A Discipline of Specification-Based Test Derivation

by

Michael R. Donat

M.Sc., University of Edinburgh, 1987

B.MATH, University of Waterloo, 1985

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

we accept this thesis as conforming
to the required standard

___

___

___

The University of British Columbia

September 1998

Abstract

System-level requirements-based testing is an important task in software development, providing evidence that each requirement has been satisfied. There are two major problems with how these tests are derived. First, the notion of coverage is subjective, i.e., there is a lack of objective definitions of coverage criteria. Second, there is a surprising lack of automation in deriving system-level requirements-based tests. Research into solutions for these problems has led to the formulation of the discipline of specification-based test derivation presented in this dissertation.

This discipline, which is based on predicate logic, provides a scientific foundation for objective definitions of coverage criteria and algorithms for partially automating test derivation. This dissertation defines some fundamental coverage criteria as examples. A *general test frame generation process* illustrates a general application of the discipline to a broad range of formal specifications, which can include existential and universal quantification. A refinement of the process can be applied to system-level requirements-based testing.

The discipline leverages work invested in compiling the requirements specification. In addition to partially automating the task of verifying that the requirements have been satisfied, the refined process automates the traceability of requirements to test descriptions. Other applications of the discipline of specification-based

test derivation include requirements validation and objective measurements for requirements complexity. The discipline can also be used to predict the expected number of tests to be derived, which can then be used for process statistics. The uses of this discipline as a basis for repeatable processes, definitions, and measurements imply that it can form part of software development processes at Capability Maturity Model (CMM) Levels 2 through 5.

Contents

Abstract ii

Contents iv

List of Tables x

List of Figures xi

Acknowledgements xii

Dedication xiv

1 Introduction 1

1.1 Objective 2

1.2 Motivation 2

1.3 Related Techniques 3

1.4 Approach 4

1.5 Contributions 6

1.6 Outline 9

2 The Problem 11

2.1 Introduction 12
2.2 Testing 14
2.3 Coverage Criteria 15
2.4 System-Level Requirements-Based Testing 16
2.5 Manual Test Frame Derivation 19
2.6 Coverage via Traceability 26
2.7 Lack of Automation 27
2.8 Towards a Solution 31
2.9 Motivation for a Mathematical Approach 33

3 Existing Solutions **37**

3.1 Introduction 37
3.2 Systematic Approaches 38
3.3 Code-based Testing 40
3.3.1 Principles 40
3.3.2 An Objective Criterion 42
3.3.3 Automation 43
3.4 Logic-Based Techniques 44
3.4.1 Finite State Machines 45
3.4.2 Logical Manipulation 48
3.4.3 Disadvantages of Modelling 50
3.4.4 Coverage Schemes 51
3.5 Conclusion 51

4 Fundamental Challenges **53**

4.1 Introduction 53

4.2 Specifications as Code 55

4.3 Structural Independence 59

4.4 Condition Dependence 60

4.5 Quantification 62

4.6 The Delta Problem 63

4.7 Summary 64

5 A Foundation for the Discipline **66**

5.1 Introduction 66

5.2 A Place to Start 68

5.3 Notation and Terminology 69

5.4 Overview 72

5.5 Test Class Normal Form 75

5.5.1 The Test Class Algorithm 75

5.5.2 Example 79

5.5.3 Existential Quantification 83

5.5.4 Demonic Choice 85

5.6 Generating Test Frames 86

5.6.1 Frame Stimuli 87

5.6.2 Coverage Schemes 91

5.7 Conclusion 94

6 Coverage Criteria **95**

6.1 Introduction 95

6.2 Objective Definitions of Coverage Criteria 97

6.3 Relative Effectiveness 98

6.4 Test Class Variations 99
6.4.1 Detailed 99
6.4.2 Focused 100
6.5 Resolving Non-Deterministic Test Classes 100
6.6 Assuming a Closed World 101
6.7 Simplifying Quantifiers 102
6.8 Mathematical Definition of Term Coverage 105
6.9 Differentiated Test Frames 106
6.10 Summary 109

7 Formal Specification-Based Testing 110

7.1 Introduction 110
7.2 Process Overview 112
7.3 Tackling Complex Specifications 114
7.4 Formalizing Domain Knowledge 115
7.4.1 Elaboration 116
7.4.2 Simplification and Infeasibility 117
7.5 Rewrite System 121
7.6 Distinguishing Stimuli and Responses 123
7.7 Algorithms for Coverage Schemes 124
7.7.1 Implicant Coverage 125
7.7.2 DNF Coverage 126
7.7.3 Term Coverage 127
7.7.4 Infeasible Test Frames and Coverage Schemes 127
7.8 Examples 128
7.8.1 Steam Boiler 128

7.8.2 North Atlantic Separation Minima 133

7.9 Conclusion 138

8 System-Level Requirements-Based Testing 140

8.1 Introduction 140

8.2 Process Overview 142

8.3 The Q Specification Language 144

8.3.1 Overview 144

8.3.2 Expressions 150

8.3.3 Predicate Definitions 151

8.3.4 Conjunctive and Disjunctive Lists 151

8.3.5 Argument-Based Conjunctions and Disjunctions 152

8.3.6 Expression Aliasing 153

8.3.7 Argument Permutation 154

8.3.8 Quantification 154

8.4 Traceability 155

8.5 Examples 155

8.5.1 CAATS SRS 155

8.5.2 ICAO Flight Plan 158

8.6 Additional Benefits 161

8.6.1 Validation 161

8.6.2 Complexity and Progress Measurement 163

8.7 Summary 163

9 Conclusions 165

9.1 Research Results 165

9.2 Foundations for Future Work 168
9.2.1 Test Frame Generation Process Improvements 168
9.2.2 Delta Heuristics 169
9.2.3 Methodology 169
9.2.4 Next Step for Q 171
9.2.5 Specification Projection 171
9.3 Epilogue 172

Bibliography **173**

Appendix A Rules for Argument-Based Conjunctions and Disjunctions **180**

Appendix B Automatically Generated Test Frames for the Steam Boiler Control **182**
B.1 S Specification 182
B.2 Base Test Frames 185
B.3 Differentiated Test Frames 191

Appendix C A Heuristic for the Delta Problem **213**

List of Tables

7.1 Numbers of Prime Implicants and Test Frames 132

8.1 An Automatically Generated Test Frame 157

List of Figures

2.1 Table A-7 from DO178B 28

3.1 Example Program 42

5.1 Entity Relationships 73

5.2 Coverage Schemes 93

7.1 Automated Test Frame Generation 112

7.2 NATS S Specification Fragment. 133

7.3 A NATS Test Frame. 134

8.1 Integrating Automated Test Frame Generation 143

8.2 ICAO Flight Plan Specification Fragment. 159

8.3 An ICAO Flight Plan Test Frame. 159

Acknowledgements

Most of all, thanks to my supervisor, Dr. Jeffrey J. Joyce, for his encouragement when the light at the end of the tunnel was only a figment of my imagination. I would like to thank my wife, Cindy Goundrey, for her understanding, support, and encouragement. I would also like to thank my fellow students and friends for their "in-the-trenches" camaraderie and inspiration. I am also deeply indebted to the members of my committee, Mark Greenstreet, Norm Hutchinson, Paul Gilmore, Philippe Kruchten, and Kal Toth, who have provided many insights from their different perspectives of my work.

I am proud of the condition of this dissertation, and owe a great deal of thanks to those who have contributed many valuable comments on earlier drafts: Jeff Joyce, my thesis committee, Tony Earnshaw, Christoph Kern, and Shauna Turner.

Comments from those in industry have had great influence on my work. I would especially like to thank Jim Ronback of Raytheon Systems Canada Ltd., and Phil Gray and Richard Yates of MacDonald Dettwiler for their questions, comments, criticisms, and insights.

This work was supported by **formalWARE**, a university-industry collaborative research project sponsored jointly by the BC Advanced Systems Institute, Raytheon Systems Canada Ltd., MacDonald Dettwiler, The University of British

Columbia and The University of Victoria.

http://www.cs.ubc.ca/**formalWARE**

Michael R. Donat

The University of British Columbia

September 1998

Chapter 1

Introduction

This dissertation proposes a discipline of deriving test descriptions, which are called *test frames*, from system-level requirements specifications. The discipline includes a *nomenclature* which consists of a collection of well-defined names of specification components and test frame properties. The nomenclature can be used to objectively define the completeness of a set of test frames relative to the requirements. Definitions of completeness, called *coverage criteria*, can be used as a basis for automatically deriving test frames from a formal specification of requirements. The discipline supports repeatability and definability, which facilitate its use in software development processes aspiring to Capability Maturity Model (CMM) Levels 2 through 5. In this dissertation, a *formal specification* is a specification written in a language that can be algorithmically transformed into a set of mathematical logic formulae.

1.1 Objective

The objective of this research has been to provide a more scientific basis for system-level requirements-based testing in order to help transform this activity from a craft requiring considerable apprenticeship and experience, into an engineering discipline. A second objective has been the partial automation of this task to improve test frame quality and to reduce the time and effort required for derivation and review, thereby reducing the overall costs of system-level requirements-based testing.

1.2 Motivation

The focus of this thesis, system-level requirements-based testing, is an important part of the disciplined development of large, software-based systems for which a detailed set of requirements is specified. This type of functional, or black-box, testing typically appears in software development processes as portions of System-Level Testing, Acceptance Testing, and Independent Validation and Verification (IV&V). The objective of *system-level requirements-based testing* is to provide evidence that each behaviour specified in the requirements has been satisfied. Documentation that a system has passed each test step in a set of test procedures is commonly used as sufficient evidence. The test steps are instances of test frames, which satisfy a given coverage criteria. Test procedures are sequences of test steps. This thesis addresses the derivation of test frames. The derivation of test steps and other types of testing, which may address properties such as robustness, performance, and availability, are not within the scope of this thesis.

Two problems motivate this research. The first is lack of objective coverage criteria. This contributes to the second problem: lack of automated analysis tools for test frame derivation. The lack of automated analysis is due partly to the subjectivity of many existing guidelines for coverage criteria. These guidelines are interpreted by specialists who decide which tests are appropriate, and how many tests are required to satisfy the guidelines. This subjectivity can lead to different opinions of what constitutes satisfaction of the guidelines. Furthermore, the communication among individuals of coverage issues is difficult. This is due to the lack of a precise vocabulary, such as a nomenclature for expressing relationships between requirements and test specifications.

Some automated tools assist in bookkeeping tasks associated with system-level requirements-based testing. However, much of the analysis required for test derivation is currently done manually. Reviews of test frames are necessary in order to ensure that they are logically consistent with the requirements, and that the appropriate coverage has been achieved. The analysis is laborious, and the results are expensive to review.

As this thesis shows, these two problems are intimately related. Atomic components of test frames are referred to in this dissertation as *frame stimuli*. A fundamental concept of this thesis is that coverage criteria describe a relationship between the frame stimuli of test frames and the frame stimuli that appear within the original requirements.

1.3 Related Techniques

The related test derivation techniques can be categorized as systematic, code-based, and logic-based techniques. Systematic techniques have the advantages that they are relatively simple and typically evolve out of a manual approach. This evolution provides a good fit between the automated tools and the current process, reducing the costs for retraining. Unfortunately, systematic techniques lack the mathematical soundness required to ensure that transformations involved in test frame derivation do not compromise the meaning of the requirements. This is substantiated in Section 3.2.

Code-based testing, traditionally applied at the unit level, is well developed. However, the circumstances and objectives of this type of testing are fundamentally different from those of system-level requirements-based testing. One issue is that, in the context of system-level requirements-based testing, test frames should be structurally independent from the requirements, i.e., the wording of a requirement should be irrelevant provided that the appropriate meaning is conveyed. Another distinction is the importance of an expressive requirements language, such as one allowing universal and existential quantification, which does not appear in a code-based context. Further issues relevant to system-level requirements-based testing include dependencies between conditions within the requirements specification, and the need to minimize the impact of requirements changes on previously generated test frames.

While techniques based on mathematical logic can provide a sound and expressive basis, they do not yet support the combination of automation and expressiveness that would allow them to be applied effectively to system-level requirements-based testing. This is substantiated in Section 3.4.

1.4 Approach

Test generation techniques based on mathematical logic that appear in the literature are based on restricted languages. The philosophy of these techniques is that certain mathematical structures have the advantage that they support capabilities such as the generation of test data for test steps, i.e., instances of test frames, and sequences of test steps. In contrast, code-based test generation techniques produce what tests they can. Rather than imposing a specification language designed for test generation, code-based techniques are required to use program source code. However, due to the undecidability of loop invariants, there are situations where code-based techniques cannot generate test data.

The approach of this thesis is similar to code-based techniques in that as few restrictions as possible are placed on the content of a specification. However, this freedom for specification authors has a cost. Logics permitting undecidable[1] formulae are used in specification languages such as Z [61] and VDM-SL [37] because these logics are *expressive*, i.e., properties can be expressed precisely and concisely. This dissertation demonstrates that specifications based on expressive mathematical logics can be manipulated algorithmically in order to produce test frames. The cost of using logics permitting undecidable formulae is that the instantiation of test frames cannot be fully automated. As is the case with code-based techniques, the approach described in this dissertation may fail to produce test frames in certain situations. However, these situations are well-defined and can be identified within $O(n \log n)$ time, where n is a measure of the size of the given specification.

A nomenclature is an important part of a scientific discipline. Names within nomenclatures are often more than simple identifiers. They may also provide functional information about the objects they identify. As an example, a classification of languages can be achieved using the nomenclature from the Chomsky Hierarchy which names the minimal type of machine needed to recognize a sentence from the language. The nomenclature presented in this dissertation provides a basis for

[1] A formula is undecidable if a proof of the truth or falsehood of the formula cannot be determined by a mechanical procedure.

classifying test frame sets using coverage criteria that relate properties of a test frame set to individual behaviours which follow logically from the requirements. The nomenclature is structured so that definitions of coverage criteria are parameters to the algorithms of the discipline. Thus, coverage criteria define the automatic production of the corresponding sets of test frames. The nomenclature allows test engineers to communicate coverage criteria details more precisely than by using the terminology in current system-level requirements-based testing guidelines found in standards documents such as DO178B, DOD-STD-2167A, ANSI/IEEE 829-1983, and MIL-STD-498 [56, 60, 35, 15].

The algorithms presented in this dissertation are based on strategies similar to those of Dick and Faivre [17], but with significant differences. However, the process of system-level requirements-based testing is not fully automated by employing the algorithms of the discipline presented in this dissertation. The original requirements specification must be formalized before test frames can be generated. Furthermore, as stated in Section 1.2, the selection of test data to satisfy a test frame is beyond the scope of this thesis. This is due to the use of expressive specification languages.

1.5 Contributions

The thesis of this dissertation has two main ideas. First, objective criteria for test coverage can be defined by embedding system-level requirements in mathematical logic. Second, such criteria form a basis for the algorithmic translation from a formal requirements specification to test frames, which can be used by test engineers to produce test steps.

The following are the major contributions of this thesis.

- *A nomenclature for defining coverage criteria relative to requirements specifications.*

  The nomenclature is based on three fundamental entities: test classes, frame stimuli, and test frames. The term "test frame" is used by Ostrand and Balcer [46]. The terms "test class" and "frame stimuli" are introduced in this dissertation. A test class is a behaviour extracted from the requirements, and is based on the required response. Frame stimuli express the elementary conditions used to determine when a response is required. Test frames prescribe particular conjunctions of frame stimuli that require a particular response. A set of test frames is derived from each test class. The frame stimulus is the principal entity on which coverage criteria are defined. Coverage criteria relate test frames to test classes.

- *Algorithms for producing sets of test frames that satisfy coverage criteria defined in terms of the nomenclature.*

  The algorithms described in this dissertation produce test frames that have the following important properties:

  - *Conservative*

    Each test frame is a logical consequence of the requirements. The soundness of the rules of logic used in the algorithms of the discipline ensures that a test frame generator that correctly implements the algorithms will produce only test frames that are implied by the requirements.

  - *Tractable*

    Test engineers can control the automatic derivation of test frames, which allows them to exercise engineering judgement.

- *Complete*

  The set of test frames is produced in compliance with a specified coverage criterion that is an objective definition of completeness.

- *Traceable*

  It is possible to determine the original elements in the formal specification from which a selected test frame was derived.

- *Definitions of fundamental coverage criteria.*

  This thesis presents a template for detailed mathematical definitions of coverage criteria. The definitions of some fundamental coverage criteria are given as examples. It is likely that more elaborate coverage criteria will be defined according to the discipline of this thesis as it becomes more widely used.

Along with the above contributions, other results of this research are noted below.

Logical expressions can be partitioned based on whether or not test frames can be algorithmically produced from them. To allow maximum expressiveness, the input to the algorithms is allowed to be as arbitrary a predicate logic expression as possible. However, certain restrictions are necessary to ensure that test frame generation can be achieved algorithmically.

A specification language, Q, has been designed for specifying system-level requirements. This language contains features specifically designed to ease the task of formalizing a natural language specification. Q specifications tend to be as concise as natural language, but, of course, do not contain ambiguities.

Requirements changes are inevitable during the course of software development. The *Delta Problem* is the problem of integrating existing test frames into new test frame sets produced from the new version of the requirements. The Delta Problem can be defined mathematically. In general, the Delta Problem is undecidable, but partial solutions exist.

The value of the formal specification of system-level requirements is increased. By providing a means of automating the generation of test frames, this thesis allows more to be done with a formal specification of system-level requirements than simple type-checking. It also provides a less expensive means than theorem-proving for deriving other artifacts from a specification.

The novelty of this thesis is that it provides a firm mathematical foundation for the automation of system-level requirements-based testing. While there are many excellent works devoted to specification-based testing, they focus on techniques that are applied to unit-level specifications or to simple models of selected aspects of system-level requirements. These techniques are too restrictive to be successfully applied to the broad range of system-level requirements addressed by this thesis.

1.6 Outline

Chapters 2 through 4 establish the research problem and motivation for the discipline, while Chapters 5 through 7 form the core of the discipline and its application. Chapter 2 describes system-level requirements-based testing in greater detail, and examines the associated problems. Chapter 3 describes existing solutions. Chapter 4 identifies the issues of system-level requirements-based testing that distinguish this level of testing from others, such as code-based testing. Chapter 5 presents a discipline of test derivation that provides a nomenclature that can be used for defining coverage criteria. Chapter 5 represents the central contribution of this thesis. Chapter 6 defines coverage criteria using the nomenclature based on extensions of the algorithms from Chapter 5. This represents the fundamental application of the discipline. Chapter 7 examines an application of this discipline in its most general setting. Chapter 8 presents one possible application of this discipline to system-level requirements-based testing. Conclusions and future work are presented in Chapter 9.

Chapter 2

The Problem

The development of test frames is a key stage in system-level requirements-based testing. This chapter identifies a set of limitations and inefficiencies encountered during this stage of test derivation. A coverage criterion determines the "completeness" of a set of test frames with respect to the requirements specification from which they were derived. Subjective coverage standards and the manual derivation of system-level requirements-based testing incur significant expense. Contributing factors include: the labour required to analyze the requirements and derive the appropriate tests; reviews ensuring the quality, correctness, and completeness of the tests; and the impact of these activities on software development schedules. This thesis focuses on decreasing the subjectivity and increasing the accuracy of test frame derivation by providing a mathematical foundation for defining coverage criteria and calculating test frames from specifications. These capabilities have the potential to reduce the costs associated with system-level requirements-based testing.

2.1 Introduction

Defective software can be frustrating, expensive, and, in the worst case, life threatening. However, for complex software systems, it is rarely possible to perform enough tests to guarantee that running yet another test is fruitless. For this reason, it is necessary to define a milestone that signals the end of testing and the point at which the software can be installed in the field. This milestone is defined as the successful completion of a set of tests. For the milestone to be credible, this test set must satisfy some pre-agreed criterion. A central concept in testing, such criteria define test set completeness, inspire methods of test generation, and provide a medium for communicating issues relevant to ensuring that software is released in both a moral and fiscally prudent fashion.

The focus of this thesis is a discipline that can be applied to system-level requirements-based testing. This type of testing considers statements of the system's required behaviour in terms of stimuli and responses. Although the internal mechanisms that implement that behaviour are not considered, an abstract view of internal states of the system typically plays a role in test derivation.

Traditionally, tests are derived manually by analyzing the natural language of the requirements. Rules of thumb are used to determine the meaning of standardized phrases within the requirements. Test engineers use common sense transformations of these phrases to derive test frames. This derivation implicitly includes an understanding of a coverage criterion. However, the particular criterion and the derivation methodology depend on the skill and experience of the test engineers. A goal of this thesis is to standardize non-domain-specific aspects of test derivation methodology, thereby allowing the skill and experience of test engineers to be focused on domain specific issues.

Assurance that a set of test frames satisfies a given criterion is achieved through a review process. This review process relies on a mechanism known as *traceability*. The traceability of requirements to test frames allows reviewers to confirm that the test frames are consistent with the requirements, and that the set of test frames satisfies the given criterion.

Although the derivation of test frames from requirements incurs significant cost, there is a surprising lack of automation of the analysis required for this task. This is partly due to the difficulties of automatically processing natural language, but a more important obstacle is the subjectivity of test derivation guidelines such as those found in DO178B, DOD-STD-2167A, ANSI/IEEE 829-1983, and MIL-STD-498. Any solution providing a means of automation, will also provide objective definitions of criteria for sets of test frames. Additional characteristics of a solution include: controls enabling test engineers to exercise engineering judgement, traceability of requirements to test frames, and some degree of containment of the impact of requirements changes.

Several qualities of system-level requirements-based testing point towards a solution based on mathematical logic. The automation of logical transformations and the requirements for test frame correctness are both addressed by such a solution, and would also provide the required objectivity. The importance of objective criteria is a central point of this chapter.

