A and B, denoted $A \cup B$, is $\{u | (u \in A) \text{ or } (u \in B)\}$.

The *intersection* of A and B, denoted $A \cap B$, is $\{u | (u \in A) \text{ and } (u \in B)\}$.

The *difference* from A to B, denoted $A - B$, is $\{u | (u \in A) \text{ and } (u \notin B)\}$.

The *complement* of A is $\bar{A} = X^* - A$.

The following definition is given in Brzozowski [3].

Definition 2.15 The set of *regular expressions* over an alphabet X is defined recursively by the following three rules.

i. Each symbol in X and the special symbols λ and $\varnothing$ are regular expressions.

ii. Let A and B be regular expressions and let f be a Boolean function of A and B. The following are also regular expressions: $(A \cdot B)$, A^*, and $f(A, B)$.

iii. Nothing is a regular expression unless it follows from a finite number of applications of the above rules.

Regular expressions denote sets. Any set which may be denoted by a regular expression is a *regular set*. We will make no notational distinction between an expression and the set denoted by that expression.

We may verify that a given regular expression denotes the empty set ($\varnothing$) by any of several methods. In particular, Brzozowski introduces the idea of the deriv-

ative of a regular expression. Brzozowski proves that any regular expression has a finite number of distinct derivatives (theorem 4.3 in [3]), and gives a procedure for generating all derivatives of a regular expression. If no derivative of a regular expression contains the empty sequence λ, then the expression is equivalent to $\varnothing$.

Definition 2.16 A mapping θ closed on the set of regular expressions is a regular operator.

Examples of regular operators are given in §5.1.

2.3 Finite State Machines

2.3.1 Automata

Definition 2.17 An *automaton* is a mathematical structure $\mathcal{A} = \langle A, X, \delta \rangle$, where A is a finite non-empty set of *states*, X is an alphabet called the *input alphabet*, and $\delta: A \times X \rightarrow A$ is a total function called the *state transition function*.

We extend the state transition function to sequences from X^* as follows:

$\forall a \in A, \delta^*(a, \lambda) = a$, and

$\forall a \in A, \forall x \in X, \forall u \in X^*, \delta^*(a, ux) = \delta(\delta^*(a, u), x)$.

Throughout the rest of this document, we will make no distinction between δ and δ^*, and will simply write δ for both. If $\delta(a, x) = b$, where $b \in A$ is a state and $x \in X$ is a symbol, then we say there is a *transition from* state a *to* state b *on* symbol x.

Let $\mathcal{A} = \langle A, X, \delta \rangle$ be an automaton, and let $a, b \in A$ be two states. If there is some string $u \in X^*$ such that $\delta(a, u) = b$ then state b is *reachable from* state a.

Definition 2.18 Let $B \subseteq A$ be a set of states, and let C be defined recursively as follows:

i. $B \subseteq C$,

ii. $b \in B$ and $x \in X$ implies $\delta(b, x) \in C$, and

iii. nothing else is in C except by a finite number of applications of the above two rules.

We refer to $\langle C, X, \delta | C \times X \rangle$ as the *subautomaton* of $\mathcal{A}$ generated by B. If an automaton can be generated by a singleton set, we say the automaton is *connected.*

Definition 2.19 Let $\mathcal{A}_1 = \langle A_1, X, \delta_1 \rangle$ and $\mathcal{A}_2 = \langle A_2, X, \delta_2 \rangle$ be two automata. The *automata product* $\mathcal{A}_1 \times \mathcal{A}_2$ is $\mathcal{A}_3 = \langle A_1 \times A_2, X, \delta_3 \rangle$, where for all $a \in A_1$, $b \in A_2$, and $x \in X$, we have $\delta_3(\langle a, b \rangle, x) = \langle \delta_1(a, x), \delta_2(b, x) \rangle$.

Note that $\mathcal{A}_1 \times \mathcal{A}_2$ and $\mathcal{A}_2 \times \mathcal{A}_1$ are isomorphic. Additionally, the automata product can be shown to be associative. We conclude this section with the following definition, which will prove useful later.

Definition 2.20 Let $\langle A, X, \delta \rangle$ be an automaton, and let $\dot{a} \in A$ be a distinguished state. We define the function $\Psi_{\dot{a}}: A \to \mathcal{P}(X^*)$ such that

$\Psi_{\dot{a}}(a) = \{u \in X^* \mid \delta(\dot{a}, u) = a\}$ for all states $a \in A$.

Note that if an automaton is connected, then there is at least one state $\dot{a} \in A$ such that $\Psi_{\dot{a}}(a) \neq \varnothing$ for all states $a \in A$.

2.3.2 Recognizers

Definition 2.21 A *recognizer* is a mathematical structure $\mathcal{R} = \langle A, X, \delta, \dot{a}, F \rangle$, where $\langle A, X, \delta \rangle$ is an automaton, $\dot{a} \in A$ is a special state called the *initial*

state, and $F \subseteq A$ is a distinguished collection of states called the *accepting states*.

We say a sequence u is *accepted* by $\mathcal{R}$ if and only if $\delta(\dot{a}, u) \in F$. Otherwise we say the sequence u is *rejected* by $\mathcal{R}$. The *language* of a recognizer is the set of all sequences accepted by the recognizer, and is denoted $L(\mathcal{R})$.

While it is common to use a zero subscript to denote the initial state (for instance, a_0) we will use the subscript as an index. We will therefore use the dot to denote the initial state; thus, $\dot{a}_i$ denotes the initial state of the ith machine in a family.

Theorems 2.3 and 2.4 in Hopcroft and Ullman [11] provide a correspondence between regular languages and recognizers. These two theorems can be paraphrased as follows.

Theorem 2.2 A language is accepted by a finite recognizer, if and only if it can be denoted by a regular expression. □

Throughout the rest of this work, it is assumed unless otherwise stated that all recognizers $\langle A, X, \delta, \dot{a}, F \rangle$ can be generated by the singleton $\{\dot{a}\}$; that is, all states are reachable from the initial state.

There are many methods for producing a recognizer for a given regular expression, however we are primarily interested in minimum recognizers; that is, recognizers for which $|A|$ is minimized.

Definition 2.22 A recognizer $\langle A, X, \delta, \dot{a}, F\rangle$ is *minimum* if and only if for every pair of states a, b and for all sequences $u \in X^*$ we have

$[\delta(a, u) \in F \Leftrightarrow \delta(b, u) \in F]$ implies $a = b$.

Theorem 2.3 Let $\mathcal{R} = \langle A, X, \delta, \dot{a}, F\rangle$ be a minimum recognizer. There is no recognizer $\mathcal{R}' = \langle A', X, \delta', \dot{a}', F'\rangle$ with $|A'| < |A|$ such that $L(\mathcal{R}) = L(\mathcal{R}')$. □

The above theorem is a consequence of the Myhill-Nerode Theorem, which is theorem 3.9 in Hopcroft and Ullman [11]. Additionally, theorem 3.10 in Hopcroft and Ullman demonstrates the uniqueness (up to an isomorphism) of the minimum automaton.

One can use derivatives of the regular expression to produce a minimum recognizer (see theorem 6.1 in Brzozowski [3]), or one may employ some other method, such as the ones outlined in Mano [14] and Hopcroft and Ullman [11], where a recognizer is produced and then minimized.

The product of two recognizers is defined in terms of the product of the underlying automata.

Definition 2.23 Let $(\mathcal{R}_i | 1 \leq i \leq k)$ be a family of recognizers in which $\mathcal{R}_i = \langle A_i, X, \delta_i, \dot{a}_i, F_i \rangle$. The *recognizer product* $\Pi(\mathcal{R}_i | 1 \leq i \leq k)$ is $\mathcal{R} = \langle A, X, \delta, \dot{a}, F \rangle$, where $\langle A, X, \delta \rangle$ is the product of all the underlying automata of the recognizers, $\dot{a} = \langle \dot{a}_1, \dot{a}_2, \ldots, \dot{a}_k \rangle$, and $F = \{a \in A | \forall 1 \leq i \leq k, \mathrm{pr}_i a \in F_i\}$.

We now return to the Ψ function and demonstrate that its range consists solely of regular sets. This will be important later in this work.

Theorem 2.4 Let $\langle A, X, \delta \rangle$ be an automaton. $\Psi_{\dot{a}}(a)$ is a regular set for any $a \in A$.

Proof: Let $\langle A, X, \delta, \dot{a}, \{a\} \rangle$ be a recognizer in which a is the only accepting state and $\dot{a}$ is the initial state. Then the language accepted by this recognizer is $\{u \in X^* | \delta(\dot{a}, u) \in \{a\}\}$ which by Theorem 2.2 is a regular set. Since $\{a\}$ is a singleton, we might as well write $\{p \in X^* | \delta(\dot{a}, p) = a\}$, which is $\Psi_{\dot{a}}(a)$, by definition. □

Definition 2.24 A regular predicate is a predicate on X^* which can be decided by a finite recognizer $\langle A, X, \delta, \dot{a}, F \rangle$; i.e., $\forall u \in X^*, p(u) \Leftrightarrow \delta(\dot{a}, u) \in F$.

2.3.3 Sequential Machines

A sequential machine specifies a mapping from sequences to responses. We will discuss two forms of sequential machines: Moore and Mealy machines.

Definition 2.25 A *Moore machine* is a mathematical structure $M = \langle A, X, Z, \delta, \gamma, \dot{a} \rangle$ where $\langle A, X, \delta \rangle$ is an automaton, $\dot{a} \in A$ is the *initial state*, Z is an alphabet called the *output alphabet*, and $\gamma: A \rightarrow Z$ is the *output mapping*. Further, we define the *machine function* $M: X^* \rightarrow Z$ to be $M(u) = \gamma(\delta(\dot{a}, u))$ for all $u \in X^*$.

Definition 2.26 A *Mealy machine* is a mathematical structure

$Me = \langle A, X, Z, \delta, \gamma, \dot{a}, \dot{z} \rangle$ where $\langle A, X, \delta \rangle$ is an automaton, $\dot{a} \in A$ is the *initial state*, Z is an alphabet called the *output alphabet*, $\dot{z} \in Z$ is a special output symbol called the *initial output*, and $\gamma: A \times X \rightarrow Z$ is the *output mapping*. Further, we define the *machine function* $Me: X^* \rightarrow Z$ to be

$Me(\lambda) = \dot{z}$ and

$Me(ux) = \gamma(\delta(\dot{a}, u), x)$, where $u \in X^*$ and $x \in X$.

As usual, we will assume unless stated otherwise that the underlying automaton of every sequential machine is connected and may be generated by the single-

ton $\{\dot{a}\}$. In a Mealy machine, whenever we have $\gamma(a, x) = z$, we say the transition from a on symbol x *produces* z. Our formulation of Mealy machines differs from that given by Hopcroft and Ullman [11] only in that it allows an initial output. This removes the difficulty in handling the empty string λ. These two formulations of a sequential machine are equivalent, as the following two theorems demonstrate.

Theorem 2.5 Let $Me = \langle A, X, Z, \delta, \gamma, \dot{a}, \dot{z} \rangle$ be a Mealy machine. Then there exists a Moore machine M' such that for all sequences $u \in X^*$, $Me(u) = M'(u)$. □

Theorem 2.6 Let $M = \langle A, X, Z, \delta, \gamma, \dot{a} \rangle$ be a Moore machine. There exists a Mealy machine Me' such that for all sequences $u \in X^*$, $M(u) = Me'(u)$. □

These two results (with the slight modification here that the two machines can now agree at λ) are proven constructively as theorems 2.6 and 2.7 in Hopcroft and Ullman [11], and give us a method for constructing a Moore machine from a Mealy machine and vice versa, such that the machine function is preserved. We will next show how a Moore machine explicitly encodes its own function decomposition.

Theorem 2.7 Let $M = \langle A, X, Z, \delta, \gamma, \dot{a} \rangle$ be a Moore machine. $\{\Psi_{\dot{a}}(a) \mid a \in A\}$ is a refinement of $X^* / (\ker M)$.

Proof: Let $a \in A$ and let $u, v \in \Psi_{\dot{a}}(a)$. Then $\delta(\dot{a}, u) = \delta(\dot{a}, v) = a$ by the definition of $\Psi_{\dot{a}}(a)$ and we have $\gamma(a) = \gamma(\delta(\dot{a}, u)) = \gamma(\delta(\dot{a}, v))$. This means $M(u) = M(v)$, and $u \equiv v \,(\ker M)$. Thus, every $\Psi_{\dot{a}}(a)$ must be contained in some block of $X^* / (\ker M)$. □

With Theorem 2.7 we can apply the basic decomposition theorem for mappings to the machine function of a Moore machine $M = \langle A, X, Z, \delta, \gamma, \dot{a} \rangle$. Each state $a \in A$ of the Moore machine corresponds to a block $\Psi_{\dot{a}}(a)$ of a partition of X^*, and we have a surjection $f' : \{\Psi_{\dot{a}}(a) \mid a \in A\} \to Z$ defined by $f'(\Psi_{\dot{a}}(a)) = \gamma(a)$. Letting ρ be the equivalence relation induced by $\{\Psi_{\dot{a}}(a) \mid a \in A\}$ on X^*, we have the decomposition $M = \text{nat}\, \rho \bullet f'$, which may be read directly from the Moore machine.

Later in this work we will be interested in combining Mealy machines and reducing them. We therefore define the product of two Mealy machines, and then extend the idea of minimality to the Mealy machine.

Definition 2.27 Let $Me_1 = \langle A_1, X, Z, \delta_1, \gamma_1, \dot{a}_1, \dot{z}_1 \rangle$ and

$Me_2 = \langle A_2, X, Z, \delta_2, \gamma_2, \dot{a}_2, \dot{z}_2 \rangle$. Let $f: Z \rightarrow \{1, 2\}$ be a function which chooses one of the two machines. The *machine product* $Me_1 \times Me_2$ is $\langle A_1 \times A_2, X, Z, \delta_3, \gamma_3, \langle \dot{a}_1, \dot{a}_2 \rangle, \dot{z}_3 \rangle$, where $\langle A_1 \times A_2, X, \delta_3 \rangle$ is the automata product of the underlying automata,

$\gamma_3(\langle a, b \rangle, x) = \gamma_1(a, x)$ when $f(\gamma_1(a, x)) = 1$ and

$\gamma_3(\langle a, b \rangle, x) = \gamma_2(b, x)$ otherwise,

and if $f(\dot{z}_1) = 1$ then $\dot{z}_3 = \dot{z}_1$, and $\dot{z}_3 = \dot{z}_2$ otherwise.

Definition 2.28 Let a and b be two states of a Mealy machine $\langle A, X, Z, \delta, \gamma, \dot{a}, \dot{z} \rangle$. We say a and b are *indistinguishable* if and only if for all sequences $u \in X^*$, and for all symbols $x \in X$, $\gamma(\delta(a, u), x) = \gamma(\delta(b, u), x)$.

Definition 2.29 A Mealy machine $\langle A, X, Z, \delta, \gamma, \dot{a}, \dot{z} \rangle$ is *minimum* if and only if for every pair of states a and b in the machine, a and b indistinguishable implies $a = b$.

Next, we show that this is the correct definition of minimality—namely that $|A|$ is minimized.

Theorem 2.8 Let $Me = \langle A, X, Z, \delta, \gamma, \dot{a}, \dot{z} \rangle$ be a minimum Mealy machine. There is no Mealy machine $Me' = \langle A', X, Z, \delta', \gamma', \dot{a}', \dot{z} \rangle$ such that $Me(u) = Me'(u)$ for all sequences $u \in X^*$ and $|A'| < |A|$.

Proof: Assume otherwise. We have $Me(u) = Me'(u)$ for all sequences $u \in X^*$, and $|A'| < |A|$. We define a relation $\tau \subseteq A \times A'$ as follows. Let $a \tau c$ exactly when there exists a sequence $u \in X^*$ such that $\delta(\dot{a}, u) = a$ and $\delta'(\dot{a}', u) = c$. Since every state $b \in A'$ is reachable, there is some sequence $u \in X^*$ such that $\delta'(\dot{a}', u) = b$. Let $\delta(\dot{a}, u) = a$. Then $a \tau b$. Since this holds for any state $b \in A'$, τ is onto. By the pigeon hole principle there must be two states $a, b \in A$, $a \neq b$, such that for some $c \in A'$ we have $a \tau c$ and $b \tau c$. We pick sequence u_1 such that $\delta(\dot{a}, u_1) = a$ and $\delta'(\dot{a}', u_1) = c$, and sequence u_2 such that $\delta(\dot{a}, u_2) = b$ and $\delta'(\dot{a}', u_2) = c$. We then proceed as follows for any sequence $v \in X^*$ and symbol $x \in X$.

$\gamma(\delta(a, v), x)$	$= \gamma(\delta(\delta(\dot{a}, u_1), v), x)$	substitution
	$= \gamma(\delta(\dot{a}, u_1 v), x)$	definition of δ
	$= Me(u_1 v x)$	definition of machine function
	$= Me'(u_1 v x)$	equality of machine functions

$= \gamma'(\delta'(\dot{a}', u_1 v), x)$ definition of machine function

$= \gamma'(\delta'(\delta'(\dot{a}', u_1), v), x)$ definition of δ'

$= \gamma'(\delta'(c, v), x)$ substitution

$= \gamma'(\delta'(\delta'(\dot{a}', u_2), v), x)$ substitution

$= \gamma'(\delta'(\dot{a}', u_2 v), x)$ definition of δ'

$= Me'(u_2 v x)$ definition of machine function

$= Me(u_2 v x)$ equality of machine functions

$= \gamma(\delta(\dot{a}, u_2 v), x)$ definition of machine function

$= \gamma(\delta(\delta(\dot{a}, u_2), v), x)$ definition of δ

$= \gamma(\delta(b, v), x)$ substitution

This contradicts Me minimum. □

2.4 Reduction Systems

When constructing specifications by enumeration, it will be convenient to use results from reduction systems. The definitions and theorem in this section are based closely on Book and Otto [1].

Definition 2.30 Let A be a set and η a reflexive and transitive binary relation on A. Then $\langle A, \eta \rangle$ is a *reduction system* and η is the *reduction relation.*

Definition 2.31 Let $a \in A$ be an element of the reduction system. If there is no $b \in A$ with $b \neq a$ such that $a \eta b$, then a is *irreducible;* otherwise a is *reducible.*

Definition 2.32 Let $a, b \in A$ be elements of the reduction system. Then b is a *normal form* for a if and only if $a \equiv b \, (\rho_\eta)$ and b is irreducible, where ρ_η is the equivalence relation resulting from taking the symmetric closure of η.

Definition 2.33 A reduction system $\langle A, \eta \rangle$ is *Church-Rosser* if and only if for all $a, b \in A$, $a \equiv b \, (\rho_\eta)$ implies there exists a $c \in A$ such that $a \eta c$ and $b \eta c$.

Definition 2.34 A reduction system $\langle A, \eta \rangle$ is *noetherian* if and only if there is no infinite sequence $a_1, a_2, \ldots \in A$ such that for all $i \geq 0$, $a_i \eta a_{i+1}$.

The following is proven as theorem 1.1.12 in Book and Otto [1].

Theorem 2.9 If $\langle A, \eta \rangle$ is both noetherian and Church-Rosser, then for every $a \in A$, $[a]\rho_\eta$ has a unique normal form. □

This result tells us when we can expect a reduction system to have exactly one irreducible element for each equivalence class.

Chapter 3

The Black Box

The first step in the specification of a software system is to identify the system boundary. The system boundary can be identified by listing the "users" of the system, where a user may be a human or another hardware or software system. The users send stimuli across the system boundary into the system, and receive responses across the system boundary from the system. Once we have identified the system boundary, we proceed to enumerate all sequences of stimuli and their responses.

A *stimulus* is a discrete event originated by a user which may have an effect on the software system's behavior. A *response* is system behavior observable by a user. Let S denote the set of all stimuli, and let R denote the set of all responses. These two sets contain all information which crosses the system boundary, and we may derive a "black box" view of the software from them.

Since software is deterministic, and since the only source of information available to our software is in S, we can specify the software's external behavior as a

complete function $BB: S^* \rightarrow R$, where S^* is the free monoid generated by S. The elements of S^* are S-sequences, which we refer to as stimulus histories. We refer to the function BB as the *black box function*, and refer to its representation as a *black box specification*, or just a *black box*.

3.1 Stimuli and Responses

We represent the stimuli and responses of complex systems in a hierarchical description, representing the stimulus and response sets as the cross product of some finite collection of sets $S = \Pi(S_i | 1 \leq i \leq m)$ and $R = \Pi(R_i | 1 \leq i \leq n)$. We make the following restrictions on these sets. The sets S_1 and R_1 are the stimulus and response *types*, respectively, and we require that they be finite. The remaining sets serve to discriminate between stimuli or responses of the same type, and we refer to them as *indices*, and say each set S_i *indexes* the stimuli. As an example, consider a graphical user interface. One might single-click or double-click on an object in a graphical user interface, with these actions giving different outcomes. S_1 might then be the set {single, double} and S_2 might be the set of "clickable" interface elements.

We allow index sets to be infinite sets—for instance, an index set may be the set of integers. This allows us to work with more familiar and understandable sets

than the finite sets of the implementation. Additionally, this allows us to defer some of the implementation details, such as how many files may be opened on the command line and how big a data set can be.

Since we wish to take advantage of formal language theory, we will require that the stimulus and response sets be finite alphabets. If we have allowed our index sets to be infinite, this will not be the case. To correct this problem, we construct finite-index, time-independent partitions S' on S and R' on R, and then work with these sets in place of S and R. One way to do this is to regard stimuli and responses which differ only by some particular index to be identical. Another way would be to create a finite-index, time-independent partition of each of the infinite index sets. The choice of partition depends in part on the problem being solved.

Some stimulus histories may not be possible because of the flow of processing (for instance, menu items may be disabled). We say these are *illegal by design*, since we wish to construct the software in a manner which makes them impossible. Other stimulus histories may not be possible because the system definition renders them impossible (for instance, we may require that every stimulus history begin with invocation of the software, so histories which do not start with invocation make no sense). We say these are *illegal by definition*.

We wish our black box function to be a complete function. That is, every stimulus history, even the illegal ones, must be mapped to some response. We introduce two special responses to handle the illegal cases. The "null" response (or no

response) denoted 0, if not already present in R, is added, and any histories which are illegal by design are mapped to this response (an error message is a legitimate response, and so histories which produce an error response are not illegal in either sense). The special response "illegal," denoted ω, is also added to the set R, and any histories which are illegal by definition are mapped to this response.

Additionally, it will be convenient to require $\mathrm{BB}(\lambda) = 0$ in all cases. This makes sense from the perspective of executing code. A software system can only respond to an event; it cannot "start itself." While we might exclude λ altogether, writing $\mathrm{BB}: S^+ \to R$, it is convenient to consider the domain to be a monoid and thus keep the identity λ. Another possibility is to require $\mathrm{BB}(\lambda) = \omega$, but this leads to some exceptions in handling illegal sequences during enumeration, and it is certainly possible for software to have an empty history.

Finally, we require that our stimulus set contain a null as well. The null stimulus, also denoted 0, is a special stimulus corresponding to "no event." The reason for this requirement will become clear in the sections on abstraction and box composition. The response to a null stimulus is always the null response (if the software does not receive an event, it cannot generate a response), giving $\forall u \in S^*, \mathrm{BB}(u0) = 0$. It will be seen that the null stimulus can usually be ignored.

3.2 The Black Box

Definition 3.1 A *black box* is a mathematical structure $\langle S, R, \Sigma, \Gamma \rangle$, where S and R are alphabets, Σ is a finite partition $(E_i | 1 \leq i \leq k)$ of S^*, and $\Gamma: \Sigma \rightarrow \mathcal{P}(R)$ is a function mapping Σ to non-empty elements of $\mathcal{P}(R)$.

So a black box consists of a partition and a function. Note that by Theorem 2.1, we have enough information to determine a unique mapping from S^* to $\mathcal{P}(R)$; this mapping is the black box function BB. We can do this as follows. Let e denote the equivalence relation on S^* such that $S^*/e = \Sigma$. Then we have

$$\forall p \in S^*, \mathrm{BB}(p) = \Gamma([p]\, e).$$

Note that we have allowed the range to consist of sets of responses, and ultimately we would like our black box function to give a unique response. We will deal with this problem in the section on refinements.

3.3 A Black Box Language

We next consider the requirements for a language in which a black box may be written such that we may always prove the completeness and consistency of the

resulting black box function. The function Γ has a finite domain, and can thus be enumerated. The primary concern is therefore the partition Σ of S^*. Let $\mathcal{L}$ be a black box language. Statements in $\mathcal{L}$ must therefore specify blocks of a partition of S^*, and the language requirements are the following.

1. An expression over $\mathcal{L}$ must denote a subset of S^*. We make no notational distinction between an expression over $\mathcal{L}$ and the set denoted by the expression.
2. For any two expressions E_1 and E_2 over $\mathcal{L}$, we can determine whether or not $E_1 \cap E_2 = \varnothing$.
3. For any family of expressions $(E_i | 1 \le i \le n)$ over $\mathcal{L}$, we can determine whether or not $\bigcup (E_i | 1 \le i \le n) = S^*$.

The first criterion simply assures us that the language applies to our problem. The second criterion guarantees that we can check the specification for consistency (verify that the blocks are disjoint, and therefore that BB is a function). The third criterion guarantees that we can check the specification for completeness (verify that the blocks cover S^*). We can then map each expression to the appropriate response (define Γ) and we have the complete black box function BB.

Note that most language classes do not satisfy these requirements. For instance, no decision procedure exists to determine whether or not two context-free grammars are disjoint (theorem 8.10, Hopcroft and Ullman [11]). The language of regular expressions does, however, satisfy the above requirements. Thus if we restrict our specification language to regular expressions, we can always check the specification for completeness and consistency.

To check completeness of a family of regular expressions $(E_i | 1 \le i \le n)$ over some alphabet X, we must verify $\bigcup (E_i | 1 \le i \le n) = X^*$. This question is equivalent to $\overline{\bigcup (E_i | 1 \le i \le n)} = \varnothing$ which can be mechanically verified using derivatives of regular expressions. To check the consistency of the family, we must verify $E_i \cap E_j = \varnothing$ whenever $1 \le i, j \le n$ and $i \ne j$. Again this can be done mechanically.

The restriction to regular expressions is confining in terms of power of representation. For instance, it generally does not allow for unbounded dynamic data. However changing to a more powerful language will, in general, preclude proof of completeness and consistency and also complicate the production of both the black and state boxes. We therefore propose the following method of iterative refinement of a black box partition.

3.3.1 Refinement

First partition S^* using regular expressions, thus obtaining some collection of regular expressions $(E_i | 1 \le i \le n)$. (A procedure to construct this partition will be given in the section on enumeration.) We can verify the completeness and consistency of this initial partition mechanically, using derivatives. Each block of the partition can then be mapped to some non-empty collection of responses, giving an initial black box. If each block can be mapped to a singleton (a unique response), then the resulting black box function gives a unique response for every stimulus sequence, and construction of the black box is done. Otherwise, we must refine the black box by refining the partition Σ until the range of Γ consists only of singletons. We can guarantee the uniqueness of this final black box by requiring that $|\mathrm{pr}_1 \Gamma| = |R|$. We can obtain this by combining any blocks E_1, E_2 for which $\Gamma(E_1) = \Gamma(E_2)$ into a new block $E = E_1 \cup E_2$ with $\Gamma(E) = \Gamma(E_1) = \Gamma(E_2)$. This gives the unique partition $S^* / (\ker \mathrm{BB})$ consisting of the level sets for black box function BB.