The importance of testing and the need for criteria signaling its completion is given in Section 2.2. The significance of these general criteria is examined further in Section 2.3. Section 2.4 describes the application domain of this thesis, system-level requirements-based testing, while Section 2.5 details the process of manual test frame derivation. The notion of the traceability of requirements to test frames and its current relevance to ensuring completeness is given in Section 2.6. Section 2.7 examines the lack of a process for automatically deriving test frames.

Based on the limitations and inefficiencies presented in Sections 2.2 through 2.7, Section 2.8 details characteristics of a solution. Section 2.9 provides the impetus for a solution based on mathematical logic, and gives a scientific perspective of the essence of system-level requirements-based testing.

2.2 Testing

A simplified view of software development identifies four basic phases: requirements specification, design, implementation, and testing. Requirements specification establishes the required behaviour of the system. The design phase determines how these requirements will be achieved. Implementation is the building and assembling of components according to the design. One of the purposes of testing is to provide a degree of confidence that each of the required behaviours is exhibited by the implementation. Testing is essential to ensuring software quality, yet it is a task that can rarely be completed to the point where nothing can be gained by performing more testing. A substantial problem in testing is determining when enough testing has been performed to ensure the desired quality with fiscal efficiency.

In general, a substantial measure of professional and public confidence in the design and implementation of a critical system is based on the assumption that the system has been "completely tested." However, it is rarely practical to test every conceivable situation in which such a system must perform flawlessly. This is partly due to the immense size of the input domain that exists for a large system. To exhaustively test the inputs for even a simple program that implements a function of two 16-bit integers requires $(2^{16})^2 = 4,294,967,296$ tests.

Clearly, then, a non-trivial software system cannot be "completely tested" in the sense that every possible situation has been accounted for. The classic "divide and conquer" approach does not work. It is impractical to decompose a non-trivial system to a level of granularity that could be completely tested and then integrate the results for the whole system. Some criteria are needed to define an adequate set of tests to be used to determine when software can be installed in the field.

2.3 Coverage Criteria

A program's proximity to being "completely tested" is determined by the properties of the set of tests as a whole. Criteria describing desirable properties are used to construct a test set which is both small, and also satisfies the chosen criteria. The intention is that these *coverage criteria* lead to test sets which exercise a sample of the program's input domain that is likely to uncover faults, if they exist. The "completeness" of a test set is measured relative to a particular coverage criterion.

Use of the word coverage stems from the notion that the criterion implies a categorization of the input domain, and that any test set satisfying the coverage criteria covers, or exercises, one or more representatives of each of the categories. The categorization is not necessarily a partition, i.e., the categories are not required to be disjoint subsets of the input domain.

Conclusions about the performance of a system are generalized from the successful completion of a test set. The validity of these conclusions depends on the coverage criteria satisfied by the test set.

There are several different types of coverage criteria, which correspond to different types of testing. Each type of organized testing focuses on a different objective and a different abstract view of the software. For example, unit testing focuses on demonstrating the correctness and robustness of individual components of the system. From this testing, conclusions may be drawn regarding each component in isolation. This type of testing supports conclusions about system components, but general conclusions cannot be drawn regarding the operation of the system as a whole, or how well the system meets the original requirements specification. Other coverage criteria focus on these latter concerns.

Coverage criteria serve a dual purpose: 1) as a definition of completeness to guide the construction of test sets and evaluate their completeness, and 2) as a description to others of the degree to which a program has been tested.

2.4 System-Level Requirements-Based Testing

This thesis addresses software development processes similar to those outlined in software system development documents such as DO178B, DOD-STD-2167A, ANSI/-IEEE 829-1983, and MIL-STD-498. In these processes, *system-level, requirements-based testing* refers to a particular level of software testing with the goal of verifying through demonstration that each of the requirements within the specification has been satisfied. This can be only a partial verification, due to the sizes of the input and state spaces. Such a demonstration assists in signaling the completion of the development cycle. In some software development processes, this demonstration is necessary for the legal completion of a contract between a customer and a software manufacturer.

One method of achieving this demonstration is by performing a number of tests. Each test is defined by a test procedure. Each test procedure is a sequence of test steps. Each test step contributes to the demonstration that a specified requirement has indeed been satisfied. Each test step involves the application of a stimulus to the software system, and a comparison of the actual response of the system with the expected response specified by the requirements.

This level of testing is "system-level" in the sense that the internal structure of the system is not visible; all testing must be performed by the application of externally generated stimuli and the observation of externally visible responses. It is "requirements-based" in contrast to other kinds of system-level testing which may, for instance, be based on scenarios intended to approximate the expected use of the system for such purposes as determining system performance or reliability.

This thesis is oriented to a very general style of requirements specification in which requirements are expressed by statements that express relationships between externally generated stimuli and externally visible responses. The requirements may also contain references to an abstract representation of the internal state in the form of pre-conditions and post-conditions. This style of requirements specification strongly discourages the description of internal processing. It is distinguishable from "model-oriented" approaches which involve the presentation of an abstract model as a means of describing the desired functionality of a system. In particular, the style of specification addressed by this thesis is characterized by logically complex statements of behaviour, relating system stimuli and responses rather than stating simple transitions amongst a complex network of states.

In a typical large system, each test procedure serves as a script for a test session that would typically require no more than several hours of effort to execute. However, many months of effort may be required to develop the test procedure. The manual development of a test procedure by a test engineer can be described in terms of two main phases.

The first phase decomposes requirement statements into a set of test frames.

This involves lexical analysis of the syntactic structure of the requirements statements guided by key words and phrases such as "and," "or," "not," "if," "unless," "whenever," "provided that," "on the condition that," and "except if one of the following conditions is true."

The second phase of a typical test procedure development process is to organize the test frames into sequences. The sequences must be arranged in order to ensure that the pre-conditions of each test frame are satisfied by the preceding sequence of test frames. The pre-conditions and post-conditions may be assertions about the internal state of the software system, or they may be assertions about parameters of the stimuli or responses.

This second phase of developing a test procedure also involves the instantiation of sequenced test frames into test steps in a test procedure by replacing data references, e.g., "the current altitude of the aircraft," with actual values, e.g., "10,000 feet." The instantiation of test frames during this second phase may involve the use of techniques such as Boundary Analysis and Equivalence Partitioning [45] to ensure that a suitable sample of actual values is used in the test procedure. A test step, an instantiated test frame, is often referred to in the literature as a test case. [1]

As stated in Section 1.2, the scope of this thesis is limited to the first of the two phases described above: the decomposition of requirements into a set of test frames. The second phase of this process, both the instantiation of test frames with actual data values and the ordering of test steps, is outside the scope of this dissertation.

[1] Other authors also use "test case" to refer to an entire test procedure. Due to the multiple meanings of this term, it is excluded from the vocabulary of this dissertation to avoid misinterpretations.

2.5 Manual Test Frame Derivation

Experienced test engineers use "rules of thumb" to decompose requirement statements into test frames. For example, the presence of the key word "or" in the antecedent of a requirement of the form,

> When Stimulus S occurs and Condition C1 or Condition C2 is true, then the system shall produce Response R

indicates that the requirement must be decomposed into at least two separate test frames - one for when Condition C1 is true, and another, separate test frame for when Condition C2 is true. This would yield a pair of test frames,

1. S and C1 and (not C2) $\Rightarrow$ R, and
2. S and (not C1) and C2 $\Rightarrow$ R,

where the symbol "$\Rightarrow$" separates the stimulus part (both the externally generated stimulus and the pre-conditions) from the response part (both the externally visible response and the post-conditions). This symbol may be read informally as "yields" or "results in." Depending on the coverage criterion used by the test engineers, additional test frames may also be generated to test for situations when the response R should not be produced, i.e., "$\Rightarrow$ not R."

It is often necessary to combine requirement statements to generate test frames. For instance, a statement of the form,

> Unless Conditions C3 and C4 are both true, the system shall also produce Response R1 whenever Response R2 is produced

needs to be paired with another statement such as, When Stimulus S occurs, then the system shall produce Response R2

to obtain an "end-to-end" stimulus-response relationship between Stimulus S and Response R1. The combination of these two statements can then be decomposed into a set of test frames.

While performing this task, test engineers manually apply rules of logical reasoning such as DeMorgan's Laws, e.g.,

```
not (A and B) = (not A) or (not B).
```

This is illustrated by the above example, which would likely involve substituting (perhaps just mentally) the phrase "unless Conditions C3 and C4 are both true" with the logically equivalent phrase "if Condition C3 is false or Condition C4 is false." The "or" in the result of this substitution could then be used to split this requirement into two test frames. Another example is the substitution of the phrase "whenever Response R2 is produced" with the phrase "when stimulus S occurs," using a rule of logical reasoning sometimes called "pre-condition strengthening." Test engineers may not be aware of the fact that they are using DeMorgan's Laws or "pre-condition strengthening," but, reassuringly, there is a correspondence between engineering intuition and formal logic.

Thus, the decomposition of requirements into test frames can be viewed as a series of lexical transformations based on rules of logical reasoning. In general, the resulting test frames are logically implied by the requirements. This makes sense from a practical engineering point of view. Obviously, it would be undesirable to test for stimulus-response relationships not implied by the requirements.

The work performed by a test engineer during this first phase is not entirely a matter of routine logical deduction. Much effort is typically spent "disambiguating" natural language in order to expose the logical structure of the requirements statements. Other considerations, such as domain knowledge, also contribute to this process. Knowledge of the application domain is needed to understand dependencies between various conditions referenced in the requirements, and to avoid the generation of impractical or infeasible combinations of conditions in test frames. For example, the conditions "is airborne" and "has landed" may appear together as conditions in a test frame which is logically derivable from the requirements for an air traffic control system – but which would be rejected by a test engineer on the basis that it is infeasible. Nevertheless, reasoning about stimulus-response relationships in a systematic manner is a central part of this task.

The effectiveness of these conventions is highly subject to the discipline of requirements authors in avoiding words or phrases which may be ambiguous or have shown a tendency to be misinterpreted. For example, experience shows that a requirement of the form,

> When Stimulus S2 occurs and Condition C3 is true, then the system shall produce Response R2 unless Condition C4 is false

is not necessarily ambiguous, but it is more likely to be misinterpreted than the following, logically equivalent, statement of this requirement:

> When Stimulus S2 occurs and Conditions C3 and C4 are both true, then the system shall produce Response R2.

The task of systematically deriving test frames becomes more complex when the interpretation of a particular requirement depends on other requirements. For example, the interpretation of the requirement,

> When Response R3 is produced and Condition C5 is true, then also produce Response R4

depends on the set of requirements which specify conditions under which Response R3 will be produced. When interpreting this requirement for the purpose of deriving test frames, one possibility is to lexically replace the phrase "When Response R3 is produced" with one of the possible conditions under which Response R3 will be produced. Another possibility is to lexically replace this phrase by the logical disjunction of all of the possible conditions under which Response R3 will be produced. For example, suppose that the conditions for producing Response R3 are expressed by the following two requirements:

> When Stimulus S4 occurs, then produce Response R3.

> When Stimulus S5 occurs, then produce Response R3.

With the first approach, lexical replacement of the phrase "When Response R3 is produced" will yield a re-statement of the original requirement in a form,

> When Stimulus S4 occurs, then also produce Response R4 if Condition C5 is true.

which would then be decomposed into a single test frame. This is different from the result of following the second approach,

> When Stimulus S4 or Stimulus S5 occurs, then also produce Response R4 if Condition C5 is true,

which would be decomposed into two distinct test frames because of the introduction of the word "or" into the text of the requirement.

Yet another source of complexity in the process of deriving test frames from requirements is illustrated by the following example requirement:

> When Stimulus S6 occurs, and ((Condition C6 is true or Condition C7 is true) and (Condition C8 is true or Condition C9 is true)), then produce Response R5.

In this example, parentheses are used to unambiguously state the requirement by clarifying the nesting of the logical connectives, "or" and "and." As an alternative to parentheses, a decision table or an itemized list of conditions may be more readable. However, the formatting style of a requirements specification is beyond the scope of this thesis.

The nesting of disjunctions, i.e., phrases containing the word "or", within a conjunction, i.e., the phrase containing the word "and", is the source of a fundamental choice of coverage in the methodology used to systematically derive test frames from a set of requirements. For the above example, this choice is a matter of deciding which subset of the following test frames are necessary to verify the above requirement:

Frame	Stimulus	Conditions	Response
1	S6	C6, C7, C8, C9	R5
2	S6	C6, C7, C8, not C9	R5
3	S6	C6, C7, not C8, C9	R5
4	S6	C6, not C7, C8, C9	R5
5	S6	not C6, C7, C8, C9	R5
6	S6	C6, not C7, C8, not C9	R5
7	S6	C6, not C7, not C8, C9	R5
8	S6	not C6, C7, C8, not C9	R5
9	S6	not C6, C7, not C8, C9	R5
10	S6	C6, C8	R5
11	S6	C7, C9	R5
12	S6	C6, C9	R5
13	S6	C7, C8	R5

Several notions of completeness are possible. Those discussed here are given names and definitions in Chapters 5 and 6. Under one notion of completeness, only the first nine test frames are necessary to claim that the verification set for the requirement is complete. Under another notion of completeness, test frames 10-13 are sufficient. The difference between these two test frame sets is the amount of detail specified in the test frames. Under yet another notion of completeness, it would be possible to reduce the verification set to just test frames 6-9. Under yet another notion of completeness, it would be possible to reduce the verification set for this requirement to just test frames 6 and 9, or, alternatively, just test frames 7 and 8. Hence, depending on the notion of completeness used in the methodology, the minimal size of the verification set for this requirement would be nine, four or two test frames. Different situations may favour a larger number of test frames or a smaller one. However, it is clear that these notions of completeness need to be distinguished in a standard way, and referred to using standard names.

The illustrative examples given above are simplistic in the sense that they amount to relatively small differences in the number of test frames required to completely verify a requirement. However, differences in the coverage criteria used to derive test frames from requirements, when applied to large, complex specifications of requirements, have potentially large differences with respect to the number of test frames required to satisfy a particular form of coverage.

Perhaps more importantly, the examples given above suggest that the derivation of test frames from a set of requirements is not necessarily a routine process that always leads to the same result independently of the skill and experience of the individuals performing the work. Some skill and experience will always be required to perform this task. However, this thesis is motivated by a desire to focus test engineer skill and experience on less tedious aspects of the task. In addition to improving the process by reducing the number of corrections that need to be made during test procedure reviews, the precise description of coverage criteria for system-level requirements-based testing provides the basis for the development of software tools to partially automate the derivation of test frames from requirements.

The extraction of test frames from a requirements specification requires a great deal of manual effort. The volume of the requirements, and the complexity that can be present through the use of decisions within the text referring to several conditions and negating such decisions, make this task tedious, routine, and error prone. Thus, additional effort must be spent in reviews to ensure that the set of test frames satisfy certain properties. Re-working test procedures as a result of specification changes is costly not only due to the effort involved, but also due to the impact on schedules.

The act of producing test frames often uncovers anomalies in the specification. However, the loose connection between specification authoring and test planning causes this feedback to be delayed until late in the authoring stages.

2.6 Coverage via Traceability

In a disciplined approach to requirements-based testing, the "completeness" of a set of test procedures is determined by inspecting the relationship between the requirements specification and the contents of steps in test procedures. A traceability mapping from requirements to individual test steps is used to demonstrate that the set of test procedures is "complete" in the sense that every functional requirement can be traced to an appropriate set of distinct steps in a test procedure [16]. The size of this appropriate set is determined by the number of choices within the requirement.

Typically, each requirement has a unique identifier, and each step in a test procedure is annotated with a list of requirement identifiers. Confirmation of the implementation of a requirement is demonstrated upon successful completion of all steps associated with that requirement. The requirement identifiers provide a method of maintaining this association.

The completeness of a set of test procedures can be determined automatically by a software tool that parses out requirement identifiers listed in the test procedure and compares this set of identifiers against a complete set of all requirement identifiers. The test set is not complete until every functional requirement has been mapped to an appropriate number of specific test steps. In this case, coverage refers to coverage of the requirements.

Traceability is necessary for providing an audit trail to support process monitoring as well as assisting in evaluating completeness. Traceability between requirements and tests also assists in determining the scope of test set changes required when requirements changes occur by providing an index that can be used to facilitate the appropriate review tasks.

However, traceability is only a partial solution to determining the completeness of a test set. This type of tool assumes that the requirement identifiers attached to the test steps are correct. More importantly, it is also assumed that the appropriate combination of tests that refer to any particular requirement has been produced. The reality of human error necessitates the use of reviews to ensure that these requirement identifiers are correct and that a suitable number of test steps has been produced for each requirement.

2.7 Lack of Automation

It is possible that the proprietary state-of-the-art is more advanced than the documents quoted below. However, the quoted documents represent the published sources of coverage criteria upon which industry standards could be based.

Requirements-based testing guidelines contained in documents such as DO178B, DOD-STD-2167A[2], ANSI/IEEE 829-1983, and MIL-STD-498 do not contain enough detail to objectively define algorithms for deriving test frames in the context of logically complex requirements specifications. Of these documents, DO178B gives the most detailed description. Paragraph 6.4.4.1(a) states:

[2] Although superseded by MIL-STD-498, some software development projects still use DOD-STD-2167A

Test cases exist for each software requirement.

Figure 2.1 is an image of Table A-7 from DO178B, and indicates the differences in the amount of detail given between requirements-based test coverage and code-based test coverage: The descriptions for code-based coverage criteria given in Rows 5 through 8 refer to specific, objective definitions. By comparison, the description of coverage in Row 3, for requirements-based testing, is not defined.

Table A-7

Verification Of Verification Process Results

	Objective		Applicability by SW Level				Output		Control Category by SW level			
	Description	Ref.	A	B	C	D	Description	Ref.	A	B	C	D
1	Test procedures are correct.	6.3.6b	●	○	○		Software Verification Cases and Procedures	11.13	②	②	②	
2	Test results are correct and discrepancies explained.	6.3.6c	●	○	○		Software Verification Results	11.14	②	②	②	
3	Test coverage of high-level requirements is achieved.	6.4.4.1	●	○	○	○	Software Verification Results	11.14	②	②	②	②
4	Test coverage of low-level requirements is achieved.	6.4.4.1	●	○	○		Software Verification Results	11.14	②	②	②	
5	Test coverage of software structure (modified condition/decision) is achieved.	6.4.4.2	●				Software Verification Results	11.14	②			
6	Test coverage of software structure (decision coverage) is achieved.	6.4.4.2a 6.4.4.2b	●	●			Software Verification Results	11.14	②	②		
7	Test coverage of software structure (statement coverage) is achieved.	6.4.4.2a 6.4.4.2b	●	●	○		Software Verification Results	11.14	②	②	②	
8	Test coverage of software structure (data coupling and control coupling) is achieved.	6.4.4.2c	●	●	○		Software Verification Results	11.14	②	②	②	

LEGEND:
- ● The objective should be satisfied with independence.
- ○ The objective should be satisfied.
- Blank Satisfaction of objective is at applicant's discretion.
- ① Data satisfies the objectives of Control Category 1 (CC1).
- ② Data satisfies the objectives of Control Category 2 (CC2).

Figure 2.1: Table A-7 from DO178B

In paragraph 4.3.4 of DOD-STD-2167A, coverage is described by referring to traceability:

> The contractor shall document the traceability of the requirements in the Software Requirements Specifications (SRSs) and Interface Requirements Specification (IRS) that are satisfied or partially satisfied by each test case identified in the Software Test Description (STD).

Paragraph 5.5 of MIL-STD-498, the document which supersedes DOD-STD-2167A, specifies only that the coverage criterion should be documented:

> 5.5 <u>Software requirements analysis</u>. The developer shall define and record the software requirements to be met by each CSCI [Computer Software Configuration Item], the methods to be used to ensure that each requirement has been met, and the traceability between the CSCI requirements and system requirements. The result shall include all applicable items in the Software Requirements Specification (SRS).

ANSI/IEEE 829-1983 is the IEEE standard for software test documentation. Its contribution is similar to that of MIL-STD-498, as it states that the approach must be documented but does not specify a collection of possible approaches for system-level requirements:

> 3.2.6 ...Identify the techniques which will be used to judge the comprehensiveness of the testing effort...

In addition to a lack of detail in the above guidelines, the definitions of coverage criteria are determined by the subjective interpretation of these documents. This subjective interpretation is the responsibility of experienced test engineers.

To prevent this interpretation from becoming too *ad hoc* or unsystematic, well established techniques impose discipline and formality, e.g., the use of requirement identifiers for tracing requirements to test steps. While established techniques provide a disciplined approach to interpreting standard guidelines, certain problems arise due to the lack of an objective definition of coverage.

As described in Section 2.5, to derive test frames from a set of requirements, a test engineer is required to break up possibly logically complex requirements into a set of atomic stimulus-response relationships. Addressing logical complexity to the degree necessary to derive an appropriate set of system-level test frames has not been addressed in software development literature. The guidelines mention data selection concepts such as “average,” “boundary,” and “out-of-bounds” values. They add only that there should exist a test for each software requirement, and that different combinations of operations should be exercised. Unfortunately, none of this describes the analysis of logical complexity, nor does it define the level of detail that should be reflected in the test steps produced.

While documents such as DO178B, DOD-STD-2167A, ANSI/IEEE 829-1983, and MIL-STD-498 provide some general guidelines for requirements-based testing, they do not provide the specific detail required to objectively decide if a particular test frame is missing from the test set, or if a particular element of the test set is redundant. Different experiences amongst senior test engineers, combined with subjective guidelines, almost guarantee that disagreements will arise. Resolutions of these disagreements can only be arbitrary, and are in danger of being inconsistent. Therefore, the lack of an objective definition becomes a management issue because it places too much dependence on engineering judgment and experience.

Communications with the customer regarding the thoroughness of the testing performed are sometimes in terms of statistics based on the amount of resources spent, rather than an objective account of the coverage achieved. This forces the customer to place a great deal of faith in the developer, or to incur additional expense to review the test process in order to become familiar with the level of testing being applied to their product.

The lack of objective definitions of coverage has resulted in a lack of automated tools for the analysis of requirements. There are commercial tools for deriving tests from executable models of requirements. The disadvantages of executable models are presented in Section 3.4. The discipline presented in this dissertation advocates the use of formal translation rather than modelling.

2.8 Towards a Solution

Any solution to the limitations and inefficiencies presented in this chapter should exhibit the following characteristics:

1. A means of defining objective coverage criteria is provided.

2. Test frame derivation is, at least partially, automated.

3. Test engineers can control the automated portions of test frame generation in order to exercise engineering judgment when necessary.

4. Traceability is supported.

5. The impact of requirements changes on previously derived test frames can be contained, to some degree. This capability requires analysis beyond traceability.

Characteristic 1), objective definitions of coverage criteria, is the most important. The existence of such definitions would:

(a) eliminate the subjectivity of current coverage guidelines upon which disagreements of interpretation are based;

(b) reduce the impact of experience on the performance of test engineers, allowing junior test engineers to perform more like senior test engineers earlier;

(c) allow completeness to be measured objectively and, perhaps, algorithmically;

(d) provide a means of partially automating the construction of test sets; and

(e) allow communications to the customer to be based on progress relative to pre-agreed coverage criteria, thus reflecting actual achievement.

Furthermore, objective definitions of coverage, standardized across the software development industry, would provide a clearer picture of the degree to which products had been tested.

Characteristic 2) could potentially reduce testing costs and increase the consistency and accuracy of the tests produced. Characteristic 3) is essential to any software development process applied to a non-trivial project. Test engineers must be able to control the test frame generation process in order to deal with special circumstances which may arise. It is also important that the test frame generator be able to build test sets around test engineer guidance, rather than simply tolerating it. Characteristic 4) provides an audit trail for various purposes. Traceability not only provides a means of reviewing the test frame sets produced by the test frame generator, but also provides a means of tracking down errors in the requirements flagged by an incorrect test frame. Characteristic 5) is important for reducing costs, but will be limited since the underlying logic allows undecidable formulae in order to be expressive.

2.9 Motivation for a Mathematical Approach

Mathematics provides the means of achieving the accuracy required to accomplish great feats. As long as 4,500 years ago, Khufu's Great Pyramid at Giza was constructed so accurately that the perimeter of the base divided by twice the height is equal to Pi to 5 decimal places. The Pont du Gard, built before the first century A.D., is an architectural masterpiece that belies the accuracy of the aqueduct it supports. This Roman aqueduct had the capacity to deliver 120,000 m^3 of water per day along a 50 km run and a mere 17 metre drop (34 cm/km) into a "castellum divisorium" with the capacity to distribute 125,000 m^3 per day. Today, mathematics provides the means of piloting spacecraft on gravity-assisted trajectories taking them close to the inner planets to steal momentum in order to hasten the spacecraft towards Jupiter and Saturn.

In the field of computer science, the construction of parsers for higher level programming languages in the late 1950's and early 1960's was a craft, not a science. It was not until Noam Chomsky proposed a mathematical hierarchy of languages, originally for classifying natural languages, that parsing became well understood. In addition, the Chomsky Hierarchy has had a profound influence on the syntactic structure of modern programming languages. This is one example of the use of mathematics to change a labour-intensive, error-prone process, e.g., parser construction, into one that is automatic and flawless.

It is reasonable to expect a similar benefit by applying a mathematical solution to automating test frame generation. The logical complexity within requirements specifications forces test engineers to perform logical reasoning (informally) during their manual derivation of test frames. This reasoning process is an excellent candidate for the application of mathematical logic.

These considerations motivate the use of rules of mathematical logic to manipulate a formal expression of the requirements specification for the purpose of calculating test frames. The idea of using logic as a medium for calculation is not new, e.g., Prolog [4] and Binary Decision Diagrams (BDDs) [8]. Using a set of mathematically sound rules guarantees that the algorithmically derived test frames are logical consequences of the specification. It is possible that an incorrect implementation of these algorithms may introduce errors into the test frames derived. However, the centralization of expression manipulation employed in theorem provers such as HOL [28], PVS [57], and Isabelle [50, 48, 47], can provide a high degree of confidence that the risk of such errors is negligible.

A scientific perspective of system-level requirements-based testing can be expressed by the following questions:

- *How can test frames be derived from an arbitrary formula expressing a relationship between stimuli and responses?*

  Currently, requirements specification authors are free to express the requirements in any way they find appropriate to accurately and efficiently convey the meaning of a requirement. Although a mathematical logic approach will impose some restrictions on authors' styles, this must be minimized in order to make the implementation of such an approach practical in an industrial setting. To achieve this, any algorithm must assume that the input format of the formal version of the requirements is as general as possible.

- *What constitutes a test frame?*

  It is likely that a test frame will consist of a list of stimuli and a list of expected responses. However, to satisfy authors' needs for expressiveness, it is likely that the underlying logic will allow quantification. The effect of quantifiers on these lists of stimuli and responses must be understood. Also, it is likely that test engineers will require some control of the amount of detail contained in the test frames.

- *What restrictions on the stimulus-response formula are necessary in order to allow automatic processing?*

  Since the elements of the logic that allow undecidable formulae cannot be removed without reducing expressiveness, it is likely that there must be some other restrictions on the use of these elements in order to allow for an algorithmic transformation from a definable class of input specifications to test frames.

- *How can the relationship between test frames and the original stimulus-response formula be described?*

  Some means of relating test frames to the stimulus-response formula supplied by the requirements authors must be possible in order to provide some measure of completeness.

A mathematical logic foundation for the definition of coverage criteria for system-level requirements-based testing should provide a depth of understanding similar to that of language syntax. Mathematics is both a method of definition and a means of calculation. Both of these aspects are present in the goals of defining objective coverage criteria and automating test frame generation. The nomenclature of coverage-criteria definitions is motivated by the goal of calculating test frames from specifications.

Chapter 3

Existing Solutions

This chapter presents existing solutions which might be applied to system-level requirements-based testing. Although each approach has advantages in deriving different types of tests, certain shortcomings remain. Examining these techniques introduces the background for underlying issues which are examined further in Chapter 4.

3.1 Introduction

The techniques examined in this chapter can be categorized as systematic, code-based, and logic-based. Some of the techniques referenced in this chapter produce test frames, while others produce test steps and test procedures. In this dissertation, when it is not necessary to distinguish the differences between these products, they are referred to simply as tests.

Perhaps the most obvious approach to automating the analysis aspects of system-level requirements-based testing is to simply define standard phrases and the systematic transformation of these phrases into test frames. Systematic approaches have the potential to be successful within the environment for which they were developed. However, the fact that they are not based on a mathematical foundation will hamper their applicability in a general setting.

Discrepancies between the circumstances and objectives of unit-level testing and those of system-level testing lead to the conclusion that system-level requirements-based testing and code-based testing are fundamentally different. Thus, although code-based testing is well understood, it does not provide a direct solution for system-level requirements-based testing. This is examined further in Section 3.3.

Techniques based on mathematical logic solve some of the problems of the systematic and code-based techniques. However, the primary difficulty with current techniques based on mathematical logic is the lack of a combination of both automation and the expressiveness to specify *what* is required without specifying *how* it is achieved.

Section 3.2 examines the systematic approach of using mechanized transformations to produce test frames from requirements. The possibility of exploiting the success of code-based techniques from unit-level testing is explored in Section 3.3. Section 3.4 examines current approaches based on mathematical logic.

3.2 Systematic Approaches

One possible approach to automating the derivation of test frames from system-level requirements is to restrict the requirements language to a standard set of phrase styles. This avoids the problems associated with parsing the ambiguities of natural language. The information within the restricted phrases could then be extracted by a parser designed for this language, and rearranged into test frames by a set of standardized transformations. The distinction between this approach and those described in Section 3.4 is that there is no mathematical basis for the soundness of these transformations. This means that situations may exist where applications of these transformations do not preserve the true meaning of a specification. Therefore, reviews are required to ensure test frame quality.

This approach would be finely tuned to the process for which it was designed, and would probably be reasonably successful. However, there are fundamental limitations to this approach. The first is that this would be a solution only for specifications that can be written using the particular set of phrases. There is also no guarantee that a subsequent specification author would not use the specification phrases in an unforeseen manner. Thus, improvements to the specification style would require changes to the test frame derivation algorithms.

A second limitation is that there is no well-founded assurance of test frame correctness. Therefore, it would be necessary to maintain a strict review process in order to monitor the test frames produced by this approach. This task may prove to be more difficult than in the manual approach of Section 2.5. For example, if an incorrect test frame is produced, there are two possibilities: 1) the error occurred in the derivation, or 2) there is an error in the specification. Since the test frame was produced automatically, there is no test engineer to justify the derivation of the test frame, as there would be in the manual process.

A further limitation is that coverage criteria defined in terms of this approach would be based on one set of phrases. Coverage criteria based on another set of phrases could potentially refer to fundamentally different entities, thereby creating confusion.

This approach might be successful in mechanizing a current process, such as the one described in Section 2.5, but would not be able to assist in improving and evolving that process beyond mechanization. This is due to the lack of a sound basis from which general conclusions about the process can be made. Such generalizations are critical for process improvement.

It is likely that examples of these techniques exist in industry. However, it is unlikely that published accounts are available for two reasons. First, they may be too specific to be of general interest. Second, they may be reguarded as a proprietary advantage.

3.3 Code-based Testing

Coverage criteria for unit and module testing are well known. These types of testing fit into a category referred to as code-based testing. Objective code-based coverage criteria are founded on a nomenclature provided by the inherent precision of code. This advantage is absent from the type of system-level requirements-based testing addressed by this thesis. This section describes code-based techniques, but also illustrates the importance of a nomenclature for defining coverage criteria.

Unfortunately, code-based techniques cannot be directly applied to system-level requirements-based testing because of differences between these levels of testing. These differences are examined in greater detail in Chapter 4.

3.3.1 Principles

Code-based testing techniques attempt to find faults in an implementation by using tests constructed from information extracted primarily from details within the program source code itself, or from design specifications of system components. These tests provide a means of evaluating the implementation components. Tests based on component design specifications are referred to as black-box, or functional, tests, while those based on source code are referred to as glass-box,[1] or structural, tests.

One class of glass-box testing derives its tests from branch and loop structures within the code. Attributes of these code structures are used to construct tests. For branches, tests are constructed to expose the difference between the true and false cases. The two tests distinguish between two different execution paths through the code. Similar tests are derived from loops. A typical set of tests for a loop will result in zero, one, and some number of iterations of the loop that reflects a "typical" use of the component.

The existence of execution paths leads to the notion of code coverage. A test set satisfying code coverage exercises each program statement at least once. A more rigorous code-based coverage criterion is path coverage in which each feasible path is executed at least once. Path coverage is rarely achieved for non-trivial components due to the large number of tests required and the fact that some paths, although feasible, may prove extremely difficult to reach due to the sequence of stimuli required.

Code-based coverage criteria need not refer only to execution paths within a program, but can refer also to equivalence classes in the input domain, the assignment and use of variables, or various other aspects of the implementation. Beizer [5] mentions over a dozen types of coverage. Code-based coverage criteria are founded on a nomenclature which is standardized by the constructs of programming languages. Programs contain a number of useful artifacts, such as branches, loops, variables, blocks, and interfaces. Coverage criteria are defined in terms of these artifacts, which are common to all programming languages.

[1] Also referred to as clear-box or white-box testing.

3.3.2 An Objective Criterion

```
Block A
if (x > 3)
   Block B
else
   Block C
   if (y > 10)
      Block D
```

Figure 3.1: Example Program

The example program in Figure 3.1 illustrates code-based testing principles and the value of an objective coverage criterion. Such a criterion, Block Coverage, is defined, simply for the purposes of this example, as:

> There exists at least one test which exercises each block of code in the program.

A block is defined as a sequence of statements containing no branches. Code-based tests can be described by a set of input settings. The test set $\{\{x = 4\}, \{x = 2, y = 11\}\}$ satisfies the example Block Coverage criterion. The test $\{x = 4\}$ exercises blocks `A` and `B` while $\{x = 2, y = 11\}$ exercises blocks `A, C` and `D`. This test set is complete because it satisfies the stated criterion. It can also be shown to be a minimal test set, since removing either of the tests results in a test set that fails to satisfy the example Block Coverage criterion.

This example provides an opportunity to show the value of an objective coverage criterion. The precision of a well defined coverage criterion promotes test team agreement of the level of confidence attained by unit and module testing.

A member of a test team might argue that the test set in this example does not adequately test this program, and that the addition of $\{x = 2, y = 5\}$ would result in an adequate test set. This is an argument that the chosen coverage criterion is inadequate, not that the test set was produced incorrectly. The team member is expressing their concern that perhaps a higher level of confidence is required for this piece of software. This concern should be addressed and a decision made as to whether a different coverage criterion should be chosen. The precision of the coverage criterion focuses discussion on the issue of determining the proper trade-off between confidence and resources. There can be no argument about the completeness of the test set, since the definition of the coverage criterion provides a simple means of evaluation.

The nomenclature provided by the precise structure of programming languages makes it possible to determine the completeness of a test set objectively. The nomenclature also provides a forum for defining and evaluating different coverage criteria. This activity also supports confidence in software by leveraging the confidence in well established code-based coverage criteria. These standards provide sufficient precision that the test steps can be derived automatically, or at least in a much more rigorous and systematic way than system-level requirements-based testing.

3.3.3 Automation

Several techniques exist for deriving some tests from code automatically. Loops provide the biggest obstacle to fully automatic code-based test generation, due to the undecidability of loop invariants. Chilenski and Newcomb's Ada Testing Workbench (ATW) [12] generates test specifications from an Ada subset and conducts coverage analysis for 21 structural coverage criteria. ATW employs a theorem prover to eliminate infeasible paths. It uses abstract syntax trees to extract structural elements from the Ada code, but does not construct test specifications for code with loops. Ferguson and Korel [24] describe a chaining approach that uses data dependencies within code to generate test data. The technique can produce test data for some simple types of loops. Voas, Payne, and Miller [65] use a simplified form of mutation testing to automate the generation of unit level tests for coverage criteria mentioned in DO178B.

Code-based techniques focus on a distinctly different level of testing from system-level requirements-based testing. One distinction between these levels of testing is that quantification appears in system-level requirements. Another distinction is that code-based tests are tightly coupled to the structure of the code from which they were derived. This means that changes in the code are likely to cause changes in the test set. This is quite desirable for unit-level testing, since the structure of the code is closely related to the machine code which is fundamental to system behaviour. Any change in the machine code warrants re-testing at the unit level. However, this is not the case for changes to system-level requirements. A change in requirements does not necessarily require re-testing. Chapter 4 addresses these differences in more detail.

3.4 Logic-Based Techniques

A number of test generation techniques make use of various types of mathematical logic specifications. These techniques have two significant advantages. The first is that they are based on logical systems that have been mathematically proven to be sound. This ensures that derivations correctly maintain the meaning of the specification. The second advantage is that many of these logics are more expressive than program source code.

Here, an *executable language* is a specification language combined with a definition of state such that transitions between states are decidable. For specifications built on primitives within an executable language, the primary advantage is that the resulting specifications can be simulated at some level of detail. Simulation allows requirements authors to interact with their specifications in order to validate that the specification implies what the author intended. Some of these techniques also provide partial code generation.

Formal specifications based on mathematical semantics provide a basis for automatic test-generation techniques. This mathematical structure allows formal specifications to be manipulated mechanically so that information contained within the specification can be isolated, transformed, assembled, and repackaged. Using rules of transformation in this manner, test frames for a system can be derived from its formal specification. If the specification is executable, test steps can also be derived. The soundness of the transformation rules and the mathematical semantics of the specification language guarantee that the tests are logical consequences of the specification. This provides a high degree of assurance in the correctness of the tests produced by test generators based on these techniques.

3.4.1 Finite State Machines

Richardson's work [22, 10, 64, 54, 53, 52] is based on specifications that are executable models. The advantages of this approach include the following:

1. the specification provides test oracles allowing testing to be fully automated once the executable model has been constructed, and

2. the specification can be simulated.

Some of the disadvantages of this approach are:

1. that test oracles assume that the portions of the specification to be tested actually terminate, and

2. that it might be quite costly to produce an executable model from a given specification that contains a similar level of detail as the original specification.

Eickelmann and Richardson's evaluation of software test environment architectures [10] takes the position that testing should be fully automated. However, the techniques addressed in their evaluation are typically applied at the unit level.

Richardson and Wolf [55] argue the importance of testing at the architectural level. They suggest that this can be accomplished by applying current techniques to an executable model of the architecture. The template for this model is called "CHAM," Chemical Abstract Machine. This machine provides the structural basis for a nomenclature for coverage criteria. The test process can be used to assess the validity and testability of the architecture and the conformity of the implementation to the architecture.

The T-VEC system by Blackburn and Busser [7] generates test vectors from hierarchical, executable requirements specifications. A test vector includes both the input data and the expected output. This allows the automation of test execution by producing a report of the success or failure for each test. However, it also requires that the specification contain a mechanism which details precisely how the desired output might be achieved. This runs counter to the philosophy of many system-level specification paradigms, which encourage specifying what is desired while refraining from specifying how it is achieved.

T-VEC specializes in dealing with non-linear inequalities. T-VEC also performs coverage analysis, test driver generation, and test results analysis. The coverage analysis is a matching of the feasible generated test vectors and the requirements from which they were generated. Any mismatches indicate anomalies in the requirements. T-VEC does not deal with quantification over infinite domains, nor condition dependencies beyond inequalities. Therefore, any specification containing such dependencies must be modeled in a way that expresses these dependencies in terms of inequalities.

The primary disadvantage of T-VEC is that it requires an executable specification. This requires that requirements be reformulated to match this model. The limited expressiveness, e.g., lack of quantification over infinite sets, of the T-VEC specification language, makes this a non-trivial and expensive task when applied to the type of specifications addressed in this dissertation.

Various techniques exist for deriving tests and test sequences from variations on finite state machines [33], e.g., those based on specification languages such as Statecharts [67], SDL [43], LOTOS [11], X-machines [42], and that of the Validator/Req [3] test generation tool. The test sequencing provided by these techniques is important for testing protocols in communication systems, e.g., specifications with simple transitions but a complex state space. In contrast, this thesis focuses on specifications that do not necessarily refer to states, but whose complexity lies in the logical relationships between stimuli and responses of the system.

These techniques have similar limitations of expression. Constructing an executable model is often a complex and expensive task. This effort is well spent if it adds value by proving certain properties of the model. However, this is a separate issue and is not the purpose of system-level requirements-based testing.

3.4.2 Logical Manipulation

Laycock [41] applied the category-partition method of Ostrand and Balcer [46] to a Z specification. The work demonstrated the feasibility of automating test generation from a formal specification.

Inspired by the work of Bernot, Gaudel, and Marre [6], Dick and Faivre [17] describe a technique for deriving test steps based on a disjunctive normal form (DNF) of a formal specification expressed as a state relation in first-order predicate calculus. The technique is based on a procedure for transforming a formal specification into a disjunctive normal form that represents the possible states of the system. Test steps are inferred from the disjuncts by determining the pre-condition for the corresponding state. A means of sequencing test steps is also given by Dick and Faivre. Their technique can produce a combinatorially large number of tests, since it produces every possible combination of choices provided by disjunctions in the specification.

First-order predicate calculus is limited for general use in specifications at the system level. Formal specification languages such as Z [61] and VDM-SL [37] are more suitable at the system level since they are more expressive, e.g., by allowing quantification. Work based on Z that is similar to the approach used by Dick and Faivre has been done by Hörcher [34]. Helke, Neustupny, and Santen [32] have re-implemented this technique using an embedding of Z in the Isabelle theorem prover [50]. This latter work demonstrates the feasibility of applying theorem-proving technology to test generation. This provides a standardized mechanism for ensuring test correctness. The underlying logic of the specification languages for these techniques is expressive enough for use in system-level requirements-based testing. However, the derivation algorithms do not deal with quantification.

Stocks and Carrington [62] have presented a framework for specification-based testing that addresses such issues as test oracles and test suite maintenance. The use of test oracles assumes that, for a given specification, the output can be computed from a given input. This assumption implies that the formal specification must be executable. The approach presented in this dissertation allows non-executable specifications, but does not produce test oracles. The importance of non-executable specifications is argued by Hayes and Jones [30].

Gaudel [6, 26] describes a theory of testing based on algebraic specifications that are characterized by the use of functions to denote operations. A set of axioms, typically expressed as universally quantified equations, defines a class of algebras. Each algebra is a model of the specification. In contrast, predicate logic specifications typically use relations between states to denote operations, and both universal and existential quantification are often present.

Hayes [31] argues that algebraic techniques are best suited to testing primitive data types and that, for more complex abstract data types, model-based specification is simpler. Hayes describes a manual technique for applying model-based specifications to module testing.

As noted by Gaudel, predicate logic specifications are more general than algebraic specifications. However, the price of this generality is the restriction that, in general, only test frames can be generated automatically. Algebraic techniques such as the one by Bernot, Gaudel, and Marre [6] can generate test data. This test data corresponds to what this thesis refers to as test steps. Test steps are instances of test frames.