Let X be an alphabet. Given a partition π_1 of X^* and a partition π_2 of some block $A \in \pi_1$, the resulting collection of blocks $\pi_3 = (\pi_1 - \{A\}) \cup \pi_2$ forms a partition of X^*, with $\pi_3 \le \pi_1$. Given an initial partition, we will construct a refine-

ment by partitioning a block. We need only show that we have properly partitioned the block to show that we have a valid partition of X^*.

Let the initial partition of S^* be $(E_i | 1 \le i \le n)$. Therefore each $E_i \subseteq S^*$ is a set of sequences. Let $p_1, p_2, \ldots, p_m$ be a collection of predicates, called *refinement predicates*, which partition S^* (that is, the predicates are characteristic predicates for the blocks of some partition of S^*). For any E_i, the sets $\{u \in E_i | p_j(u)\}$ for $1 \le j \le m$ form a partition of E_i. We denote these sets $E_i[p_j]$ for $1 \le j \le m$.

The notation for the refinement predicates depends on the language extension used, and is discussed in the following section and in the chapter on advanced refinement techniques. Additionally, we wish to consider what restrictions must be placed on refinement predicates and their usage to insure that we can mechanically verify completeness and consistency of the resulting specification.

3.3.2 Predicates

A computable predicate on a set can be used to refine a partition of the set, since it induces a simple two-block partition on the set, one block for which the predicate holds and one block for which the predicate fails.

Let p and q be computable predicates on S^*, and let E be a block to refine. We can partition E into three blocks using these two predicates as follows.

1. $E[p]$ denotes the block for which p holds.
2. $E[\neg p]\ [q]$ denotes the block for which p fails and q holds.
3. $E[\neg p]\ [\neg q]$ denotes the block for which both p and q fail.

Note that we have used p to refine E, and we have used q to refine $E[\neg p]$. Clearly these blocks are disjoint and cover E.

If practitioners are required to use only the predicate name or its negation, then the check for completeness and consistency may be automated, and one need not even consider the semantics of the predicates to ensure completeness and consistency. We next consider how such a check may be performed.

To mechanically verify that predicates have been applied completely and consistently we construct a truth table. We list the predicates across the top, then list their possible values in the table. Finally we add a column in which we place the numbers of the expressions for which the given truth values hold. For instance the above three-expression example yields the truth table shown in Table 3.1. Note that expression one occurs twice in the table, since the value of q is irrelevant when p is true. If any cell in the third column is empty, then the predicates are

incompletely applied. If any cell in the third column has more than one entry, then the expressions listed in that cell are inconsistent.

Table 3.1: Truth Table for Example

p	q	Expression
true	true	1
true	false	1
false	true	2
false	false	3

While it may be tempting to create a predicate defined only on the particular block being partitioned, this should be avoided. Predicates so constructed are unlikely to be reusable and are likely to be harder to verify. The use of predicates will always complicate the production of a state box from the black box, as will be explained in the chapter on state boxes, however sometimes there is no alternative.

A regular predicate is a predicate that can be decided by a finite recognizer. Since the class of finite recognizers is equivalent to the class of regular expressions, all regular predicates can be specified by regular expressions. Since predicates complicate the production of a state box, any regular predicates should be replaced with the corresponding regular expressions. For instance let p be a regular predicate which is decidable by recognizer $\mathcal{R}$. Let $D = L(\mathcal{R})$ be the regular expression denoting the language of the recognizer $\mathcal{R}$. We have the following two relations.

$$E[p] = D \cap E$$

$$E[\neg p] = \overline{D} \cap E$$

The expression D may be very complex, and we may regard the predicate p as a shorthand for the above while constructing the specification. However, before producing the state box, one should replace the instances of p with its regular expression equivalent.

Finally, practitioners must ensure that any predicates used are computable for all stimulus histories. Otherwise the resulting specification (as a specification for executable software) will be useless.

3.4 Enumeration

The primary result of this research is the sequence enumeration method of specification writing. Straightforward, systematic eumeration of all sequences to produce an arguably complete, consistent, and correct specification is made practical by a collection of techniques based on controlling the growth of an inherently combinatorial process.

3.4.1 The Interface and Natural Language Specifications

Before one can begin production of a black box, one must have a description of the system. This description typically consists of two documents which form a bridge between the informal customer requirements and the formal black box specification. They are produced once a conceptual design has been selected which satisfies the requirements. The first is the *interface description*, a description of all external interfaces which explicitly identifies the system boundary and all stimuli. The second is the *natural language specification*, which describes the intended software response for all circumstances of use. Neither document needs to be complete or consistent—they can be corrected and improved as the black box is produced. Indeed, as the black box is produced it will likely be necessary to revise one or both documents. For instance, if one detects seeming "non-determinism" in the black box, one should look for an incomplete or inconsistent system boundary in the interface specification.

3.4.2 Sequence Enumeration

Throughout this section it will be necessary to distinguish between two black box functions. The first is the intended function $\beta: S^* \rightarrow R$ which is the function we intend the black box to denote when we have finished it. The second is the function denoted by the black box at any time during its construction. This function is

denoted simply BB . When finished, we hope to have $\beta = BB$, with the black box being an explicit representation of the intended function. The intended black box function is embodied in a combination of the natural language specification and the understanding of the practitioners and domain experts involved.

Once the interface and natural language specification have been created and the set of stimuli S and an initial response set R are known, the production of a black box specification begins. Practitioners generate sequences from S^* according to some fixed "alphabetical" ordering; first all sequences of length one are generated in alphabetical order, then all sequences of length two in alphabetical order, etc. (Note that the only sequence of length zero, λ, and any sequence ending in 0 can be automatically mapped to 0, and need not be explicitly generated.) As each sequence is generated, practitioners specify some response from R for the sequence (possibly the special responses 0 or ω) by consulting the natural language specification. It is possible, if some partition of S is being used in place of the actual stimulus set, that practitioners will not be able to specify the particular response for a sequence. In this case, practitioners should list each possible response, and the conditions under which each of the listed responses would be chosen. Practitioners may choose to refine their partitioning of the stimulus set to resolve this problem, or may wait and use predicates or other techniques to refine the resulting specification.

Definition 3.2 Let A be a countable set and B a set. Let $f{:}A \rightarrow B$ be a function. An *enumeration* of f is a well-ordered listing of the pairs comprising f.

An enumeration of the black box function lists sequences from S^* by length and by alphabetical order within a length, and identifies the appropriate response for each sequence listed. We denote the mapping of a sequence u to response r in an enumeration by $u \rightarrow r$.

This process of enumeration continues until the enumeration is complete or until practitioners discover a "rule" for specifying sequences which give a particular response or for which the same refinement criteria (such as a predicate) must be used to determine the response. For example, one may notice that all sequences which end in a particular stimulus give the same response, or that all sequences which end in a particular stimulus give one of two responses, but the specific response can only be determined using a certain predicate on the sequence. One then describes the rule using regular expressions and any required refinements. The enumeration of sequences and discovery of rules proceeds in this fashion until the rules are determined to be complete. Rule discovery may be partially automated, as will be discussed later in the section on implicit rules.

As each rule is produced, it should be checked against all other rules for consistency, as described earlier. If two rules are found to be inconsistent, one or both must be revised to correct the problem. One may create a finite recognizer to

accept or generate only the inconsistently-mapped sequences and thus gain insight into exactly which sequences are inconsistently mapped.

In the following discussion, several enumeration techniques are described in terms appropriate to enumeration practice. These could be stated and proved as theorems, but would appear as weak or obvious results.

Technique 3.1 (Illegal Prefix) Any sequence which has a prefix u mapped to ω must also be mapped to ω.

During enumeration if a sequence u is mapped to ω one need not consider any extensions of u.

Further, one may discover two types of equivalence among sequences in an enumeration.

Definition 3.3 Two sequences $u, v \in S^*$ are *Moore equivalent*, denoted $u \equiv v\ (\rho_M)$, if and only if $\forall w \in S^*, \beta(uw) = \beta(vw)$.

In other words, two sequences are Moore equivalent if they are intended to always give the same response when extended by identical futures. Note that we have allowed $w = \lambda$, and so we have $\beta(u) = \beta(v)$. This equivalence relation is used by Parnas in the Trace Assertion Method [19].

Definition 3.4 Two sequences $u, v \in S^*$ are *Mealy equivalent*, denoted $u \equiv v\,(\rho_{Me})$, if and only if $\forall w \in S^+, \beta(uw) = \beta(vw)$.

In other words, two sequences are Mealy equivalent if and only if they are intended to always give the same response when extended by identical nonempty futures. Mealy equivalence was used by Mao in his Box-Structure Development Method [15]. Note that we have specifically excluded $w = \lambda$, and may have the case that $\beta(u) \neq \beta(v)$. These two equivalences are related by the following result.

Theorem 3.1 $\rho_M \subseteq \rho_{Me}$.

Proof: Assume $u \equiv v\,(\rho_M)$. Then $\forall w \in S^*, \beta(uw) = \beta(vw)$ and, in particular, $\forall w \in S^+, \beta(uw) = \beta(vw)$, and we have $u \equiv v\,(\rho_{Me})$. □

Thus we see two Moore equivalent sequences must also be Mealy equivalent, whereas two Mealy equivalent sequences will also be Moore equivalent (if not already) after one more stimulus is added to the history.

As we are enumerating sequences in order by length and alphabetically within a each length, we may discover a Moore or Mealy equivalence between a sequence and a previously-mapped sequence. If we discover a Moore equivalence

in the enumeration, we have likely missed a Mealy equivalent shorter sequence in the enumeration. If we discover that the current sequence u is Moore equivalent to the earlier-mapped sequence v, then we simply note this in the enumeration. If we discover that the current sequence u is Mealy equivalent to the earlier-mapped sequence v, then we must not only note the equivalence, but also give the response for the sequence u. To prevent the mistake of forgetting to note the response in the case of a Mealy equivalence, we can require practitioners always to specify the response. Throughout the rest of this work, whenever two sequences are said to be equivalent, ρ_{Me} is intended unless otherwise specified.

We denote the mapping of a sequence u to the response r with equivalence to the previously-mapped sequence v by $u \rightarrow r/\equiv v$. Note that for any $u \in S^*$ we have $u0 \rightarrow 0/\equiv u$. Thus we need never consider the null stimulus in enumerations.

Definition 3.5 Given equivalence relation ρ_{Me}, let relation η be given by

> $\forall u, v \in S^*$, $u \eta v$ if and only if u comes no later than v in the alphabetical ordering and $u \equiv v\,(\rho_{Me})$. It follows from Definition 2.30 that $\langle S^*, \eta \rangle$ is a reduction system.

Lemma 3.1 The reduction system $\langle S^*, \eta \rangle$ is noetherian and Church-Rosser.

Proof: $\langle S^*, \eta \rangle$ has a least element (λ), and reduction is always to sequences earlier in alphabetical order. The reduction system is therefore noetherian, since every descending chain of reductions must terminate.

Choose sequences $u, v \in S^*$ such that $u \equiv v\,(\rho_{Me})$. Either $u = v$, or one of the two comes first in the alphabetical ordering. Thus, assume without loss of generality that either $u = v$ or u comes before v in the alphabetical ordering. Then we observe that $v \eta u$ and $u \eta u$. This holds for any pair of sequences equivalent under ρ_{Me}, and thus the reduction system is Church-Rosser. □

Theorem 3.2 (Canonical Sequence) Let $u \in S^*$ be a sequence. Then there is a unique sequence $v \in [u]\,\rho_{Me}$ which is irreducible.

Proof: Follows directly from Lemma 3.1 and Theorem 2.9. □

The unique normal forms for the reduction system defined by ρ_{Me} are called the *canonical sequences,* following the usage of Parnas [19].

A *reduced sequence* is a sequence in an enumeration which is noted as equivalent to a previously-mapped sequence. That is, if we have $u \rightarrow r / \equiv v$, then u is a reduced sequence. An *unreduced sequence,* then, is any sequence in an enumeration

which has not been reduced. An unreduced sequence may or may not be a canonical sequence, but all canonical sequences must be unreduced in an enumeration.

Technique 3.2 (Reduction) If $u \rightarrow r/\equiv v$ is in an enumeration for reduced v, then there is an unreduced sequence w in the enumeration such that u is equivalent to w, and we can replace $u \rightarrow r/\equiv v$ with $u \rightarrow r/\equiv w$.

We require practitioners to use only unreduced sequences in equivalences.

Technique 3.3 (Reduced Prefix) If, for a sequence u, we have $u \rightarrow r/\equiv v$ for unreduced previously enumerated sequence v, then no sequence with proper prefix u need be included in the enumeration.

If we have $u \rightarrow r/\equiv v$ in our enumeration, we need not consider any sequences with the proper prefix u, since those are already given by the enumeration (by replacing the prefix u with unreduced v). This provides the notion of a "complete" enumeration.

Definition 3.6 An enumeration is *complete* if and only if all sequences of some length $n \geq 0$ are either reduced to a previous sequence in the enumeration, or are mapped to ω.

Lemma 3.2 All sequences of length one will appear in a complete enumeration.

Proof: Since the only sequence of length zero, λ, is unreduced and legal, it must be extended by every $x \in S$. The resulting sequences are all sequences of length one. □

Theorem 3.3 (Complete Enumeration) A complete enumeration specifies a response for every sequence $u \in S^*$.

Proof: This theorem is a direct consequence of the Illegal Prefix (Technique 3.1) and Reduced Prefix (Technique 3.3) Techniques. From a complete enumeration, we may determine the response for any sequence $u \in S^*$ by the following algorithm.

i. If u is in the enumeration, we are done. Recall that we require a response be specified even when an equivalence is also specified.

ii. If u is not in the enumeration, find the longest proper prefix v of u in the enumeration. Such a v must exist, since all sequences of length one are guaranteed to be in the enumeration by Lemma 3.2. Additionally, such a v will either be mapped to ω or will be equivalent to some previously-enumerated sequence v'. (If an equivalence is not given for v and v is not mapped to ω, we must have the sequences vx for all $x \in S$

in the enumeration and can thus find a longer prefix.) If v is mapped to ω, then u must also be mapped to ω. If v is equivalent to previously-mapped v', then replace v with v' in u and repeat step i.

This algorithm must eventually terminate since we always have λ mapped to the null response, and we will thus find a response for u. □

The above results are formalized as Algorithm A.1.1: Enumeration.

If, as one is enumerating, one discovers that the enumeration is getting too long with little or no additional information being added, one may use a technique such as abstraction or stimulus reduction to reduce the size of the enumeration. For instance, one may discover that only a few problems remain to be solved, but these problems will not be encountered until sequences of length 100 are enumerated. One would then stop enumeration, construct an abstract stimulus set to address the remaining behavior, and enumerate with the new stimulus set. These techniques are discussed in Chapter 5.

It would be convenient if ρ_M or ρ_{Me} were refinements of ker β. Then the equivalences we discover would contribute to our knowledge of the intended black box partition $\Sigma = S^*/\ker \beta$.

Theorem 3.4 $\rho_M \subseteq \ker \beta$.

Proof: Assume $u \equiv v\,(\rho_M)$. Then $\forall w \in S^*, \beta(uw) = \beta(vw)$. In particular we have

$\beta(u\lambda) = \beta(v\lambda)$, and $\beta(u) = \beta(v)$, giving us $u \equiv v\,(\ker \beta)$. □

Discovering a Moore equivalence contributes to our knowledge of ker β. However, we can usually discover a Mealy equivalence among shorter sequences, and if we detect the equivalence among shorter sequences we will have to enumerate fewer sequences in total. We can always convert Mealy equivalences to Moore equivalences as follows. Assume that $u \rightarrow r/\equiv v$ for unreduced v (by Theorem 3.2 we can always find such an unreduced v). We replace this with $u \rightarrow r$ and add the sequences ux to the enumeration for each $x \in S$. If v is mapped to a response other than ω, then clearly each ux is Moore equivalent to vx, which must be in the complete enumeration. If v is mapped to ω, then clearly each ux is Moore equivalent to v. So we can always remove the troublesome Mealy equivalences from our enumeration after we are done.

We return to the discussion of equivalences by investigating the relations between the equivalences ρ_M, ρ_{Me}, and ker β. The following result shows when we can expect $\rho_M = \rho_{Me}$ to occur.

Theorem 3.5 $\rho_{Me} \subseteq \ker \beta$ if and only if $\rho_M = \rho_{Me}$.

Proof: Assume $\rho_M = \rho_{Me}$. Then by Theorem 3.4 we have $\rho_{Me} \subseteq \ker \beta$. Assume $\rho_{Me} \subseteq \ker \beta$. If $u \equiv v\,(\rho_{Me})$ then $\forall w \in S^+, \mathrm{BB}(uw) = \mathrm{BB}(vw)$. By our assumption we have $\mathrm{BB}(u) = \mathrm{BB}(v)$, and we can conclude that $\forall w \in S^*, \mathrm{BB}(uw) = \mathrm{BB}(vw)$, or $u \equiv v\,(\rho_M)$. By Theorem 3.1, this gives $\rho_M = \rho_{Me}$. □

From the above result we see that since in general we cannot conclude that $\rho_M = \rho_{Me}$, we cannot conclude that $\rho_{Me} \subseteq \ker \beta$. The opposite inclusion is also unlikely, as the following result shows.

Theorem 3.6 $\ker \beta \subseteq \rho_{Me}$ implies $\ker \beta = \rho_M$.

Proof: Assume $\ker \beta \subseteq \rho_{Me}$. Then $\mathrm{BB}(u) = \mathrm{BB}(v)$ implies $\forall w \in S^+, \mathrm{BB}(uw) = \mathrm{BB}(vw)$. Thus we have $\forall w \in S^*, \mathrm{BB}(uw) = \mathrm{BB}(vw)$, and by Theorem 3.4 we have $\ker \beta = \rho_M$. □

This process of explicit sequence enumeration insures the traceability-to-correctness of the resulting specification, and provides a systematic way to discover incompleteness and inconsistency in the natural language specification and interface specification. As a special effort to construct a correct specification, several

practitioners may concurrently perform the enumeration with a mechanical comparison of the results. Disagreements can then be resolved by clarifying and correcting the natural language specification and interface specification.

3.4.3 Implicit Rules

An enumeration, even a partial enumeration, encodes "implicit rules." These may be inferred from the enumeration by constructing a Mealy machine, obtaining the equivalent Moore machine, and then inferring regular expressions from the Moore machine. It will be seen that these rules give the Σ and Γ required for constructing a black box specification.

Let $\mathcal{E}$ be an enumeration. If the enumeration is incomplete, it is first completed by mapping any remaining unmapped sequences to the special response "incomplete," denoted ι. We can construct a special Mealy machine $Me_{\mathcal{E}} = \langle A, S, Y, \delta, \gamma, \lambda, 0 \rangle$ using Algorithm A.1.2: Enumeration Mealy Machine Construction. Now we need to show that the resulting machine function $Me_{\mathcal{E}}: S^* \rightarrow Y$ correctly encodes the enumeration.

Lemma 3.3 If $u \rightarrow r$ $(r \neq \omega)$ is in $\mathcal{E}$, then $\delta(\lambda, u) = u$ and $Me_{\mathcal{E}}(u) = r$.

Proof: We proceed by induction on $|u|$. If $u = \lambda$ then we note $\lambda \rightarrow 0$, $Me_{\mathcal{E}}(\lambda) = 0$, and $\delta(\lambda, \lambda) = \lambda$.

Assume the lemma holds for all $u \in S^*$ with $|u| \leq n$ for some $n \geq 0$. Assume $ux \rightarrow r$ is in $\mathcal{E}$. Then $\delta(u, x) = ux$ and $\gamma(u, x) = r$ by the construction of $M_{\mathcal{E}}$. By the inductive hypothesis we have $\delta(\lambda, u) = u$ and thus $ux = \delta(u, x) = \delta(\delta(\lambda, u), x) = \delta(\lambda, ux)$. Finally we observe that $Me_{\mathcal{E}}(ux) = \gamma(\delta(\lambda, u), x) = \gamma(u, x) = r$. □

Lemma 3.4 If $u \rightarrow \omega$ is in $\mathcal{E}$, then $\forall w \in S^*, Me_{\mathcal{E}}(uw) = \omega$.

Proof: We know $u \neq \lambda$, since $\lambda \rightarrow 0$. Let $u = u'x$ where $u' \in S^*$ and $x \in S$. By the construction of $Me_{\mathcal{E}}$ we have $\delta(u', x) = \omega$ and $\gamma(u', x) = \omega$. By Lemma 3.3 this gives $\delta(\lambda, u) = \omega$ and $Me_{\mathcal{E}}(u) = \gamma(\delta(\lambda, u'), x) = \omega$.

Let $w \in S^*$ and $y \in S$. By construction, $\delta(\omega, wy) = \omega$ and $\gamma(\delta(\omega, w), y) = \omega$. Thus $\forall v \in S^+, Me_{\mathcal{E}}(uv) = \omega$. □

Lemma 3.5 If no response for sequence u was given in the original enumeration $\mathcal{E}$, then $\forall w \in S^*, Me_{\mathcal{E}}(uw) = \iota$.

Proof: We know $u \neq \lambda$, since $\lambda \rightarrow 0$ is always in the enumeration. Let $u = u'x$ where $u' \in S^*$ and $x \in S$. By completion of the enumeration we know $u \rightarrow \iota$. By the construction of $Me_{\mathcal{E}}$ we have $\delta(u', x) = \iota$ and $\gamma(u', x) = \iota$. By Lemma 3.3 this gives $\delta(\lambda, u) = \iota$ and $Me_{\mathcal{E}}(u) = \gamma(\delta(\lambda, u'), x) = \iota$.

Let $w \in S^*$ and $y \in S$. By construction, $\delta(\iota, wy) = \iota$ and $\gamma(\delta(\iota, w), y) = \iota$. Thus $\forall v \in S^+, Me_{\mathcal{E}}(uv) = \iota$.

Lemma 3.6 If $u \rightarrow r/\equiv v$ is in $\mathcal{E}$, then $\forall w \in S^+, Me_{\mathcal{E}}(uw) = Me_{\mathcal{E}}(vw)$.

Proof: By Lemma 3.3 we know $\delta(\lambda, v) = v$. We know $u \neq \lambda$, since $\lambda \rightarrow 0$. Let $u = u'x$ where $u' \in S^*$ and $x \in S$. By Lemma 3.3 we have $\delta(\lambda, u') = u'$ and from the construction we have $\delta(u', x) = v$ and $\gamma(u', x) = r$. This gives $\delta(\delta(\lambda, u'), x) = v = \delta(\lambda, v)$. Thus, for all $w' \in S^*$ and $y \in S$ we have

$$
\begin{aligned}
Me_{\mathcal{E}}(uw'y) &= \gamma(\delta(\lambda, uw'), x) && \text{by definition of machine function} \\
&= \gamma(\delta(\delta(\lambda, u), w'), x) && \text{by definition of } \delta \\
&= \gamma(\delta(v, w'), x) && \text{substitution} \\
&= \gamma(\delta(\delta(\lambda, v), w'), x) && \text{substitution} \\
&= \gamma(\delta(\lambda, vw'), x) && \text{by definition of } \delta \\
&= Me_{\mathcal{E}}(vw'x) && \text{by definition of machine function}
\end{aligned}
$$

□

From Lemma 3.3, Lemma 3.4, Lemma 3.5, and Lemma 3.6 we can conclude that $Me_{\mathcal{E}}$ correctly encodes the mapping denoted by the enumeration $\mathcal{E}$.

We now have a Mealy machine $Me_{\mathcal{E}}$ which encodes the information obtained from our enumeration. We convert this into an equivalent Moore machine $M = \langle A', S, Y, \delta', \gamma', \dot{a} \rangle$ and note that, by Theorem 2.7, we can read off Γ and Σ as follows: $\Sigma = \{\Psi_{\dot{a}}(a) \mid a \in A'\}$ and $\Gamma(\Psi_{\dot{a}}(a)) = \gamma'(a)$ for all states $a \in A'$. This gives us the black box $\langle S, R, \Sigma, \Gamma \rangle$. It is important to note that this black box is composed of regular expressions only, since every $\Psi_{\dot{a}}(a)$ is a regular set by Theorem 2.4. Algorithm A.1.3: Black Box Construction follows naturally from Theorem 2.4 and Theorem 2.7. Algorithms for producing an equivalent Moore machine from a Mealy machine and for constructing a regular expression from a recognizer may be found in [11]. The resulting Σ and Γ are the partition and mapping guaranteed by Theorem 2.7.

Chapter 4

The State Box

The black box function is a total function from the set of all stimulus histories to the set of all responses. From an implementation view, we consider only the *essential* information from the stimulus history. In other words we wish to store some *state* information and then define a new function whose mapping is determined only by the current state and stimulus; given the present state the future behavior is independent of the past state.

This state-based view is called a *state box*, and in defining it we may make reference to stimuli, responses, and a state space. Given a black box several elements of the state box are fixed. First, the state box must denote the same total function as the black box, and thus have the same domain, range, and mapping. Second, the minimum state space is completely determined by the black box, since we need only store the "essential" information from the history. Thus it is possible to derive a state box from a given black box, although this may or may not be the state box that is ultimately coded.

New information may be introduced into the state box. The black box determines the minimum size of the state space, but not its makeup (how we encode or name the state space). We may also introduce redundant state information as a concession to the need for human understanding of the resulting state box. In particular, we would like to introduce state variables and specify their ranges, writing the state box in terms of these variables. This will ease our transition to executable code. These variables and ranges must fulfill the minimum state requirements, but beyond that practitioners are free to modify the state space for readability and understandability, as long as the total function is preserved.

4.1 Minimum State Derivation

4.1.1 Moore Machine Derivation

We wish to convert the black box into a state box, discovering the minimum state requirements. We do this by first converting the black box into a Moore machine and then transforming this Moore machine into the minimum Mealy machine. The resulting machine is then transformed directly into a state box.

At each step we show that the total function is preserved. The null stimulus 0 mentioned earlier (§3.1) will be ignored throughout this process. We simply require that for all states a, $\delta(a, 0) = a$ and $\gamma(a, 0) = 0$.

Definition 4.1 Let $\langle S, R, \Sigma, \Gamma \rangle$ be a black box. We combine any elements of Σ which are distinguished by non-regular refinements, giving a new partition Σ' composed only of regular sets with $\Sigma \leq \Sigma'$. Let Γ' be the mapping corresponding to Σ'; that is, let $\Gamma'(\sigma') = \{r \in \Gamma(\sigma) | \sigma \in \Sigma, \sigma \subseteq \sigma'\}$ for all $\sigma' \in \Sigma'$. The black box $\langle S, R, \Sigma', \Gamma' \rangle$ is the *regular approximation* of $\langle S, R, \Sigma, \Gamma \rangle$. A black box which denotes the same function as its regular approximation is a *regular black box.*

Let $\langle S, R, \Sigma, \Gamma \rangle$ be the black box. Σ is given by a collection of regular expressions which partition S^*, possibly with some additional non-regular refinements. For the time being we will remove all non-regular refinements from our partition, re-introducing them after we have constructed the Mealy machine. Let $\Sigma' = (E_i | 1 \leq i \leq k)$ be the resulting collection of regular expressions, let Γ' be the corresponding mapping, and let BB′ be the corresponding black box function. Thus $\langle S, R, \Sigma', \Gamma' \rangle$ is the regular approximation of $\langle S, R, \Sigma, \Gamma \rangle$.

By Theorem 2.2 every regular expression can be expressed as a finite recognizer. Construct the minimum recognizer $\mathcal{R}_i = \langle A_i, S, \delta_i, \dot{a}_i, F_i \rangle$ for each expression E_i in Σ', such that $L(R_i) = E_i$ for all $1 \leq i \leq k$. We consider some of the properties of this collection of recognizers. For the following results we fix a regu-

lar black box $\langle S, R, \Sigma', \Gamma' \rangle$ and the corresponding collection of recognizers $(\mathcal{R}_i \mid 1 \le i \le k)$.