3.4.3 Disadvantages of Modelling

The techniques described in the previous section are based on mathematical modelling. A disadvantage of the above techniques, in the context of system-level requirements-based testing for large projects, is that determining the underlying primitives for the model is often a non-trivial task, which is outside the bounds of typical requirements authoring. Constructing a model that supports the appropriate dependencies between conditions within the requirements is a fundamentally different skill from the presentation of system-level requirements. This is because the model contains technical detail particular to the model, or modelling language, that is not readable by typical requirements authors. Thus, specifications based on mathematical logic often require the maintenance of two specifications: one readable by requirements authors for contract purposes, and the formal version used to generate tests. Thus, mathematical logic approaches usually incur additional costs associated with this second specification.

This approach also requires a review process to ensure that the two specifications remain synchronized as changes are made. This tends to delay the derivation of the formal specification in order to ensure that changes are minimized. However, the process of generating tests often uncovers inaccuracies within the requirements specification. This feedback is critical for requirements authors. The result is that a possibly lengthy delay for testing-to-requirements-author feedback is built into the process.

For specifications where a large number of requirements can be based on a small number of primitives with relative ease, these modelling costs are usually repaid in ensuring consistency within the specification. This is due to the high degree of interdependence between requirements. However, for specifications with a large number of independent requirements, the costs of modelling are less fruitful, simply because there are fewer possibilities for inconsistencies. In such cases, a manual review is likely to be less expensive and just as effective in discovering them.

3.4.4 Coverage Schemes

Some work has explored issues of coverage schemes. A coverage scheme is an algorithm for constructing a test set that satisfies a given coverage criterion. MacColl, Carrington, and Stocks [44] describe a mechanized but not automated approach to deriving test steps from formal specifications. They provide for a variety of derivation strategies which could embody different coverage schemes. Ammann and Offutt [2] describe each-choice-used and base-choice coverage. These coverage criteria dramatically reduce the number of test steps produced. These criteria are different from code-based criteria, since they describe coverage in terms of a relationship between two behaviours of the system. These coverage criteria are examined further in Chapter 6.

The author has introduced a framework for several coverage criteria based on prime implicants of a partitioning of the specification referred to as test classes [18]. The same paper presents details of generating test frames from a formal specification containing universal and existential quantification. This thesis is the fruition of this earlier work.

3.5 Conclusion

This chapter has described three categories of techniques that might serve as a basis for a solution to the problem described in Chapter 2. Systematic techniques lack the mathematical soundness required to ensure test frames correctness. Code-based techniques, while providing well developed notions of coverage, do not address features found in more expressive specification languages that are suitable for system-level requirements. Current logic-based techniques lack a combination of automation and expressiveness.

Chapter 4

Fundamental Challenges

A central conclusion of this research is that the problem of generating test frames algorithmically from a set of requirements for the purpose of system-level testing is significantly different from the problem of generating test frames from code for the purpose of unit level testing. This chapter examines the challenges that illustrate this difference.

4.1 Introduction

There are four fundamental challenges to system-level requirements-based testing: structural independence, condition dependence, quantification, and the Delta Problem. Code-based techniques provide a rich vocabulary for describing coverage criteria, the means of evaluating the coverage achieved by a given test set, and, to some degree, a means of automatically generating tests. However, techniques for code-based testing do not need to address the fundamental challenges of system-level requirements-based testing.

At the system level, the readability of the requirements specification is of primary concern. The purpose of this specification is to communicate what is required of the system so that the appropriate stakeholders, e.g., customers, requirements authors, software designers, government regulators, can comprehend and discuss requirements issues as easily as possible. To ensure that previously generated test frames are not made obsolete by simple changes in presentation to address readability issues, it is essential that the derivation of test frames be structurally independent of how the requirements are stated.

Recognizing dependencies between conditions within the requirements is necessary to avoid generating infeasible test frames. Depending on the way in which requirements are specified, different strategies for recognizing condition dependencies may be more or less appropriate. For example, properties of well understood primitives can be used to compute dependencies between conditions defined in terms of these primitives. However, in more abstract specifications, other techniques may be more appropriate.

Existential and universal quantification are logic mechanisms that reflect phraseology commonly found in natural language. These mechanisms provide a means of describing what is required, rather than how it is achieved. For example, it is easier to state universally that "all men are mortal," than to enumerate the fact for each and every man. Thus, quantification is an important quality of a system-level specification language.

The impact of specification changes on previously generated test frames is an important consideration when applying any automated test frame derivation technique to large projects. When generating new test frames, it is expensive to ignore test results based on existing test frames that are still valid. A *valid* test frame is one that is logically implied by the specification. The Delta Problem is to integrate existing valid test frames into new test frame sets. Structural independence is mandatory, but additional capability is required to solve the Delta Problem.

Section 4.2 examines an application of a code-based approach to system-level requirements. This leads to the issue of structural independence, which is elaborated further in Section 4.3. Section 4.4 examines the impact of specification type on the choice of condition recognition strategy. Section 4.5 presents the importance of universal and existential quantification to system-level requirements-based testing. This is followed by Section 4.6, a description of the Delta Problem.

4.2 Specifications as Code

The systematic derivation of tests based on the structure of code for the purpose of testing software components is well-established. It is sensible, therefore, to consider the possibility of simply lifting this idea up to the level of requirements-based testing for the purpose of generating test frames.

It is relatively easy to translate stimulus-response statements, provided they do not require quantification, into a logical representation using simple code-like constructs such as **if-then-endif**, **if-then-else-endif**, **and**, **or**, and **not**. For example, the requirement,

> When Stimulus S occurs and Condition C1 or Condition C2 is true, then
> the system shall produce Response R

could be translated into the following code-like representation:

> if S and (C1 or C2) then R endif.

This simple approach takes into account only the top-level logical structure. The phrases represented symbolically by S, C1, C2 and R would correspond to phrases such as "the aircraft is airborne," which are left unformalized. Such phrases could be represented formally in a parseable notation such as S [39], which allows text strings such as "the aircraft is airborne" to be introduced as uninterpreted constants.

This simple approach would yield a code-like representation, in the sense that it would have a logical structure expressed by standard logical operators of common programming languages. This logical structure serves as the basis for generating tests from code using well-known techniques.

For example, the following code-like statement,

```
if (S1 and S3)
        or ((not S1) and S2)
        or ((not S1) and (not S3)) then
    R
endif
```

could be used as input to a test frame generation tool based on the condition/decision coverage criterion defined by Chilenski and Newcomb [12]. Their definition of condition/decision coverage is:

> Every possible decision and condition has taken all possible outcomes at least once.

For the above example, the decision is,

(S1 and S3) or ((not S1) and S2) or ((not S1) and (not S3))

and the conditions are: S1, S2, and S3.

A test frame generation tool based on condition/decision coverage must generate a set of test frames that includes at least one test frame in which the decision evaluates to true, and at least one test frame in which the decision evaluates to false. Also, for each condition, S1, S2 and S3, there must be at least one test frame in which the condition is true, and another test frame in which the condition is false. A minimal set of test frames satisfying condition/decision coverage is,

1. S1 and (not S2) and (not S3) $\Rightarrow$ not R, and

2. (not S1) and S2 and S3 $\Rightarrow$ R

where, as before, the symbol "$\Rightarrow$" is used to separate the stimuli part of the test frame from the response part.

The first test specifies that when S1 is true and S2 and S3 are false in the environment, the system should respond in a manner consistent with "not R." Under the truth values specified by the first test, the decision in the specification evaluates to false. In the second test, the decision evaluates to true and the appropriate response is R. Since each of the conditions takes on the values true and false in at least one test, these two tests satisfy the condition/decision coverage criterion. The set is minimal because there must be at least two tests: one in which the decision evaluates to true, and a second in which the decision evaluates to false.

So it may appear that the methods previously developed for algorithmically generating tests from code can simply be re-used. These methods are based exclusively on structure, which, in this example, is expressed by code-like constructs, e.g., **if-then-endif**, **if-then-else-endif**, **and**, **or** and **not**.

However, a limitation of this simple approach is illustrated by the fact that the statement,

```
if (S1 and S3)
        or ((not S1) and S2)
        or ((not S1) and (not S3)) then
    R
endif
```

might alternatively have been written in the logically equivalent form:

```
if (S1 and S3) then
    R
else
    if not S1 then
        if S2 or (not S3) then
            R
        endif
    endif
endif
```

For this alternative form, the test frames,

1. S1 and (not S2) and (not S3) $\Rightarrow$ not R, and

2. (not S1) and S2 and S3 $\Rightarrow$ R

would not satisfy the condition/decision coverage criterion. This is because the structure of the alternative form involves a different set of decisions than the original form, even though they are logically equivalent.

The fact that this coverage criterion yields different results for logically equivalent statements is not surprising, given that it is intended to be a coverage criterion for code at the unit level. This is partially explained by the fact that the structure of the code directly affects compilation in terms of which instructions are executed, and the order in which they are executed. For unit testing, the test set must be structurally tied to the implementation, since a change in implementation source code actually changes the underlying system.

The situation is very different for system-level, requirements-based testing, where it is likely to be highly undesirable for two semantically equivalent, but structurally different, statements to yield a different set of tests. Hence, the above example suggests that the usefulness of techniques based purely on code-like structure may be limited as the basis for automating the task of generating test frames from formalized requirements for the purpose of verifying requirements.

4.3 Structural Independence

The term *structural dependence* refers to the coupling between the structure of the input of a test frame generation process, and the test frames produced. Ideally, test frames should be structurally independent from the specification from which they were derived. The output of a test frame generation process should be affected by requirements changes only to the extent that the revised requirements differ semantically from the original requirements. Two structurally different, but semantically equivalent, versions of the requirements should ideally produce the same set of test frames.

This conjecture is based on the observation that, for a variety of reasons, requirements may be organized structurally in a manner that is not conducive to generating tests. It would be undesirable for redundant test frames to be generated simply because of the structure of the requirements. Also, for a variety of reasons, a significant change to the structure of the requirements may be made with little or no semantic change, i.e., no implementation changes are required. It would be undesirable for such changes to yield a significantly different set of test frames if this entails re-working existing test procedures, and/or repeating previously executed tests.

One approach to addressing structural dependence may be to impose constraints on the formal representation of requirements so that there is only one way express the requirements. However, it is doubtful that it is possible to devise an effective set of constraints that would gain wide acceptance. Instead, the strategy adopted in Chapter 5 is based on the transformation of sets of requirements into a normal form using rules of logical reasoning.

Unfortunately, complete structural independence cannot be achieved. A condition expression can be rephrased such that the new form cannot be recognized as being equivalent to the original by automatic means. In mathematical logic terms, complete structural independence cannot be achieved because the truth of a conjecture of the equivalence of two general formulae may be undecidable.

4.4 Condition Dependence

The term *condition dependence* refers to logical relationships between conditions within a requirements specification. It is often the case that these dependencies are not explicitly documented in the requirements, though they impact the derivation of test frames.

For instance, the requirements specification for an air traffic control system may use phrases such as “is airborne,” “has landed,” and “is cleared for departure,” as primitive terms. The choice of these phrases as primitives rests upon the assumption that the users of the specification have enough common domain knowledge to recognize dependencies between these primitives. For example, an aircraft cannot simultaneously satisfy the condition "is airborne" and "has landed."

The set of primitive terms used in a natural language requirements specification of a system constitutes the level of abstraction used by the requirements authors. One approach to addressing condition dependence is to reduce the number of primitive terms to a very small number of purely mathematical primitives. Decision procedures can then be used to search for dependencies at this standard level. In effect, this lowers the level of abstraction in a manner analogous to the refinement of a requirements specification into executable code. Whereas the primitives in code are operations on bits, the primitives in this unrestrained style of formalization are, for instance, operations on mathematical sets. In both cases, the result is a much more detailed description that blurs the distinction between "what" and "how" in the specification of the required functionality. In more practical terms, the refinement of hundreds or thousands of primitives down to the level of abstract mathematics, though it may be intellectually challenging, is an indirect and costly way to address condition dependence.

The strategy presented in Chapter 7 allows the level of abstraction used by the domain experts to be maintained by introducing the primitive terms of the natural language specification as uninterpreted elements of the formal representation. Many formal specification notations allow elements such as types, constants, functions and predicates to be introduced as uninterpreted elements. In simple terms, this means that names for these elements may be declared as part of the working vocabulary of the formal representation without providing a definition of the element in terms of some previously introduced or built-in element. Condition dependence is addressed in this dissertation by allowing the user to selectively provide some forms of domain knowledge as input to the test frame generator. This domain knowledge takes the form of axiom schemata that define mutually exclusive conditions and conditions forming partial orders and states. This provides the required information in order to determine dependencies between conditions. This approach is described in further detail in Chapter 7.

4.5 Quantification

Finite forms of quantification are, of course, expressible in any programming language. Universal quantification over a finite set of values can be expanded into a conjunction of conditions. Similarly, existential quantification over a finite set of values can be expanded into a disjunction of conditions. However, formal specifications often involve quantification over sets of values that are not necessarily finite, or whose members are left unspecified. Even in the case of quantifying over some finite sets, it may not be practical to expand the quantification into a conjunction or disjunction if the finite set is large, e.g., the set of all 32-bit integers.

Section 4.3 outlined how a modest level of formalization could be achieved using only simple code-like structures, such as **if-then-endif**, **if-then-else-endif**, **and**, **or**, and **not**. However, this propositional logic style of formal specification may not be adequate in all cases. Circumstances may require more expressive kinds of formal specification, based, for instance, on predicate logic with quantifiers.

The ability to quantify universally, i.e., "for all," or existentially, i.e., "there exists," over a set of values often allows the expression of requirements in the formal representation to more closely correspond to their expression in natural language. This is often a matter of being able to express what functionality is required, rather than how the function is to be realized. Quantifiers are also useful when specifying global constraints that influence the interpretation of other requirement statements.

For this reason, quantification is also a fundamental challenge which must be addressed by any practical approach to generating test frames from formalized requirements. Obviously, existing techniques for generating test frames from code are not equipped to accept input containing quantifiers, since programming languages do not include general quantifiers as operators.

4.6 The Delta Problem

The Delta Problem, which is the integration of existing tests with new ones, requires analysis and is different from structural independence. Structural independence provides a degree of latitude that allows the test generator to produce tests to fit certain criteria. This also allows the test generator to integrate existing tests with new ones.

When specification changes occur, it is necessary to minimize their impact on existing test sets previously constructed. Although generating a completely new test set is possible, this is undesirable if testing has already begun. Assuming that the requirements changes do not require any implementation changes, it is less expensive to perform a few new tests to augment positive results already obtained than to dismiss previous positive results and perform a larger number of different tests. For example, if a portion of the requirements is re-worded for clarity or contractual reasons, but no implementation changes are necessary and the test generator produced different tests based on the re-wording, then unnecessary and perhaps costly testing would be performed. Thus, existing tests must be integrated with any new tests by the test generator.

This capability is not necessary in the context of code-based testing. A rearrangement of conditions within coded decisions rarely results in a situation where the implementation does not need to be re-tested. This is because such a change usually results in a change to the implementation. For example, in a C program, simply changing `if (a || b)` to `if (b || a)` changes the order of evaluation. Since the implementation has changed, it must be re-tested; therefore, generating new tests is not wasteful.

In order to minimize test set impact due to specification changes, a test frame generator should accept two inputs: the specification for which test frames are to be derived, and the previous set of test frames. To the extent possible, the test frame generator should attempt to use the previously generated test frames as a starting point for constructing a test set that satisfies the given coverage criterion. This should be the case whether the specification or the coverage criterion is changed. Extending this idea, it is desirable to allow test engineers to specify the "previous tests." This would provide a means of allowing test engineers to mandate certain tests, and to use the test frame generator to complete the test set according to a chosen coverage scheme.

4.7 Summary

This chapter has examined the possibility of writing requirements like program code to take advantage of well-known, existing code-based techniques. This has led to the identification of certain challenges to be overcome by a technique that can be applied to requirements-based testing. The challenges, structural independence, condition dependence, quantification, and the Delta Problem, distinguish requirements-based testing from code-based testing.

The first challenge is that requirements-based tests should be structurally independent of the way in which the requirements are written. This is not required of code-based test generation techniques which produce tests that are structurally dependent. Unfortunately, complete structural independence cannot be achieved for all specification languages.

The second challenge is to capture condition dependencies amongst conditions that may not be defined in terms of primitives, as is the case in code. These dependencies are necessary in order to avoid generating infeasible tests and to simplify those that are feasible.

Quantification provides an expressiveness that is useful for describing requirements at the system level. This challenge does not exist in the domain of code-based techniques, but must be addressed in a discipline of requirements-based testing.

The fourth challenge is the Delta Problem. Wasteful rework can be avoided with the ability to integrate existing tests into new test sets when requirements changes occur. This challenge is specific to requirements-based testing, because a substantial re-wording of the requirements does not necessitate the obsolescence of all existing tests.

Chapter 5

A Foundation for the Discipline

This chapter presents a discipline of test derivation which includes algorithms for generating test frames from formal specifications containing universal and existential quantification. A nomenclature for defining specification-based coverage criteria is based on the parameters of these algorithms. The foundation of this technique on formal rules of logical derivation ensures that the test frames produced are logical consequences of the specification. Since this technique deals with quantification, it can be applied to more expressive specifications than previous approaches. This also makes the technique applicable to specifications written at the system requirements level.

5.1 Introduction

It is well recognized that there is an important distinction between specifying what a system should do, and how this goal is to be achieved. In particular, when specifying system-level requirements it is important to focus on "what," while specifying as little "how" as possible. Mathematical logic provides a means of describing "what" without describing "how." Conversely, code is well suited to describing "how," but is more difficult to use when trying to describe "what" without "how." For this reason, along with the issues raised in the previous chapter, logic-based approaches seem to be better suited as a foundation for automating system-level requirements-based testing.

The most appropriate existing test-generation technique for the type of specifications addressed by this thesis is the DNF approach, which arose from the work of Dick and Faivre [17]. However, this approach has certain limitations. An alternative to the DNF approach forms the basis of the discipline of specification-based test derivation presented in this dissertation.

There are three fundamental entities that highlight the intermediate stages to generating test frames: test classes, frame stimuli, and test frames. These entities form the basis of the nomenclature which will be used in Chapter 6 to define coverage criteria. During the production of test classes, certain forms of specifications can be flagged as possible specification errors. Test class normal form is the key mechanism by which system behaviours are grouped. The production of test frames introduces the notion of specification coverage. The terms test class, test frame, and frame stimuli form the foundation for the nomenclature that will be used to define coverage criteria. The basic coverage concepts introduced in this chapter are extended further in Chapter 6.

Section 5.2 begins this chapter by detailing some of the limitations of test generation techniques based on the work of Dick and Faivre [17]. Section 5.3 introduces the notation and fundamental terminology for the discipline. This is followed by Section 5.4, which defines test classes, test frames, and test steps and provides an overview of the relationships between them. Section 5.5 presents one of the fundamental ideas of this thesis: Test Class Normal Form. Section 5.6 deals with coverage schemes and the actual generation of test frames from test classes.

5.2 A Place to Start

The DNF approach is based on a procedure for transforming a formal specification into a disjunctive normal form that represents the possible states of the system. Test steps are inferred from the disjuncts by determining the pre-condition for the corresponding state. Specifications are transformed using logical manipulations such as

$$A \Rightarrow B = \neg A \vee (A \wedge B), \text{ and}$$
$$A \vee B = (A \wedge \neg B) \vee (\neg A \wedge B) \vee (A \wedge B).$$

An example from Dick and Faivre's original paper [17] illustrates their process. The specification $(max = a \vee max = b) \wedge max \geqslant a \wedge max \geqslant b$ is transformed and simplified into the set of state descriptions:

$$\{max = a \wedge max = b, max = a \wedge max > b, max = b \wedge max > a\}.$$

Each element of the above set represents a possible state of the system.

A limitation of this approach is that disjunction and implication are treated differently. This implies that if an author wrote $B \vee \neg A$ or $\neg B \Rightarrow \neg A$ rather than $A \Rightarrow B$, different tests would result. The limitations of this type of structural dependence were presented in Section 4.3.

Care must be taken when dealing with non-determinism in the context of the DNF approach. This thesis does not consider the merits or problems associated with non-deterministic specifications, but acknowledges their existence. Hayes and

Jones [30] describe situations where non-deterministic specifications are particularly useful. The non-deterministic specification $S \wedge (R_1 \vee R_2)$ leads to three possible states:

$$S \wedge \neg R_1 \wedge R_2,\ S \wedge R_1 \wedge \neg R_2, \text{ and } S \wedge R_1 \wedge R_2.$$

However, these three states do not directly correspond to three valid tests, i.e., tests that will not reject a correct program. This is different from the first example, where each state corresponds to a valid test. Clearly, it would be more appropriate not to split the original disjunction in this case. This problem hints that there is a fundamental difference between stimuli and responses, which needs to be addressed when generating tests.

A further limitation of this approach is that it does not explicitly address the presence of universal and existential quantifiers within the specification. Along with addressing quantification issues, the discipline presented in this dissertation takes a slightly different approach to test derivation. Rather than producing a disjunction of all possible states, a conjunction of the stimulus-response behaviours of the system is produced. In the specification of possible states produced by the DNF approach, stimuli, responses, and non-determinism are not obvious. Tests can be more readily derived from stimulus-response descriptions, since the stimuli and responses are explicitly separated.

5.3 Notation and Terminology

The technique presented in this dissertation is based on the logical relationships between elements within the specification. Since it is not tied to a particular specification language such as S [39] or Z [61], standard logical expressions shall be used in the text below. The technique is composed of two algorithms, which are founded on the following definitions:

1. A predicate represents a parameterised truth value.[1] The symbols $\top$ and $\perp$ represent the Boolean values true and false.

2. An atom is either a predicate or a negated predicate.

3. A stimulus is an atom that only refers to the state of the system before an operation is performed.

4. A stimulus expression is a predicate logic expression where each atom is a stimulus.

5. A *frame stimulus* is a restricted form of stimulus expression. The exact definition of a frame stimulus for a particular test class is provided algorithmically in Section 5.6.1. A frame stimulus has one of the following forms:

   (a) an atom,

   (b) a universally quantified atom,

   (c) a universally quantified disjunction of stimulus expressions, or

   (d) a universally quantified stimulus expression which is itself existentially quantified, e.g., $\forall\, x. \exists\, y. E(x, y)$, where E is a stimulus expression.

6. A response is an atom that contains at least one reference to the state of the system after the operation has completed, and may also refer to the previous state, i.e., any atom which is not a stimulus is a response.

[1] In this dissertation, the term predicate refers to the predicate symbol and its parameters.

7. A response expression is a predicate logic expression where each atom is a response.

A specification of a system is a logical expression relating the state of the system at the time a stimulus occurs, to the state of the system at the time the response is produced. The expression is constructed from predicates, the logical connectives conjunction, disjunction, implication, and negation, along with universal and existential quantification (the standard logic symbols are $\vee, \wedge, \Rightarrow, \neg, \forall$, and $\exists$, respectively). A system specification may be of the form:

$$(S_1 \Rightarrow R_1) \wedge (S_2 \Rightarrow R_2) \wedge \ldots \qquad (5.1)$$

where the S_i are stimulus expressions and the R_i are response expressions. This specifies a system that will satisfy R_i when given the stimulus S_i. In this specification, each implication describes a class of behaviour to be exhibited by the system. However, a specification is not restricted to this form. The restrictions on specification form are given in Sections 5.5.3 and 5.5.4.

The following example illustrates the above definitions. The specification used in this example is a Z adaptation of a portion of the VDM-SL style RSL solution by Schinagl [58] to Abrial's steam boiler specification problem [1]. Modifications were made to construct a concise example, but these changes do not affect its logical complexity. Test frames generated from a larger portion of Schinagl's specification are given in Appendix B.

Abrial's specification problem is to formally specify requirements for a control system responsible for maintaining the correct level of water in a boiler attached to a steam, driven turbine. One of the requirements of this system is to identify whether or not any inconsistencies exist in the sensor readings.

$$\begin{array}{l}\neg\ \mathit{OutOfOrder}' \Leftrightarrow \\ \quad (\exists!\, n : \mathbb{N} \bullet \mathit{Level}\ n) \wedge \\ \quad (\exists!\, n : \mathbb{N} \bullet \mathit{Steam}\ n) \wedge \\ \quad (\forall\, i : \mathit{PUMP} \bullet \mathit{PumpState}(i, \top) \Leftrightarrow \neg\ (\mathit{PumpState}(i, \bot))) \wedge \\ \quad (\forall\, i : \mathit{PUMP} \bullet \exists\, b : \mathit{bool} \bullet \mathit{PumpCtrState}(i, b))\end{array}$$

$A \Leftrightarrow B$ is defined as $(A \Rightarrow B) \wedge (B \Rightarrow A)$. Exists unique, $\exists!$, is defined as,

$$\exists! x.S\ x = \exists\, x.S\ x \wedge (\forall\, x, y.S\ x \wedge S\ y \Rightarrow (x = y))$$

This specification requires that the "out of order" indicator, *OutOfOrder*, is true if and only if there is a detected malfunction. The predicates *Level, Steam, PumpState,* and *PumpCtrState* represent the presence of various messages just received from the sensors. *Level* indicates the quantity of water in the boiler, *Steam* indicates the quantity of steam coming from the boiler, *PumpState* indicates whether the given pump, i, is turned on ($\top$) or off ($\bot$), *PumpCtrState* indicates whether or not water is circulating from the pump, i, to the boiler. Primed variables are references to the after state, thus $\neg \mathit{OutOfOrder}'$ is a response. All the other atoms, such as $\mathit{PumpState}(i, \top)$, are stimuli.

This specification is a relationship between the response and various stimuli. Although it is not written directly in the form of (5.1), it can be translated into that form as part of test frame generation.

5.4 Overview

Requirements specifications are written to be understood at particular levels of abstraction. For this reason, many details are hidden within definitions of more abstract concepts. Issues of clarity are left to the discretion of the specification authors. Hence, it must be assumed that the specification is an arbitrary logical expression and there is some means of distinguishing stimuli from responses.

Test classes are the intermediate step between the specification and test frames. The derivation of test classes requires a means of distinguishing stimuli from responses. A test class isolates one behaviour from the specification. The test class can be considered as a standard format for writing requirements. However, for practical reasons, it is unlikely that all specifications would be written as a simple conjunction of test classes as in (5.1).

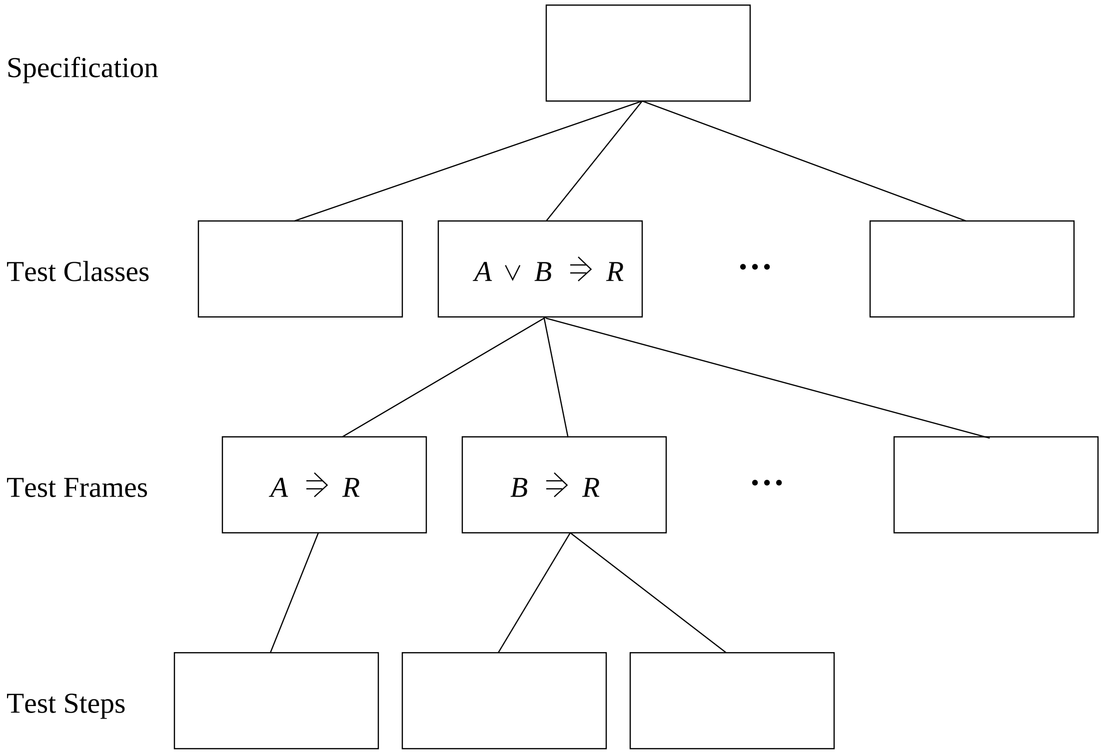

Figure 5.1: Entity Relationships

Figure 5.1 illustrates the relationships between the specification, test classes, test frames, and test steps.

Definition 1 *A* test class *is an implication $S \Rightarrow R$, which may be quantified, where S is a stimulus expression and R is a response expression. Quantifiers may appear anywhere in the test class, and may also bind variables occurring in both S and R.*

The purpose of the test class is to isolate a class of behaviour based on the response. The first step of the test frame generation process is to transform the specification into its *test class normal form* such as (5.1). Details of this transformation are presented in Section 5.5.

A set of test frames is produced from each test class.

Definition 2 *A* test frame *is an implication $A \Rightarrow R$, which may be quantified, where A is a conjunction of frame stimuli and R is the response expression from the corresponding test class. Quantifiers may also bind variables occurring in both A and R. A test frame $A \Rightarrow R$ generated from the test class $S \Rightarrow R$ has the property that $A \Rightarrow S$.*

The generation of test frames is presented in Section 5.6.

Definition 3 *A* test step *is an implication $t \Rightarrow R$, where t is a conjunction of atoms and R is a response expression. Quantifiers can only occur in R.*

Although it is desirable to derive test steps, these cannot, in general, be generated automatically from the type of specifications considered in this dissertation. However, much of the effort required to generate a test step can be performed automatically by producing a test frame. As stated in Section 1.2, the instantiation of test frames into test steps is beyond the scope of this thesis.

The computation of test frames from a specification can be performed within any logic consistent with the manipulations used in this chapter. The algorithms do not diverge, due to the use of convergent subsets of logical inferences when transforming portions of the specification.

5.5 Test Class Normal Form

This section presents the underlying algorithm for producing test classes. Variations of this algorithm are presented later in Section 6.4. This algorithm has the following important properties:

1. For non-demonic[2] formal specifications, test class normal form can be computed in $O(n \log n)$ time in the size of the specification.

2. It is founded on rules of mathematical logic, which ensures that the algorithm is logically sound.

Definition 4 *Test class normal form is a conjunction of test classes with distinct responses.*

It can be achieved by applying the test class algorithm to a specification which is a logical relation with restrictions (Sections 5.5.3, 5.5.4). Test class normal form is not canonical.

5.5.1 The Test Class Algorithm

The test class algorithm can be described as a function on logical expressions. The result of applying this function to an expression, E, is a conjunction of test classes

[2]Demonic specifications are described in Section 5.5.4.

which is logically equivalent to E. The test class algorithm rewrites the specification into its test class normal form. This does not alter its logical content.

Assuming R is a response and S is a stimulus, a definition for the recursive test class algorithm, TC, is:

$$
\begin{array}{rcll}
TC(A \wedge B) & = & \mathit{RewriteAnd}(TC(A) \wedge TC(B)) & \text{conjunction} \\
TC(A \vee B) & = & \mathit{RewriteOr}(TC(A) \vee TC(B)) & \text{disjunction} \\
TC(\forall x.P) & = & \mathit{ForallIn}(\forall x.TC(P)) & \text{quantification} \\
TC(\exists x.P) & = & \mathit{ExistsIn}(\exists x.TC(P)) & \text{quantification} \\
TC(A \Rightarrow B) & = & TC(\neg A \vee B) & \text{implication} \\
TC(R) & = & \top \Rightarrow R & \text{response} \\
TC(S) & = & \neg S \Rightarrow \bot & \text{stimulus}
\end{array}
$$

S and R can refer to negated predicates. Negated expressions are dealt with by applying DeMorgan's laws and double negation to move the negation inwards and proceeding.

$$\neg(A \vee B) = \neg A \wedge \neg B$$

$$\neg(A \wedge B) = \neg A \vee \neg B$$

$$\neg\neg A = A$$

In the descriptions below, it is assumed that the TC algorithm is operating on an expression that has a test class normal form. Expressions that do not have a test class normal form are addressed in Sections 5.5.3 and 5.5.4, below.

The algorithm *RewriteAnd* operates on a conjunction of test classes and combines any like antecedents and consequents using the equivalences:

$$(A \Rightarrow B) \wedge (A \Rightarrow C) = A \Rightarrow (B \wedge C)$$

$$(A \Rightarrow C) \wedge (B \Rightarrow C) = (A \vee B) \Rightarrow C$$

Combining response expressions is preferred over combing stimuli expressions. Applications of these equivalences may require a rearrangement of the two implications to be combined. For example, a conjunction such as,

$$(\mathit{Steam}\ x \Rightarrow \bot) \wedge C \wedge (\mathit{Level}\ y \Rightarrow \bot)$$

would be rewritten to:

$$((\mathit{Steam}\ x \vee \mathit{Level}\ y) \Rightarrow \bot) \wedge C$$

The algorithm *RewriteOr* operates on a disjunction of two conjunctions of test classes and first reduces any AND/OR connectives above these test classes to conjunctive normal form. Next, any universal and existential quantifiers are moved outside the disjunctions. This is done using the equivalences:

$$(\forall\, x.Q) \vee P = \forall\, x.Q \vee P$$

$$P \vee (\forall\, x.Q) = \forall\, x.P \vee Q$$

$$(\exists\, x.Q) \vee P = \exists\, x.Q \vee P$$

$$P \vee (\exists\, x.Q) = \exists\, x.P \vee Q$$

where x is alpha converted if necessary to avoid capturing any free occurrence of x in P. Finally, the test classes are OR'd together using the equivalence

$$(S_1 \Rightarrow R_1) \vee (S_2 \Rightarrow R_2) = S_1 \wedge S_2 \Rightarrow R_1 \vee R_2 \tag{5.2}$$

The *RewriteOr* algorithm is illustrated with the following example. When manipulating the expression,

$$((\mathit{Steam}\ x \Rightarrow \neg \mathit{OutOfOrder}') \wedge (\top \Rightarrow \mathit{OutOfOrder}')) \vee (\forall\, x.\mathit{Level}\ x \Rightarrow \bot)$$

the first step is to produce the conjunctive normal form:

$$((\mathit{Steam}\ x \Rightarrow \neg \mathit{OutOfOrder}') \vee (\forall\, x.\mathit{Level}\ x \Rightarrow \bot)) \wedge$$
$$((\top \Rightarrow \mathit{OutOfOrder}') \vee (\forall\, x.\mathit{Level}\ x \Rightarrow \bot))$$

Next, the universal quantifiers are moved outside the disjunctions. Here, the variable x_1 is introduced to avoid capturing the x of *Steam* x.

$$(\forall\, x_1.(\mathit{Steam}\ x \Rightarrow \neg \mathit{OutOfOrder}') \vee (\mathit{Level}\ x_1 \Rightarrow \bot)) \wedge$$
$$(\forall\, x.(\top \Rightarrow \mathit{OutOfOrder}') \vee (\mathit{Level}\ x \Rightarrow \bot))$$

The last step in the *RewriteOr* algorithm is to use Equation (5.2) to remove the disjunctions between the implications.

$$(\forall\, x_1.(\mathit{Steam}\ x \wedge \mathit{Level}\ x_1) \Rightarrow \neg \mathit{OutOfOrder}') \wedge (\forall\, x.\mathit{Level}\ x \Rightarrow \mathit{OutOfOrder}')$$

For non-demonic specifications, the *RewriteOr* algorithm is $\mathrm{O}(n \log n)$ since at least one of $TC(A)$ and $TC(B)$ in $\mathit{RewriteOr}(TC(A) \vee TC(B))$ produces a single intermediate test class.[3]

The algorithm *ForallIn* operates on a conjunction of test classes and moves the universal quantifier into the conjunction, if possible, using the equivalences:

$$(\forall\, x.P \wedge Q) = (\forall\, x.P) \wedge Q$$
$$(\forall\, x.Q \wedge P) = Q \wedge (\forall\, x.P)$$
$$(\forall\, x.M \wedge P) = (\forall\, x.M) \wedge (\forall\, x.P)$$
$$(\forall\, x.P \Rightarrow Q) = (\exists\, x.P) \Rightarrow Q$$
$$(\forall\, x.Q \Rightarrow P) = Q \Rightarrow (\forall\, x.P)$$

[3] Implications formed during the production of test classes are referred to as *intermediate test classes*.

where x is free in P and M, and x is not free in Q.

The algorithm *ExistsIn* operates on a conjunction of test classes and moves the existential quantifier, if possible, into the test class using the equivalences:

$$(\exists\, x.P \wedge Q) = (\exists\, x.P) \wedge Q$$

$$(\exists\, x.Q \wedge P) = Q \wedge (\exists\, x.P)$$

$$(\exists\, x.P \Rightarrow Q) = (\forall\, x.P) \Rightarrow Q$$

$$(\exists\, x.Q \Rightarrow P) = Q \Rightarrow (\exists\, x.P)$$

$$(\exists\, x.M \Rightarrow P) = (\forall\, x.M) \Rightarrow (\exists\, x.P)$$

where x is free in P and M, and x is not free in Q.

Some expressions do not have a test class normal form due to the arrangement of quantifiers. It is also possible for the conjunctive normal form produced by *RewriteOr* to be combinatorially large. These types of specifications are examined in Sections 5.5.3 and 5.5.4.

5.5.2 Example

This example illustrates the derivation of the test class normal form of the specification given in Section 5.3 above. The derivation is the evaluation of

$$\begin{aligned}
&TC(\neg \mathit{OutOfOrder}' \Leftrightarrow\\
&\;(\exists ! n.\mathit{Level}\ n) \wedge\\
&\;(\exists ! n.\mathit{Steam}\ n) \wedge\\
&\;(\forall\, i.\mathit{PumpState}(i, \top) \Leftrightarrow \neg(\mathit{PumpState}(i, \bot))) \wedge\\
&\;(\forall\, i.\, \exists\, b.\mathit{PumpCtrState}(i, b))
\end{aligned}$$

As a preliminary step in the derivation, the definition of $\exists!$ is expanded to obtain:

$$TC(\neg \mathit{OutOfOrder}' \Leftrightarrow E)$$

where E is:

$$\begin{array}{l}
((\exists\, n.\mathit{Level}\ n) \wedge \\
(\forall\, n, m.(\mathit{Level}\ n) \wedge (\mathit{Level}\ m) \Rightarrow (n = m)) \wedge \\
(\exists\, n.\mathit{Steam}\ n) \wedge \\
(\forall\, n, m.(\mathit{Steam}\ n) \wedge (\mathit{Steam}\ m) \Rightarrow (n = m)) \wedge \\
(\forall\, i.\mathit{PumpState}(i, \top) \Leftrightarrow \neg \mathit{PumpState}(i, \bot)) \wedge \\
(\forall\, i.\, \exists\, b.\mathit{PumpCtrState}(i, b))))
\end{array}$$

Next, the definition of $\Leftrightarrow$ is used to derive:

$$TC((\neg \mathit{OutOfOrder}' \Rightarrow E) \wedge (E \Rightarrow \neg \mathit{OutOfOrder}'))$$

Following this, the application of the TC algorithm's conjunction rule yields:

$$= \quad \mathit{RewriteAnd}(TC(\neg \mathit{OutOfOrder}' \Rightarrow E) \wedge TC(E \Rightarrow \neg \mathit{OutOfOrder}'))$$

The next operation is to rewrite the implication of the first TC term and then use the rule for disjunction (the ... represent unaffected subexpressions):

$$= \quad \mathit{RewriteAnd}(\mathit{RewriteOr}(TC(\neg\neg \mathit{OutOfOrder}') \vee TC(\ldots)) \wedge TC(\ldots))$$

The double negation is removed and the response rule is then applied:

$$= \quad \mathit{RewriteAnd}(\mathit{RewriteOr}((\top \Rightarrow \mathit{OutOfOrder}') \vee TC(\ldots)) \wedge TC(\ldots))$$

Using the rule for conjunction on the next TC term produces:

$$= \ \mathit{RewriteAnd}(\mathit{RewriteOr}((\top \Rightarrow \mathit{OutOfOrder}') \vee$$
$$\mathit{RewriteAnd}(TC(\exists\, n.\mathit{Level}\ n) \wedge TC(\ldots)) \wedge TC(\ldots)))$$

The quantification rule followed by the stimulus rule gives:

$$= \ \mathit{RewriteAnd}(\mathit{RewriteOr}((\top \Rightarrow \mathit{OutOfOrder}') \vee$$
$$\mathit{RewriteAnd}(\mathit{ExistsIn}(\exists\, n.\neg(\mathit{Level}\ n \in \mathit{inmess}) \Rightarrow \bot) \wedge TC(\ldots))$$
$$\wedge\ TC(\ldots)))$$

Applying $\mathit{ExistsIn}$ yields:

$$= \ \mathit{RewriteAnd}(\mathit{RewriteOr}((\top \Rightarrow \mathit{OutOfOrder}') \vee$$
$$\mathit{RewriteAnd}(((\forall\, n.\neg(\mathit{Level}\ n)) \Rightarrow \bot) \wedge TC(\ldots)) \wedge TC(\ldots)))$$

A full application of the algorithm to the next TC term produces:

$$= \ \mathit{RewriteAnd}(\mathit{RewriteOr}((\top \Rightarrow \mathit{OutOfOrder}') \vee$$
$$\mathit{RewriteAnd}(((\forall\, n.\neg(\mathit{Level}\ n)) \Rightarrow \bot) \wedge$$
$$(((\exists\, n, m.(\mathit{Level}\ n) \wedge (\mathit{Level}\ n) \wedge \neg(n = m)) \vee$$
$$(\forall\, n.\neg(\mathit{Steam}\ n)) \vee$$
$$(\exists\, n, m.(\mathit{Steam}\ n) \wedge (\mathit{Steam}\ n) \wedge \neg(n = m)) \vee$$
$$(\exists\, i.(\mathit{PumpState}(i, \top) \wedge \mathit{PumpState}(i, \bot)) \vee$$
$$(\neg(\mathit{PumpState}(i, \top)) \wedge \neg(\mathit{PumpState}(i, \bot)))) \vee$$
$$(\exists\, i.\,\forall\, b.\neg(\mathit{PumpCtrState}(i, b) \in \mathit{inmess})))$$
$$\Rightarrow \bot)) \wedge$$
$$TC(\ldots)))$$