Lemma 4.1 Let u be an arbitrary stimulus history. If Σ' is a partition of S^*, then exactly one of the recognizers $\mathcal{R}_i$ will accept u.

Proof: Assume both $\mathcal{R}_i$ and $\mathcal{R}_j$ accept u with $i \neq j$. Then it must be the case that $u \in L(\mathcal{R}_i) = E_i$ and $u \in L(\mathcal{R}_j) = E_j$, and thus $u \in E_i \cap E_j$, which contradicts E_i and E_j disjoint. Assume none of the recognizers accept u. Then it must be the case that $u \notin L(R_i) = E_i$ for any $E_i \in \Sigma'$. Thus $u \notin \bigcup \Sigma'$, a contradiction. □

Next we construct the connected recognizer product $\mathcal{R} = \Pi(\mathcal{R}_i \mid 1 \le i \le k)$ $= \langle A, S, \delta, \dot{a}, F \rangle$ and explore some of the properties of $\mathcal{R}$. This will give a second way to check completeness and consistency of the regular expressions in Σ' (verify that Σ' is a partition of S^*).

Theorem 4.1 If Σ' is a regular partition of S^* with $|\Sigma'| = k$ and $\mathcal{R} = \langle A, X, \delta, \dot{a}, F \rangle$ is the connected recognizer product formed for Σ', then for every $a \in A$ there is a unique i, $1 \leq i \leq k$, such that $\mathrm{pr}_i a \in F_i$.

Proof: Since every state in A can be generated from $\dot{a}$, for an arbitrary state $a \in A$ we know that there is some string $u \in S^*$ such that $\delta(\dot{a}, u) = a$. By Lemma 4.1 we know that at exactly one of the component recognizers, say $\mathcal{R}_i$, must accept u and therefore $\delta_i(\dot{a}_i, u) \in F_i$, and this value is unique. □

We can use the recognizer product to test the specification for completeness and consistency (at the level of regular expressions), and thus a separate test for completeness and consistency is not required. If incompleteness or inconsistency is detected, a regular expression for the inconsistently mapped or unmapped sequences can be created and inspected by the practitioners to characterize these sequences. For incompleteness the expression of interest is $\overline{\bigcup \Sigma'}$. For inconsistency, assume we know $\mathrm{pr}_i a \in F_i$ and $\mathrm{pr}_j a \in F_j$ for $i \neq j$ and some state $a \in A$. The expression of interest is therefore $E_i \cap E_j$.

We next transform $\mathcal{R}$ into a Moore machine $M = \langle A, S, Z, \delta, \gamma, \dot{a} \rangle$, where A, X, δ, and $\dot{a}$ are from $\mathcal{R}$ and Z is $\mathrm{pr}_2\Gamma'$. For each $a \in A$ and $x \in S$, let $\gamma(a, x) = \Gamma'(E_i)$ if and only if $\mathrm{pr}_i a \in F_i$. By Theorem 4.1 the value of i is unique.

Theorem 4.2 Let $M = \langle A, S, Z, \delta, \gamma, \dot{a} \rangle$ be the Moore machine produced for regular black box $\langle S, R, \Sigma', \Gamma' \rangle$ denoting function BB′. Then $\forall u \in S^*, M(u) = \mathrm{BB}'(u)$.

Proof: $M(u) = \gamma(\delta(\dot{a}, u))$ by definition of machine function. Let $a = \delta(\dot{a}, u)$, and let i be the value guaranteed by Theorem 4.1 such that $\mathrm{pr}_i a \in F_i$. Choose equivalence relation e such that $S^*/e = \Sigma'$. Then $u \in E_i = [u]\,e$, and $\gamma(a) = \Gamma'(E_i)$ by the construction. Substitution gives $\Gamma'(E_i) = \Gamma'([u]\,e)$, and by the definition of BB′, $M(u) = \Gamma'([u]\,e) = \mathrm{BB}'(u)$. □

Choose $Z = \{z_1, z_2, \ldots, z_k\}$ such that $\Gamma'(E_i) = z_i$ for all $1 \le i \le k$. Then $Z = \mathrm{pr}_2\Gamma'$ as required.

Theorem 4.3 Let $M = \langle A, S, Z, \delta, \gamma, \dot{a} \rangle$ be the Moore machine produced from the recognizer product as specified above. Then

$$\forall a \in A, \forall u \in S^*, \forall 1 \le i \le k, \ (\gamma(\delta(a, u)) = z_i) \Leftrightarrow (\delta_i(\mathrm{pr}_i a, u) \in F_i) \ .$$

Proof: Choose $a \in A$, $u \in S^*$, and $1 \le i \le k$.

$\gamma(\delta(a, u)) = z_i \Leftrightarrow \gamma(\delta(a, u)) = \Gamma'(E_i)$ substitution

$\Leftrightarrow \mathrm{pr}_i\delta(a, u) \in F_i$ construction of Moore machine

$\Leftrightarrow \delta_i(\mathrm{pr}_i a, u) \in F_i$ construction of recognizer product □

4.1.2 Derivation of the Minimum Mealy Machine

By Theorem 2.6 we can construct a Mealy machine, $Me' = \langle A, S, Z, \delta, \gamma', \dot{a}, \dot{z} \rangle$, which is equivalent to the Moore machine derived in the last section, and thus we have $\forall u \in S^*, \mathrm{BB}'(u) = Me'(u)$. We can do this by moving the response from a state to all arcs directed into that state. In other words, we have $\dot{z} = \gamma(\dot{a})$ and $\gamma'(a, x) = \gamma(\delta(a, x))$ for all $a \in A$ and $x \in S$. This allows us to preserve the state set and, in particular, preserve the useful properties of the state set, such as $\mathrm{pr}_i a \in A_i$.

We combine this result with the results from the last section to obtain a procedure for generating a Mealy machine $Me' = \langle A, S, Z, \delta, \gamma', \dot{a}, \dot{z} \rangle$ from a regular

black box $\langle S, R, \Sigma', \Gamma' \rangle$. Recall that we already have $Z = \mathrm{pr}_2\Gamma'$. The resulting procedure is Algorithm A.2.1: Black Box Mealy Machine Construction Algorithm.

We next consider how to reduce this machine to obtain the minimum Mealy machine. By Theorem 2.8 we know if we have a minimum Mealy machine, we have the smallest state set and also a unique machine (up to an isomorphism). This gives a unique Mealy machine for each black box. Unfortunately, the definition of minimality given in Definition 2.29 does not provide a simple means to reduce a Mealy machine to the minimum machine. We next consider such a method, and show that it does, in fact, produce the minimum Mealy machine for our particular case. It will be convenient to introduce the following connection between the Mealy machine and the recognizers used in the product.

Lemma 4.2 $\forall a \in A, \forall x \in S, \forall u \in S^*, \forall 1 \le i \le k,$

$$(\gamma'(\delta(a, u), x) = z_i) \Leftrightarrow (\delta_i(\mathrm{pr}_i a, ux) \in F_i) .$$

Proof: Let $1 \le i \le k$, $u \in S^*$, $x \in S$, and $a \in A$. Then

$\gamma'(\delta(a, u), x) = z_i \Leftrightarrow \gamma(\delta(a, ux)) = z_i$ construction of Mealy machine

$\Leftrightarrow \delta_i(\mathrm{pr}_i a, ux) \in F_i$ by Theorem 4.3 □

Definition 4.2 Two states a and b of a Mealy machine $\langle A, S, Z, \delta, \gamma, \dot{a}, \dot{z}\rangle$ are *trivially indistinguishable* if and only if for all symbols $x \in S$ we have

$\delta(a, x) = \delta(b, x)$ and $\gamma(a, x) = \gamma(b, x)$.

Lemma 4.3 Let a and b be trivially indistinguishable states of a Mealy machine $\langle A, S, Z, \delta, \gamma, \dot{a}, \dot{z}\rangle$. Then $\delta(a, u) = \delta(b, u)$ for all strings $u \in S^+$.

Proof: Let $Z = \{z_1, z_2, \ldots, z_k\}$ be the set of responses, where $\Gamma'(E_i) = z_i$ for each $1 \le i \le k$. We will prove this by induction on the length of the string u. If $|u| = 1$ then the lemma holds because the states are trivially indistinguishable. Assume the lemma holds for all strings u where $1 \le |u| \le n$ for some $n \ge 1$, and let $x \in X$ be an arbitrary symbol.

$$\begin{aligned} \delta(a, ux) &= \delta(\delta(a, u), x) && \text{by definition of } \delta \\ &= \delta(\delta(b, u), x) && \text{by inductive hypothesis} \\ &= \delta(b, ux) && \text{by definition of } \delta \end{aligned}$$ □

Note that a state is always trivially indistinguishable from itself. In fact, *trivially indistinguishable* can be shown to be an equivalence relation. We next couple this definition with the idea of a minimum Mealy machine.

Theorem 4.4 (Minimality) A Mealy machine $\langle A, S, Z, \delta, \gamma, \dot{a}, \dot{z}\rangle$ with no unequal trivially indistinguishable states constructed from the recognizer product of minimum recognizers is minimum.

Proof: Assume otherwise. Thus there are states a and b in the machine with $a \neq b$ such that $\forall u \in S^*, \forall x \in S, \gamma(\delta(a, u), x) = \gamma(\delta(b, u), x)$.

When $\gamma(\delta(a, u), x) = z_m$ then $\delta_m(\mathrm{pr}_m a, ux) \in F_m$ and $\delta_m(\mathrm{pr}_m b, ux) \in F_m$, by Lemma 4.2. Likewise, when $\gamma(\delta(a, u), x) \neq z_m$ then $\delta_m(\mathrm{pr}_m a, ux) \notin F_m$ and $\delta_m(\mathrm{pr}_m b, ux) \notin F_m$. Thus we have for all sequences $u \in S^*$ and symbols $x \in S$ that $\delta_m(\mathrm{pr}_m a, ux) \in F_m$ if and only if $\delta_m(\mathrm{pr}_m b, ux) \in F_m$, which we note holds for all $1 \leq m \leq k$.

Let $x \in S$ be a symbol and let $\delta(a, x) = e$ and $\delta(b, x) = f$. This gives $\delta_m(\mathrm{pr}_m e, u) \in F_m$ if and only if $\delta_m(\mathrm{pr}_m f, u) \in F_m$ for all $u \in S^*$. Since the component recognizers are minimum, we have $\mathrm{pr}_m e = \mathrm{pr}_m f$ for all $1 \leq m \leq k$, or simply that $e = f$. The above argument holds for any symbol x, and we have $\forall x \in X, \delta(a, x) = \delta(b, x)$. Thus the states must be trivially indistinguishable, a contradiction. □

From the above result we see that if we use minimum recognizers in the construction, we have only to remove all unequal trivially indistinguishable states

from the Mealy machine to produce the minimum Mealy machine. We next consider how to remove these trivially indistinguishable states without altering the machine function.

Theorem 4.5 Let $Me = \langle A, S, Z, \delta, \gamma, \dot{a}, \dot{z} \rangle$ be a Mealy machine with trivially indistinguishable states a and b, with $a \neq b$ and $b \neq \dot{a}$. There exists a Mealy machine Me' with state set $A' = A - \{b\}$ such that

$\forall u \in S^*, Me'(u) = Me(u)$.

Proof: Let $Me' = \langle A', S, Z, \delta', \gamma', \dot{a}, \dot{z} \rangle$, where $A' = A - \{b\}$, $\gamma' = \gamma | (A' \times S)$, and for all states $c \in A'$ and symbols $x \in S$, $\delta'(c, x) = a$ if $\delta(a, x) = b$, and $\delta'(c, x) = \delta(c, x)$ otherwise.

It remains to show that these two machines have identical machine functions. Assume they do not. That is, there is some sequence $u \in S^*$ such that $Me(u) \neq Me'(u)$. By definition, $Me(\lambda) = \dot{z} = Me'(\lambda)$, and so $u \neq \lambda$. Let $u = vx$, where $v \in S^*$ and $x \in S$. This gives $Me(u) = \gamma(\delta(\dot{a}, v), x)$ and $Me'(u) = \gamma'(\delta'(\dot{a}, v), x)$. Either $\delta(\dot{a}, u) = \delta'(\dot{a}, u)$, or $\delta(\dot{a}, u) = b$ and $\delta'(\dot{a}, u) = a$.

Assume $\delta(\dot{a}, u) = \delta'(\dot{a}, u) = c$. This gives $Me'(ux) = \gamma'(c, x)$ and $Me(ux) = \gamma(c, x)$. From the definition of γ' and since $c \neq b$, we know $\gamma'(c, x) = \gamma(c, x)$ and consequently $Me(ux) = Me'(ux)$.

Assume $\delta(\dot{a}, u) = b$ and $\delta'(\dot{a}, u) = a$. This gives $Me'(ux) = \gamma'(a, x)$ and $Me(ux) = \gamma(b, x)$. But since a and b are trivially indistinguishable, $\gamma'(a, x) = \gamma(b, x)$ and consequently $Me(ux) = Me'(ux)$. □

Definition 4.2 allows us to easily detect trivially indistinguishable states and Theorem 4.5 tells us that for every pair of trivially indistinguishable states we find, we can remove one without changing the machine function. Since the machine has a finite number of states, this process of discovery and removal of trivially indistinguishable states must terminate and, by Theorem 4.4 the resulting Mealy machine will by minimum and thus unique (up to an isomorphism).

Theorem 4.6 Let $Me = \langle A, S, Z, \delta, \gamma, \dot{a}, \dot{z} \rangle$ be a Mealy machine formed from the recognizer product of minimum recognizers as above. If two states $a, b \in A$ are indistinguishable, they are also trivially indistinguishable.

Proof: Assume otherwise. Then there are two states $a, b \in A$ with $a \neq b$ such that $\forall u \in S^*, \forall x \in S, \gamma(\delta(a, u), x) = \gamma(\delta(b, u), x)$, and $\exists y \in S$ such that

$\delta(a, y) \neq \delta(b, y)$. Let $\delta(a, y) = e$ and $\delta(b, y) = f$ with $e \neq f$. If $\gamma(a, y) = z_m$ then $\mathrm{pr}_m e \in F_m$ and $\mathrm{pr}_m f \in F_m$. Finally, $\delta_m(\mathrm{pr}_m a, ux) \in F_m$ if and only if $\delta_m(\mathrm{pr}_m b, ux) \in F_m$ which, extended by our result for e and f gives $\forall u \in S^*$, $\delta_m(\mathrm{pr}_m e, u) \in F_m$ if and only if $\delta_m(\mathrm{pr}_m f, u) \in F_m$, which contradicts minimality. □

Theorem 4.6 tells us that removing one of a pair of trivially-indistinguishable states cannot "reveal" a new pair of trivially indistinguishable states. It is therefore acceptable to first find all pairs of trivially indistinguishable states, and then remove one of each pair of states.

We can now summarize all these results in Algorithm A.2.2: Black Box Mealy Machine Minimization. The input is a Mealy machine $Me = \langle A, S, Z, \delta, \gamma, \dot{a}, \dot{z} \rangle$ formed from the recognizer product of minimum recognizers, and the output is the minimum Mealy machine. The computational complexity [2] of Algorithm A.2.2: Black Box Mealy Machine Minimization is of particular interest, and it is $O(|A|^2)$.

4.2 The State Box

Definition 4.3 A *state box* is a mathematical structure $\langle T, S, R, \Delta, \Gamma, \dot{t}, \dot{r} \rangle$, where S and R are alphabets, T is a *state set*, $\Delta: T \times S \rightarrow T$ is the *state transition function*, $\Gamma: T \times S \rightarrow R$ is the *output mapping*, $\dot{t} \in T$ is a distinguished state called the *initial state*, and $\dot{r} \in R$ is a distinguished element called the *initial response*.

From Definition 4.3 we see that every Mealy machine defines a state box, and we can transform any Mealy machine $Me = \langle A, X, Z, \delta, \gamma, \dot{a}, \dot{z} \rangle$ into a state box by choosing $T = A$, $S = X$, $R = Z$, $\Delta = \delta$, $\Gamma = \gamma$, $\dot{t} = \dot{a}$, and $\dot{r} = \dot{z}$. If we extend the definition of Δ to sequences, as we did for automata, and let $\mathrm{SB}(\lambda) = \dot{r}$ and $\mathrm{SB}(ux) = \Gamma(\Delta(\dot{t}, u), x)$ for all sequences $u \in X^* = S^*$ and symbols $x \in X = S$, then $\forall u \in S^*, Me(u) = \mathrm{SB}(u)$. We refer to SB as the *state box function*. We note that by the preceding constructions, we can produce a unique state box such that $\forall u \in S^*, \mathrm{SB}(u) = \mathrm{BB}'(u)$.

Given the above definition for a state box, we can investigate its relationship to the state box proposed by Mao [15]. Mao's functions f_R and f_Q are the functions Γ and Δ, respectively. The change in notation in this work is only to empha-

size the connection with Mealy machines. In particular, the extension of next state function Δ to sequences is identical to Mao's *state trace function*, and the function $\Delta_i: S^* \rightarrow T$, given by $\Delta_i(u) = \Delta(i, u)$ for all $u \in S^*$, is Mao's *state encapsulation function*.

4.2.1 A State Box Language

We next consider the requirements for a language in which we may represent a state box. A *state box specification language* must be defined that will allow us to alter the state space in ways which preserve the function (construct isomorphisms of the state box), discover common elements, and possibly discover state migration details. Since we are drawing closer to implementation, and the purpose of a state box is to provide a human-readable specification which can be implemented as code, we will introduce variable names along with assignment and comparison of values. This allows practitioners to choose state variable names and ranges which are meaningful to them. We will therefore need a formal syntax, which we introduce next. While a state-based language such as Z [21] could have been used, we chose instead to construct a very simple state box language tailored to our requirements.

Definition 4.4 Let S and R be the stimulus and response sets, respectively, with $\dot{r}, r_1, r_2, \ldots, r_m \in R$ for some $m \geq 1$. For some $n \geq 1$ let $(T_i | 1 \leq i \leq n)$ be a family of sets and let $T = \Pi(T_i | 1 \leq i \leq n)$ and $\dot{t} = \langle \dot{t}_1, \dot{t}_2, \ldots, \dot{t}_n \rangle$ be an element of T. Let $(p_i | 1 \leq i \leq m)$ be a family of predicates on $T \times S$ which partition $T \times S$. Let $(\phi_i | 1 \leq i \leq m)$ be a family of automorphisms on T. Let $\{\mathbf{s}, \mathbf{r}, \mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_n\}$ be a collection of variable names. Every *state box specification* in our state box specification language is of the form

begin

s in S;

r in R **initial** $\dot{r}$;

state

$\mathbf{t}_1$ **in** T_1 **initial** $\dot{t}_1$,

$\mathbf{t}_2$ **in** T_2 **initial** $\dot{t}_2$,

...

$\mathbf{t}_n$ **in** T_n **initial** $\dot{t}_n$;

transition

$p_1 \rightarrow \phi_1$ **response** r_1

| $p_2 \rightarrow \phi_2$ **response** r_2

| ...

| $p_m \rightarrow \phi_m$ **response** r_m

end.

The variables have ranges as follows. The special variable **s** has range S and is called the *current stimulus*. The special variable **r** has range R and is called the *next response*. Each $\mathbf{t}_i$ has range T_i. Let $\sigma(\mathbf{v})$ denote the value of a variable **v**, and for variables $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_l$ let $\sigma(\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_l) = \langle \sigma(\mathbf{v}_1), \sigma(\mathbf{v}_2), \ldots, \sigma(\mathbf{v}_n) \rangle$. Thus the value $\sigma(\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_n)$ is in T. The predicates p_i are evaluated at $\langle \sigma(\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_n), \sigma(\mathbf{s}) \rangle$ and may not refer to the value of **r**. Whenever a predicate $p_i(\sigma(\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_n), \sigma(\mathbf{s}))$ holds, the corresponding automorphism ϕ_i is applied to the state space to determine a new value t_j' for each variable $\mathbf{t}_j$ as $\langle t_1', t_2', \ldots, t_n' \rangle = \phi_i(\sigma(\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_n))$. Finally, the current response is assigned $\mathbf{r}' = r_i$.

Given the above definition, we can determine a state box $\langle T, S, R, \Delta, \Gamma, i, \dot{r} \rangle$, where T, S, R, i, and $\dot{r}$ are given explicitly in the state box specification, and the mappings Δ and Γ can be determined from the transition section as follows. For all $t \in T$ and $s \in S$, we have

$$[p_1(t, s) \Rightarrow (\Delta(t, s) = \phi_1(t)) \wedge (\Gamma(t, s) = r_1)] \wedge$$

$$[p_2(t, s) \Rightarrow (\Delta(t, s) = \phi_2(t)) \wedge (\Gamma(t, s) = r_2)] \wedge$$

...

$$[p_m(t, s) \Rightarrow (\Delta(t, s) = \phi_m(t)) \wedge (\Gamma(t, s) = r_m)] .$$

Since we require the predicates $p_1, p_2, \ldots, p_m$ to partition $T \times S$, Δ and Γ are well-defined. Note that if S, R, and the sets T_i for $1 \leq i \leq n$ are all finite, the state box may be transformed directly into a Mealy machine, which may then be minimized.

Informally, we can think of the state box as "starting" with $\sigma(\mathbf{r}) = \dot{r}$ and $\forall 1 \leq i \leq n, \sigma(\mathbf{t}_i) = \dot{t}_i$. The current response $\mathbf{r}$ is output. The state box then "executes" by reading a single stimulus into $\mathbf{s}$ and evaluating each of the predicates $p_i(\sigma(\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_n), \sigma(\mathbf{s}))$. When a predicate holds (exactly one will), then the new value of the state space is given by $\phi_i(\sigma(\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_n))$, and the next response is r_i. The response is then output and the state box continues with the next stimulus. This procedure is given as Algorithm A.2.3: State Box.

4.2.2 State Partitioning

We next consider how to construct a state box specification given a Mealy machine $Me = \langle A, S, R, \delta, \gamma, \dot{a}, \dot{z} \rangle$. If we have produced a Mealy machine from a

recognizer product, there is a natural way in which to do this; we regard each recognizer as a state variable on the finite range of its state set (that is, $T_i = A_i$ for each $1 \leq i \leq k$). We can then implement the predicates and mappings using simple tests of equality and assignments to constant values. We formalize this in Algorithm A.2.4: State Box Specification, whose input is a Mealy machine $Me = \langle A, S, Z, \delta, \gamma, \dot{a}, \dot{z} \rangle$, whose state set is still given as ordered tuples from the recognizer product ($\forall 1 \leq i \leq k, \forall a \in A, \mathrm{pr}_i a \in A_i$) and whose output is the state box specification components (state space, predicates, automorphisms, and responses).

Algorithm A.2.4: State Box Specification will yield a state box specification. However, the purpose of a state box specification is to provide a human-readable specification which can be transformed into code. Therefore it is important that the state variables have meaningful names and that the ranges of the state variables are also meaningful to the practitioners. We therefore consider another method for transforming our Mealy machine into a state box specification.

Consider an example in which the set of states A for a Mealy machine can be partitioned into three blocks: one in which some project file is open, one in which no file is open, and one in which this distinction does not apply (perhaps during initialization of the software). This suggests a state variable **project** with range {open, not_open, n/a}. If one carries this partitioning of the state space far enough, one would eventually reach the case in which every element of the state

space could be uniquely specified by giving a value for every state variable. At this point, one could immediately transform the Mealy machine into a state box specification with the chosen variables and state space.

This is precisely the process which should be used to transform the Mealy machine into a state box. The following definition will help clarify the goal of this process.

Definition 4.5 Let B be a set, and let $\pi_1, \pi_2, \ldots, \pi_l$ be partitions of B. Each of these partitions defines an equivalence relation $\rho_1, \rho_2, \ldots, \rho_l$ such that $\pi_i = B/\rho_i$ for all $1 \leq i \leq l$. A set $C \subseteq B$ is *unpartitioned* if and only if $|C| > 1$ and $\forall b, c \in C, \forall 1 \leq i \leq l, b \equiv c\ (\rho_i)$. If B contains no unpartitioned classes, then we say $\pi_1, \pi_2, \ldots, \pi_l$ *completely distinguish* B. Let $b, c \in B$. If there is no unpartitioned set $C \subseteq B$ such that $b, c \in C$, we say $\pi_1, \pi_2, \ldots, \pi_l$ *distinguish* b and c.

Practitioners must construct a set of partitions $\pi_1, \pi_2, \ldots, \pi_l$ which completely distinguish the state space A. The practitioners construct each partition π_i by declaring a state variable $\mathbf{t}_i$ and giving the value of the state variable $\sigma(\mathbf{t}_i)$ for every state in A. The collection of these values is the range T_i of the state variable $\mathbf{t}_i$. States which have the same value for the state variable are in the same block of the

partition. In other words, we can consider each state variable to be a mapping $\mathbf{t}_i : A \to T_i$, and the partition is the set of level sets $\pi_i = A/\ker \mathbf{t}_i$.

As each new state variable is created, A should be inspected for unpartitioned classes. If any are found, the practitioner must continue the process. Once no more unpartitioned classes remain, the state space $T = \Pi(T_i | 1 \le i \le n)$ is completely defined, and the resulting partitions completely distinguish A. We can then replace A with T and construct a state box specification in the state box specification language.

Before presenting an algorithm for this process, we will need the following result.

Theorem 4.7 Let $(\rho_i | 1 \le i \le n)$ be a family of equivalence relations on A, and choose $\pi_i = A/\rho_i$ for each $1 \le i \le n$. Then $\left|\bigcap (\rho_i | 1 \le i \le n)\right| \ge |A|$ with equality if and only if $(\pi_i | 1 \le i \le n)$ completely distinguish A.

Proof: Since each ρ_i is an equivalence relation on A, $\{\langle a, a\rangle | a \in A\} = \iota_A \subseteq \rho_i$ for every $1 \le i \le n$. Thus $\iota_A \subseteq \bigcap (\rho_i | 1 \le i \le n)$, and we have

$$\left|\bigcap (\rho_i | 1 \le i \le n)\right| \ge |\iota_A| = |A|$$

as required. Equality implies $\bigcup (\rho_i | 1 \le i \le n) = \iota_A$, and the partitions completely distinguish A. Assume the partitions completely distinguish A. A consequence of this is $\forall a, b \in A, (\forall 1 \le i \le n, a \equiv b (\rho_i)) \Rightarrow (a = b)$, and again we have $\bigcup (\rho_i | 1 \le i \le n) = \iota_A$. □

The above result leads to the construction of an algorithm for state space partitioning. In the Algorithm A.2.5: State Space Partitioning, we use ρ to model the intersection of the equivalence relations induced by the state variables.

4.2.3 State Box Generation

Let $Me = \langle A, S, Z, \delta, \gamma, \dot{a}, \dot{z} \rangle$ be the Mealy machine constructed from the black box specification. Let $\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_n$ be the chosen state variables with corresponding ranges $T_1, T_2, \ldots, T_n$, and let $\mathbf{t}$ denote $\langle \mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_n \rangle$. Let the value $\langle t_1, t_2, \ldots, t_n \rangle$ which corresponds to state $a \in A$ be denoted t_a. The remaining components of the state box specification are determined as follows.

$$\forall a \in A, \forall x \in S, \qquad \begin{aligned} p_{a,x} &\text{ is } (\mathbf{t} = t_a \wedge \mathbf{s} = x) \\ \phi_{a,x} &\text{ is } [\mathbf{t} := t_{\delta(a,x)}] \\ r_{a,x} &\text{ is } \gamma(a, x) \end{aligned}$$

Note that the predicates above do not necessarily partition $T \times S$. They are, however, clearly disjoint on $T \times S$.