Since the consequents of the two inner-most implications are identical ($\perp$), applying the inner-most *RewriteAnd* produces:

$$
\begin{aligned}
= \quad & \mathit{RewriteAnd}(\mathit{RewriteOr}((\top \Rightarrow \mathit{OutOfOrder}') \vee \\
& \quad (((\forall\, n. \neg(\mathit{Level}\; n)) \vee \\
& \quad (\exists\, n, m. (\mathit{Level}\; n) \wedge (\mathit{Level}\; n \in \mathit{inmess}) \wedge \neg(n = m)) \vee \\
& \quad (\forall\, n. \neg(\mathit{Steam}\; n)) \vee \\
& \quad (\exists\, n, m. (\mathit{Steam}\; n) \wedge (\mathit{Steam}\; n \in \mathit{inmess}) \wedge \neg(n = m)) \vee \\
& \quad (\exists\, i. (\mathit{PumpState}(i, \top) \wedge \mathit{PumpState}(i, \perp)) \vee \\
& \quad\; (\neg(\mathit{PumpState}(i, \top)) \wedge \neg(\mathit{PumpState}(i, \perp)))) \vee \\
& \quad (\exists\, i. \forall\, b. \neg(\mathit{PumpCtrState}(i, b) \in \mathit{inmess}))) \\
& \quad \Rightarrow \perp) \wedge \\
& \quad \mathit{TC}(\ldots)))
\end{aligned}
$$

Applying *RewriteOr* combines the response and stimuli to produce the first test class:

$$
\begin{aligned}
= \quad & \mathit{RewriteAnd}(\\
& \quad (((\forall\, n. \neg(\mathit{Level}\; n)) \vee \\
& \quad (\exists\, n, m. (\mathit{Level}\; n) \wedge (\mathit{Level}\; n \in \mathit{inmess}) \wedge \neg(n = m)) \vee \\
& \quad (\forall\, n. \neg(\mathit{Steam}\; n)) \vee \\
& \quad (\exists\, n, m. (\mathit{Steam}\; n) \wedge (\mathit{Steam}\; n \in \mathit{inmess}) \wedge \neg(n = m)) \vee \\
& \quad (\exists\, i. (\mathit{PumpState}(i, \top) \wedge \mathit{PumpState}(i, \perp)) \vee \\
& \quad\; (\neg(\mathit{PumpState}(i, \top)) \wedge \neg(\mathit{PumpState}(i, \perp)))) \vee \\
& \quad (\exists\, i. \forall\, b. \neg(\mathit{PumpCtrState}(i, b) \in \mathit{inmess})))
\end{aligned}
$$

$\Rightarrow \mathit{OutOfOrder}') \wedge$

$TC(\ldots)))$

Continuing with the remaining TC term produces the second test class:

$$(\exists\, n.\mathit{Level}\ n) \wedge$$
$$(\forall\, n, m.\neg(\mathit{Level}\ n) \vee \neg(\mathit{Level}\ m \in \mathit{inmess}) \vee (n = m)) \wedge$$
$$(\exists\, n.\mathit{Steam}\ n) \wedge$$
$$(\forall\, n, m.\neg(\mathit{Steam}\ n) \vee \neg(\mathit{Steam}\ m \in \mathit{inmess}) \vee (n = m)) \wedge$$
$$(\forall\, i.(\neg(\mathit{PumpState}(i, \top)) \vee \neg(\mathit{PumpState}(i, \bot))) \wedge$$
$$(\mathit{PumpState}(i, \top) \vee \mathit{PumpState}(i, \bot)) \wedge$$
$$(\forall\, i.\exists\, b.\mathit{PumpCtrState}(i, b))$$
$$\Rightarrow \neg \mathit{OutOfOrder}'$$

5.5.3 Existential Quantification

Specifications employing certain uses of existential quantification impose limitations on the test class algorithm, TC. Even so, such specifications can be converted algorithmically into specifications from which the TC algorithm can produce a test class normal form.

The limitations are manifested in the quantification rules of the TC algorithm as follows. *ForallIn* will not be successful in moving the universal quantifier into the conjunction if there is an existential quantifier in the way,

$$\text{e.g.,}\ \ \forall\, x.\exists\, y.(S_1 \Rightarrow R_1) \wedge (S_2 \Rightarrow R_2), \tag{5.3}$$

where y is free in at least one of S_1 and R_1, and also in at least one of S_2 and R_2. This occurs when an existential quantifier straddles two intermediate test classes.

An example of a specification similar to Equation (5.3) is:

> The system shall ensure that there is at least one printer satisfying the following:
>
> 1. if a job is printing on the printer, it will be completed within ten minutes; and
> 2. if there is a job about to be printed on the printer, it will commence printing within 15 minutes.

The intermediate expression encountered by the TC algorithm would be:

$$\exists\, printer.(\forall\, job.job\ PrintingOn\ printer \Rightarrow CompletedWithinTenMinutes\ job) \wedge$$
$$(\forall\, job.job\ FirstToPrintFor\ printer \Rightarrow StartsWithin15Minutes\ job)$$

In a specification, one would expect that the individual, *printer*, would be named explicitly, rather than implicitly by using an existential. Specifications such as these can be flagged by the TC algorithm. Alternatively, the existential variable can be replaced by a Skolem constant, e.g., in the case of Equation (5.3), f, a function of x, where f was not previously a free variable of the specification.

If desired, the existential quantifier in Equation (5.3) can be pushed inwards using the theorem

$$(\exists\, x.P(x) \wedge Q(x)) \Rightarrow ((\exists\, x.P(x)) \wedge (\exists\, x.Q(x)))$$

However, the use of this theorem produces a set of test classes that are implied by the original specification, rather than a set whose conjunction is logically equivalent to the original specification. Thus, this theorem cannot be used to produce a test class normal form of a specification.

It is possible that this existential quantification issue can also be addressed by other means.

5.5.4 Demonic Choice

Some forms of non-determinism, e.g., $S \Rightarrow (R_1 \vee R_2)$, are of no consequence to the test class algorithm. Demonic choice is a form of non-determinism which allows the implementation to behave according to more than one specification, arbitrarily. The demonic specification

$$(S_1 \Rightarrow R_1) \vee (S_2 \Rightarrow R_2)$$

does not force an implementation to produce R_1 in response to S_1, since it has the option of behaving like $S_2 \Rightarrow R_2$ and ignoring S_1. An implementation of this specification is not required to produce a response unless confronted with the stimulus $S_1 \wedge S_2$. In this case, it may elect to produce either R_1, R_2, or both, and still perform according to the specification.

The following example illustrates consequences of the demonic specification:

> The system shall arbitrarily perform at least one of the following actions:
>
> 1. Call the fire department, if there is a fire.
> 2. Call the police, if there is an explosion.

A formal version of this specification is:

$$(\mathit{fire} \Rightarrow \mathit{Call\ fire\_dept}) \vee (\mathit{explosion} \Rightarrow \mathit{Call\ police}).$$

The specification requires the system to respond only when there is both a fire and an explosion. When the system responds, it is allowed to call either the fire department or the police. The specification would be satisfied by a system that never called the fire department, even when there was a fire.

The test class algorithm can be applied to a demonic specification. However, this type of specification can cause a combinatorially large test class normal form due to the definition of *RewriteOr*. For example, the intermediate expression $(C_1 \wedge C_2) \vee (C_3 \wedge C_4)$ is converted to $(C_1 \vee C_3) \wedge (C_1 \vee C_4) \wedge (C_2 \vee C_3) \wedge (C_2 \vee C_4)$ before the disjunctions of test classes are combined using (5.2).

The author's experience suggests that this type of specification does not typically arise in system-level specifications. Each time a specification has been flagged as demonic by the TC algorithm, it has turned out to be a specification error rather than an intended behaviour.

5.6 Generating Test Frames

As defined in Section 5.4, a test frame from a given test class $S \Rightarrow R$ is an implication $A \Rightarrow R$, where $A \Rightarrow S$, A is a conjunction of frame stimuli, and R is a response expression. Quantifiers may also bind variables occurring in both A and R.

A variety of different test frame sets can be constructed from a test class. One possible set of test frames is the one derived from a disjunctive normal form (DNF) of the test class antecedent. However, the test class antecedent may have more than one DNF, e.g., the function $(a \wedge \neg c) \vee (\neg b \wedge c) \vee (\neg a \wedge b)$ and its alter ego $(a \wedge \neg b) \vee (\neg a \wedge c) \vee (b \wedge \neg c)$. In the context of the Delta Problem of Section 4.6, this raises an issue. If an existing test set contains a valid test frame which does not correspond to a term in the DNF of the antecedent of the test class, it will not be recognized as valid and will be replaced. This is not desirable, since tests should be replaced only when necessary.

Assuming that the frame stimuli in the existing test frame set form a subset of the frame stimuli in the test class antecedent, the problem of determining a set of test frames that satisfy a given criteria is NP-hard. A solution to this problem would also solve what Garey and Johnson [25] refer to as "[LO7] SATISFIABILITY OF BOOLEAN EXPRESSIONS." The solution would be to use the given Boolean expression as a test class antecedent and a criteria that requires at least one test frame, if any exist, the expression is satisfiable if and only if the set of test frames is non-empty. The binary decision diagram (BDD) [8] is a convenient tool for addressing this type of problem. The technique described here uses BDDs to perform test frame construction and selection. The strategy for generating test frame antecedents is:

1. Assign BDD variables to each frame stimulus.
2. Generate the set of prime implicants[4] for the antecedent of the test class.
3. Using the heuristic to be described in Appendix C, attempt to identify any existing or mandated valid test frames that can contribute to the coverage of the current test class. This forms the initial set of test frames.
4. Augment this set with other elements from the set of prime implicants to construct a set satisfying the desired coverage criterion.

5.6.1 Frame Stimuli

BDDs encode unquantified Boolean expressions. Quantifiers within the test class place a limit on the granularity of the terms which appear in test frames. To

[4]An implicant of a formula is a conjunction of variables or negated variables which imply the formula. An implicant is prime if it implies no other implicant. For example, A and $\neg B$ are prime implicants of $A \vee \neg B$. $A \wedge \neg B$ is an implicant, but is not prime because it implies at least one other implicant, e.g., $\neg B$.

obtain an unquantified expression from the test class antecedent, quantifiers are pushed inwards to be grouped as tightly as possible to the stimuli that they quantify. Existential quantifiers which are not blocked by universal quantifiers are then moved outside the implication, where they become universal quantifiers. This minimizes the number of quantifiers in the test class antecedent.

The theorems used for determining frame stimuli are:

$$\forall\, x.P \vee Q = (\forall\, x.P) \vee Q \qquad \exists\, x.P \wedge Q = (\exists\, x.P) \wedge Q$$
$$\forall\, x.Q \vee P = Q \vee (\forall\, x.P) \qquad \exists\, x.Q \wedge P = Q \wedge (\exists\, x.P)$$

$$\forall\, x.P \wedge Q = (\forall\, x.P) \wedge Q \qquad \exists\, x.P \vee Q = (\exists\, x.P) \vee Q$$
$$\forall\, x.Q \wedge P = Q \wedge (\forall\, x.P) \qquad \exists\, x.Q \vee P = Q \vee (\exists\, x.P)$$
$$\forall\, x.P \wedge M = (\forall\, x.P) \wedge (\forall\, x.M) \qquad \exists\, x.P \vee M = (\exists\, x.P) \vee (\exists\, x.M)$$

$$\forall\, x.\forall\, y.P(x, y) = \forall\, y.\forall\, x.P(x, y) \qquad \exists\, x.\exists\, y.P = \exists\, y.\exists\, x.P$$

$$(\exists\, x.P) \Rightarrow Q = \forall\, x.P \Rightarrow Q$$

where x is free in M, x and y are free in P, and x is not free in Q. Although the rules for swapping quantifiers could cause a rewrite system to diverge, they are only applied in a controlled manner. These rules are used to move quantifiers of specific variables to positions within the expression where they can be pushed inwards.

An illustration of this process is as follows:

$$\begin{aligned}
& (\forall\, x.\exists\, y.A(x) \wedge (B \vee C(y))) \Rightarrow R \\
= \;& ((\forall\, x.A(x)) \wedge (B \vee \exists\, y.C(y))) \Rightarrow R \\
= \;& (\exists\, y.(\forall\, x.A(x)) \wedge (B \vee C(y))) \Rightarrow R \\
= \;& \forall\, y.((\forall\, x.A(x)) \wedge (B \vee C(y))) \Rightarrow R.
\end{aligned}$$

Applying this process to the steam boiler test classes results in:

$$
\begin{aligned}
&\forall\, n, m, i. \\
&\quad (\forall\, n. \neg(\mathit{Level}\ n)) \vee \\
&\quad (\mathit{Level}\ n \wedge \mathit{Level}\ m \wedge \neg(n = m)) \vee \\
&\quad (\forall\, n. \neg(\mathit{Steam}\ n)) \vee \\
&\quad (\mathit{Steam}\ n \wedge \mathit{Steam}\ m \wedge \neg(n = m)) \vee \\
&\quad ((\mathit{PumpState}(i, \top) \wedge \mathit{PumpState}(i, \bot)) \vee \\
&\quad\quad (\neg(\mathit{PumpState}(i, \top)) \wedge \neg(\mathit{PumpState}(i, \bot)))) \vee \\
&\quad (\forall\, b. \neg(\mathit{PumpCtrState}(i, b))) \\
&\quad \Rightarrow \mathit{OutOfOrder}'
\end{aligned}
\tag{5.4}
$$

and

$$
\begin{aligned}
&\forall\, n_1, n_2. \\
&\quad (\mathit{Level}\ n_1) \wedge \\
&\quad (\forall\, n, m. \neg(\mathit{Level}\ n) \vee \neg(\mathit{Level}\ m) \vee (n = m)) \wedge \\
&\quad (\mathit{Steam}\ n_2) \wedge \\
&\quad (\forall\, n, m. \neg(\mathit{Steam}\ n) \vee \neg(\mathit{Steam}\ m) \vee (n = m)) \wedge \\
&\quad (\forall\, i. \neg(\mathit{PumpState}(i, \top)) \vee \neg(\mathit{PumpState}(i, \bot))) \wedge \\
&\quad (\forall\, i. \mathit{PumpState}(i, \top) \vee \mathit{PumpState}(i, \bot)) \wedge \\
&\quad (\forall\, i. \exists\, b. \mathit{PumpCtrState}(i, b)) \\
&\quad \Rightarrow \neg \mathit{OutOfOrder}'.
\end{aligned}
$$

A BDD representation is constructed by substituting a variable for each quantified subexpression and unquantified stimulus. The quantified subexpressions and unquantified stimuli represented by BDD variables are referred to as *frame stimuli.*

The antecedent of Equation (5.4) can be represented with the unquantified expression:

$$V_1 \vee (V_2 \wedge V_3 \wedge \neg E) \vee W_1 \vee (W_2 \wedge W_3 \wedge \neg E) \vee ((X \wedge Y) \vee (\neg X \wedge \neg Y)) \vee Z$$

where

$$\begin{array}{lcl@{\qquad\qquad}lcl} V_1 & = & \forall\, n.\neg(\mathit{Level}\ n) & W_1 & = & \forall\, n.\neg(\mathit{Steam}\ n) \\ V_2 & = & \mathit{Level}\ n & W_2 & = & \mathit{Steam}\ n \\ V_3 & = & \mathit{Level}\ m & W_3 & = & \mathit{Steam}\ m \end{array}$$

$$\begin{array}{lcl@{\qquad\qquad}lcl} X & = & \mathit{PumpState}(i, \top) & Z & = & \forall\, b.\neg(\mathit{PumpCtrState}(i, b)) \\ Y & = & \mathit{PumpState}(i, \bot) & E & = & (n = m) \end{array}$$

The set of prime implicants is then generated from the BDD representation of this expression. For this particular specification, Implicant, DNF, and Term Coverage, defined in Section 5.6.2 below, result in the same test frames. Test frames are constructed around the prime implicants, which can be seen in the following test frame antecedents:

$$(\forall\, n. \neg(\mathit{Level}\ n)) \qquad\qquad (\forall\, n. \neg(\mathit{Steam}\ n))$$
$$\Rightarrow \mathit{OutOfOrder}' \qquad\qquad \Rightarrow \mathit{OutOfOrder}'$$

$$\forall\, n, m. \mathit{Level}\ n \wedge \qquad\qquad \forall\, n, m. \mathit{Steam}\ n \wedge$$
$$\mathit{Level}\ m\ \wedge \neg(n = m) \qquad\qquad \mathit{Steam}\ m\ \wedge \neg(n = m)$$
$$\Rightarrow \mathit{OutOfOrder}' \qquad\qquad \Rightarrow \mathit{OutOfOrder}'$$

$$\forall\, i. \mathit{PumpState}(i, \top) \wedge \qquad\qquad \forall\, i. \neg(\mathit{PumpState}(i, \top)) \wedge$$
$$\mathit{PumpState}(i, \bot) \qquad\qquad \neg(\mathit{PumpState}(i, \bot))$$
$$\Rightarrow \mathit{OutOfOrder}' \qquad\qquad \Rightarrow \mathit{OutOfOrder}'$$

$$\forall\, i. (\forall\, b. \neg(\mathit{PumpCtrState}(i, b)))$$
$$\Rightarrow \mathit{OutOfOrder}'$$

Although quantifiers were used liberally throughout the specification, reasonable test frames could still be generated automatically. It is less tedious and less error-prone to manually derive test steps from these test frames than from the original specification.

5.6.2 Coverage Schemes

This section introduces the basics of coverage schemes which are algorithms for selecting test frames to satisfy the corresponding coverage criteria. The topic of coverage is examined in greater detail in Chapter 6.

A major concept of this thesis is that coverage of a test class by its test frames is described by relating the test frame antecedents to the antecedent of the test class.

A test frame is uniquely identified within a test class by its antecedent. In general, a coverage scheme is a function, C, from a set of possible test frame antecedents, I, to a subset, F, of I chosen by the coverage scheme, and a flag, r, which indicates whether F satisfies the coverage criterion. The coverage scheme builds F by repeatedly selecting test frame antecedents from the given set of possibilities, I, until this set of selections, F, satisfies the corresponding coverage criterion, or no more selections from I can make a further contribution to satisfying the coverage criterion.

A coverage scheme can be used to evaluate a given test frame set, T, by evaluating $C(Ant(T)) = (F, r)$, where Ant provides the set of antecedents of the given set of test frames. The redundant test frames are those represented in T but not in F. The completeness of T is given by r.

The author proposes the following terms for some fundamental coverage schemes:

1. **All points:** This is similar to the DNF of Dick and Faivre, where each test frame specifies the truth or falsehood of each of the frame stimuli from the test class stimulus expression.

2. **Implicant:** Test frames are produced for each prime implicant.

3. **DNF:** Test frames are produced for a subset of prime implicants. The disjunction of this subset corresponds to a DNF of the test class stimulus expression.

4. **Partition:** A subset of prime implicants is used to determine an implicant set which is similar to DNF coverage, but the implicants are pair-wise contradictory. There is no test step that will satisfy any two test frames.

5. **Term:** Test frames are produced for a subset of prime implicants such that each frame stimuli from the test class stimulus expression is present in at least one of the selected prime implicants. A precise mathematical definition of Term Coverage is given in Section 6.8.

The differences between these coverage schemes can be illustrated by considering the number of terms produced when applied to the expression in Figure 5.2. This figure shows the points where the expression is true (black dots), and compares the Karnaugh maps [40] corresponding to the coverage schemes defined above. Each bubble represents the antecedent of a test frame. One antecedent may cover several points. This occurs when the truth value of some variables is not specified. The coverage schemes produce 8, 5, 4, 4, and 3 test frames, respectively.

Comparison of coverage schemes applied to
$(\neg W \wedge \neg X \wedge \neg Y) \vee (\neg W \wedge \neg Y \wedge Z) \vee (X \wedge Y) \vee (W \wedge Y \wedge \neg Z)$

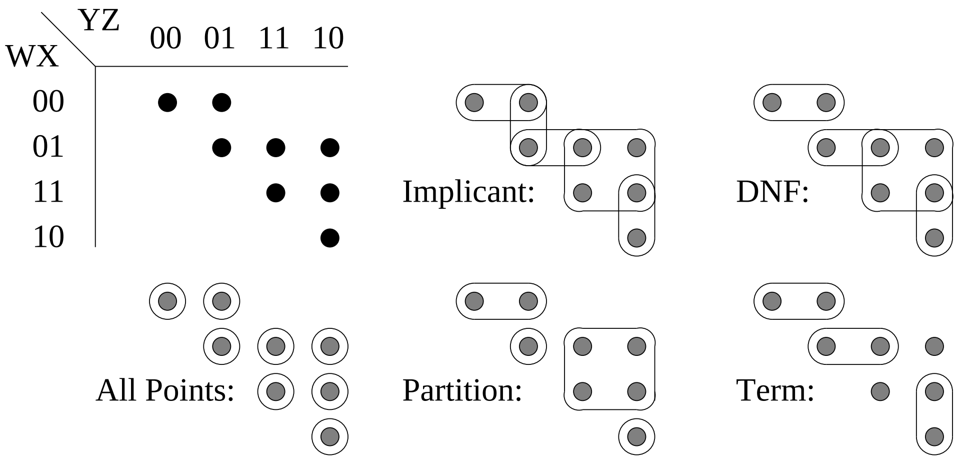

Figure 5.2: Coverage Schemes

Term coverage is of interest, since the size of the corresponding test frame set is linear with respect to the size of the test class rather than combinatorial, as are the other coverage schemes. Term coverage does not produce test frames that cover two of the eight all-points cases, $W \wedge X \wedge Y \wedge Z$ and $\neg W \wedge X \wedge Y \wedge \neg Z$. This is the compromise made in order to produce fewer tests in situations where it is appropriate to do so.

The steam boiler example used in this chapter focuses on issues of determining frame stimuli. In this example, there are several frame stimuli but few combinations of logical conjunction and disjunction (AND and OR). Hence, this example produces the same number of tests for either Term Coverage or DNF Coverage. Section 7.8.1 notes that for a more complex version of the steam boiler example, there are 22 test frames for Term Coverage, 47 for DNF Coverage, and 84 for Implicant Coverage.

5.7 Conclusion

This chapter has presented the fundamental algorithms that form the foundation of the discipline presented in this dissertation. This foundation is based on mathematical rules of logical manipulation which ensure that the algorithms are sound. The definitions of test classes, frame stimuli, and test frames form the basis of a nomenclature for naming coverage criteria. This nomenclature is extended in the next chapter.

Although rewrite rules are used in various contexts to produce test frames, the entire set of these rewrite rules is not confluent[5]. This implies that test frame production is more complex than blindly rewriting the specification using a confluent set of rewrite rules.

[5] A confluent set of rewrite rules is also Church-Rosser [9].

Chapter 6

Coverage Criteria

This chapter defines a nomenclature for naming coverage criteria with tuples of arguments to a test frame generation process. This process is an extension of the basic algorithms of Chapter 5. The parameters of the process establish the nomenclature for defining a wide range of specification-based coverage criteria.

6.1 Introduction

In this discipline of specification-based testing, a coverage criterion is named by specifying arguments to the test frame generation process. Although the coverage scheme is the fundamental component of a coverage criterion, there are additional parameters to test frame generation. These parameters are based on logically sound extensions to the algorithms presented in Chapter 5.

The first group of extensions focuses on aspects of the test class normal form. Derivation of the test class normal form can be produced in any of three variations. Each of these can be achieved with minor adjustments to a portion of the *TC* algorithm. In certain situations, it is possible to specialize a test class to eliminate non-determinism caused by disjunctions in the response expression. This is referred to as response-response resolution. One parameter affecting test classes indicates whether a "closed world" should be assumed or not. This is a common assumption made in specifications, and can significantly reduce the size of a specification.

Once the set of test classes has been determined, the granularity of frame stimuli can be addressed. In special circumstances, universally quantified frame stimuli can be broken down into unquantified components. Frame stimuli are the common components of both test classes and test frames. Coverage schemes, the algorithms for selecting test frames, are defined based on frame stimuli relationships. As examples, mathematical definitions of two variations of Term Coverage are given.

According to the basic algorithms, test frames are produced in their most general forms. However, they can also be specialized in order to differentiate when responses are due to particular stimuli.

An objective coverage criterion is defined in terms of arguments to this extended process. Comparisons of the effectiveness of these criteria are based on a partial order of coverage criteria.

Section 6.2 gives a definition of objective coverage criteria and some examples. The relative effectiveness of coverage criteria is examined in Section 6.3. Section 6.4 presents the test class variations. Response-response resolution is described in Section 6.5. Section 6.6 examines the effects of assuming a closed world. Section 6.7 describes the simplification of quantified frame stimuli. Two examples of mathematical definitions of coverage criteria are given in Section 6.8. Test frame differentiation is examined in Section 6.9.

6.2 Objective Definitions of Coverage Criteria

The extensions to the test frame generation process that support the nomenclature are described in later sections. Using the nomenclature, several parameters need to be specified to identify a particular specification-based coverage criteria:

1. Test class type: pure, detailed, or focused.
2. Response-response resolution: none, embellish, or eliminate.
3. Closed world or not.
4. Frame stimuli simplification: none, single, all, pairs, power set.
5. Selection scheme: All points, Implicant, DNF, Term.
6. Test frame style: base, differentiated.

It is likely that the above list will grow with the evolution of the discipline presented in this dissertation. Although the selection scheme generally has the most dramatic impact on test selection, each of these elements must be specified in order to define a particular coverage criteria.

This nomenclature can be used to specifically name a large number of different criteria. For example, using this nomenclature, a relatively small set of test frames can be specified using the criteria (focused, eliminate, not closed, no simplification, Term, base). A much more extensive notion of coverage which corresponds to a much larger set of test frames, depending on the specification, is named by (pure, embellish, closed, power set, all points, differentiated).

6.3 Relative Effectiveness

The *effectiveness* of a specification-based coverage criterion refers to its ability to produce a test set which uncovers discrepancies between the requirements and the implementation. Containment provides a simple means of comparing coverage criteria effectiveness. By considering coverage criteria as relations between specification expressions and sets of test frames, criterion A is more effective than criterion B when test frames produced by A imply those produced by B, i.e.,

$$\forall\, S, t_1, t_2.(A(S, t_1) \wedge B(S, t_2)) \Rightarrow (\bigwedge t_1 \Rightarrow \bigwedge t_2),$$

where $A(S, t)$ means that the set of test frames t satisfies criterion A for specification S. The sets of test frames that satisfy criteria A and B are represented by t_1 and t_2, respectively.

Using this means of comparison, the Implicant, All-points, and DNF Coverage criteria are equally effective, and each is more effective than Term Coverage. Another means of comparing coverage criteria is to compare the number of test frames produced. The principle advantage of Term Coverage is that it is less expensive, since it produces dramatically fewer tests while still including a test involving each frame stimulus.

Apart from this trivial definition of relative effectiveness, this thesis does not address the issue of determining which coverage criteria are more appropriate for specific testing objectives. Without a more satisfactory mathematical notion of effectiveness, the relative merits of coverage criteria will need to be determined on the basis of empirical study. It is likely that definitions of coverage criteria will include some domain specific elements.

6.4 Test Class Variations

The *TC* algorithm of Chapter 5 produces test classes in what is referred to as their *pure* form. This pure test class normal form is logically equivalent to the original specification. This is due to the fact that the *TC* algorithm is based on equivalences. There are two additional variations on this algorithm which involve slight changes to the *RewriteAnd* function.

6.4.1 Detailed

Some test engineers may require that the responses be as detailed as possible. For example, the pure test class normal form of the specification,

$$S_1 \Rightarrow (R_1 \wedge (S_2 \Rightarrow R_2)) \tag{6.1}$$

is:

$$(S_1 \Rightarrow R_1) \wedge (S_1 \wedge S_2 \Rightarrow R_2).$$

The test class $S_1 \wedge S_2 \Rightarrow R_2$ may be regarded by some as incomplete since S_1 will cause the response R_1 in addition to the response R_2. The desired test class, $S_1 \wedge S_2 \Rightarrow R_1 \wedge R_2$, is an example of the *detailed* form of test classes.

Detailed test classes are produced by augmenting the processing of conjunctions in *RewriteAnd* by using the equivalence:

$$\forall\, P, S, R. (\top \Rightarrow P) \wedge (S \Rightarrow R) = (\top \Rightarrow P) \wedge (S \Rightarrow P \wedge R). \tag{6.2}$$

As with pure test classes, the conjunction of detailed test classes derived from a specification is logically equivalent to the specification.

The detailed test class normal form of (6.1) is

$$(S_1 \Rightarrow R_1) \wedge ((S_1 \wedge S_2) \Rightarrow (R_1 \wedge R_2)).$$

6.4.2 Focused

In some testing situations, tests from the test class $S_1 \Rightarrow R_1$, above, may be deemed redundant, since R_1 can be observed in the test steps for the test frame $S_1 \wedge S_2 \Rightarrow R_1 \wedge R_2$. A *Focused* set of test classes which eliminates this type of test class is produced by augmenting the processing of conjunctions in *RewriteAnd* by using the inference

$$\forall\, P, S, R.(\top \Rightarrow P) \wedge (S \Rightarrow R) \Rightarrow (S \Rightarrow P \wedge R). \tag{6.3}$$

The conjunction of focused test classes is implied by the specification, but is not equivalent to it. This is due to the use of the inference (6.2), rather than the exclusive use of logical equivalences.

The focused test class normal form of (6.1) is

$$(S_1 \wedge S_2) \Rightarrow (R_1 \wedge R_2).$$

6.5 Resolving Non-Deterministic Test Classes

Response-response resolution refers to the process of eliminating certain kinds of non-determinism from test class response expressions where possible. Test class combinations of the form

$$(S_1 \Rightarrow \neg R_1) \wedge (S_2 \Rightarrow (R_1 \vee R_2))$$

can be used to derive additional test classes as follows:

$$\begin{aligned}
& (S_1 \Rightarrow \neg R_1) \wedge (S_2 \Rightarrow (R_1 \vee R_2)) \\
= \quad & (S_1 \Rightarrow \neg R_1) \wedge ((S_2 \wedge \neg R_1) \Rightarrow R_2) \\
= \quad & (S_1 \Rightarrow \neg R_1) \wedge (S_2 \Rightarrow (R_1 \vee R_2)) \wedge (S_1 \wedge S_2 \Rightarrow R_2) \\
\Rightarrow \quad & (S_1 \Rightarrow \neg R_1) \wedge (S_1 \wedge S_2 \Rightarrow R_2)
\end{aligned}$$

As indicated by the last two steps of the derivation above, the additional test classes can be used to embellish the original set, or the non-deterministic test classes can be eliminated. The elimination of non-deterministic test classes is similar to the production of focused test classes in that the resulting test classes are implied by the specification, rather than equivalent to it. Some test engineers may deem the test frames derived from the omitted test classes to be of no real value. For example, some test engineers may require tests that specify deterministic responses. Thus, there is no significant consequence in the loss of logical equality of the conjunction of the set of remaining test classes to the original specification.

6.6 Assuming a Closed World

A closed world, or complete knowledge, assumption [51] is common in many specifications. The assumption is that a given response can be produced only in those cases prescribed in the specification, and in no others. For example, assuming a closed world, $\neg A \wedge \neg B \Rightarrow \neg R$ is valid if the specification $(A \Rightarrow R) \wedge (B \Rightarrow R)$ is valid. This can be achieved by augmenting the test class normal form of a specification with the appropriate test classes, prior to test frame generation.

A closed world assumption can have a dramatic effect on the number of test frames produced from a specification. For example, the specification $((A \wedge B) \vee (C \wedge D) \vee (E \wedge F)) \Rightarrow R$ has three DNF test frames. However, the same specification in a closed world has 11 DNF test frames.

6.7 Simplifying Quantifiers

The simplification of quantified frame stimuli is performed during the determination of frame stimuli. Removing quantifiers produces simpler frame stimuli, which are easier to instantiate manually into test steps. This simplification assumes that the domain of the quantified variable is a set, and requires that this set be identified as either static or dynamic. Any element of the specification is *dynamic* if it can be different in different contexts of the specification. Any element is *static* if it is not dynamic, e.g., the set of natural numbers is a static element. For example, the expression $\forall\, x.P\ x$, where x has the type corresponding to the set of aircraft within an airspace, A, is interpreted as $\forall\, x \in A.P\ x$. Since there can be different numbers of aircraft within an airspace at any given time, A is dynamic.

The test class and test frame algorithms process specifications which may include quantification. Quantifiers in the specification often appear in test frame stimuli expressions, as illustrated in the example of Section 5.3. Without further processing, quantified frame stimuli would normally be addressed during the manual instantiation of test frames into test steps. To reduce the labour required for this task, it is beneficial to automatically process quantified frame stimuli where possible.

The following example illustrates where quantifiers can be simplified. The expression $(\exists! x.S\ x) \wedge w < 2 \Leftrightarrow R$, where R is the only response, produces the following term coverage test frames:

$\forall\, x.(S\ x \wedge (\forall\, x, y.\neg S\ x \vee \neg S\ y \vee (x = y)) \wedge w < 2 \Rightarrow R)$,

$(\forall\, x.\neg S\ x) \Rightarrow \neg R$,

$w \geq 2 \Rightarrow \neg R$, and

$\forall\, x, y.(S\ x \wedge S\ y \wedge x \neq y \Rightarrow \neg R)$.

Instantiating a test frame into a test step is the process of determining an instance of input variables which satisfies each frame stimulus. For an unquantified frame stimulus, such as $w \geq 2$, instantiation is simply a matter of selecting appropriate data values for the variables, e.g., $w = 2$. However, satisfying quantified frame stimuli, such as $\forall\, x, y. \neg S\ x \vee \neg S\ y \vee (x = y)$ and $\forall\, x. \neg S\ x$ above, can be quite complex, since the stimuli expression can be undecidable.

The test frame generation algorithms guarantee that the first quantifier of a quantified frame stimulus is universal.[1] A quantified variable is associated with a set of values. This set is either dynamic or static. Thus, there are three categories of quantified frame stimuli:

1. the quantified variable is associated with a static set and the frame stimulus contains no free variables that represent the system environment;
2. the quantified variable is associated with a static set and the frame stimulus contains a free environment variable; or
3. the quantified variable is associated with a dynamic set.

In the first category, the environment has no effect on the truth value of the frame stimulus which is either true or false, e.g., $\forall\, n. n^2 > n$. It is suspicious that a system would be required to produce a response depending on the truth or falsehood of a stated theorem. In such cases, it is likely that the source of the frame stimulus is incorrectly specified.

Frame stimuli from the second category express a property of the free variable. This is illustrated by the frame stimulus $\forall\, y. x \,\mathsf{mod}\, y \neq 0 \vee y = 1 \vee y = x$, which expresses the property that x is a prime number. When the static set associated

[1] If it were existential, it could be moved outside the antecedent of the test class to universally quantify the test class.

with the quantified variable is infinite, instances of frame stimuli for this category must be determined manually. When the static set associated with the quantified variable is finite, a frame stimulus of the form $\forall\, x \in \{x_i \mid 1 \leqslant i \leqslant n\}.P\, x$ can be simplified using the theorem,

$$\forall\, x \in \{x_i \mid 1 \leqslant i \leqslant n\}.P\, x = \bigwedge\{Px_i \mid 1 \leqslant i \leqslant n\},$$

where $\bigwedge(\{x\} \cup A) = x \wedge (\bigwedge A)$ and $\bigwedge \emptyset = \top$.

The third category is particularly interesting from a coverage point of view. In this case, the set associated with the quantified variable contains an arbitrary number of elements. For example, in the frame stimulus $\forall\, x \in$ Aircraft.Is_Taxiing $x \vee$ Is_Boarding x, the set Aircraft represents all the aircraft within the operating environment of the system. In the context of an air traffic control system, the contents of this set are constantly changing. For these frame stimuli, the question is: What instances of this set, e.g., Aircraft, should be used in test frames to ensure adequate coverage?

The frame stimulus

$$\forall\, x \in X.P_1\, x \vee P_2\, x \vee \ldots \vee P_n\, x,$$

can be satisfied by the singleton instance $X = \{c\}$, where c has one of the properties $P_i, 1 \leqslant i \leqslant n$. This is certainly a light notion of coverage. A more reasonable notion of coverage might be to conduct n tests, each one addressing a different P_i. Another alternative is to set $X = \{x_i \mid P_i\, x_i, 1 \leq i \leq n\}$, a single set of n elements, each of which satisfies at least one P_i. This would require one test.

The soundness of the above substitutions is assured by the theorems

$$(X = \{x\}) \wedge P_1 x \vee \ldots \vee P_n x \Rightarrow \forall\, x \in X.P_1 x \vee \ldots \vee P_n x \qquad (6.4)$$

$$(X = \{x_i \mid P_i\, x_i, 1 \leq i \leq n\}) \wedge P_1 x_1 \wedge \ldots \wedge P_n x_n \Rightarrow$$
$$\forall\, x \in X . P_1 x \vee \ldots \vee P_n x \tag{6.5}$$

where $x, x_1, \ldots, x_n$ are constants that have not yet been introduced into the specification, and reflect a particular instance of the type of the quantified variable. In terms of the test frame generation process, quantified frame stimulus simplification can be performed in at least three modes: none, single, or all, where single and all refer to the use of inferences (6.4) and (6.5), respectively. Another alternative is to combine these techniques and conduct $\binom{n}{2}$ tests where each test involves a pair of elements that satisfy distinct properties, i.e., the $\binom{n}{2}$ instances of X such that $X = \{(x, y)\}$ and $\exists\, i, j . 1 \leqslant i, j \leqslant n \wedge i \neq j \wedge P_i\, x \wedge P_j\, y$. A further, perhaps extreme, alternative is to conduct 2^n tests based on the power set of the P_i's.

6.8 Mathematical Definition of Term Coverage

The definition of Term Coverage expresses a relationship between frame stimuli within test frames and the frame stimuli of a test class normal form of the specification. The mathematical definition of Term Coverage follows.

The following definitions are made:

- Let $C_i, 1 \leqslant i \leqslant n$, represent the n test classes of specification Q, i.e., $Q = C_1 \wedge \ldots \wedge C_n$.
- Let c_i represent the test class antecedent of C_i.
- Let Conj(E) represent the set of conjuncts in an expression E.

Now, let $S(E)$ represent the set of frame stimuli in the test class normal form of an expression, E, i.e.,

$$S = \{s \mid \exists\, i.C_i \in \mathrm{Conj}(TC(E)) \wedge s \in FS(c_i)\},$$

where TC is the test class algorithm from Section 5.5 and $FS(c)$ represents the set of frame stimuli obtained from the test class antecedent, c, as determined by the procedure from Section 5.6.

Let f_{ik} represent the antecedent of the k^{th} test frame F_{ik} derived from C_i, i.e.,

$$\forall\, ik.(f_{ik} \Rightarrow c_i) \wedge \forall\, e.(e \Rightarrow c_i) \Rightarrow \mathrm{Conj}(e) \not\subset \mathrm{Conj}(f_{ik}). \tag{6.6}$$

Equation (6.6) states that F_{ik} is a valid test frame of test class C_i and f_{ik} is a prime implicant. The F_{ik} test frames satisfy Term Coverage of a specification, E, when:

$$\forall\, s \in S(E).\, \exists\, ik.s \in \mathrm{Conj}(f_{ik}). \tag{6.7}$$

An alternative variation of Term Coverage is where the coverage of the F_{ik} test frames is measured relative to each individual test class, rather than to the specification as a whole:

$$\forall\, i.C_i \in \mathrm{Conj}(TC(E)) \Rightarrow \forall\, s \in S(C_i).\, \exists\, k.s \in \mathrm{Conj}(f_{ik}). \tag{6.8}$$

6.9 Differentiated Test Frames

The test frames generated using the basic algorithm of Chapter 5 are referred to as base-style test frames. This style of test frame specifies the most general constraints on test frame stimulus expressions. For various reasons, it may be desirable to produce more specific test frames, such as the pair below from Section 2.5.

1. S and C1 and (not C2) $\Rightarrow$ R

2. S and (not C1) and C2 $\Rightarrow$ R

This section examines a method of producing test frames in a different style.

Differentiated test frames include additional constraints to ensure that there does not exist a test step which is an instance of more than one test frame for a test class. Differentiated test frames may be required to ensure that frame stimuli are tested in isolation.

For example, the test class $(A \vee B) \Rightarrow R$ has base test frames $A \Rightarrow R$ and $B \Rightarrow R$. The test step $(A \wedge B) \Rightarrow R$ is an instance of both test frames, and it may not be clear which stimulus was actually being tested.

Definition 5 *A set of test frames is* differentiated *when the antecedents of the test frames, $\hat{f}_i, 1 \leqslant i \leqslant n$, are pair-wise contradictory, i.e.,*

$$\forall\, i, j . 1 \leqslant i, j \leqslant n \wedge i \neq j \Rightarrow (\hat{f}_i \wedge \hat{f}_j = \perp).$$

Differentiation is performed after the coverage scheme has selected a set of test frames. A differentiated test frame, $\hat{F}_k$, can be computed from the corresponding base test frame, F_k. To correctly compute differentiated test frames when quantifiers are present requires the use of adjusted test frames. An *adjusted test frame* is a test frame where universal quantifiers exterior to the implication have been pushed into the antecedent, if possible. The antecedent, $\hat{f}_k$, for the differentiated test frame, F_k, can be computed from the test frame antecedent f_k and the $n-1$ adjusted test frame antecedents, $f_i, 1 \leqslant i \leqslant n$ and $i \neq k$, using the formula

$$\hat{f}_k = \text{ArbPI}(f_k \wedge \neg(f_1 \vee \ldots \vee f_{k-1} \vee f_{k+1} \vee \ldots \vee f_n)),$$

where ArbPI(e) represents an arbitrary, feasible prime implicant of frame stimuli from expression e.

While this technique ensures that frame stimuli can be tested in isolation, there are two disadvantages to differentiated test frames. Since this method of differentiation involves an arbitrary choice from a set of alternatives, it is possible that a test frame generator may make a choice other than that desired by a test engineer. In addition, differentiation involves computing prime implicants and selecting a feasible one. Thus, when selecting the representative differentiated test frame, simplification and infeasibility checking will also need to be performed and may be a prohibitively expensive computation. Simplification and infeasibility are examined further in Section 7.4.2.

Differentiated test frames are similar to Ammann and Offutt's base-choice coverage [2]. Ammann and Offutt's each-choice-used coverage is similar to Term Coverage (6.7), with the difference that the tests are based on a partitioning of the input domain alone, rather than on test classes which partition the stimulus-response behaviours of the system. Base-choice coverage requires specifying a base input in addition to a system behaviour. Test inputs are selected by negating one predicate that describes the base input.

A test class approach has the advantage that test classes correspond to base behaviours associated with the base inputs of Ammann and Offutt. Although differentiated test frames are based on a single behaviour, they produce tests similar to those satisfying base-choice coverage. Thus, a base behaviour does not need to be specified in order to produce base-choice-like tests.

For example, the differentiated test frames of $A \wedge B \wedge C \Leftrightarrow R$ are:

$$A \wedge B \wedge C \Rightarrow R$$

$$\neg A \wedge B \wedge C \Rightarrow \neg R$$

$$A \wedge \neg B \wedge C \Rightarrow \neg R$$

$$A \wedge B \wedge \neg C \Rightarrow \neg R$$

The test frames with response $\neg R$ correspond to those obtained by base-choice coverage that uses the antecedent $A \wedge B \wedge C$ as the base input. With differentiated test frames, however, the latter three test frames follow directly from the test class $\neg A \vee \neg B \vee \neg C \Rightarrow R$, and are not based on any other behaviour.