Definition 4.6 An *invalid state* is any value $\sigma(\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_n)$ which does not correspond to some $a \in A$.

These values are all in T, but since they do not correspond to a state in A, our predicates do not address them. These states should never be reached during the execution of the state machine. They correspond to states which cannot be generated from $\dot{a}$, but which are nonetheless in the product T. We deal with these by adding an additional predicate p_{error} which is true exactly when all the other predicates are false. For simplicity we can list this predicate last, and represent it with the keyword **else** following standard practice. When this predicate is added to the family $(p_a | a \in A)$, the resulting collection of predicates does partition $T \times S$ as required. We let ϕ_{error} be the identity mapping and let $r_{\text{error}} = \omega$.

4.2.4 Re-Introduction of Predicate Refinements

We explicitly removed non-regular refinements in §4.1.1. We now re-introduce the predicate refinements into the state box specification. We do this by splitting tran-

sitions in the state box. One can also view this as splitting the input alphabet, or refining the output mapping.

The state box may be evaluated on stimulus histories by applying the sequence of stimuli in the history. In the following discussion, let u denote the stimulus history excluding the last ("most recent") event.

For every block E_i which was partitioned with predicates previously, we must split any transition whose response is z_i. For instance, if E_3 was partitioned into $E_3[p]$ and $E_3[\neg p]$, and we have $\mathbf{t} = t_a \wedge \mathbf{s} = x \rightarrow \phi$ **response** z_3, we split the transition into two transitions, one for the stimulus x when $p(ux)$ holds, denoted $x[p]$, and one for the stimulus x when $p(ux)$ fails, denoted $x[\neg p]$. The new transitions still go to the same next state. That is ϕ remains unchanged. However we can now specify different responses for the two transitions. Whenever the machine must decide on these two transitions, the predicate p is evaluated on ux. If true, the response for $x[p]$ is generated, and if false, the response for $x[\neg p]$ is generated.

Once we have explicitly encoded the predicate refinements into the state box, we next must resolve these refinements through the introduction of special state variables on possibly-infinite domains (note that a predicate which can be resolved using only state variables on finite domains is a regular predicate, so this is how we introduce non-regular refinements). The introduction of infinite state data increases the modelling power of the state box since we now have an

unbounded state space and, in particular, we can no longer directly transform our state box into a Mealy machine.

Choose a predicate p which was introduced above, and let the new state variables be $u_1, u_2, \ldots, u_n$. These new state variables must be used to encode the relevant parts of u required to decide $p(ux)$. Each time a stimulus is read, $u_1, u_2, \ldots, u_n$ must be updated, and thus a transformation on $u_1, u_2, \ldots, u_n$ will be part of every automorphism in the transition section. Finally, predicates of the form $\mathrm{s} = x\,[p]$ must be replaced with the identical predicates $p'(u_1, u_2, \ldots, u_n) \wedge \mathrm{s} = x$, where the p' is chosen appropriately based on the choice of $u_1, u_2, \ldots, u_n$.

Choosing the state variables, the state transformation, and the replacement predicate for the predicate refinement must be done carefully, and is certainly a potential source of errors in the final state box. For this reason, a suite of commonly-used predicate refinements should be developed, along with proofs of their computability on the set of all stimulus histories, the required resolutions for the state box (new state variables and replacement predicate), and proofs that the resolutions are correct.

4.3 State Box Composition

Reuse of software components requires that we are able to describe the system formed by the composition of the components. Since the black box denotes a function, normal function composition might seem possible, but consider connecting two black boxes in series where responses from the first will be stimuli to the second. Unfortunately, this results in a domain mismatch. The second black box operates on sequences of stimuli, S^*, but the first black box produces only single events, R. A better model for composition is one that does not require sequences of events in the domain. The state box is such a model. Therefore, reuse composition should be modelled using state-based specifications rather than sequence-based specifications.

We will discuss two ways of composing state boxes: composition in series, in which the responses from one state box are stimuli to a second state box; and composition in parallel, in which stimuli are passed concurrently to two state boxes, both of which generate responses. For the following sections, we fix two state boxes

$$\mathrm{SB}_1 = \langle T_1, S_1, R_1, \Delta_1, \Gamma_1, \dot{t}_1, \dot{r}_1 \rangle$$
$$\mathrm{SB}_2 = \langle T_2, S_2, R_2, \Delta_2, \Gamma_2, \dot{t}_2, \dot{r}_2 \rangle$$

and will describe their composition by defining a new state box

$$\mathrm{SB} = \langle T, S, R, \Delta, \Gamma, \dot{t}, \dot{r} \rangle .$$

Finally, we note that a black box may be derived from the resulting compound state box SB. A new state box can then be derived from this black box and minimized, which may reveal additional information about the system.

4.3.1 Series Composition

We wish to connect the two state boxes in series (Fig. 4.1), so that the responses from one will be stimuli to the second. Because of this, we will require $R_1 \subseteq S_2$. Additionally, it will be helpful to require $\dot{r}_1 = 0$. This will prevent any problems with the initial response of SB_1.

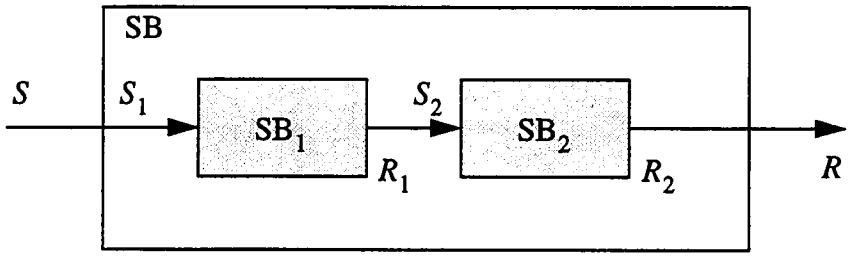


Figure 4.1: Series Composition

We construct the new state box SB as follows. Let $T = T_1 \times T_2$, $S = S_1$, $R = R_2$, $\dot{t} = \langle \dot{t}_1, \dot{t}_2 \rangle$, and $\dot{r} = \dot{r}_2$. We then define Δ and Γ as follows for all $t_1 \in T_1$, $t_2 \in T_2$, and $s \in S$.

$$\Delta(\langle t_1, t_2 \rangle, s) = \langle \Delta_1(t_1, s), \Delta_2(t_2, \Gamma_1(t_1, s)) \rangle$$
$$\Gamma(\langle t_1, t_2 \rangle, s) = \Gamma_2(t_2, \Gamma_1(t_1, s))$$

Thus the new state box keeps track of the states of the component state boxes, and issues the responses generated by SB_2.

4.3.2 Parallel Composition

We wish to connect the two state boxes in parallel (Fig. 4.2). Stimuli are presented to both state boxes simultaneously. Any stimuli which are outside the domain of the state box are ignored. Note that we make no restrictions on S_1 and S_2. In particular, $S_1 \cap S_2$ may not be empty. The output will be an ordered pair in which the first element is the output from SB_1 and the second is the output from SB_2. If either state box gives no output, we will simply treat it as the null output 0.

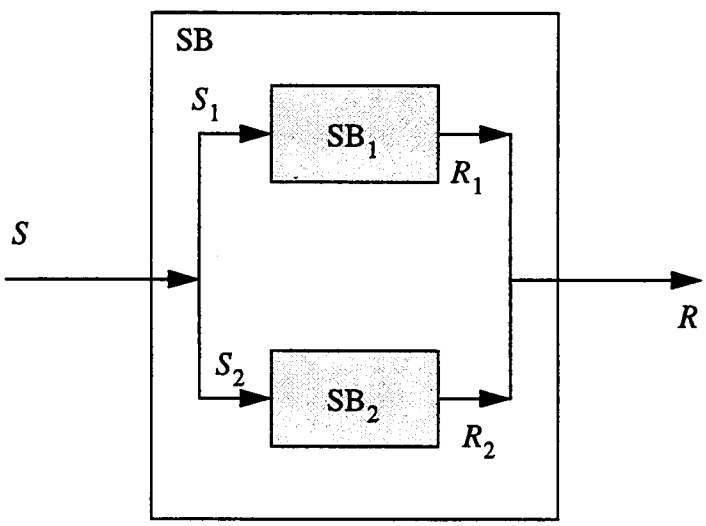


Figure 4.2: Parallel Composition

We construct the new state box SB as follows. Let $T = T_1 \times T_2$, $S = S_1 \cup S_2$, $R = R_1 \times R_2$, $i = \langle i_1, i_2 \rangle$, and $\dot{r} = \langle \dot{r}_1, \dot{r}_2 \rangle$. We then define Δ and Γ as follows for all $t_1 \in T_1$, $t_2 \in T_2$, and $s \in S$.

$$s \in S_1 \wedge s \notin S_2 \Rightarrow (\Delta(\langle t_1, t_2 \rangle, s) = \langle \Delta_1(t_1, s), t_2 \rangle) \wedge (\Gamma(\langle t_1, t_2 \rangle, s) = \langle \Gamma_1(t_1, s), 0 \rangle)$$
$$s \notin S_1 \wedge s \in S_2 \Rightarrow (\Delta(\langle t_1, t_2 \rangle, s) = \langle t_1, \Delta_2(t_2, s) \rangle) \wedge (\Gamma(\langle t_1, t_2 \rangle, s) = \langle 0, \Gamma_2(t_2, s) \rangle)$$

$$s \in S_1 \wedge s \in S_2 \Rightarrow$$
$$((\Delta(\langle t_1, t_2 \rangle, s) = \langle \Delta_1(t_1, s), \Delta_2(t_2, s) \rangle) \wedge (\Gamma(\langle t_1, t_2 \rangle, s) = \langle \Gamma_1(t_1, s), \Gamma_2(t_2, s) \rangle))$$

Thus the new state box keeps track of the states of the component state boxes, and issues the responses for each. The above definition is similar to the definition of synchronous composition (§5.1.3).

4.4 Other State Box Issues

4.4.1 Data State and Control State

Data and *control state* arise as a consequence of initially partitioning the stimulus and response sets as mentioned in §3.1. Let S_A and R_A denote the true, "atomic" stimulus and response sets, and let S and R be the partitions used during enu-

meration and construction of the state box. Let $S_A/\rho_A = S$ and $R_A/\rho_A = R$. Let $BB_A: S_A^* \to R_A$ and let $BB: S^* \to R$ be the corresponding black box functions.

We define a homomorphism $\phi: S_A^* \to S^*$ as follows.

$$\phi(\lambda) = \lambda$$
$$\forall u \in S_A^*, \forall x \in S_A, \phi(ux) = \phi(u) \cdot [x]\rho_S$$

This gives us the relationship shown in the commutative diagram (Fig. 4.3). Thus we have $BB = \phi^{-1} \cdot BB_A \cdot \text{nat}\ \rho_R$.

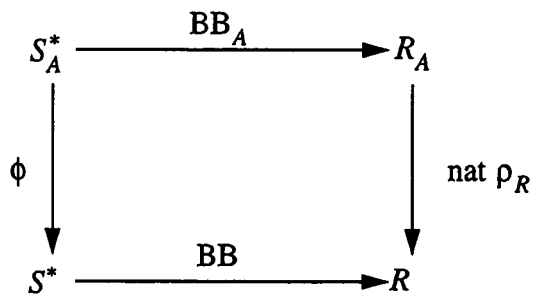


Figure 4.3: Relationship Between Black Box Functions

Definition 4.7 State data discovered to compute BB is *control state.*

Let $u \in S_A^*$. $BB(\phi(u))$ is in R. Given this response, we must now decide on a response in R_A. We need a function which combines $BB(\phi(u))$ with some additional information to yield a response in R_A. That is, we need a function such that

$$f(BB(\phi(u)), \mathcal{D}) = BB_A(u)$$

where $\mathcal{D}$ is the "additional information" mentioned earlier. Since the right hand side of this equation depends only on u, we conclude that the left hand side must as well, and thus $\mathcal{D}$ must depend only on u. We re-write $\mathcal{D}$ as $d(u)$.

Definition 4.8 Additional state information required to compute BB_A from BB for a sequence is *data state.*

We must decide on the essential parts—the extra data state $d(u)$—required to get from R to R_A. This new data state information must then be added to the state box.

Since we only have data state when we partition the response set, we can detect where this additional state is needed by examining the responses in our state box. Wherever a non-singleton response block from R is mentioned, we must introduce additional information to distinguish between the responses in the response block. Depending on the problem at hand, this may be simple or complex. As with the resolution of the predicate refinements in the last section, this must be conducted by adding additional state variables and propagating them throughout the predicates and automorphisms.

4.4.2 State Migration

Many of the assertions in the transition section of the state box specification may have similarities in the predicates and automorphisms. Consider the following example.

$$(\mathbf{t}_1 = \text{true}) \wedge (\mathbf{t}_2 = 7) \wedge (\mathbf{s} = x_1) \rightarrow (\mathbf{t}_1 := \text{true}) \wedge (\mathbf{t}_2 := 8) \ \textbf{response}\ r_1$$

$$\mid \ (\mathbf{t}_1 = \text{true}) \wedge (\mathbf{t}_2 = 8) \wedge (\mathbf{s} = x_1) \rightarrow (\mathbf{t}_1 := \text{true}) \wedge (\mathbf{t}_2 := 0) \ \textbf{response}\ r_1$$

The above two assertions contain several identical parts. We can factor the predicate as follows.

$$(\mathbf{t}_1 = \text{true}) \wedge (\mathbf{s} = x_1) \wedge$$

$$[$$

$$(\mathbf{t}_2 = 7) \rightarrow (\mathbf{t}_1 := \text{true}) \wedge (\mathbf{t}_2 := 8) \ \textbf{response}\ r_1$$

$$\mid \ (\mathbf{t}_2 = 8) \rightarrow (\mathbf{t}_1 := \text{true}) \wedge (\mathbf{t}_2 := 0) \ \textbf{response}\ r_1$$

$$]$$

We can then factor the automorphisms as follows.

$$(\mathbf{t}_1 = \text{true}) \wedge (\mathbf{s} = x_1) \wedge$$

$$[$$

$$(\mathbf{t}_2 = 7) \rightarrow (\mathbf{t}_2 := 8) \ \textbf{response}\ r_1$$

$$\mid \ (\mathbf{t}_2 = 8) \rightarrow (\mathbf{t}_2 := 0) \ \textbf{response}\ r_1$$

$$] \quad \wedge (\mathbf{t}_1 := \text{true})$$

We could also factor out the responses, which are the same for everything within the brackets. Finally, we may choose to discard the $\mathbf{t}_1$:= true altogether, since it does not change the state space in this case.

If all access to a particular state variable can be logically grouped by factoring, that state variable is an excellent candidate for deferment of the state variable's implementation and maintenance to a lower level of design. Likewise, we may also discover that responses, certain predicates, and certain transformations can also be logically grouped. If so, these need not be global, either.

Definition 4.9 The design decision that state box components (data, tests, transformations, and responses) "will be aggregated in various ways and each aggregation [...] hidden from and unavailable to specifications regarding all other aggregations" is *state migration* [15, p. 87].

Since we may not know which parts of the state box will require lower-level design phases until we begin construction of a clear box and confront some of the implementation issues, the determination of which state box components to migrate cannot be finalized until the clear box construction phase. The aggregation of state box components in the state box specification (through factoring) may expose migration details which would otherwise be overlooked. Designers should arrange the state space in as logical and compact a fashion as possible, and inspect the state box thus produced. During this inspection, one may gain insight

into ways to redistribute the state space. For instance, one may discover that the state box can be factored in two ways, each of which precludes the other.

Certain refinement techniques such as those discussed in §5.2 (stimulus deletion, indexed stimulus deletion, and abstraction) lead to families of Mealy machines. Each machine in the family can be regarded as a state box, and a separate state box specification produced. The resulting specifications represent a logical aggregation of state by context.

Chapter 5

Advanced Specification Techniques

In §3.3.1, stepwise-refinement of black boxes through refinement of the black box partition was introduced. One refinement technique, the use of predicates, was discussed (§3.3.2). This technique left practitioners with some hard problems when constructing the state box (§4.2.4). In this chapter we will present alternate black box specification techniques which make it easier to produce a state box.

In the following, a specification means a black box specification derived through stimulus enumeration. We will discuss two classes of specification techniques: the use of special operators and the use of abstraction techniques.

5.1 Special Operators

One alternative to refining a partition, is to expand the language of regular expressions by adding operators. These regular operators would allow practitio-

ners to work with some reduced set of histories and then apply companion operators to recover the full set of histories. Since a regular operator is closed on the set of regular expressions, it adds no power to the specification language and is simply a notational convenience. The proof of regularity for such an operator should be done constructively, to facilitate the removal of the operator prior to constructing a state box.

5.1.1 Special Sequence Operator

Consider the following case. An item of software is to be specified which can, at any time, receive a stimulus sequence from the set denoted by the regular expression R. All such sequences have no bearing on future responses. For instance, if the software is running on a UNIX system, it may receive a signal to terminate. Immediately the software pops up a dialog box for the user to select "terminate" or "cancel." If the use selects "cancel," the software continues execution as though the signal were never received, giving the sequence $R = \text{signal} \cdot \text{cancel}$. We wish to ignore R as we construct our initial partitioning using regular expressions, and then apply an operator to account for R.

Let θ_R be defined for any sequence in S^* as follows.

$$\theta_R(\lambda) = \lambda$$
$$\forall u \in S^*, \forall x \in S, \theta_R(ux) = \theta_R(u) \cdot (R + \lambda) \cdot x$$

We next extend the definition to sets of sequences as follows.

$$\theta_R(E) = \{u \in \theta_R(v) \mid v \in E\}$$

Having defined the operator, we must now prove that it is, indeed, a regular operator. We proceed as follows.

Lemma 5.1 $\forall u \in S^*, \forall v \in S^*, \theta_R(uv) = \theta_R(u) \cdot \theta_R(v)$.

Proof: Follows by induction on the length of v. □

Theorem 5.1 Let E and F be regular expressions. Then the following hold.

i. $\theta_R(E \cdot F) = \theta_R(E) \cdot \theta_R(F)$,

ii. $\theta_R(E \cup F) = \theta_R(E) \cup \theta_R(F)$,

iii. $\theta_R(E \cap F) = \theta_R(E) \cap \theta_R(F)$,

iv. $\theta_R(E - F) = \theta_R(E) - \theta_R(F)$,

v. $\theta_R(E^*) = (\theta_R(E))^*$, and

vi. $\theta_R(\bar{E}) = (\theta_R(S))^* - \theta_R(E)$.

Proof: Each part of the theorem follows directly from the application of definitions and Lemma 5.1. □

Theorem 5.2 θ_R is a regular operator.

Proof: By definition, if θ_R is applied to any sequence, the result is a regular expression. This serves as the basis case of a proof by structural induction that uses Theorem 5.1. □

5.1.2 Reset Operator

Another important operator is the unconditional reset. Let $x \in S$ be a stimulus which initializes the system; thus any stimulus history up to and including the most recent x is irrelevant to future responses. Let $u \in S^*$ be a sequence and let E be a set of sequences.

$$\theta_x(u) = S^* \cdot x \cdot u \qquad \text{definition of } \theta_x$$

$$\theta_x(E) = \{u \in \theta_x(v) \mid v \in E\} \qquad \text{extension to sets of sequences}$$

$$= \{u \in S^* \cdot x \cdot v \mid v \in E\} \qquad \text{definition of } \theta_R$$

$$= S^* \cdot x \cdot E$$

Thus, if E is a regular expression, so is $\theta_x(E)$.

5.1.3 Concurrency Operators

Two special operators are used to model concurrency: the interleave and synchronous composition. These two operators (along with the closure of the interleave) are used in [8] to provide an algebraic representation of Petri net languages (see also [20,22]). We will show that if one does not allow closure of the interleave operator (the alpha-closure), then the resulting system is regular.

The interleave is used to model a system formed with several independent actors. The synchronous composition is used to model a system formed with actors which must share some actions. Garg and Ragunath discuss the application of these operators in detail in [8], and also introduce the closure of the interleave, called the *alpha-closure*, which they prove is not a regular operator. The alpha-closure is only required when one has a possibly-unbounded number of independent actors. If one specifies a finite number of independent actors using the nth interleave, one can obtain an operator which is regular.

Definition 5.1 Let X and Y be alphabets, let $u \in X^*$ and $v \in Y^*$ be sequences, and let $x \in X$ and $y \in Y$ be symbols. The *interleave*, denoted $\|$, is given by

$$\lambda \| u = u \| \lambda = u$$
$$ux \| vy = (ux \| v) \cdot y \cup (u \| vy) \cdot x$$

for sequences, and is extended to sets of sequences $E \subseteq X^*$ and $F \subseteq Y^*$ by

$$E \| F = \{u \in v \| w | v \in E \wedge w \in F\} .$$

Definition 5.2 The n*th interleave* (for positive n) of a set of sequences E, denoted $E^{[n]}$, is given by $E^{[1]} = E$ and $E^{[n+1]} = E^{[n]} \| E$ for any $n \geq 1$.

Theorem 5.3 The interleave of regular sets is regular.

Proof: Let E_1 and E_2 be regular sets, and let $\mathcal{R}_1 = \langle A_1, X_1, \delta_1, \dot{a}_1, F_1 \rangle$ and $\mathcal{R}_2 = \langle A_2, X_2, \delta_2, \dot{a}_2, F_2 \rangle$ be finite recognizers with $L(\mathcal{R}_1) = E_1$ and $L(\mathcal{R}_2) = E_2$. Construct a non-deterministic finite recognizer $\mathcal{R} = \langle A, X, \delta, \dot{a}, F \rangle$ as follows. Let $A = A_1 \times A_2$, $\dot{a} = \langle \dot{a}_1, \dot{a}_2 \rangle$, $X = X_1 \cup X_2$, and $F = \{\langle a_1, a_2 \rangle | a_1 \in F_1 \wedge a_2 \in F_2\}$. Then

$$\forall x \in X_1, \delta(\langle a_1, a_2 \rangle, x) = \langle \delta_1(a_1, x), a_2 \rangle$$
$$\forall x \in X_2, \delta(\langle a_1, a_2 \rangle, x) = \langle a_1, \delta_2(a_2, x) \rangle$$

Let w be an element of $E_1 \| E_2$. We can find sequences $u \in E_1$ and $v \in E_2$ such that $w \in u \| v$. Since $\delta_1(\dot{a}_1, u) \in F_1$ and $\delta_2(\dot{a}_2, v) \in F_2$, $\mathcal{R}$ can accept w. Let $w \notin E_1 \| E_2$. Since we cannot break w up as above, $\mathcal{R}$ cannot accept it.

Thus $L(\mathcal{R}) = E_1 \parallel E_2$, and since the language is accepted by a recognizer, it is regular. □

Theorem 5.4 The nth interleaving of any regular set is regular.

Proof: Let E be a regular set. Then $E^{[1]}$ is regular, by definition. Assume $E^{[i]}$ is regular, for all $i \leq n$ for some constant $n \geq 1$. It remains to show that $E^{[n+1]}$ is regular. $E^{[n+1]} = E^{[n]} \parallel E$, and since we know $E^{[n]}$ and E are regular, by Theorem 5.3 $E^{[n+1]}$ must also be regular. □

5.1.4 Synchronous Composition Operator

We next consider the synchronous composition operator, which is used to model actors which must synchronize on some shared events.

Definition 5.3 For any alphabet X, we define the *stimulus deletion homomorphism* $\phi_X: (Y^* \rightarrow X^*)$ for arbitrary alphabet Y as follows.

$$\begin{aligned} \phi_X(\lambda) &= \lambda \\ \forall u \in Y^*, \forall y \in Y, \phi_X(uy) &= \phi_X(u) \cdot y \text{ if } y \in X \\ &= \phi_X(u) \text{ otherwise} \end{aligned}$$

Definition 5.4 Let X and Y be alphabets, and let $E \subseteq X^*$ and $F \subseteq Y^*$ be sets of sequences. The *synchronous composition* of E and F, denoted $E[]F$, is the set $\{u \in (X \cup Y)^* \mid \phi_X(u) \in E \wedge \phi_Y(u) \in F\}$.

Note that if the alphabets from which the sets of sequences are drawn are disjoint, the synchronous composition reduces to the interleave. Further, if the sequences are drawn from the same alphabets, the synchronous composition reduces to the intersection. However, when the alphabets overlap but are not identical, we discover that the resulting sequences must agree, or *"synchronize,"* at the shared symbols.

Theorem 5.5 The synchronous composition of regular sets is regular.

Proof: Let E_1 and E_2 be regular sets, and let $\mathcal{R}_1 = \langle A_1, X_1, \delta_1, \dot{a}_1, F_1 \rangle$ and $\mathcal{R}_2 = \langle A_2, X_2, \delta_2, \dot{a}_2, F_2 \rangle$ be finite recognizers with $L(\mathcal{R}_1) = E_1$ and $L(\mathcal{R}_2) = E_2$. Construct a finite recognizer $\mathcal{R} = \langle A, X, \delta, \dot{a}, F \rangle$ as follows.

Let $A = A_1 \times A_2$, $\dot{a} = \langle \dot{a}_1, \dot{a}_2 \rangle$, $X = X_1 \cup X_2$, and

$F = \{\langle a_1, a_2 \rangle \mid a_1 \in F_1 \wedge a_2 \in F_2\}$. Then $\forall x \in X$

$$x \in X_1 \wedge x \notin X_2 \Rightarrow \delta(\langle a_1, a_2 \rangle, x) = \langle \delta_1(a_1, x), a_2 \rangle$$
$$x \in X_1 \wedge x \in X_2 \Rightarrow \delta(\langle a_1, a_2 \rangle, x) = \langle \delta_1(a_1, x), \delta_2(a_2, x) \rangle$$
$$x \notin X_1 \wedge x \in X_2 \Rightarrow \delta(\langle a_1, a_2 \rangle, x) = \langle a_1, \delta_2(a_2, x) \rangle$$

Elements of $X_1 - X_2$ change only the state associated with $\mathcal{R}_1$, elements of $X_2 - X_1$ change only the state associated with $\mathcal{R}_2$, and elements of $X_1 \cap X_2$ change both states simultaneously. Thus $L(\mathcal{R}) = E_1[]E_2$. □

5.2 Abstraction Techniques

Systematic enumeration of all sequences to produce an arguably complete, consistent, and correct specification is made practical by a collection of techniques based on controlling the growth of an inherently combinatorial process. Abstraction techniques are very powerful for controlling growth.

5.2.1 Abstraction

During enumeration, one may see that no more details are going to be discovered for some time. For instance, the next detail may be associated with a clock pulse happening 1,000 times, and it is impractical to keep on enumerating to sequences of length 1,000 just to discover this detail. In another case, one may discover that certain details may continue to repeat in the history, resulting in wasted time deal-

ing with these sequences. In these cases it is important to get away from these details and continue enumerating at a higher level of abstraction. For instance, a single stimulus could be introduced which carries the information that 1,000 clock pulses have occurred, or that some well-understood sequence of events has happened. Such stimuli are called *abstract stimuli*, since they are really a many-to-one mapping. The principle idea is to omit the details of what particular sequence of stimuli compose a transaction, and deal with the transaction as a unit.

Definition 5.5 Let S and Y be an alphabets. An *abstraction* is a monoid homomorphism $\phi: S^* \rightarrow Y^*$.

An abstraction ϕ_Y may be specified by a set of disjoint predicates P defined on S^*. For each predicate $p_i \in P$, let y_i denote the predicate name, with Y being the complete set of predicate names. The resulting abstraction $\phi_Y: S^* \rightarrow Y^*$ is defined as follows.

$$\phi_Y(\lambda) = \lambda$$

$$\begin{aligned} \forall u \in S^*, \forall x \in S, \phi_Y(ux) &= \phi_Y(u) \cdot y_1 \text{ if } p_1(ux) \\ &= \phi_Y(u) \cdot y_2 \text{ if } p_2(ux) \\ &\cdots \\ &= \phi_Y(u) \cdot y_n \text{ if } p_n(ux) \\ &= \phi_Y(u) \text{ if } \neg p_1(ux) \wedge \neg p_2(ux) \wedge \cdots \wedge \neg p_n(ux) \end{aligned}$$

Note that we compute the predicates on the entire history up to, and including, the stimulus of interest. If one of the characteristic predicates p_i holds, we replace the stimulus of interest with the corresponding predicate name. If none of the characteristic predicates holds, we discard the stimulus of interest and continue back through the history until we reach the basis case (λ).

As an example, consider characteristic predicate τ (or "time") defined as "there have been 1,000 clock pulses in the history." One would now enumerate with the new stimulus set (which would possibly include some of the original stimuli from S with a simple characteristic predicate), and would not have to examine sequences of length 1,000 before dealing with the 1,000 clock pulses. This would produce a specification at the new level of abstraction.

A specification produced under a new level of abstraction can be used to refine the system specification.

Theorem 5.6 Let $\phi_Y: S^* \rightarrow Y^*$ be an abstraction, let $(F_i \mid 1 \leq i \leq m)$ be a partition of Y^*, and let $E \subseteq S^*$ be a set of sequences. Then, using $E[\phi_Y, F]$ to denote the set $\{u \in E \mid \phi_Y(u) \in F\}$, $(E[\phi_Y: F_i] \mid 1 \leq i \leq m)$ is a partition of E.

Proof: We know from the definition that every $E[\phi_Y: F_i]$ is a subset of E. It remains to show that every element $u \in E$ is in one and only one set

$E[\phi_Y{:}F_i]$. Let $u \in E$ be a sequence. $\phi_Y(u)$ is a unique element of Y^* and is thus in one and only one block F_i of the partition. Thus u must be in $E[\phi_Y{:}F_i]$ and can be in no other block. Thus $(E[\phi_Y{:}F_i] \mid 1 \le i \le m)$ is a partition of E. □

The above result permits use of the specification derived for the abstract stimulus set to refine the system specification. The consequences of using such a refinement are as follows.

Theorem 5.7 Let ϕ_Y be an abstraction defined by predicates P, let $E \subseteq S^*$ be a set of sequences, and let $F \subseteq Y^*$ be a set of sequences from the abstract alphabet. If P consists of only regular predicates, and both E and F are regular sets, then $E[\phi_Y{:}F]$ is also a regular set.

Proof: Let p_0 be the predicate $\neg p_1 \wedge \neg p_2 \wedge \cdots \wedge \neg p_n$, and let $P' = P \cup \{p_0\}$. Let y_0 be the name for predicate p_0, and define $Y' = Y \cup \{y_0\}$. Since all the predicates in P are regular, p_0 must also be regular, and $\phi_{Y'}$ is an abstraction in which the predicates in P' partition S^*. For $0 \le i \le n$, let $E_i = \{u \in S^* \mid p_i(u)\}$. The resulting family of sets $(E_i \mid 1 \le i \le n)$ form a par-

tition of S^*. Let $\Gamma(E_i) = y_i$ for all $0 \le i \le n$. Then $\langle S, Y', (E_i | 1 \le i \le n), \Gamma \rangle$ is a regular black box. Following Chapter 4, construct a Mealy machine $M = \langle A, S, Y', \delta, \gamma, \dot{a}, \dot{z} \rangle$ such that $\forall u \in S^*, M(u) = y_i$ if and only if $p_i(u)$ is true.

Let $\mathcal{R}_F = \langle A_F, Y, \delta_F, \dot{a}_F, F_F \rangle$ be a recognizer such that $L(\mathcal{R}_F) = F$. Construct a recognizer $\mathcal{R}' = \langle A', S, \delta', \dot{a}', F' \rangle$ such that $L(\mathcal{R}') = \{u \in S^* | \phi_Y(u) \in F\}$ as follows. Let $A' = A \times A_F$, $\dot{a}' = \langle \dot{a}, \dot{a}_F \rangle$, $F' = \{\langle a, b \rangle \in A | b \in F_F\}$, and finally let δ' be given by

$$\delta'(\langle a, b \rangle, x) = \langle \delta(a, x), \delta_F(b, \gamma(a, x)) \rangle \quad \text{if } \gamma(a, x) \neq y_0, \text{ and}$$

$$= \langle \delta(a, x), b \rangle \quad \text{otherwise.}$$

It is clear from the above that $\mathrm{pr}_1 \delta'(\dot{a}', u) = \delta(\dot{a}, u)$ for all $u \in S^*$. We now claim that $\mathrm{pr}_2 \delta'(\dot{a}', u) = \delta_F(\dot{a}_F, \phi_Y(u))$ for all $u \in S^*$. If this is true, then $L(\mathcal{R}') = \{u \in S^* | \phi_Y(u) \in F\}$.

We proceed by induction on the length of a string u. If $u = \lambda$ we have $\mathrm{pr}_2 \delta'(\dot{a}', \lambda) = \mathrm{pr}_2 \dot{a}' = \dot{a}_F = \delta_F(\dot{a}_F, \lambda)$. Assume the claim holds for all $u \in S^*$ with $0 \le |u| \le m$ for some constant $m \ge 0$. Let $u \in S^*$ with $|u| = m$ and let $x \in S$ be a symbol. Assume $p_i(ux)$ is true (there must be one such predicate which holds), and that $p_i \neq p_0$.

$$\begin{aligned}
\mathrm{pr}_2\delta'(\dot{a}', ux) &= \mathrm{pr}_2\delta'(\delta'(\dot{a}', u), x) && \text{from the definition of } \delta' \\
&= \mathrm{pr}_2\delta'(\langle \delta(\dot{a}, u), \delta_F(\dot{a}_F, \phi_Y(u))\rangle, x) && \text{by inductive hypothesis} \\
&= \delta_F(\delta_F(\dot{a}_F, \phi_Y(u)), y_i) && \text{from construction} \\
&= \delta_F(\dot{a}_F, \phi_Y(ux)) && \text{by definition of } \phi_Y
\end{aligned}$$

If we assume that $p_i = p_0$, then

$$\begin{aligned}
\mathrm{pr}_2\delta'(\dot{a}', ux) &= \mathrm{pr}_2\delta'(\delta'(\dot{a}', u), x) && \text{from the definition of } \delta' \\
&= \mathrm{pr}_2\delta'(\langle \delta(\dot{a}, u), \delta_F(\dot{a}_F, \phi_Y(u))\rangle, x) && \text{by inductive hypothesis} \\
&= \delta_F(\dot{a}_F, \phi_Y(u)) && \text{from construction} \\
&= \delta_F(\dot{a}_F, \phi_Y(ux)) && \text{by definition of } \phi_Y
\end{aligned}$$

Since we now have a recognizer which accepts $\{u \in S^* | \phi_Y(u) \in F\}$, it follows that this is a regular set. Let this set be denoted Φ. Since we already know E is a regular set, we can conclude that $E \cap \Phi$ is regular, and thus that $E[\phi_Y{:}F] = E \cap \Phi$ is also regular. □

The above result tells us that if we use only regular characteristic predicates for our abstraction, we can discover the state space required to represent the abstraction.

The remaining techniques are specific cases of abstraction which are useful for certain specification problems.

5.2.2 Stimulus Deletion Abstraction

It may be natural to specify the behavior of a part of a system apart from the behavior of the rest of the system. We call such a part a *context*. For example, a software system may process two data streams which are largely independent. To obtain the response for any stimulus, we need to consider the history of the data stream from which the current stimulus originates. In this case, we would like to do the following.

1. Produce a specification for each of the data streams (contexts), indicating the places at which we cannot decide on the response without consulting the history of the other stream (the points at which context independence is violated).
2. Construct a block in the system specification which includes sequences (all or possibly only some) which depend only on the history of one data stream (context).
3. Use the specification produced in 1 to refine the block constructed in 2.

Let $Y \subseteq S$ be the set of stimuli for a context. Following Definition 5.3, for $E \subseteq S^*$ and $F \subseteq Y^*$, we let $E[Y{:}F] = \{u \in E | \phi_Y(u) \in F\}$.

Theorem 5.8 Let $E \subseteq S^*$ be a set of sequences, $Y \subseteq S$ the context stimuli, and $(F_i \mid 1 \le i \le n)$ a partition of Y^*. Then $(E[Y{:}F_i] \mid 1 \le i \le n)$ is a partition of E.

Proof: By definition, every $E[Y{:}F_i]$ is a subset of E. The image $\phi_Y(u)$ is a unique element of Y^* and is thus in one and only one of the sets F_i. Therefore, u can be in one and only one of the sets $E[Y{:}F_i]$. □

Theorem 5.9 Let $Y \subseteq S$ be the context stimuli, $E \subseteq S^*$ a set of sequences, and $F \subseteq Y^*$ a set of sequences from the context stimuli. If E and F are regular sets, so is $E[Y{:}F]$.

Proof: Let $\mathcal{R}_E = \langle A_E, S, \delta_E, \dot{a}_E, F_E \rangle$ be a recognizer such that $L(\mathcal{R}_E) = E$. Likewise, let $\mathcal{R}_F = \langle A_F, S, \delta_F, \dot{a}_F, F_F \rangle$ be a recognizer such that $L(\mathcal{R}_F) = F$, with the following exception. We extend the alphabet of $\mathcal{R}_F$ to S, and let $\forall a \in A_F, \forall x \in S - Y, \delta(a, x) = a$. Let $\mathcal{R} = \langle A, S, \delta, \dot{a}, F' \rangle$ be the recognizer product $\mathcal{R} = \mathcal{R}_E \times \mathcal{R}_F$. We claim that $L(\mathcal{R}) = E[Y{:}F]$.

Let $u \in S^*$. Since elements of $S - Y$ do not cause $\mathcal{R}_F$ to change state, we have that $\phi(u) \in F$ if and only if $\delta_F(\dot{a}_F, u) \in F_F$. Since $u \in E$ if and only if

$\delta_E(\dot{a}_E, u) \in F_E$, and from the definition of recognizer product, we have

$u \in E \wedge \phi(u) \in F$ if and only if $\delta(\dot{a}, u) \in F'$. Thus $L(\mathcal{R}) = E[Y{:}F]$. □

The first result shows that we can directly apply our specification for a context to refine a block in our system specification. This is called a *stimulus deletion* refinement. The second result shows that we can replace stimulus deletion refinements with equivalent regular expressions, and also shows how to construct a state machine for a stimulus deletion refinement.

Two approaches are now available. If the context black box is regular, practitioners may choose to remove the stimulus deletion refinements from the specification, replacing them with regular expressions. If the context black box is not regular, or if practitioners wish to postpone merging the specifications until the state box, they may derive a separate state box for the context specification, and combine the two state boxes using the machine product with choice function f choosing the context machine whenever the response depends on the refinement. The latter approach has the advantage that details relating only to the context are kept separate from the rest of the system details.

5.2.3 Indexed Stimulus Deletion Abstraction

Suppose we have many contexts, all of which share the same specification. Let the contexts be indexed by the set $\mathbf{I}$, and let the stimuli associated with the ith context be denoted Y_i. We note that the stimuli in these sets differ only by index, and we therefore discuss the set of *indexed stimuli* Y. We have a family of stimulus deletion homomorphisms $\left(\phi_{Y_i} \middle| i \in \mathbf{I} \right)$. We would like to use a context specification to refine the system specification, as in the previous section. We note that for a fixed index i, Theorem 5.8 and Theorem 5.9 apply.

1. Fix an index i, and produce a specification for the ith context, indicating the points at which we cannot determine the response without consulting the history of some other context. The resulting specification applies to all contexts.
2. Construct a block in the system specification which includes some or all of the sequences which depend only on the history of the context with the same index as the most recent stimulus. We are using the most recent stimulus to select a particular index from among the many possibilities. Thus we require that the block contain only histories which end in an indexed stimulus.
3. Use the specification produced in 1 to refine the block constructed in 2.

A refinement based on the above is called an *indexed deletion* refinement. Let $E \subseteq S^*$ be a set of sequences all of which end in some stimulus from Y. Let $F \subseteq Y^*$ be a sequence constructed from the indexed stimuli. Let $E[\langle Y, \mathbf{I} \rangle : F]$ denote the set $\{u \in E | u = vx \wedge x \in Y_i \wedge \phi_{Y_i}(u) \in F\}$.

Since specification for a given index applies to all indices, the refined system specification need not specify a particular index. If the index set is small, we may choose to treat the system as though all contexts exist at all times. However, if the index set is large, we will need to regard the contexts as being constructed and destructed. This adds an additional feature to the state box.

We construct a state box for the context specification, and then regard this as a family of state boxes indexed by the set $\mathbf{I}$. Whenever a state box for a particular index i changes state from the initial state to some other state, we regard the state box for that index as "created." Whenever the state box for a particular index i changes state to the initial state, we regard the state box for that index as "destroyed." The indexed state boxes thus model dynamic data.

Note that we have not said much about $\mathbf{I}$. In particular, we have not said $\mathbf{I}$ must be finite. Further, there is no reason to suspect that creation of indices will always work properly. In a real system we may simply run out of memory at some point. To capture this behavior, we recommend that all "creation" transitions in the state box be refined using a "creation success" predicate. Whenever a creation transition is selected, and the creation success predicate holds, the nor-

mal transition is selected (the automorphism is applied, the response generated). Whenever the creation success predicate fails on a creation transition, the state remains unchanged and a response appropriate to the failure is generated (such as a "no memory" message to the user). Refinement of transitions using predicates is discussed in §4.2.4.

One may also use indexed deletion to refine blocks whose members do not end in an indexed stimulus. We allow three types of refinements in this case: existential, universal, and partial. These refinement predicates (see §3.3.2) are constructed as follows.

Letting $u \in S^*$ be a sequence and $F \subseteq Y^*$ a regular set, we define $C_F(u) = \{i \in \mathbf{I} | \phi_{Y_i}(u) \in F\}$ to be the set of all indices i such that the Y_i stimulus deletion of u is a sequence from F. Thus $C_F: S^* \rightarrow \mathcal{P}(\mathbf{I})$ is a mapping. $C_F(u) = \varnothing$ if none of the context histories in u are in F, and $C_F(u) = \mathbf{I}$ if all the context histories in u are in F. If we allow our index set $\mathbf{I}$ to be infinite, the latter condition can only happen if $\lambda \in F$ since u is finite and thus some context histories must be empty.

Definition 5.6 Let $E \subseteq S^*$ be a set of sequences, let $\mathbf{I}$ be an index set, let Y be a set of indexed stimuli, let $F \subseteq Y^*$ be a regular set, and let p be a computable predicate on $\mathcal{P}(\mathbf{I})$. We define the following sets.

$$E[\langle Y, \mathbf{I}\rangle{:}\exists F] = \{u \in E | C_F(u) \neq \emptyset\}$$

$$E[\langle Y, \mathbf{I}\rangle{:}\forall F] = \{u \in E | C_F(u) = \mathbf{I}\}$$

$$E[\langle Y, \mathbf{I}\rangle{:}pF] = \{u \in E | p(C_F(u))\}$$

These sets are the *existential, universal,* and *partial* sets of E, respectively. Their negations are denoted $E[\langle Y, \mathbf{I}\rangle{:}\neg\exists F]$, $E[\langle Y, \mathbf{I}\rangle{:}\neg\forall F]$, and $E[\langle Y, \mathbf{I}\rangle{:}\neg pF]$, respectively.

The existential set consists of those sequences for which there is at least one context history matching F. The universal set consists of those sequences for which all context histories match F. The partial set consists of those sequences such that the set of contexts in F satisfy p. Note that the universal and existential are special cases of the partial.

Theorem 5.10 Let $E \subseteq S^*$ be a set of sequences, let $\mathbf{I}$ be an index set and let Y be the set of indexed stimuli, and let $F \subseteq Y^*$ be a regular set. Then $E[\langle Y, \mathbf{I}\rangle{:}\neg\exists \bar{F}] = E[\langle Y, \mathbf{I}\rangle{:}\forall F]$.

Proof: We begin by proving that $C_{\bar{F}}(u) = \emptyset$ if and only if $C_F(u) = \mathbf{I}$. We will prove the "only if" part; the proof of the "if" is symmetric. Assume

$C_{\bar{F}}(u) = \varnothing$. This means $\neg\exists i \in \mathbf{I}, \phi_{Y_i}(u) \in \bar{F}$, and thus that $\forall i \in \mathbf{I}, \phi_{Y_i}(u) \notin \bar{F}$. Since every sequence $\phi_{Y_i}(u)$ must be in either F or $\bar{F}$, we have $\forall i \in \mathbf{I}, \phi_{Y_i}(u) \in F$, which is equivalent to $C_F(u) = \mathbf{I}$. We are now ready to prove the main result.

$$
\begin{aligned}
E[\langle Y, \mathbf{I}\rangle{:}\neg\exists\bar{Y}] &= \{u \in E \mid C_{\bar{F}}(u) = \varnothing\} && \text{by definition} \\
&= \{u \in E \mid C_F(u) = \mathbf{I}\} && \text{previous result} \\
&= E[\langle Y, \mathbf{I}\rangle{:}\forall F] && \text{by definition}
\end{aligned}
$$

□

From the above result we see that a universal refinement can always be replaced with an equivalent existential refinement. Since we have required that F be a regular set, there is no problem in constructing $\bar{F}$, and the substitution can be automated. If we replace all the universals with existentials, once we construct the state box we need answer only the question "are any of the indexed state boxes in a given state?" This question can be answered with a simple counter which is incremented whenever a state box enters one of the special states, and is decremented whenever a state box leaves one of the special states. Note that the "special states" are those states which are reached by sequences in F. We can insure that these states are represented in the final machine through a recognizer product with the recognizer for F.

Chapter 6

Conclusions and Future Research

The method presented here offers a systematic way to produce specifications from customer requirements, insuring completeness, consistency, and correctness traceability throughout the process. Much of the process can be automated, including the checks for completeness and consistency, sequence generation during enumeration, production of a black box from an enumeration, and the derivation of a state box from a black box. Algorithms have been provided where appropriate. Practitioners are relieved of the necessity of proving the correctness of much of the work, since the process itself can serve in that capacity.

Reuse requires specification composition. The state box has been shown to be the correct level of abstraction to model such composition.

Several problems remain to be solved. In particular, the derivation of a clear box from the state box remains to be explored; however, given a well-formed state box, one could easily argue that the implementation can be performed by implementing each piece of the state box independently, with the state box structure

serving to organize the result. Such an approach lends itself well to object-oriented programming techniques, since the nature of objects and their required methods can be determined from the state box. The development of a program from a specification is discussed at length elsewhere [5,12,16,18].

Other problems include the management of state partitioning during the creation of the state box, capturing data state (possibly during enumeration for reintroduction into the state box), and resolving predicate refinements in the state box. General guidelines need to be created for these tasks, with possible additional automated support.

As this method of writing specifications is used, a catalog of regular operators and abstraction templates will be developed to expedite the work. Automated tool support together with the catalog of operators and templates will make sequence-based box structures applicable to a wide range of applications.

The Trace Assertion Method of Parnas [19] is closely related to the Box Structure Method. Parnas has devoted some attention to the problem of developing automated testing oracles from Trace Assertion specifications. An important avenue of research would be to follow Parnas' lead and explore the implications of developing an automated testing oracle from the sequence-based specifications presented here.

REFERENCES

References

[1] Book, Ronald V. and Friedrich Otto. *String Rewriting Systems.* Springer-Verlag, New York, New York. 1993.

[2] Brassard, Gilles, and Paul Bratley. *Algorithmics, Theory and Practice.* Prentice Hall: Englewood Cliffs, New Jersey. 1988.

[3] Brzozowski, Janusz A. "Derivatives of Regular Expressions." *Journal of the ACM, Volume 11, Number 4.* pp. 481-494, October 1964.

[4] Cohn, P. M. *Universal Algebra.* Harper and Row, Publishers: New York, New York. 1965.

[5] Dromey, Geoff. *Program Derivation: The Development of Programs from Specifications.* Addison-Wesley Publishing Company: Reading, Massachusetts. 1989.

[6] Fetzer, D. and J. H. Poore. "Using Box Structures with the Z Notation." *Proceedings of the 25th Hawaii International Conference on System Sciences.* IEEE, January 1992.

[7] Francez, Nissim. *Program Verification.* Addison-Wesley Publishing Company: Reading, Massachusetts. 1992.

[8] Garg, Vijay K. and M.T. Ragunath. "Concurrent regular expressions and their relationship to Petri nets." *Theoretical Computer Science 96.* pp. 285-304. Elsevier Science Publishers B. V. 1992.

[9] Gécseg, Ferenc. *Products of Automata.* Springer-Verlag: Berlin. 1986.

[10] Grätzer, George. *Universal Algebra (Second Edition).* Springer-Verlag: New York, New York. 1979.

[11] Hopcroft, John E. and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company: Reading, Massachusetts. 1979.

[12] Linger, Richard C, Harlan D. Mills, Bernard I. Witt. *Structured Programming: Theory and Practice.* Addison-Wesley Publishing Company: Reading, Massachusetts. 1979.

[13] Loeckx, Jacques and Kurt Sieber. *The Foundations of Program Verification.* John Wiley and Sons: New York, New York 1987.

[14] Mano, M. Morris. *Digital Logic and Computer Design.* Prentice-Hall, Inc.: Englewood Cliffs, New Jersey 1979.

[15] Mao, Hailong. *The Box-Structure Development Method (Dissertation).* University of Tennessee at Knoxville. December 1993.

[16] Mills, Harlan D. "Stepwise Refinement and Verification in Box-Structured Systems." *Computer.* pp. 23-36. June 1988.

[17] Mills, Harlan D., R. C. Linger, and A. R. Hevener. "Box Structured Information Systems," *IBM Systems Journal, Volume 26, Number 4.* pp. 395-413. 1987.

[18] Mills, Harlan D., R. C. Linger, and A. R. Hevener. *Principles of Information Systems Analysis and Design.* Academic Press, Inc.: Orlando, Florida. 1986.

[19] Parnas, David L., and Yabo Wang. *The Trace Assertion Method of Module Interface Specification.* CRL Report No. 244. Communications Research Laboratory, McMaster University: Hamilton, Ontario. 1992.

[20] Peterson, James L. "Computation Sequence Sets." *Journal of Computer and System Sciences, Volume 13.* pp. 1-24. 1976.

[21] Spivey, J. M. *The Z Notation—A Reference Manual.* Prentice-Hall: Englewood Cliffs, New Jersey. 1989.

[22] Valk, Rüdiger and Guy Vidal-Naquet. "Petri Nets and Regular Languages." *Journal of Computer and System Sciences, Volume 23.* pp. 299-325. 1981.

APPENDICES

Appendix A

Algorithms

A.1 Black Box

Algorithm A.1.1: Enumeration

Inputs: Intended black box function β

Outputs: Enumeration $\mathcal{E}$

---

1. **begin** [*Construct a complete enumeration.*]
2. **let** $\mathcal{E} := \{\lambda \rightarrow 0\}$;
3. **let** $l := 0$;
4. **for each** $u \in \mathcal{E}_l$ in alphabetical order
5. **do** [*Extend sequence* u *by each stimulus.*]
6. **for each** $x \in S$ in alphabetical order
7. **do** [*Extend sequence* u *by stimulus* x.]
8. examine ux for intended equivalence to previous sequences v ;
9. **if** ux is equivalent to v

10. **then** [*Sequence ux is equivalent to previous sequence v. Find an unreduced sequence equivalent to v and add the equivalence to the enumeration.*]
11. **if** v is reduced
12. **then**
13. find an unreduced sequence w which is equivalent to v (guaranteed by the Reduction Technique, Technique 3.2);
14. **let** $v := w$
15. **fi**; [*Sequence ux is equivalent to unreduced prior sequence v. Add this information to the enumeration.*]
16. **let** $\mathcal{E} := \mathcal{E} \cup \{ux \rightarrow \beta(ux)/\equiv v\}$
17. **else** [*An equivalence for sequence ux was not found. Add it as an unreduced sequence (it may be illegal).*]
18. **let** $\mathcal{E} := \mathcal{E} \cup \{ux \rightarrow \beta(ux)\}$
19. **fi** [*Sequence ux has been added to the enumeration.*]
20. **od**; [*This loop terminates when all stimuli have been considered.*]
21. **let** $l := l + 1$
22. **od** [*This loop terminates when the enumeration is complete.*]
23. **end.**

At each step in the above algorithm, we consider only the unreduced sequences of length l not mapped to ω. By the Illegal Prefix Technique (Technique 3.1) we need not consider any sequences mapped to ω. By the Reduced Prefix Technique (Technique 3.3) we need not consider any reduced sequences. We thus generate all sequences of length one (since we start with $l = 0$ and $\mathcal{E}_o = \{\lambda\}$), as guaranteed by Lemma 3.2. Note that when $\mathcal{E}_l$ is empty, the enumeration is complete and, by the Complete Enumeration Theorem (Theorem 3.3) we have all the information needed to produce a black box specification.

Algorithm A.1.2: Enumeration Mealy Machine Construction

Inputs: An enumeration (possibly incomplete) $\mathcal{E}$

Outputs: A Mealy machine $Me_{\mathcal{E}} = \langle A, S, Y, \delta, \gamma, \lambda, 0 \rangle$ which implements the function denoted by the enumeration

---

1. **begin** [*Construct a Mealy machine for the given enumeration.*]
2. **let** $A := \{\lambda\}$, $\delta := \varnothing$, $\gamma := \varnothing$;
3. **let** m be the length of the longest history in $\mathcal{E}$;
4. **for each** n, $1 \leq n \leq m$, in ascending order
5. **do** [*Add all the sequences of length n and the corresponding responses and equivalences to the Mealy machine.*]
6. **for each** $u \rightarrow r$ in $\mathcal{E}$ with $|u| = n$, $r \neq \omega$, $r \neq \iota$, and $u = u'x$ ($u' \in S^*$ and $x \in S$)
7. **do** [*Add the information for a particular unreduced legal sequence u to the machine. Unreduced sequences correspond to states in the machine.*]
8. **let** $A := A \cup \{u\}$;
9. **if** $r \notin Y$
10. **then** [*The response r is not in the machine's response set. Add it.*]
11. **let** $Y := Y \cup \{r\}$
12. **fi**;
13. **let** $\delta(u', x) := u$ and $\gamma(u', x) := r$
14. **od**; [*This loop terminates when all unreduced legal sequences have been added to the machine.*]
15. **for each** $u \rightarrow \omega$ in $\mathcal{E}$ with $|u| = n$ and $u = u'x$ ($u' \in S^*$ and $x \in S$)
16. **do** [*Add a single illegal sequence u to the machine. These sequences all cause the machine to move to the special trap state* ω.]
17. **if** $\omega \notin A$
18. **then** [*If the trap state* ω *has not yet been added, add it now. The state is only added when it is required.*]
19. **let** $A := A \cup \{\omega\}$;
20. **let** $Y := Y \cup \{\omega\}$;
21. **let** $\delta(\omega, y) := \omega$ and $\gamma(\omega, y) := \omega$ for all $y \in S$;
22. **fi**;
23. **let** $\delta(u', x) := \omega$ and $\gamma(u', x) := r$
24. **od**; [*This loop terminates when all illegal sequences have been added to the machine.*]
25. **for each** $u \rightarrow \iota$ in $\mathcal{E}$ with $|u| = n$ and $u = u'x$ ($u' \in S^*$ and $x \in S$)
26. **do** [*Add an unmapped sequence to the machine. These sequences all cause the machine to move to the special trap state* ι.]

27. **if** $\iota \notin A$
28. **then** [*If the trap state* ι *has not already been added, add it now. The state is only added when required.*]
29. **let** $A := A \cup \{\iota\}$;
30. **let** $Y := Y \cup \{\iota\}$;
31. **let** $\delta(\iota, y) := \iota$ and $\gamma(\iota, y) := \iota$ for all $y \in S$;
32. **fi**;
33. **let** $\delta(u', x) := \iota$ and $\gamma(u', x) := \iota$
34. **od**; [*This loop terminates when all unmapped sequences have been added to the machine.*]
35. **for each** $u \rightarrow r/\equiv v$ in $\mathcal{E}$ with $|u| = n$ and $u = u'x$ ($u' \in S^*$ and $x \in S$)
36. **do** [*Add a single reduced sequence to the machine. These sequences denote movement between states added by previously-enumerated unreduced sequences.*]
37. **if** $r \notin Y$
38. **then** [*If the response* r *is not in the machine's response set, add it now.*]
39. **let** $Y := Y \cup \{r\}$;
40. **let** $\delta(u', x) := v$ and $\gamma(u', x) := r$
41. **od** [*This loop terminates when all reduced sequences have been added to the machine.*]
42. **od** [*This loop terminates when all sequences in the enumeration have been added to the machine.*]
43. **end.**

Algorithm A.1.3: Black Box Construction

Inputs: Mealy machine $Me_{\mathcal{E}} = \langle A, S, Y, \delta, \gamma, \lambda, 0 \rangle$

Outputs: Black box $\langle S, Y, \Sigma, \Gamma \rangle$

---

1. **begin** [*Construct a black box specification from a Mealy machine.*]
2. let $\Sigma := \varnothing$, $\Gamma := \varnothing$;
3. create the equivalent Moore machine $M = \langle A', S, Y, \delta', \gamma', \dot{a} \rangle$;
4. **for each** $a \in A'$
5. **do** [*Each state in the machine corresponds to a block in the black box specification.*]

6. **let** $R := \langle A', S, \delta', \dot{a}, \{a\} \rangle$ (note R is a recognizer);
7. **let** $E = L(R)$ be the regular expression for the language accepted by R;
8. **let** $\Sigma := \Sigma \cup \{E\}$;
9. **let** $\Gamma(E) := \gamma'(a)$
10. **od** [*This loop terminates when all states have been coverted to blocks in the black box specification.*]
11. **end.**

A.2 State Box

Algorithm A.2.1: Black Box Mealy Machine Construction Algorithm

Inputs: A regular black box $\langle S, R, \Sigma', \Gamma' \rangle$

Outputs: A Mealy machine $\langle A, S, Z, \delta, \gamma', \dot{a}, \dot{z} \rangle$ which implements the regular black box

---

1. **begin** [*Construct a Mealy machine to implement a regular black box.*]
2. **for each** i, $1 \leq i \leq k$
3. **do** [*Construct a minimum recognizer for each block of the regular black box. Set the initial response for the machine.*]
4. **let** $\mathcal{R}_i := \langle A_i, S, \delta_i, \dot{a}_i, F_i \rangle$ be the minimum recognizer such that $L(\mathcal{R}_i) = E_i$;
5. **if** $\lambda \in E_i$
6. **then** [*This block contains the empty history. Its response must be the initial response. The response should be* 0.]
7. **let** $\dot{z} = \Gamma'(E_i)$
8. **fi**
9. **od**; [*This loop terminates when a recognizer has been built for each block of the specification.*]

```
10.   let A := ∅;
11.   let U := { ⟨ȧ_1, ȧ_2, ..., ȧ_k⟩ } ;
12.   while U ≠ ∅
13.   do [For each unexpanded state in the set U, explore the application of every
            stimulus to that state. Any new states found go in U to be expanded
            next time through the loop. If we don't find any states we are done.]
14.     let A := A ∪ U;
15.     for each a ∈ U
16.     do [Expand a single state by applying every stimulus to it.]
17.       for each x ∈ S
18.       do [Apply stimulus x to state a by applying the stimulus to the correspond-
                ing states of each of the recognizers.]
19.         for each i, 1 ≤ i ≤ k
20.         do [Apply stimulus x to the i th recognizer at state pr_i a. The result is
                the i th component of the resulting Mealy machine.]
21.           let a_i := δ_i(pr_i a, x);
22.           if a_i ∈ F_i
23.           then [This is an accepting state. Since the machine accepts, we
                    assign the response for the corresponding block to the current tran-
                    sition.]
24.             let γ'(a, x) := Γ'(E_i)
25.           fi
26.         od; [This loop terminates when we have considered the stimulus at each
                of the recognizers and generated a Mealy machine state.]
27.         if ⟨a_1, a_2, ..., a_k⟩ ∉ A
28.         then [The state we just generated is new. We add it to the unexpanded
                state set.]
29.           let U := U ∪ { ⟨a_1, a_2, ..., a_k⟩ } ;
30.           let δ(a, x) := ⟨a_1, a_2, ..., a_k⟩
31.         fi
32.       od; [This loop terminates when we have considered each stimulus.]
33.       let U := U − {a}
34.     od [This loop terminates when we have considered each state in the unex-
            panded state set.]
35.   od [This loop terminates when an expansion does not discover any new states.]
```

36. **end.**

The above algorithm operates by generating the Mealy machine from the initial state, thus producing the connected Mealy machine. Brzozowski [3] provides a method for determining the condition in line four, given that E_i is regular. Since Σ' partitions S^*, we know that λ must be in exactly one block E_i, so when we exit the first loop we will have $\dot{z}$ set correctly.

We use U to keep track of the "unexpanded" states. We execute the main loop as long as we have states left to expand. First we put the unexpanded states in A, then we "expand" each state by considering each symbol $x \in S$ from each unexpanded state $a \in U$. We note that by Theorem 4.1, exactly one value of i will satisfy the predicate in line 22, and we will correctly set γ'. Finally, we check to see if we have already added $\delta(a, x)$ to A, and if not, we add it to the unexpanded state set when the loop iterates.

Algorithm A.2.2: Black Box Mealy Machine Minimization

Inputs: The black box Mealy machine $\langle A, S, Z, \delta, \gamma, \dot{a}, \dot{z} \rangle$

Outputs: The minimum Mealy machine $\langle A', S, Z, \delta', \gamma, \dot{a}, \dot{z} \rangle$

---

1. **begin** [*Remove any indistinguishable states from the Mealy machine.*]
2. **let** $L := \varnothing$;
3. **let** $R := \varnothing$;
4. **for each** $a \in A$
5. **do** [*Search the machine to see if a should be removed.*]
6. **if** $a \notin R$ and $a \neq \dot{a}$
7. **then** [*The state is a candidate for removal. Look for a trivially indistinguishable state b.*]
8. **for each** $b \in A$ with $b \neq a$
9. **do** [*See if a and b are trivially indistinguishable.*]
10. **if** $b \notin R$
11. **then** [*Since b is not scheduled for removal, attempt to show that a and b are not trivially indistinguishable.*]

```
12.                 let t := true;
13.                 for each x ∈ S
14.                 do [Consider each stimulus and compare next states and responses.]
15.                    if δ(a, x) ≠ δ(b, x) or γ(a, x) ≠ γ(b, x)
16.                    then [The two states are not trivially indistinguishable.]
17.                       let t := false
18.                    fi
19.                 od; [This loop terminates after each stimulus has been considered.]
20.                 if t = true and a ∉ R
21.                 then [Schedule state a for removal.]
22.                    let R := R ∪ {a};
23.                    let L := L ∪ {⟨a, b⟩}
24.                 fi
25.              fi
26.           od [This loop terminates after each state has been compared to a.]
27.        fi
28.     od; [This loop terminates after every state has been considered for removal.]
29.     for each ⟨a, b⟩ ∈ L
30.     do [Remove the extra states we discovered in the previous loop.]
31.        let A := A − {a};
32.        for each c ∈ A
33.        do [Adjust the next state function, if necessary.]
34.           for each x ∈ S
35.           do [Adjust the next state function for stimulus x, if necessary.]
36.              if δ(c, x) = a
37.              then [Adjust the next state function for stimulus x.]
38.                 let δ(c, x) = b
39.              fi
40.           od [This loop terminates when each stimulus has been considered.]
41.        od [This loop terminates when the next state function has been fixed.]
42.     od [This loop terminates when all states scheduled for removal have been
                removed.]
43. end.
```

In the above algorithm we use R to keep track of the states we plan to remove. We use L to keep track of pairs of trivially indistinguishable states in which one state is to be removed. The first main loop schedules states for removal, and the second main loop removes them.

Algorithm A.2.3: State Box

Inputs: Sequence of stimuli

Outputs: State box responses

---

1. **begin** [*Simulate the operation of the state box on a sequence of inputs.*]
2. **output** $\dot{r}$; [*Output the initial response.*]
3. **for** each $1 \leq i \leq m$
4. **do** [*Initialize tjhe state variables to their initial values.*]
5. **let** $\mathbf{t}_i := \dot{t}_i$
6. **od**;
7. **while** input not exhausted
8. **do** [*Read an input, change state, and write an output.*]
9. **read** **s**;
10. **if** $p_1(\langle \mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_n \rangle, \mathbf{s})$
11. **then** [*The first predicate holds. Output the response and change state.*]
12. **output** r_1;
13. **let** $\langle \mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_n \rangle := \phi_1(\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_n)$
14. **else**
15. **if** $p_2(\langle \mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_n \rangle, \mathbf{s})$
16. **then** [*The second predicate holds. Output the response and change state.*]
17. **output** r_2;
18. **let** $\langle \mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_n \rangle := \phi_2(\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_n)$
19. **else**
20. ...
21. **fi**
22. **fi**
23. **od** [*This loop terminates when all input has been read.*]
24. **end**.

Algorithm A.2.4: State Box Specification

Inputs: Black box Mealy machine $\langle A, S, Z, \delta, \gamma, \dot{a}, \dot{z} \rangle$

Outputs: State box specification

1. **begin** [*Create a state box specification from a black box Mealy machine.*]
2. **for each** $a \in A$
3. **do** [*Encode states as state variables.*]
4. **for each** i, $1 \le i \le k$
5. **do** [*Add the state's value to the range for the state variable.*]
6. **let** $T_i := T_i \cup \mathrm{pr}_i a$
7. **od** [*This loop terminates when the state's values have been added to the state variables' ranges.*]
8. **od**; [*This loop terminates when the state variable ranges match the state set.*]
9. **let** $j := 1$;
10. **for each** $a \in A$
11. **do** [*Encode the information for a each state in the Mealy machine.*]
12. **for each** $x \in S$
13. **do** [*Encode the information for each transition from a given state.*]
14. **let** $r_j := \gamma(a, x)$;
15. **let** $\phi_j := [\mathbf{t}_1 := \mathrm{pr}_1\delta(a, x), \mathbf{t}_2 := \mathrm{pr}_2\delta(a, x), \ldots, \mathbf{t}_k := \mathrm{pr}_k\delta(a, x)]$;
16. **let** $p_j := [\,(\mathbf{t}_1 = \mathrm{pr}_1 a) \wedge (\mathbf{t}_2 = \mathrm{pr}_2 a) \wedge \cdots \wedge (\mathbf{t}_k = \mathrm{pr}_k a) \wedge (\mathbf{s} = x)\,]$;
17. **let** $j := j + 1$
18. **od** [*This loop terminates when all transitions from the chosen state have been encoded.*]
19. **od**; [*This loop terminates when all states have been encoded.*]
20. **let** $i := j$
21. **for each** $x \in S$
22. **do** [*Add the trap for invalid states.*]
23. **let** $r_j := 0$;
24. **let** $\phi_j := [\ ]$;
25. **let** $p_j := [\neg p_1 \wedge \neg p_2 \wedge \cdots \wedge \neg p_{i-1} \wedge (\mathbf{s} = x)\,]$
26. **od** [*This loop terminates when the trap for invalid states has been added.*]
27. **end**.

Algorithm A.2.5: State Space Partitioning

Inputs: State space A

Outputs: State variables which completely partition the state space

---

1. **begin** [*Get state variables and ranges which completely partition the state space.*]
2. **let** $i := 1$;
3. **let** $\rho = A \times A$;
4. **while** $|\rho| > |A|$
5. **do** [*Define a new state variable and range each time, until the state space is completely partitioned.*]
6. **let** $T_i := \varnothing$;
7. **get** state variable name $\mathbf{t}_i$;
8. **for each** $a \in A$
9. **do** [*Get the value corresponding to each state in the machine.*]
10. **get** value $\mathbf{t}_i(a)$;
11. **let** $T_i := T_i \cup \{\mathbf{t}_i(a)\}$;
12. **for each** $b \in A$
13. **do** [*Use the new values to eliminate equivalences and refine the partitioning.*]
14. **if** $\mathbf{t}_i(a) \neq \mathbf{t}_i(b)$ and $\langle a, b \rangle \in \rho$
15. **then**
16. **let** $\rho := \rho - \{\langle a, b \rangle, \langle b, a \rangle\}$;
17. **fi**
18. **od** [*This loop terminates when the partitioning induced by the state variable has been updated.*]
19. **od**; [*This loop terminates when the state variable has been given a value for each state in the machine.*]
20. **let** $i := i + 1$
21. **od** [*This loop terminates when a set of state variables has been defined which completely partition the state space of the machine.*]
22. **end.**

Appendix B

Prime Number Switch

B.1 Requirements

B.1.1 Tagged Requirements

A piece of software should accept a series of integers as input.[R1] For every integer accepted, one of two reports is to be generated.

- R+: report the maximum integer seen so far, the minimum integer seen so far, and the sum of the most recent integer and the previous integer.[R2]
- R*: report the maximum integer seen so far, the minimum integer seen so far, and the product of the most recent integer and the previous integer.[R3]

Begin by generating report R+ [R4] until the current integer is prime. Then switch, and for that integer issue report R*.[R5] Continue issuing report R* until the current integer is again prime; then switch, and issue report R+.[R6] Continue in this manner, switching each time the current integer is prime.[R7]

B.1.2 Interface Specification

The software is not complex enough to warrant a formal interface specification, or a formal natural language specification. The tagged requirements can serve as the

natural language specification, and the following discussion can serve as the interface specification.

The stimulus set will be $S' = \mathbf{Z}^+$. The software does not print any special prompt on invocation, or any message on termination. Further, no information is maintained across invocations of the software. For this reason, invocation and termination details are not considered here, but are deferred to the implementation. It will be assumed that any stimulus sequence contains an implicit initial invocation.

Following the approach given in §3.1, the set S' can be partitioned into prime integers and composite integers, giving the stimulus class set $S = \{P, C\}$. The function $\phi_S: S' \rightarrow S$ can be defined such that $\phi_S(s) = P$ whenever $s \in S'$ is prime, and $\phi_S(s) = C$ whenever $s \in S'$ is composite.

The response set is defined $R = \{R+, R*, 0, \omega\}$, where the null and illegal stimuli (0 and ω) are added in case they are needed. Note that by partitioning the responses in this way, additional information must be added after the regular black box is constructed, and then carried forward as data state in the state box construction (as described in §4.4.1).

B.2 Black Box Construction

B.2.1 Enumeration

The sequences from S^* are enumerated and mapped to responses in R. Whenever the tagged requirements specify a response, it is noted. Whenever the tagged requirements do not specify a response, an assumption is made and tagged.

$\lambda \rightarrow 0$	Required (§3.1).
$P \rightarrow 0$	[A1] There is no previous integer with which to compute the sum or product. To avoid the need for a new response, the null response is used.
$C \rightarrow 0$	[A1]
$PP \rightarrow R+/\equiv C$	[R4], [R5], [R6]
$PC \rightarrow R*/\equiv P$	[R4], [R5]

$CP \rightarrow R*/\equiv P$ [R4], [R5]

$CC \rightarrow R+/\equiv C$ [R4]

As the sequences were enumerated, Technique 3.1 and Technique 3.3 were used to eliminate sequences from consideration. The final enumeration satisfies Definition 3.6 for a complete enumeration, and by Theorem 3.3 gives a response for every sequence in S^*.

B.2.2 Black Box Derivation

Next, Algorithm A.1.2: Enumeration Mealy Machine Construction is used to generate a Mealy machine to encode the implicit rules of the enumeration. The Mealy machine is then converted into a Moore machine, shown below (Fig. B.1).

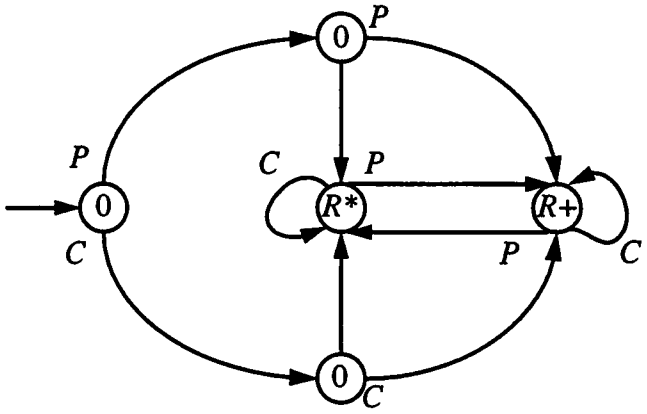


Figure B.1: Enumeration Moore Machine

From the machine, the following black box specification is discovered.

1. $\lambda + P + C \rightarrow 0$
2. $(PC^+ + PPC^*PC^* + C^+PC^*)\ (PC^*PC^*)^* \rightarrow R+$
3. $(C^*PC^*PC^* + CC^*CC^*)^+ \rightarrow R*$

This mapping does not specify which particular response in a response block is to be given for a particular history. To add this final detail, four specification functions must be introduced. The response blocks can then be given in terms of these specification functions. Each specification function will be a mapping from sequences of stimuli to a particular response value. The specification functions required are given below.

- max: $(S')^* \rightarrow \mathbf{Z}^+$
  $\max(\lambda) = 1$
  $\forall u \in (S')^*, \forall x \in S', \max(ux) = x \text{ if } x \geq \max(u)$
  $\max(ux) = \max(u) \text{ if } x < \max(u)$
- min: $(S')^* \rightarrow \mathbf{Z}^+$
  $\min(\lambda) = \infty$
  $\forall u \in (S')^*, \forall x \in S', \min(ux) = x \text{ if } x \leq \min(u)$
  $\min(ux) = \min(u) \text{ if } x > \min(u)$
- prod: $(S')^* \rightarrow \mathbf{Z}^+$
  $\text{prod}(\lambda) = 1$
  $\forall x \in S', \text{prod}(x) = x$
  $\forall u \in (S')^*, \forall x \in S', \forall y \in S', \text{prod}(uxy) = xy$
- sum: $(S')^* \rightarrow \mathbf{Z}^+$
  $\text{sum}(\lambda) = 1$
  $\forall x \in S', \text{sum}(x) = x$
  $\forall u \in (S')^*, \forall x \in S', \forall y \in S', \text{sum}(uxy) = x + y$

Each of the responses can be re-written using the above specification functions. For history $u \in (S')^*$, the possible response values are:

- $R+ = \langle \max(u), \min(u), \text{sum}(u) \rangle$ and
- $R* = \langle \max(u), \min(u), \text{prod}(u) \rangle$.

B.3 State Box Construction

B.3.1 Minimum Recognizer Construction

Minimum recognizers are constructed for each of the expressions in the black box (Fig. B.2, Fig. B.3, and Fig. B.4). This is the first part of Algorithm A.2.1: Black Box Mealy Machine Construction Algorithm.

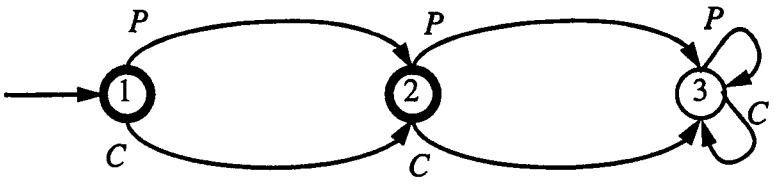


Figure B.2: Minimum Recognizer for Block 1

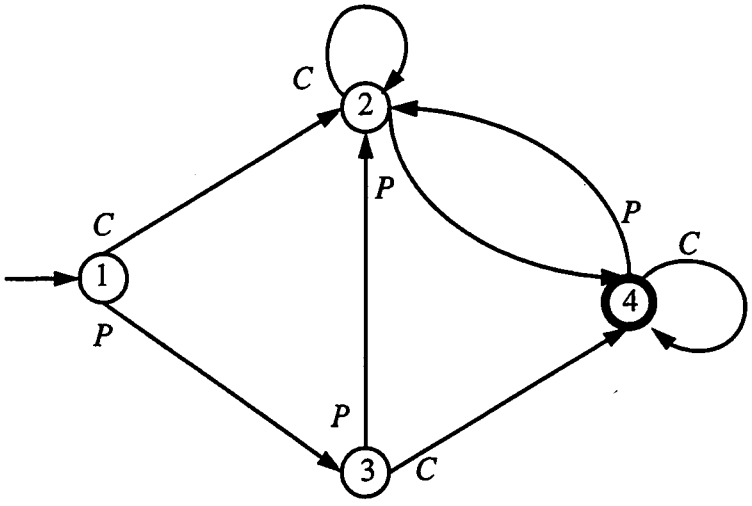


Figure B.3: Minimum Recognizer for Block 2

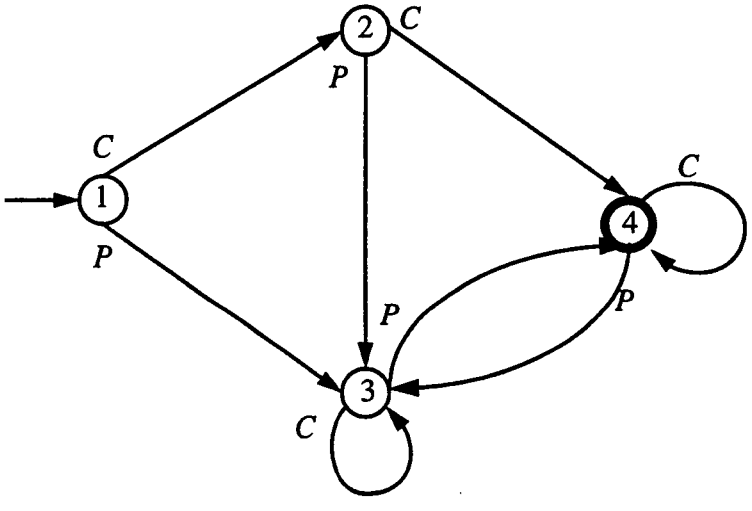


Figure B.4: Minimum Recognizer for Block 3

B.3.2 Mealy Machine Construction

Algorithm A.2.1: Black Box Mealy Machine Construction Algorithm continues by combining the minimum recognizers into a Mealy machine. The resulting Mealy machine is given below (Fig. B.5). The numbers used for the state names correspond to states in the minimum recognizers.

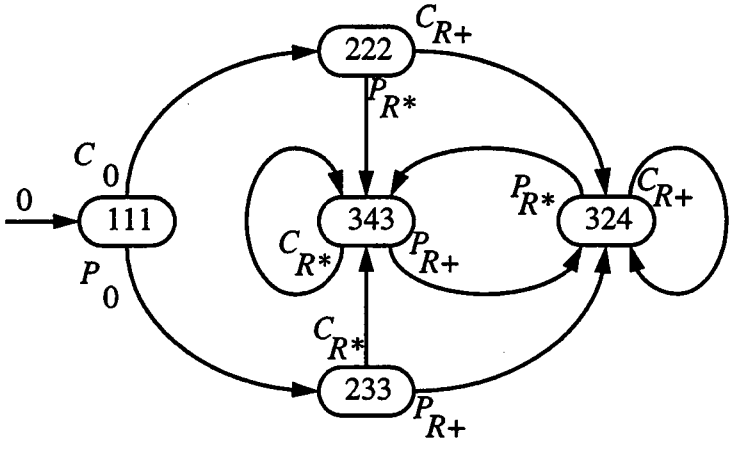


Figure B.5: Black Box Mealy Machine

B.3.3 Mealy Machine Minimization

The next step is to reduce the Mealy machine, if possible, by applying Algorithm A.2.2: Black Box Mealy Machine Minimization. The algorithm identifies two pairs of trivially indistinguishable states: 222 and 324 are trivially indistinguishable, as are 233 and 343. States 324 and 343 are removed, yielding the minimum Mealy machine, which has only three states (Fig. B.5). By Theorem 4.4 this is the minimum Mealy machine.

B.3.4 State Space Partitioning

In order to produce a state box specification, the state space of the Mealy machine must be completely partitioned (Definition 4.5). This can be done by applying Algorithm A.2.5: State Space Partitioning. A single variable "Report" with a three-value range is chosen (Table B.1).

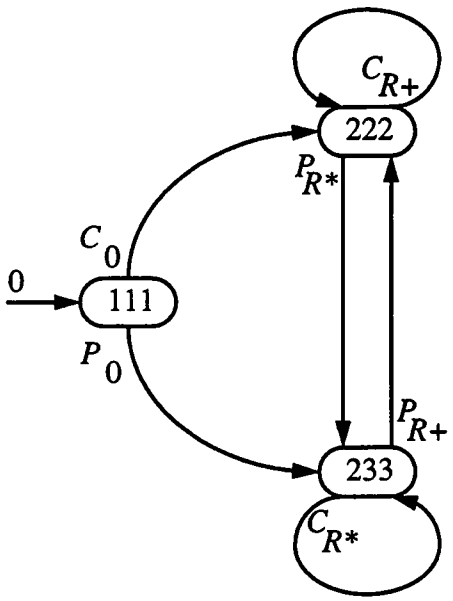


Figure B.6: Black Box Mealy Machine

Table B.1: State Variable

Variable	Value on:	111	222	233
Report		N/A	+	*

B.3.5 State Box Specification

Given the state space partitioning induced by the state variable "Report" and the minimum Mealy machine, the following state box specification is determined using Algorithm A.2.4: State Box Specification.

begin
  s in $\mathbf{Z}^+$;
  r in $\{R+, R*, 0\}$ **initial** 0;
  state
    Report **in** {N/A, +, *} **initial** N/A;
  transition
    (Report = N/A) and (ϕ_S(s) = P) $\rightarrow$ Report := * **response** 0
  | (Report = N/A) and (ϕ_S(s) = C) $\rightarrow$ Report := + **response** 0
  | (Report = +) and (ϕ_S(s) = P) $\rightarrow$ Report := * **response** $R*$

| (Report = +) and (ϕ_S(s) = C) → Report := + **response** R+

| (Report = *) and (ϕ_S(s) = P) → Report := + **response** R+

| (Report = *) and (ϕ_S(s) = C) → Report := * **response** R*

end.

Due to the response partitioning (as was noted during the black box construction), this state box specification captures control state but not data state. To capture data state, new state data items are introduced (Table B.1). These new items are based on the specification functions used in the black box to compute the precise values of the responses.

Table B.2: State Variables

Variable	Range
Max	$\mathbf{Z}^+$
Min	$\mathbf{Z}^+$
Prev	$\mathbf{Z}^+$

Additionally, the state box can be factored and items which do not change state can be removed. This results in the following state box. Note that the response set is now the unpartitioned atomic response set. To simplify this state box, a "followed by" structure is used to imply sequencing of evaluations. That is, when two parts are separated by "followed by" in the transition section, the first is evaluated and any automorphisms applied, then the second is evaluated on the new state box, and any additional automorphisms applied. Note that with such a structure responses should only be specified in one block of a sequence. This structure can always be reduced to the original structure (without "followed by") using function and predicate composition.

begin
  s in $\mathbf{Z}^+$;
  r in $(\mathbf{Z}^+ \times \mathbf{Z}^+ \times \mathbf{Z}^+) \cup \{0\}$ **initial** 0;
  state
    Report **in** {N/A, +, *} **initial** N/A,
    Max **in** $\mathbf{Z}^+$ **initial** 0,
    Min **in** $\mathbf{Z}^+$ **initial** ∞,
    Prev **in** $\mathbf{Z}^+$ **initial** 0;
  transition
    [s < Min → Min := s

```
    ]
    followed by
    [   s > Max → Max := s
    ]
    followed by
    [   (φS(s) = P) and
        [   Report = N/A →  (Report := *) and (Min := s) and (Max := s)
            response 0
        |   Report = + → Report := * response ⟨Max, Min, Prev × s⟩
        |   Report = * → Report := + response ⟨Max, Min, Prev + s⟩
        ]
    |   (φS(s) = C) and
        [   Report = N/A →  (Report := +) and (Min := s) and (Max := s)
            response 0
        |   Report = * →  response ⟨s, s, Prev × s⟩
        |   Report = + →  response ⟨s, s, Prev + s⟩
        ]
    ]   and (Prev := s)
end.
```

Appendix C

Plain Ordinary Telephone System

C.1 Tagged Requirements

C.1.1 Introduction

A software system is to be developed to control the operations of a simple telephone switching system called POTS, or Plain Ordinary Telephone System. Our POTS system will not allow long-distance calls, and there will be no lines from other exchanges. An operator must be able to interact with the system to add new subscribers or disconnect existing subscribers. The scenario is illustrated in Fig. C.1.

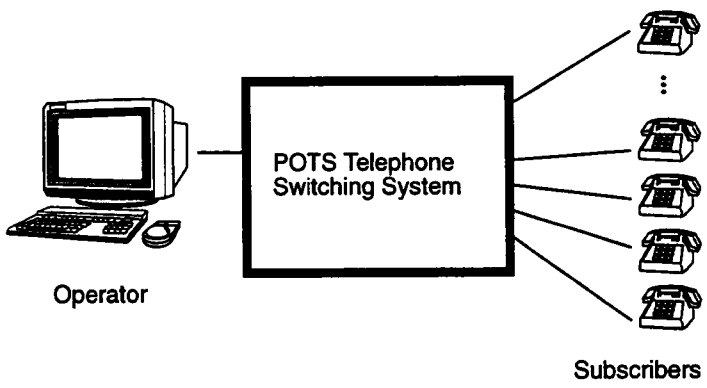


Figure C.1: The POTS Exchange

The POTS system will consist of a simplified digital subscriber switch and its associated control hardware (see Fig. C.2). The exchange, which includes no "intelligence," will forward stimuli from subscribers' phones and the operator console to the control hardware, and will deliver responses from the control hardware to subscribers' phones and the operator console. Additionally, the control hardware will be able to send stimuli to a connector, which can connect phones to each other, or to one of several tones produced by a tone generator.

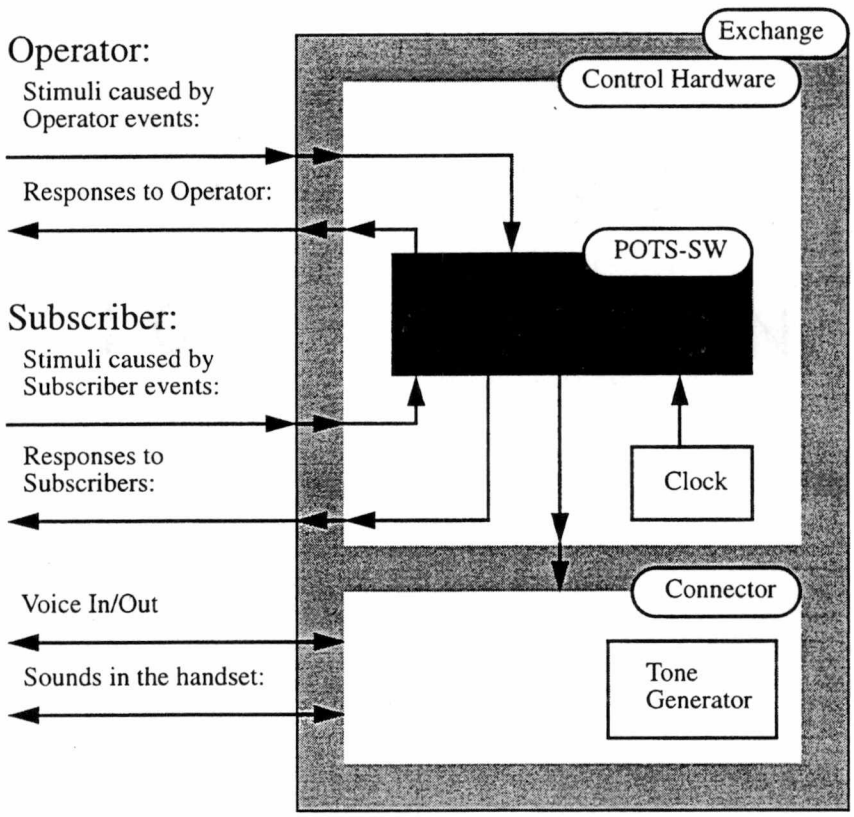


Figure C.2: The Exchange Model

The control hardware will have a microprocessor and memory, and will run an embedded software system called POTS-SW.

C.1.2 POTS Requirements

The requirements for the control hardware are divided into two sections: the hardware specification and the software requirements. Since the software must

interface directly with the hardware, the hardware specifications must be included.

C.1.2.A Specification of the POTS Control Hardware

The POTS control hardware is composed of six components, which are described below (see Fig. C.3).

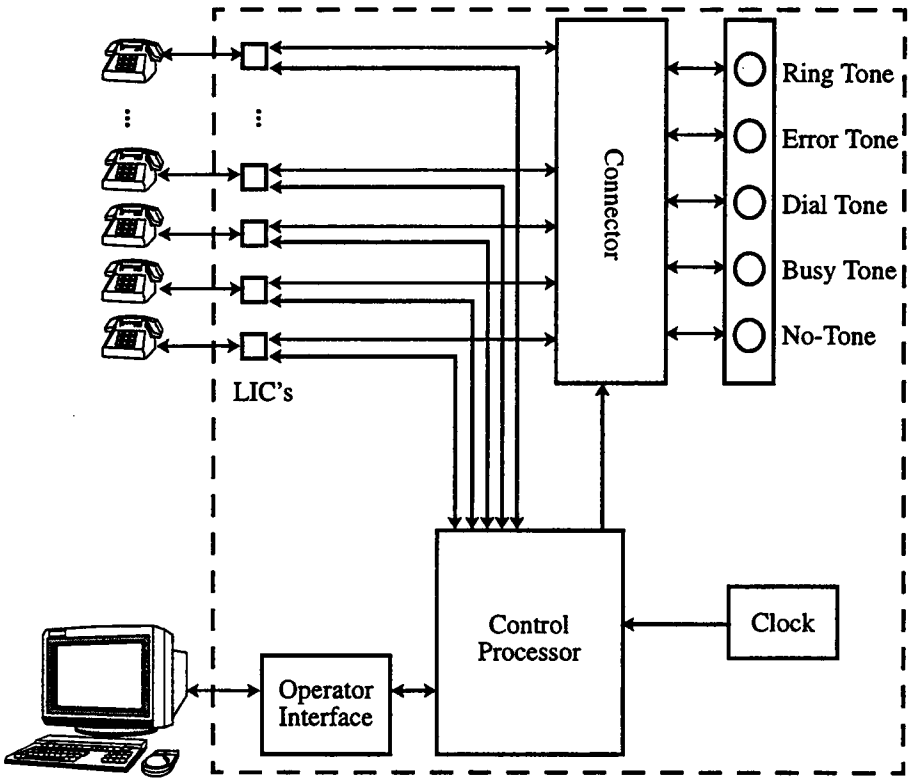


Figure C.3: POTS Hardware

1. Control Processor (CP)

   The control processor contains a CPU, RAM, and ROM for the control software, a two-way bus for communicating with the LIC's, and another one-way bus for communicating with the connector.

   The control processor receives input signals from the LIC's, and sends responses to the LIC's and the connector.[R1.2.1:1]

2. Line Interface Circuits (LIC)

There are 1,000 line interface circuits (LIC's) in a POTS switch.[R1.2.1:2] Each LIC connects one phone to the rest of the POTS hardware, so there may be no more than 1,000 phones connected to POTS.[R1.2.1:3] Each LIC has a line to the connector, which delivers voice and tones.[R1.2.1:4] Each LIC is also connected to two-way bus, allowing communication with the control processor.[R1.2.1:5]

The LIC's each have:

* a power feed to telephone,
* continuous detection of off-hook and on-hook conditions for a connected phone,
* decoding of digit 0-9 presses on a connected phone (* and # do not exist on phones connected to POTS), and
* an A/D converter for starting and stopping the ringing of a connected phone.

The LIC's provides only one service: start or stop a connected phone ringing.[R1.2.1:6] The LIC's provide the following information: notification on transition from off-hook to on-hook, or vice-versa, and notification of digit presses on the attached phone when the phone is off-hook.[R1.2.1:7]

The LIC signals the control processor whenever the connected phone goes on-hook or off-hook, and whenever a digit is pressed with the following exceptions.[R1.2.1:8]

* The LIC ignores any digits pressed while the phone is on-hook.[R1.2.1:9]
* If the attached phone is off-hook at system invocation or restart, the LIC will ignore any events until after it first detects on-hook.[R1.2.1:10]

Other inputs to the LIC are voice and tones, which are not passed through the control processor and thus need not be considered for this problem.

3. Tone Generator (TG)

The tone generator constantly generates the following five tones:

* a ring tone, which is a long beep repeating after an equally long period of silence;
* a dial tone, which is a continuous tone;
* an error tone, which is a sequence of three short beeps, each at a progressively higher pitch, which repeats after an equally long period of silence;
* a busy tone, which is a short tone repeating after a very short period of silence; and

* no-tone, which is continuous silence.

The connector may connect an LIC to at most one of these tones.[R1.2.1:11] The connected tone will be continually broadcast until the connector disconnects the LIC from the tone.[R1.2.1:12] Any number of LIC's may be connected to a given tone, but each LIC may be connected to only one tone at a time.[R1.2.1:13] Connection to a tone breaks any connection to another LIC.[R1.2.1:14]

4. Connector

The connector can connect an LIC to either the tone generator or another LIC.[R1.2.1:15] The connector can also disconnect an LIC from all other lines (no connection).[R1.2.1:16] The connector makes or breaks connections at the request of the control processor, to which it is connected by a one-way bus.[R1.2.1:17]

Any number of LIC's may be connected to the same tone, but each LIC may be connected to only one tone or LIC at a time. Thus connecting an LIC to a tone or another LIC automatically breaks any existing connection.[R1.2.1:18]

5. System Clock

The system clock sends periodic signals (once every 100 milliseconds) to the control processor.[R1.2.1:19] The clock pulses are used by the control processor to synchronize control actions.

6. Operator Interface (OI)

There is an operator interface, so the operator can send messages, including invocation or re-initialization to the system. The operator interface includes external hardware and software. The functions which may be performed from the operator interface are:

* request addition or deletion of a phone and[R1.2.1:20]
* initialize or re-initialize the system.[R1.2.1:21]

C.1.2.B Hardware Assumptions

We make the following simplifying assumptions about the hardware system.

1. No hardware failures occur during execution.[R1.2.2:1]
2. The hardware is sufficient for any request—the bus does not get overloaded, for instance.[R1.2.2:2]
3. Each of the 1,000 LIC's is assigned a physical location which is a three-digit number (000-999). The telephone number associated with an LIC is the same as its physical location.[R1.2.2:3]
4. Each LIC is connected to exactly one phone.[R1.2.2:4]
5. The connector is block-free. All connection requests from the control processor complete.[R1.2.2:5]
6. Internal hardware latencies are small enough to ignore for this problem. [R1.2.2:6]
7. The memory capacity, etc., of the control processor will be chosen to fit whatever is required by the software.[R1.2.2:7]
8. POTS will directly use non-volatile storage for added phone numbers. Thus explicit saving and loading of the subscriber list will not be necessary.[R1.2.2:8]

C.1.2.C Requirements for POTS-SW

The control software must satisfy the following requirements.

1. Subscribers can place calls from connected phones by dialing three digits after they have a dial tone.[R1.2.3:1] A no-tone occurs between digits.[R1.2.3:2] Subscribers have eight seconds after they begin dialing to complete dialing the number (see requirement seven, below).[R1.2.3:3]
2. Phone number validation will occur after all digits are entered.[R1.2.3:4]
3. An error tone will occur when the number dialed is not currently assigned to a subscriber.[R1.2.3:5]
4. A busy tone will occur when the number dialed is either off-hook or ringing.[R1.2.3:6]
5. If an assigned number is dialed and the corresponding phone is on-hook and not ringing, then the A subscriber (the call originator) will hear a ring tone, and the B subscriber's (the call receiver's) phone will ring.[R1.2.3:7]
6. If a phone is picked up while ringing, then a connection will be made between the A and B subscribers, and the two can speak to one another.[R1.2.3:8]
7. For any situation where the subscriber is given a dial tone or a no-tone for more than eight seconds (no-tone occurs after the first digit is dialed from a connected phone with dial tone, and dial tone occurs after going off-hook on a connected phone), the subscriber will receive an error tone.[R1.2.3:9] The error tone will last until the subscriber goes on-hook.[R1.2.3:10] The subscriber cannot make calls during an error tone.[R1.2.3:11]
8. The subscriber may go on hook at any time, and this will result in all appropriate disconnections (phone is disconnected from any tones and other phones, any phone which was connected to a phone which goes on hook is discon-

nected from that phone, and a phone which was started ringing by the phone which went on hook is disconnected from the ring signal).[R1.2.3:12]

9. The operator will invoke the system, which must remember the list of subscribers entered prior to the start.[R1.2.3:13] All phones are disconnected at system start.[R1.2.3:14] Any phones which are off-hook at system start must be placed back on-hook before they can be used.[R1.2.3:15] This detail is handled by the LIC's, which monitor the phone state.[R1.2.3:16] An LIC does not report phone events after system start after until the phone is on-hook.[R1.2.3:17] Any phones receiving any tone or signal at system start are immediately disconnected.[R1.2.3:18]
10. The first time the system is invoked, the list of added subscribers is empty.[R1.2.3:19]
11. The operator can add subscribers to the POTS system by giving a unique phone number (3 digits).[R1.2.3:20] A connected telephone which is not added to the system by the operator will not operate.[R1.2.3:21]
12. The operator can delete subscribers from the POTS system by giving the phone number with the delete command.[R1.2.3:22] Upon deletion, the deleted phone is immediately disconnected (no matter what its status) and will not operate until added by the operator.[R1.2.3:23]

C.2 Interface Specification

The control software receives stimuli from three sources.

1. LIC's (see R1.2.1:1).
2. Operator Interface (see R1.2.1:20,21).
3. Clock (see R1.2.1:19).

The control software issues responses to three destinations. Note that the operator interface is not explicitly identified in the tagged requirements, but it must be included since acknowledgments must be sent for operator actions.

1. LIC's (see R1.2.1:1).
2. Connector (see R1.2.1:1).
3. Operator Interface [A2:1]

Additionally, we add the following "standard" stimuli and responses: the null stimulus 0, and the null and illegal responses 0 and ω, respectively.

C.2.1 LIC Interface

C.2.1.A Stimuli

The following stimuli are generated by an LIC.

Off-hook for phone *n*. Generated by phone *n* going off-hook after having been on-hook (see R1.2.1:7,8). Let $N(n)$ denote this stimulus.

On-hook for phone *n*. Generated by phone *n* going on-hook after having been off-hook previously, given that it was on-hook at system initialization or went on-hook subsequent to system initialization and prior to the current on-hook (see R1.2.1:7,8,10 and R1.2.3:15-17). Let $F(n)$ denote this stimulus.

Digit *d* pressed on phone *n*. Generated when a digit is pressed on a phone which is off-hook and which was either on-hook at system initialization or went on-hook subsequent to system initialization and prior to the current digit press (see R1.2.1:7-10 and R1.2.3:15-17). Let $D(n, d)$ denote this event.

C.2.1.B Responses

The control software generates the following responses to the LIC's.

Start phone *n* ringing. The control software may request that an LIC start its associated phone ringing (see R1.2.1:6). Let RingStart(n) denote this response.

Stop phone *n* ringing. The control software may request that an LIC stop its associated phone ringing (see R1.2.1:6). Let RingStop(n) denote this response.

C.2.1.C Concerns

Requirement R1.2.1.10 states that, under certain circumstances, the LIC will ignore all events from a connected phone. Further, the control software may assume the phone is on-hook. If the control software requests that the LIC start or stop a phone ringing, it will get no confirmation from the LIC in the event the phone is being ignored. This leads to the curious case that a phone which is off-hook may be started ringing, and must be placed back on-hook before it can be answered. The LIC is a pre-existing part, and changes in it would be very costly. Therefore the above behavior is judged acceptable for this version of POTS.

C.2.2 Operator Interface

C.2.2.A Stimuli

Initialize the system. Generated by the operator invoking the system for the first time, or by the operator activating a system-reset switch (see R1.2.1:21). Let *I* denote this stimulus.

Add phone *n* to the system. Generated by the operator (see R1.2.1:20). Let *A*(*n*) denote this stimulus.

Remove phone *n* from the system. Generated by the operator (see R1.2.1:20). Let *R*(*n*) denote this stimulus.

C.2.2.B Responses

Based on an examination of the operator's role, the following is the initial response set for the operator.

"OK". A general acknowledgment, indicating that a requested action has been carried out. Let OK denote this response. [A2.2.2:1]

"Not Added". Notification that the attempt to remove a phone failed, because the phone had not been added. Let Not Added denote this response. [A2.2.2:2]

"Already Added". Notification that the attempt to add a phone failed, because the phone had already been added. Let Already Added denote this response. [A2.2.2:3]

C.2.2.C Concerns

The structure of the operator interface is not clear from the requirements. Additional work should be done to clarify the operator interface requirements.

C.2.3 Connector

C.2.3.A Stimuli

The connector provides no stimuli to the control software (see R1.2.1:1,17).

C.2.3.B Responses

The connector accepts the following responses from the control software.

Connect phone *n* to tone *t*. This is a request to connect a phone to a specific tone of the tone generator (see R1.2.1:11,15). This results in any connections the phone may already have being severed (see R1.2.1:13,14,18). The valid values for *t* are busy, ring, dial, error, and no (see R1.2.1:11). Let Tone(*n*, *t*) denote this response.

Connect phone *n* and phone *m*. This is a request to connect two phones (see R1.2.1:15). Any connections the two phones may already have are severed (see R1.2.1:18). Let Connect(*n*, *m*) denote this response.

Disconnect phone *n* from all connections. This is a request to completely disconnect phone *n* (see R1.2.1:16). Let Disconnect(*n*) denote this response.

C.2.3.C Concerns

What happens if a phone is connected to itself? Upon investigation, it is discovered that the connector ignores any request to connect a phone to itself.[A2.3.3:1]

C.2.4 Clock

C.2.4.A Stimuli

Clock pulse. Generated every 100 milliseconds by the clock (see R1.2.1:19). Let *C* denote this stimulus.

C.2.4.B Responses

The clock accepts no responses from the control software. This is implied by the tagged requirements.[A2.4.2:1]

C.2.4.C Concerns

No concerns are noted at this time.

C.3 Natural Language Specification

The POTS system's response to each stimulus identified in the Interface Specification is given below.

C.3.1 Off-hook for phone *n* *N(n)*

If the phone has been added by the operator and is not ringing, the phone is given dial tone: Tone(*n*, dial) [A3.1:1].

If the phone has been added by the operator and is ringing due to a call from phone *m*, then the ringing is stopped [A3.1:2] and the two phones are connected: RingStop(*n*), Connect(*n*, *m*) (see R1.2.3:8).

If the phone has not been added by the operator, there is no response: 0 (see R1.2.3:21).

C.3.2 On-hook for phone *n* *F(n)*

If the phone has been added by the operator and currently has any tone, the tone is stopped: Disconnect(*n*) (see R1.2.3:12).

If the phone has been added by the operator and was connected to another phone *m*, the phones are disconnected (see R1.2.3:12) and phone *m* is given dial tone: Tone(*m*, dial) [A3.2:1]. Note that this results in phone *n* being disconnected.

If the phone has not been added by the operator, there is no response: 0 (see R1.2.3:21).

C.3.3 Digit *d* pressed on phone *n* *D(n, d)*

If the phone has been added by the operator and is currently connected to dial tone, the dial tone is replaced by no-tone: Tone(*n*, no) (see R1.2.3:2).

If the phone has been added by the operator, is connected to no-tone, and this is not the third digit press since receiving no-tone, the no-tone continues (there is no response): 0 (see R1.2.3:1,2,4).

If the phone has been added by the operator, is connected to no-tone, and this is the third digit press since receiving no-tone, then the following occurs (see R1.2.3:1,4). Let the three digit presses be $d_1 d_2 d_3$, and $m = 100d_1 + 10d_2 + d_3$ the corresponding phone number. If m has not been added by the operator, n is given error tone: Tone(*n*, error) (see R1.2.3:5). If m is currently off-hook or ringing, give busy tone: Tone(*n*, busy) (see R1.2.3:6). If m is on-hook, has been added by the operator, and is not ringing, start m ringing and give n ring tone: RingStart(*m*), Tone(*n*, ring) (see R1.2.3:7).

If the phone has been added by the operator and is connected to error, busy, or ring tone, give no response: 0 [A3.3:1] (and see R1.2.3:11).

If the phone has not been added by the operator, give no response 0 (see R1.2.3:21).

C.3.4 Initialize the system *I*

Disconnect all phones and stop all phones from ringing:
$\forall 0 \le n \le 999$, RingStop(*n*), Disconnect(*n*) (see R1.2.3:14,18). Additionally, give the operator an acknowledgment: OK (see A2.2.2:1).

C.3.5 Add phone *n* to the system *A(n)*

If the phone has already been added to the system, notify the operator: Already Added (see A2.2.2:3).

If the phone has not been added and, add the phone and notify the operator: OK (see A2.2.2:1, R1.2.3:20).

C.3.6 Remove phone n from the system $R(n)$

If the phone has not been added to the system, notify the operator: Not Added (see A2.2.2:2).

If the phone has been added to the system and is not connected, remove it and notify the operator: OK (see A2.2.2:1, R1.2.3:22,23). If the phone has been added to the system and is connected either to tone or another phone, disconnect it, remove it, and notify the operator: Disconnect(n), OK (see A2.2.2:1, R1.2.3:22,23).

C.3.7 Clock pulse C

If any phone n has been receiving no-tone or dial tone for more than 80 clock pulses (80 pulses $\times$ (100 ms/1 pulse) $\times$ (1 second/10^3 ms) = 8 seconds), give the phone error tone: Tone(n, error) (see R1.2.3:9).

C.4 Enumeration

C.4.1 First Enumeration

Given the Natural Language Specification and the Interface Specification, the next task is to enumerate stimuli. However, the total number of stimuli for the system is $1,0 \times 5 + 1,0 \times 4 + 2 = 9,002$. Thus it is impractical to enumerate even sequences of length one. To get around this constraint, we note that each phone is identical in operation to other phones. Thus, the sequences $A(007)$ and $A(931)$ differ only in the specific phone to which any response is directed: they can be characterized by $A(a)$, where a is any phone number. In sequences of greater length, such as $A(007)A(007)$ and $A(007)A(931)$, we are only concerned with the numbers being different. These two sequences can be characterized by $A(a)A(a)$ and $A(a)A(b)$, where we require only that $a \neq b$. This observation motivates an approach to the enumeration. Further, we note that until we consider sequences of length at least six (invoke, add a phone, take phone off hook, press three digits) we will not need to worry about specific digit values.

Let d be any digit, and a, b, and c be any phone numbers which satisfy the constraint $a \neq b \neq c \neq a$. When enumerating, then, we need only enumerate $A(a)$

at length one, since this can stand for adding any phone. We only use b in the enumeration when we already have an a, and similarly for c and b. Such a constraint is even regular, and can be expressed using regular expressions. Let $\mathbf{A} = \{A(a), R(a), F(a), N(a), D(a, d)\}$, $\mathbf{B} = \{A(b), R(b), F(b), N(b), D(b, d)\}$, and $\mathbf{C} = \{A(c), R(c), F(c), N(c), D(c, d)\}$. We are going to generate sequences from the set $\mathbf{A}^* + \mathbf{A}^*\mathbf{B}(\mathbf{A}+\mathbf{B})^* + \mathbf{A}^*\mathbf{B}(\mathbf{A}+\mathbf{B})^*\mathbf{C}(\mathbf{A}+\mathbf{B}+\mathbf{C})^*$. We can also view this approach as an abstraction. With 1,000 such variables, we can map any sequence to a unique abstract sequence by assigning variables to phone numbers throughout the sequence on a first-seen, first-assigned basis.

We enumerate and assign responses as follows.

$\lambda \rightarrow 0$	Required.
$F(a) \rightarrow \omega$	Cannot receive stimuli prior to first invocation. [A4.1:1]
$N(a) \rightarrow \omega$	(see A4.1:1)
$D(a, d) \rightarrow \omega$	(see A4.1:1)
$C \rightarrow \omega$	(see A4.1:1)
$A(a) \rightarrow \omega$	(see A4.1:1)
$R(a) \rightarrow \omega$	(see A4.1:1)
$I \rightarrow$ OK	§C.3.4
$IF(a) \rightarrow 0$	§C.3.1
$IN(a) \rightarrow \omega$	Can't receive on-hooks prior to first off-hook. (see R1.2.1:10)
$ID(a, d) \rightarrow \omega$	Can't receive digits prior to first off-hook. (see R1.2.1:9,10)
$IC \rightarrow 0/ \equiv I$	§C.3.7
$IA(a) \rightarrow$ OK	§C.3.5
$IR(a) \rightarrow$ Not Added$/ \equiv I$	§C.3.6
$II \rightarrow$ OK$/ \equiv I$	§C.3.4
$IF(a)F(a) \rightarrow \omega$	An off-hook phone must go on-hook prior to another off-hook. [A4.1:2] (and see R1.2.1:8)
$IF(a)F(b) \rightarrow 0$	§C.3.1
$IF(a)N(a) \rightarrow 0/ \equiv I$	§C.3.2
$IF(a)N(b) \rightarrow \omega$	(see R1.2.1:10)
$IF(a)D(a, d) \rightarrow 0/ \equiv IF(a)$	§C.3.3

$IF(a)D(b, d) \rightarrow \omega$	(see R1.2.1:9,10)
$IF(a)C \rightarrow 0$	§C.3.7
$IF(a)A(a) \rightarrow$ Tone(a, dial) $\wedge$ OK	§C.3.5 If we know an added phone is off-hook when added, give it dial tone. [A4.1:3]
$IF(a)A(b) \rightarrow$ OK	§C.3.5
$IF(a)R(a) \rightarrow$ Not Added/ $\equiv IF(a)$	§C.3.6
$IF(a)R(b) \rightarrow$ Not Added/ $\equiv IF(a)$	§C.3.6
$IF(a)I \rightarrow$ OK/ $\equiv I$	§C.3.4 (see also R1.2.1:10, R1.2.3:15-17)
$IA(a)F(a) \rightarrow$ Tone(a, dial)/ $\equiv IF(a)A(a)$	§C.3.1
$IA(a)F(b) \rightarrow 0$	§C.3.1
$IA(a)N(a) \rightarrow \omega$	(see R1.2.1:10)
$IA(a)N(b) \rightarrow \omega$	(see R1.2.1:10)
$IA(a)D(a, d) \rightarrow \omega$	(see R1.2.1:9,10)
$IA(a)D(b, d) \rightarrow \omega$	(see R1.2.1:9,10)
$IA(a)C \rightarrow 0/ \equiv IA(a)$	§C.3.7
$IA(a)A(a) \rightarrow$ Already Added/ $\equiv IA(a)$	§C.3.5
$IA(a)A(b) \rightarrow$ OK	§C.3.5
$IA(a)R(a) \rightarrow$ OK/ $\equiv I$	§C.3.6
$IA(a)R(b) \rightarrow$ Not Added/ $\equiv IA(a)$	§C.3.6
$IA(a)I \rightarrow$ OK/ $\equiv IA(a)$	§C.3.5

A machine can be produced for the above enumeration (following Algorithm A.1.2: Enumeration Mealy Machine Construction), and rules derived from the machine (following Algorithm A.1.3: Black Box Construction). Note that this will result in a large "incomplete" block mapped to ι. Reversal of the initial abstraction is not necessary, as we can recover the information later as data state.

Examining any such machine, we would note that the states were occurring whenever we had legal unreduced sequences. These occur whenever a phone goes off-hook or is added by the operator. Going on-hook and being removed by the operator reverse these actions.

Also, several assumptions have been made in this enumeration. These would be reflected back in the natural language specification, which serves two purposes:

1. domain experts who may be unfamiliar with formal specifications can read and validate the natural language specification, and
2. all details are collected in the natural language specification, which is a more accessible starting point for writing user documentation and getting a general understanding of the system specification.

The formal black box which will also be developed is necessary for verification of completeness and consistency (when it is not evident from enumerations), and also for derivation of state data.

C.4.2 Second Enumeration (Deletion)

The previous enumeration was becoming quite long without producing many new results. However, the operator context is largely independent of everything else. We can use stimulus deletion to eliminate the details about other phones from the enumeration and concentrate only on the operator details. The operator stimulus set is $\{A(n), R(n), I\}$, where $0 \le n \le 999$. We will again use a, b, and c as phone numbers. Only responses to the operator are listed.

$\lambda \rightarrow 0$	Required.
$I \rightarrow$ OK	§C.3.4
$A(a) \rightarrow \omega$	(see A4.1:1)
$R(a) \rightarrow \omega$	(see A4.1:1)
$II \rightarrow$ OK/ $\equiv I$	§C.3.4
$IA(a) \rightarrow$ OK	§C.3.5
$IR(a) \rightarrow$ Not Added/ $\equiv I$	§C.3.6
$IA(a)I \rightarrow$ OK/ $\equiv IA(a)$	§C.3.5
$IA(a)A(a) \rightarrow$ Already Added/ $\equiv IA(a)$	§C.3.5
$IA(a)A(b) \rightarrow$ OK	§C.3.5
$IA(a)R(a) \rightarrow$ OK/ $\equiv I$	§C.3.6
$IA(a)R(b) \rightarrow$ Not Added/ $\equiv IA(a)$	§C.3.6

This enumeration does not add much information, but it makes the role of the add and remove stimuli very clear. Enumerating any further would add no additional information. We can encode this information as a computable predicate (perhaps for later use in an abstraction). To insure that predicates are computable, we will write them as recursive rules.

$$\forall 0 \le n \le 999,\ u \in S^*,\ x \in S,$$

$$\begin{aligned}
\text{valid}(\lambda, n) &= \text{true} \\
\text{valid}(ux, n) &= \text{true if } x = A(n) \\
&= \text{false if } x = R(n) \\
&= \text{valid}(u, n) \text{ otherwise}
\end{aligned}$$

The previous enumeration indicated the need to know whether or not a phone is off hook. We can introduce a predicate for that as well.

$$\forall 0 \le n \le 999,\ u \in S^*,\ x \in S,$$

$$\begin{aligned}
\text{offhook}(\lambda, n) &= \text{false} \\
\text{offhook}(ux, n) &= \text{false if } x = I \\
&= \text{false if } x = N(n) \\
&= \text{true if } x = F(n) \\
&= \text{offhook}(u, n) \text{ otherwise}
\end{aligned}$$

C.4.3 Third Enumeration (Indexed Deletion)

The previous enumerations uncovered some details. However we did not, in particular, get to digit presses. As we observed earlier, each phone operates identically to the other phones. This indicates that indexed deletion may be helpful.

We fix a value n, and enumerate the stimuli for that value. Additionally, we can ignore the specific value of any digits pressed (at least until we have three digits). The stimulus set is therefore $\{F, N, D, C, A, R, I\}$. There may be cases in the following enumeration in which information about other phones is required to determine the complete response. Whenever this occurs, it is indicated with a "?" in the response column.

$\lambda \rightarrow 0$	Required.
$F \rightarrow \omega$	(see A4.1:1)
$N \rightarrow \omega$	(see A4.1:1)
$D \rightarrow \omega$	(see A4.1:1)
$C \rightarrow \omega$	(see A4.1:1)
$A \rightarrow \omega$	(see A4.1:1)
$R \rightarrow \omega$	(see A4.1:1)

$I \rightarrow$ OK	§C.3.4
$IF \rightarrow 0$	§C.3.1
$IN \rightarrow \omega$	(see R1.2.1:10)
$ID \rightarrow \omega$	(see R1.2.1:9,10)
$IC \rightarrow 0/ \equiv I$	§C.3.7
$IA \rightarrow$ OK	§C.3.5
$IR \rightarrow$ Not Added$/ \equiv I$	§C.3.6
$II \rightarrow$ OK$/ \equiv I$	§C.3.4
$IFF \rightarrow \omega$	(see A4.1:2, R1.2.1:8)
$IFN \rightarrow 0/ \equiv I$	§C.3.2
$IFD \rightarrow 0/ \equiv IF$	§C.3.3
$IFC \rightarrow 0/ \equiv IF$	§C.3.7
$IFA \rightarrow$ OK, Tone(n, dial)	§C.3.5 (see also A4.1:3)
$IFR \rightarrow$ Not Added$/ \equiv IF$	§C.3.6
$IFI \rightarrow$ OK$/ \equiv I$	§C.3.4
$IAF \rightarrow$?	§C.3.1 This phone may be ringing.
$IAN \rightarrow \omega$	(see R1.2.1:10)
$IAD \rightarrow \omega$	(see R1.2.1:9,10)
$IAC \rightarrow 0/ \equiv IA$	§C.3.7
$IAA \rightarrow$ Already Added$/ \equiv IA$	§C.3.5
$IAR \rightarrow$ OK$/ \equiv I$	§C.3.6
$IAI \rightarrow$ OK$/ \equiv IA$	§C.3.4
$IFAF \rightarrow \omega$	(see A4.1:2, R1.2.1:8)
$IFAN \rightarrow$ Disconnect(n)$/ \equiv IA$	§C.3.2
$IFAD \rightarrow$ Tone(n, no)	§C.3.3
$IFAC \rightarrow 0$	§C.3.7
$IFAA \rightarrow$ Already Added$/ \equiv IFA$	§C.3.5
$IFAR \rightarrow$ OK, Disconnect(n)$/ \equiv IF$	§C.3.6
$IFAI \rightarrow$ OK, Disconnect(n)$/ \equiv IA$	§C.3.4
$IFADF \rightarrow \omega$	(see A4.1:2, R1.2.1:8)
$IFADN \rightarrow$ Disconnect(n)$/ \equiv IA$	§C.3.2

$IFADD \rightarrow 0$	§C.3.3
$IFADC \rightarrow 0$	§C.3.7
$IFADA \rightarrow$ Already Added/ $\equiv IFAD$	§C.3.5
$IFADR \rightarrow$ OK, Disconnect(n)/ $\equiv IF$	§C.3.6
$IFADI \rightarrow$ OK/ $\equiv IA$	§C.3.4
$IFACF \rightarrow \omega$	(see A4.1:2, R1.2.1:8)
$IFACN \rightarrow$ Disconnect(n)/ $\equiv IA$	§C.3.2
$IFACD \rightarrow$ Tone(n, no)/ $\equiv IFAD$	§C.3.3
$IFACC \rightarrow 0$	§C.2.4
$IFACA \rightarrow$ Already Added/ $\equiv IFAC$	§C.3.5
$IFACR \rightarrow$ OK, Disconnect(n)/ $\equiv IF$	§C.3.6
$IFACI \rightarrow$ OK/ $\equiv IA$	§C.3.4
$IFADDF \rightarrow \omega$	(see A4.1:2, R1.2.1:8)
$IFADDN \rightarrow$ Disconnect(n)/ $\equiv IA$	§C.3.2
$IFADDD \rightarrow ?$	§C.3.3 The state of the phone dialed is unknown.
$IFADDC \rightarrow 0$	§C.3.7
$IFADDA \rightarrow$ Already Added/ $\equiv IFADD$	§C.3.5
$IFADDR \rightarrow$ OK, Disconnect(n)/ $\equiv IF$	§C.3.6
$IFADDI \rightarrow$ OK/ $\equiv IA$	§C.3.4
$IFADCF \rightarrow \omega$	(see A4.1:2, R1.2.1:8)
$IFADCN \rightarrow$ Disconnect(n)/ $\equiv IA$	§C.3.2
$IFADCD \rightarrow 0$	§C.3.3
$IFADCC \rightarrow 0$	§C.3.7
$IFADCA \rightarrow$ Already Added/ $\equiv IFADC$	§C.3.5
$IFADCR \rightarrow$ OK, Disconnect(n)/ $\equiv IF$	§C.3.6
$IFADCI \rightarrow$ OK/ $\equiv IA$	§C.3.4
$IFACCF \rightarrow \omega$	(see A4.1:2, R1.2.1:8)
$IFACCN \rightarrow$ Disconnect(n)/ $\equiv IA$	§C.3.2
$IFACCD \rightarrow$ Tone(n, no)/ $\equiv IFAD$	§C.3.3
$IFACCC \rightarrow 0$	§C.3.7

$IFACCA \rightarrow$ Already Added/ $\equiv IFACC$	§C.3.5
$IFACCR \rightarrow$ OK, Disconnect(n)/ $\equiv IF$	§C.3.6
$IFACCI \rightarrow$ OK/ $\equiv IA$	§C.3.4

The above enumeration will yield a partial specification (some class will be mapped to ι) for the behavior of a single phone. Using the approach in §5.2.3, one could use this information to refine the ι-block of the prior specification.

C.4.4 Abstraction

Our ability to complete the enumeration is compromised by three factors. First, we cannot tell if the phone dialed is valid, on hook, and not ringing. Second, we do not have enough information to determine whether or not phone n is ringing when it goes off-hook after being added. Third, we have to count 80 clock pulses to detect the time-out.

To deal with these problems we will introduce abstract stimuli. First, we want to have as precise a definition as possible for the abstract stimuli. During enumeration, however, we may discover that our definitions are inadequate in certain ways, and we can then correct them.

To deal with the clock pulse problem, we introduce an abstract stimulus τ which will occur when there have been 80 clock pulses since the most recent dial or no-tone. The characteristic predicate for this stimulus will have the following form.

$$\begin{aligned} \forall 0 \le n \le 999, u \in S^*, \tau(u, n) &= \text{true if phone } n \text{ has just timed out} \\ &= \text{false otherwise} \end{aligned}$$

A phone times out after 80 clock pulses. If we had a function which counted clock pulses (and reset the count when appropriate), we could write τ as follows.

$$\begin{aligned} &\forall 0 \le n \le 999, u \in S^*, \\ &\quad \tau(\lambda, n) = \text{false} \\ &\quad \tau(u, n) = \text{true if } \text{pulses}(u, n) = 80 \\ &\qquad\quad = \text{false otherwise} \end{aligned}$$

If we had a function which told us when to reset the pulse count, we could write pulses as follows.

$$\begin{aligned}
&\forall 0 \le n \le 999, u \in S^*, x \in S, \\
&\quad \text{pulses}(\lambda, n) = 0 \\
&\quad \text{pulses}(ux, n) = 0 \text{ if } \text{reset}(ux, n) \\
&\qquad\qquad = \text{pulses}(u, n) \text{ if } \neg\text{reset}(ux, n) \text{ and } x \neq C \\
&\qquad\qquad = 1 + \text{pulses}(u, n) \text{ if } \neg\text{reset}(ux, n) \text{ and } x = C
\end{aligned}$$

We need to reset the count whenever a valid phone goes off hook while not ringing, and whenever we get the first digit press. We need two functions, one to count the digits and one to determine when a phone is ringing. We will need the ringing function for other things as well. Given functions digitcount(u, n) and $\rho(u, n)$ which return the digit count for phone n and true when phone n is ringing, respectively, we can write the reset function.

$$\begin{aligned}
&\forall 0 \le n \le 999, u \in S^*, x \in S, \\
&\quad \text{reset}(\lambda, n) = \text{true} \\
&\quad \text{reset}(ux, n) = \text{true if } x = F(n) \\
&\qquad\qquad = \text{true if } x = D(n, d) \text{ and digitcount}(u, n) = 0 \\
&\qquad\qquad = \text{true if } x = A(n) \text{ and } \neg\text{valid}(u, n) \text{ and offhook}(u, n) \\
&\qquad\qquad = \text{true if } \neg\text{valid}(ux, n) \text{ or } \neg\text{offhook}(ux, n) \\
&\qquad\qquad = \text{false otherwise}
\end{aligned}$$

The function digitcount can be written, given the existing functions and predicates.

$$\begin{aligned}
&\forall 0 \le n \le 999, u \in S^*, x \in S, \\
&\quad \text{digitcount}(\lambda, n) = 0 \\
&\quad \text{digitcount}(ux, n) = 0 \text{ if } \neg\text{offhook}(ux, n) \text{ or } \neg\text{valid}(ux, n) \\
&\qquad\qquad = \text{digitcount}(u, n) \text{ if } x \neq D(n, d) \text{ and offhook}(ux, n) \\
&\qquad\qquad\quad \text{and valid}(ux, n) \\
&\qquad\qquad = \text{digitcount}(u, n) + 1 \text{ if } x = D(n, d) \text{ and offhook}(ux, n) \\
&\qquad\qquad\quad \text{and valid}(ux, n)
\end{aligned}$$

We have written everything but the function to decide whether or not a phone is ringing. We want a function like the following.

$$\begin{aligned}
&\forall 0 \le n \le 999, u \in S^*, \rho(u, n) = \text{true if phone } n \text{ is ringing} \\
&\qquad\qquad\qquad\qquad = \text{false otherwise}
\end{aligned}$$

To create such a function, we must know when one phone dials another. We need another function like the following.

$$\forall 0 \le n \le 999, u \in S^*, \delta(u, m, n) = \text{true if phone } m \text{ just dialed phone } n \text{ without timeout}$$
$$= \text{false otherwise}$$

If we can write the above function, we can define ρ as follows.

$$\forall 0 \le n \le 999, u \in S^*, x \in S,$$
$$\rho(\lambda, n) = \text{false}$$
$$\rho(ux, n) = \text{true if valid}(ux, n) \text{ and onhook}(ux, n) \text{ and } \exists 0 \le m \le 999, \delta(ux, m, n)$$
$$= \text{false otherwise}$$

This means we are going to have mutual recursion. We must insure that δ is defined such that the recursion will always terminate.

$$\forall 0 \le n \le 999, u \in S^*, x \in S,$$
$$\delta(\lambda, m, n) = \text{false}$$
$$\delta(ux, m, n) = \text{true if pulses}(ux, m) < 80 \text{ and } x = D(m, d) \text{ and digitcount}(ux, m) = 3 \text{ and number}(ux, m) = n \text{ and } \neg\rho(u, n)$$

Now we need the number function, which returns the last number dialed by the phone. This function only has to remember the last three digits dialed historically.

$$\forall 0 \le n \le 999, u \in S^*, x \in S,$$
$$\text{number}(\lambda, n) = 0$$
$$\text{number}(ux, n) = \text{number}(u, n) \text{ and } x \neq D(n, d)$$
$$= [10 \times \text{number}(u, n) + d] \bmod 1000 \text{ if } x = D(n, d)$$

Finally, it would be convenient to know the last caller of a particular phone. Given the functions and predicates we already have, this is fairly easy.

$$\forall 0 \le n \le 999, u \in S^*, x \in S,$$
$$\text{lastcaller}(\lambda, n) = \text{nobody}$$
$$\text{lastcaller}(ux, n) = m \text{ if } \delta(ux, m, n) \text{ and } \neg\rho(ux, n) \text{ and onhook}(ux, n) \text{ and valid}(ux, n)$$
$$= \text{lastcaller}(u, n) \text{ otherwise}$$

We already know the behavior of the system in most cases. To get the remaining cases, we need a new level of abstraction. The following are the characteristic predicates for the abstraction. Let $u \in S^*$ and $x \in S$. The predicates are all false at λ. Note that we are still operating with a fixed $0 \le n \le 999$ (indexed deletion).

- System invocation (I).

  $I(ux)$ = true if $x = I$
  = false otherwise

- Addition of phone n (A).

  $A(ux)$ = true if $x = A(n)$
  = false otherwise

- Removal of phone n (R).

  $R(ux)$ = true if $x = R(n)$
  = false otherwise

- Phone n going on hook (N).

  $N(ux)$ = true if $x = N(n)$
  = false otherwise

- Phone n going off hook (F).

  $F(ux)$ = true if $x = F(n)$
  = false otherwise

- Phone n timing out (τ).

  $\tau(ux)$ = true if $\tau(ux, n)$
  = false otherwise

- Phone n dialing a phone which is currently off-hook (δ_F).

  $\delta_F(ux)$ = true if $\delta(ux, n,$ number$(ux, n))$ and offhook$(ux,$ number$(ux, n))$
  and valid$(ux,$ number$(ux, n))$
  = false otherwise

- Phone n dialing a phone which is currently ringing (δ_R).

$\delta_R(ux)$ = true if $\delta(ux, n, \text{number}(ux, n))$ and $\rho(ux, \text{number}(ux, n))$
= false otherwise

- Phone n dialing a phone which is currently not valid (δ_I).

$\delta_I(ux)$ = true if $\delta(ux, n, \text{number}(ux, n))$ and $\neg\text{valid}(ux, \text{number}(ux, n))$
= false otherwise

- Phone n dialing a valid phone which is on hook and not ringing (δ_G).

$\delta_G(ux)$ = true if $\delta(ux, n, \text{number}(ux, n))$ and $\neg\delta_F(ux)$ and $\neg\delta_R(ux)$ and $\neg\delta_I(ux)$
= false otherwise

- Another phone dialing phone n (i).

$i(ux)$ = true if $\exists m, \delta(ux, m, n)$
= false otherwise

- The phone which originated the incoming call is removed by the operator (R_i).

$R_i(ux)$ = true if $x = R(m)$ where $m = \text{lastcaller}(ux, n)$
= false otherwise

- The phone which originated the incoming call goes on hook (N_i).

$N_i(ux)$ = true if $x = N(m)$ where $m = \text{lastcaller}(ux, n)$
= false otherwise

- The phone called by phone n is removed by the operator (R_o).

$R_o(ux)$ = true if $x = R(m)$ where $n = \text{lastcaller}(ux, m)$
= false otherwise

- The phone called by phone n goes on hook (N_o).

$N_o(ux)$ = true if $x = N(m)$ where n = lastcaller(ux, m)
= false otherwise

- The phone called by phone n goes off hook (F_o).

$F_o(ux)$ = true if $x = F(m)$ where n = lastcaller(ux, m)
= false otherwise

We can now enumerate abstract sequences. For the sake of brevity, illegal sequences are not included here. In the following let l = lastcaller(u, n) and let m be the value which satisfies n = lastcaller(u, m) for history u.

$\lambda \rightarrow 0$	Required.
$I \rightarrow$ OK	§C.3.4
$IA \rightarrow$ OK	§C.3.5
$IR \rightarrow$ Not Added/ $\equiv I$	§C.3.6
$IF \rightarrow 0$	§C.3.1
$Ii \rightarrow$ Tone(i, error)/ $\equiv I$	§C.3.3
$IAI \rightarrow$ OK/ $\equiv IA$	§C.3.4
$IAA \rightarrow$ Already Added/ $\equiv IA$	§C.3.5
$IAR \rightarrow$ OK/ $\equiv I$	§C.3.6
$IAF \rightarrow$ Tone(n, dial)	§C.3.1
$IAi \rightarrow$ RingStart(n), Tone(i, ring)	§C.3.3
$IFI \rightarrow$ OK/ $\equiv I$	§C.3.4
$IFA \rightarrow$ Tone(n, dial)/ $\equiv IAF$	§C.3.5
$IFR \rightarrow$ Not Added/ $\equiv IF$	§C.3.6
$IFN \rightarrow 0$/ $\equiv I$	§C.3.2
$IFi \rightarrow$ Tone(i, error)/ $\equiv IF$	§C.3.3
$IAFI \rightarrow$ OK, Disconnect(n)/ $\equiv IA$	§C.3.4
$IAFA \rightarrow$ Already Added/ $\equiv IAF$	§C.3.5
$IAFR \rightarrow$ OK, Disconnect(n)/ $\equiv IF$	§C.3.6
$IAFN \rightarrow$ Disconnect(n)/ $\equiv IAF$	§C.3.2
$IAF\tau \rightarrow$ Tone(n, error)	§C.3.7

$IAF\delta_F \rightarrow$ Tone(n, busy)/ $\equiv IAF\tau$	§C.3.3
$IAF\delta_R \rightarrow$ Tone(n, busy)/ $\equiv IAF\tau$	§C.3.3
$IAF\delta_I \rightarrow$ Tone(n, error)/ $\equiv IAF\tau$	§C.3.3
$IAF\delta_G \rightarrow$ Tone(n, ring), RingStart(m)	§C.3.3
$IAFi \rightarrow$ Tone(i, busy)/ $\equiv IAF$	§C.3.3
$IAiI \rightarrow$ OK, Disconnect(n), RingStop(n)/ $\equiv IA$	§C.3.4
$IAiA \rightarrow$ Already Added/ $\equiv IAi$	§C.3.5
$IAiR \rightarrow$ OK, RingStop(n), Disconnect(i)/ $\equiv I$	§C.3.6
$IAiF \rightarrow$ RingStop(n), Connect(i, n)	§C.3.1
$IAiR_i \rightarrow$ RingStop(n), OK, Disconnect(l)/ $\equiv IA$	§C.3.6 and stop any associated phone from ringing. [A4.4:1]
$IAiN_i \rightarrow$ RingStop(n)/ $\equiv IA$	When a phone hangs up, stop any associated phone from ringing. [A4.4:2]
$IAF\tau I \rightarrow$ OK, Disconnect(n)/ $\equiv IA$	§C.3.4
$IAF\tau A \rightarrow$ Already Added/ $\equiv IAF\tau$	§C.3.5
$IAF\tau R \rightarrow$ OK, Disconnect(n)/ $\equiv IF$	§C.3.6
$IAF\tau N \rightarrow$ Disconnect(n)/ $\equiv IA$	§C.3.2
$IAF\tau i \rightarrow$ Tone(l, busy)/ $\equiv IAF\tau$	§C.3.3
$IAF\delta_G I \rightarrow$ OK, RingStop(m), Disconnect(n)/ $\equiv IA$	§C.3.4
$IAF\delta_G A \rightarrow$ Already Added/ $\equiv IAF\delta_G$	§C.3.5
$IAF\delta_G R \rightarrow$ OK, RingStop(m), Disconnect(n)/ $\equiv IF$	§C.3.6 (and see A4.4:1)
$IAF\delta_G N \rightarrow$ RingStop(m), Disconnect(n)/ $\equiv IA$	§C.3.2 (and see A4.4:2)
$IAF\delta_G R_o \rightarrow$ RingStop(m), Tone(n, error)/ $\equiv IAF\tau$	(see A4.4:1) Issue an error tone to the previously-connected phone when a phone is removed. [A4.4:3]
$IAF\delta_G F_0 \rightarrow$ RingStop(m), Connect(m, n)	§C.3.1
$IAiFI \rightarrow$ OK, Disconnect(n)/ $\equiv IA$	§C.3.4
$IAiFA \rightarrow$ Already Added/ $\equiv IAiF$	§C.3.5

$IAiFR \rightarrow$ OK, Disconnect(n), Tone(l, error)/ $\equiv IF$	§C.3.6
$IAiFN \rightarrow$ Disconnect(n), Tone(l, dial)/ $\equiv IA$	§C.3.2 On hanging up, give a previously-connected phone dial tone. [A4.4:4]
$IAiFR_i \rightarrow$ OK, Disconnect(l), Tone(n, error)/ $\equiv IAF\tau$	(see A4.4:3)
$IAiFN_i \rightarrow$ Disconnect(l), Tone(n, dial)/ $\equiv IAF$	(see A4.4:4)
$IAF\delta_G F_o I \rightarrow$ OK, Disconnect(n)/ $\equiv IA$	§C.3.4
$IAF\delta_G F_o A \rightarrow$ Already Added/ $\equiv IAF\delta_G F_o$	§C.3.5
$IAF\delta_G F_o R \rightarrow$ OK, Disconnect(n), Tone(m, error)/ $\equiv IF$	§C.3.6
$IAF\delta_G F_o N \rightarrow$ Disconnect(n), Tone(m, dial)/ $\equiv IAF\delta_G F_o$	§C.3.2
$IAF\delta_G F_o i \rightarrow$ Tone(l, busy)/ $\equiv IAF\delta_G F_o$	§C.3.3
$IAF\delta_G F_o R_o \rightarrow$ OK, Disconnect(m), Tone(n, error)/ $\equiv IAF\tau$	§C.3.6 (and see A4.4:3)
$IAF\delta_G F_o N_o \rightarrow$ Disconnect(m), Tone(n, dial)/ $\equiv IAF$	§C.3.2 (and see A4.4:4)

The enumeration is complete.

Examining the above enumeration reveals that our definition for reset is insufficient. We have added several new points at which dial tone begins (when one party in a call hangs up). Fortunately, by generating implicit rules from the above enumeration, we can see exactly what sequences are mapped to a response which includes dial tone, and can easily correct the definition of reset.

C.4.5 Conclusion

The remainder of the process involves generating implicit rules for each enumeration and combining the resulting specifications into a single black box. Since the process is mechanical from this point to the final result, we conclude the example.

Vita

Stacy J. Prowell was born in Nashville, Tennessee, on 22 January 1968. He graduated from Manchester Central High School in May 1986. Mr. Prowell then attended the University of Tennessee, where he first studied aerospace engineering before switching to computer science.

From August 1992 until December 1995, Mr. Prowell worked as a Research Assistant in the Software Quality Research Laboratory, where he contributed to various research, development, and technology transfer efforts, and also worked as the lab's system administrator. During the summers of 1992 and 1993, Mr. Prowell left the university to work for Software Engineering Technology in Lanham, Maryland. In December 1992, Mr. Prowell was awarded a B.S. in Computer Science, and in May 1995, Mr. Prowell was awarded an M.S. in Computer Science. Mr. Prowell currently works as a consultant for Software Engineering Technology, Inc.

Mr. Prowell was awarded the Ph.D. in computer science in May 1996. He is a member of the Association for Computing Machinery and the American Association for the Advancement of Science.