The differentiated version of the test frame

$$\forall\, i.PumpState(i, \top)\ \wedge PumpState(i, \bot) \Rightarrow OutOfOrder'$$

from Section 5.6.1 is:

$$\begin{aligned}
&\forall\, i, n_1, n_2.\\
&\quad PumpState(i, \top)\ \wedge PumpState(i, \bot) \wedge\\
&\quad Steam\ n_2 \wedge\\
&\quad (\forall\, p.\, \exists\, b.PumpCtrState(p, b)) \wedge\\
&\quad (\forall\, p.PumpState(p, \top) \vee PumpState(p, \bot)) \wedge\\
&\quad (\forall\, x.\neg(Level\ x) \vee \forall\, y.\neg(Level\ y) \vee (x = y)) \wedge\\
&\quad (\forall\, x.\, \forall\, y.(x = y) \vee \neg(Steam\ x) \vee \neg(Steam\ y)) \wedge\\
&\quad Level\ n_1\\
&\quad \Rightarrow OutOfOrder'
\end{aligned}$$

6.10 Summary

This chapter has defined extensions to the test class and test frame algorithms of Chapter 5. Parameters to these extensions form the nomenclature for naming coverage criteria for sets of test frames.

Chapter 7

Formal Specification-Based Testing

This chapter describes an application of the discipline of Chapters 5 and 6 to general formal specification-based testing. It defines a general test frame generation process that can be applied to a wide range of formal specifications. It also provides details of the design of a particular implementation of this process.

7.1 Introduction

Although the discipline presented in this dissertation is designed to be applied to system-level requirements specifications, the generality of this discipline allows it to be applied to a wide variety of formal specifications. Applying this discipline to a formal specification assumes:

- that stimuli can be distinguished from responses by some means, and
- the specification language can be founded on a logic that is consistent with

the logical inferences used in the algorithms of Chapter 5.

The *general test frame generation process* is based on a test frame generator that implements the algorithms of Chapters 5 and 6. The test frame generator takes a formal specification, a coverage criterion as defined in Chapter 6, user-mandated tests, existing test frames, and specified domain knowledge, and produces a set of test frames that satisfies the given criterion.

Many of the details required to implement such a test frame generator were given in Chapters 5 and 6. This chapter provides process details for:

- the iterative application of the general test frame generation process which allows test frames to be generated for a specification that cannot be processed within available memory or time limits, and
- the types of domain knowledge applicable to this process, how domain knowledge can be formalized, and a general decision procedure for applying this knowledge.

This chapter also provides the following details of one possible implementation:

- a rewrite system used in order to increase the assurance that logical manipulations carried out by the the test frame generator are sound,
- techniques for distinguishing between stimuli and responses, and
- algorithms for three of the coverage schemes defined in Section 5.6.2.

Section 7.2 provides an overview of the general test frame generation process. A method for processing large and logically complex specifications is described in Section 7.3. Section 7.4 describes how this process makes use of domain knowledge.

The remainder of this chapter focuses on aspects of one possible implementation of a test frame generator. The rewrite system is described in Section 7.5. Techniques for distinguishing stimuli from responses are described in Section 7.6. Section 7.7 outlines three algorithms for implementing coverage schemes. Examples of the application of a general test frame generation tool to a portion of a formal specification from the literature [58] and to another specification with a complex logical structure are examined in Section 7.8.

7.2 Process Overview

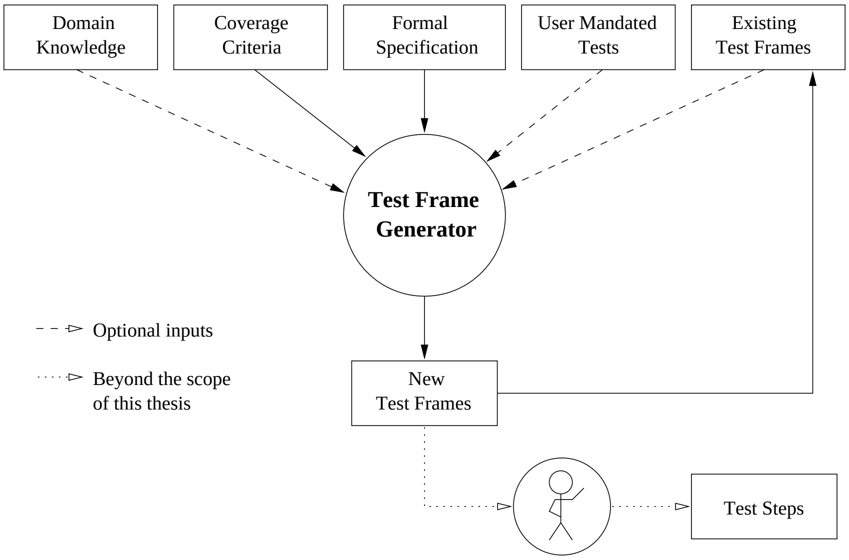

Figure 7.1: Automated Test Frame Generation

Figure 7.1 illustrates a process based on the discipline of specification-based test frame derivation presented in this dissertation. This process automatically generates test frames from a formal specification. The required inputs are the formal specification and the coverage criteria. Optional inputs are domain knowledge, user mandated tests, and existing test frames. As stated in Section 1.2, the selection of test data to derive test steps from test frames is outside the scope of this thesis.

The formal specification is assumed to be a logical expression relating stimuli and responses. Uses of existential quantification in the form of Equation (5.3) and demonic specifications are flagged and rejected during test class generation. These must be corrected by the specification author. The selected coverage criterion determines the precise relationship between the test frames to be produced and the given specification.

The input labeled "Domain Knowledge" in Figure 7.1 describes logical relationships amongst stimuli and amongst responses separately. This domain knowledge can be selectively provided by the user to control three aspects of test frame production:

1. the level of abstraction expressed in the test frames;
2. the elimination of infeasible test frames; and
3. the simplification of those test frames that are feasible.

"User Mandated Tests" provide the test engineer with the option of directly specifying some of the test frames to be included in the output. User-mandated tests are specified as test frames that are either fully or partially instantiated. User-mandated tests are not simply appended to the automatically generated test frames. Rather, the test frame generator integrates the user-mandated tests to reduce the generation of redundant, or partially redundant, test frames.

Software requirements often change during the development of a system. When requirements change, it is highly desirable to limit the impact of the changes on existing sets of test frames. For this reason, the user may optionally provide the set of "existing test frames" as input to the test frame generator. The test frame generator attempts to limit the number of arbitrary differences between the new and existing sets of test frames. It will also flag test frames in the previous set which are no longer implied by the specification.

The integration of user mandated tests and existing test frames is an instance of the Delta Problem presented in Section 4.6. This capability has not been implemented, but a heuristic algorithm for this intractable problem is given in Appendix C.

7.3 Tackling Complex Specifications

Automatically generating test frames for large specifications can be impractical, typically due to the amount of time required for the computation. In situations where limits on time and memory resources are exceeded, the specification can be processed iteratively as follows:

1. Limit the amount of detail in the specification in order to provide a more abstract view of the specification. This can be accomplished by instructing the test frame generator not to expand specific terms in the specification by their definitions during the derivation of test classes. In some situations, it may be necessary to limit detail by defining complex portions of the requirements as abstract terms, then suppressing the expansion of these abstract terms.

2. Generate test frames from the abstract view of the specification.

3. Use each test frame containing an abstract term combined with the definition of the abstract term as the specification for the next input to the test frame

generator. When using the Term Coverage scheme, only a single test frame for each abstract term is required.

4. Repeat steps 1 to 3 until test frames no longer contain abstract terms.

5. During data selection, when instantiating an abstract term, choose one instance for that term.

This iterative approach was used in preparing the examples presented in Sections 7.8.2, 8.5.1, and 8.5.2.

In many situations it may be desirable to use the iterative approach above, but use different coverage criteria at the various levels of abstraction. This provides test engineers with another means of control.

7.4 Formalizing Domain Knowledge

Domain knowledge encompasses a number of facts that can be used for different purposes in the test frame derivation process. Some of this domain knowledge expresses the interaction between the environment and the system by defining translations between the conditions used to describe the environment, and those used to specify the system requirements. In this dissertation, this type of domain knowledge can be expressed via *elaboration*. Elaboration can be used to ensure that test frames are composed of terms at the appropriate level of abstraction for testing purposes.

Other domain knowledge expresses condition dependencies that must be taken into account to disregard infeasible test frames and simplify those feasible test frames that are selected by the coverage scheme. This domain knowledge is expressed as theorems about mutually exclusive conditions, those forming partial orders, and those that represent states.

7.4.1 Elaboration

In this dissertation, elaboration refers to a mechanism for expanding stimuli and responses in the requirements into other combinations of stimuli and responses, respectively. This addresses some of the specification forms introduced in Section 2.5. These relationships may be part of the domain knowledge supplementing the requirements specification. They may also be parts of the requirements that express relationships between different levels of abstraction of the stimuli and responses. Elaboration allows test engineers to use a more detailed level of abstraction to describe tests, if necessary.

For example, tests may need to be expressed in terms of the Computer-Human Interface, which may be specified separately from the system requirements. The advantage of this type of elaboration is that it uses a supplement to the requirements specification. This ensures that constraints on the terminology used in testing do not affect the level of abstraction expressed in the system requirements.

There are two mechanisms for elaboration: definition and implication. Definitions correspond to rewrite axioms of the form:

$$\forall\, x.A(x) = E(x),$$

where A is either a stimulus or response predicate, and E is any predicate logic expression. As the TC algorithm computes test classes, defined terms are expanded according to their definitions.

Implication relationships amongst stimuli that do not involve a response, and similar relationships amongst responses that do not involve stimuli, are expressed as axioms of the following forms:

$$\forall\, x.E_S(x) \Rightarrow S(x)$$

$$\forall x. R(x) \Rightarrow E_R(x)$$

where $S(x)$ is a stimulus, $E_S(x)$ is a stimulus expression, $R(x)$ is a response, and $E_R(x)$ is a response expression.

Implications formed during the production of test classes are referred to as *intermediate test classes.* When a stimulus is first formed into an intermediate test class by the test class algorithm, it has the form $S(a) \Rightarrow \bot$, as described in Section 5.5. When this intermediate test class is formed, any relevant elaboration axioms are used to form the equivalent intermediate test class, $S(a) \vee E_S(a) \Rightarrow \bot$. The original stimulus, $S(a)$, is retained in the antecedent to ensure that the test class normal form is logically equivalent to the original specification. Similarly, an intermediate test class for a response, $\top \Rightarrow R(a)$, is replaced with the equivalent intermediate test class, $\top \Rightarrow R(a) \wedge E_R(a)$.

7.4.2 Simplification and Infeasibility

Domain knowledge involving condition dependencies can be provided as a supplement to the requirements. These are used during test frame selection to disregard infeasible test frames, and to simplify the antecedents of selected test frames. For the purpose of identifying infeasible test frames, it is necessary to identify the logic dependencies between conditions. The system-level specifications examined during the research for this thesis contain relatively few dependencies of this sort between conditions. It is likely that this is due to the system-level descriptions of stimuli, which are more abstract than the detailed descriptions that might be found in unit-level specifications. This motivates the use of axiom schemata to define dependencies, rather than requiring some underlying formal model to support these schemata as theorems.

In the general test frame generation process, known dependencies between conditions are specified using any of three axiom schemata:

1. $\forall\, x.G \Rightarrow \mathrm{MutEx}[P_1\, x;\; P_2\, x;\; \ldots P_n\, x]$,

2. $\forall\, x.G \Rightarrow \mathrm{Subsm}[P_1\, x;\; P_2\, x;\; \ldots P_n\, x]$, and

3. $\forall\, x.G \Rightarrow \mathrm{States}[P_1\, x;\; P_2\, x;\; \ldots P_n\, x]$.

These provide a means of defining condition dependencies. The MutEx form is used to define dependencies between mutually exclusive conditions. Conditions that form partial orders can be defined using Subsm. The States form defines conditions that represent a set of system states. The symbol G represents an optional guard which can refer to any of the quantified variables from the vector x. The guard provides a means of converting the dependency into a standard domain for which the test frame generator has a decision procedure. As an example of defining a partial order, assuming a decision procedure for simple arithmetic, theorem schema $\forall\, x, y.x > y \Rightarrow \mathrm{Subsm}[P\, x;\; P\, y]$ allows the test frame generator to simplify $P\,1 \wedge P\,2$ to $P\,1$.

The axioms defined by these schemata are given below:

$$
\begin{aligned}
&\forall\, x.G \Rightarrow \mathrm{MutEx}[P_1\, x;\; P_2\, x;\; \ldots P_n\, x] \vdash \\
&\quad \forall\, x, i, j.1 \leqslant i \leqslant n \wedge 1 \leqslant j \leqslant n \wedge i \neq j \wedge G \Rightarrow \\
&\qquad (P_i\, x \wedge P_j\, x = \bot) \wedge \\
&\qquad (P_i\, x \Rightarrow \neg P_j\, x)
\end{aligned}
$$

$$\forall\, x.G \Rightarrow \mathrm{Subsm}[P_1\, x;\;\; P_2\, x;\;\; \ldots P_n\, x] \vdash$$
$$\forall\, x, i, j.1 \leqslant i < j \leqslant n \wedge G \Rightarrow$$
$$(P_j\, x \Rightarrow P_i\, x) \wedge$$
$$(\neg\, P_i\, x \Rightarrow \neg\, P_j\, x)$$

$$\forall\, x.G \Rightarrow \mathrm{States}[P_1\, x;\;\; P_2\, x;\;\; \ldots P_n\, x] \vdash$$
$$\forall\, x, i, j.1 \leqslant i \leqslant n \wedge 1 \leqslant j \leqslant n \wedge i \neq j \wedge G \Rightarrow$$
$$(P_i\, x \wedge P_j\, x = \bot) \wedge$$
$$(P_i\, x \Rightarrow \neg P_j\, x) \wedge$$
$$(\neg P_1\, x \wedge \ldots \wedge \neg P_n\, x = \bot)$$

Dependencies between predicates, such as "Is_In_Canada," can be specified as

$$\forall\, x.\mathrm{MutEx}[\mathrm{Is\_In\_Canada}\, x;\;\; \mathrm{Is\_In\_USA}\, x;\;\; \mathrm{Is\_In\_Mexico}\, x],$$

$$\forall\, x.\mathrm{Subsm}[\mathrm{Is\_In\_Canada}\, x;\;\; \mathrm{Is\_In\_BC}\, x;\;\; \mathrm{Is\_Over\_Vancouver}\, x],$$

and

$$\forall\, x.\mathrm{States}[\mathrm{Is\_In\_Canada}\, x;\;\; \mathrm{Is\_Outside\_Canada}\, x].$$

The theorems are applied to the conjunctions of frame stimuli found in prime implicants corresponding to potential test frames.

In addition to condition dependencies, a confluent set of rewrite rules can also be specified as axioms. As an example of these techniques, reasoning about conjunctions of linear inequalities can be specified using the following rewrite rules:

$$\forall\, a, b.\neg(a < b) = b \leqslant a$$
$$\forall\, a, b.\neg(a > b) = a \leqslant b$$

$$\forall\, a, b. \neg(a \leqslant b) = b < a$$

$$\forall\, a, b. \neg(a \geqslant b) = a < b$$

$$\forall\, a, b. a > b = b < a$$

$$\forall\, a, b. a \geqslant b = b \leqslant a$$

$$\forall\, a, b. a \leqslant a = \top$$

$$\forall\, a, b. a < a = \bot$$

together with the following dependencies:

$$\forall\, x, y. \mathrm{Subsm}[x \leqslant y;\; x < y] \tag{7.1}$$

$$\forall\, x, y, z. (y < z) \Rightarrow \mathrm{Subsm}[x \leqslant z;\; x < z;\; x \leqslant y;\; x < y] \tag{7.2}$$

$$\forall\, x, y, z. (z < y) \Rightarrow \mathrm{Subsm}[z \leqslant x;\; z < x;\; y \leqslant x;\; y < x] \tag{7.3}$$

$$\forall\, x, y, z. (y \leqslant z) \Rightarrow \mathrm{MutEx}[x < y;\; z < x] \tag{7.4}$$

$$\forall\, x, y, z. (y < z) \Rightarrow \mathrm{MutEx}[x \leqslant y;\; z \leqslant x] \tag{7.5}$$

$$\forall\, x, y, z. (y \leqslant z) \Rightarrow \mathrm{MutEx}[x \leqslant y;\; z < x] \tag{7.6}$$

$$\forall\, x, y, z. (y \geqslant z) \Rightarrow \mathrm{MutEx}[y \leqslant x;\; x < z] \tag{7.7}$$

As a simple example, the conjunction $x \leqslant 0 \wedge x > 1$ is found to be infeasible as follows. Since rewrite axioms are applied during the determination of the test class normal form, $x > 1$ will be rewritten to $1 < x$ before a coverage scheme subjects the conjunction to feasibility analysis. Theorem schema (7.6) produces a match where the instantiation of the guard is $0 \leqslant 1$, which is resolved to $\top$ by a built-in decision procedure for simple arithmetic. Thus, it can be concluded that $x \leqslant 0 \wedge x > 1 = \bot$ and the corresponding test frame is infeasible. Similarly, $x < 3 \wedge x \leqslant 4$ produces a match in theorem schema (7.2) in the first and fourth positions of the list with guard $3 < 4$. Matches at other list positions do not allow the guard to be reduced to $\top$ by the decision procedure. Thus, this conjunction is simplified to $x < 3$.

This approach has certain limitations. To handle situations where the condition dependencies within test frames can be complex, it may be more efficient to provide a domain-specific decision procedure similar to the built-in decision procedure for arithmetic. For example, although the complex contradiction $a < b \wedge b < c \wedge c < a$ could be deduced by matching the guard of theorem schema (7.4) with $c < a$ in theorem schema (7.1), this type of reasoning is expensive to compute within the framework presented here. However, this research suggests that condition dependencies at the system-level typically involve pairs of conditions, rather than an interaction between three or more conditions.

In situations where there are relatively few dependencies between conditions, such as system-level requirements, condition dependencies can be addressed by specifying the theorems that an underlying model should support. The theorem schema forms MutEx, Subsm, and States allow a reasonably concise means of specifying these theorems. This approach tends to work well in the context of the system-level requirements specifications addressed by this thesis, since the dependencies between conditions can be expressed with relatively few axiom schemata. It is not necessary to document dependencies between every pair of conditions within the specification. It is necessary only to document those dependencies for frame stimuli which appear within the same test class antecedent.

7.5 Rewrite System

To increase the reliability of the test frame generator, a rewrite system is used to perform logical manipulations. The rewrite system described in this section differs from some well-known rewrite systems, such as the one found in HOL [28]. For performance reasons, the prototype test frame generator does not use the rewrite system during portions of simplification and infeasibility processing. The rewrite system assumes the correctness of each of the rewrite rules provided. To increase the assurance of correctness of the rewrite rules used in this dissertation, a HOL version of each rule was proved to be a theorem using the HOL system.

Rewrite rules are stated as universally quantified equalities, e.g., $\forall\, x.E_1(x) = E_2(x)$, where x is a vector of variables. For rules specifying rewrites involving quantifiers, the system assumes the following rules:

1. variable capture is avoided using alpha conversion; and

2. if variable release occurs, the rewrite fails.

The concept of *variable release* is the opposite of variable capture. During rewriting, if a variable is quantified in an expression matching the left-hand side of the rewrite rule and is unquantified in the corresponding instance of the right-hand side, variable release has occurred. For example, applying $\forall\, P, Q.(\forall\, x.P \vee Q) = ((\forall\, x.P) \vee Q)$ to $\forall\, x.f\, x \vee y$ is valid. However, applying the same rule to $\forall\, x.f\, x \vee g\, x$ is invalid because the x of $g\ x$ is released, i.e., x has become unquantified because it was free in Q. Rewrite rules requiring conditions on free variables can often be stated in terms of variable release.

By failing rewrites in which variable release occurs, the rewrite system allows the specification of rewrite rules such as:

$$\forall\, P, Q.(\forall\, x.P \vee Q) = (\forall\, x.P) \vee Q$$

$$\forall\, P, Q.(\forall\, x.P \vee Q) = P \vee (\forall\, x.Q)$$

$$\forall\, P, Q.(\forall\, x.P \wedge Q) = (\forall\, x.P) \wedge Q$$

$$\forall\, P, Q.(\forall\, x.P \wedge Q) = P \wedge (\forall\, x.Q)$$

$$\forall P, Q.(\forall x.P \wedge Q) = (\forall x.P) \wedge (\forall x.Q)$$

$$\forall P.(\forall x.P) = P$$

The last rule specifies that a universal quantifier can be removed if the quantified variable is not free in the expression, P.

This type of quantifier manipulation is not performed in HOL by general theorems used as rewrite rules, as above. Instead, it is performed using functions called *conversions*, which produce a theorem for the specific context only if such a theorem exists. The rewrite system described in this section is a simpler approach to rewriting, which does not require the specification of converters.

The rewrite system also recognizes alpha equivalence, e.g., $(\lambda x.E(x)) = \lambda a.E(a)$. These capabilities allow most of the logical manipulation done by the test frame generator to be performed by the rewrite system.

7.6 Distinguishing Stimuli and Responses

The algorithms in Chapter 5 rely on the distinction of stimuli from responses, but precisely how this is done has not yet been presented. There are two primary means of distinguishing stimuli and responses. The first is through the literal used to name the predicate. The prototype test frame generator described in this dissertation uses this approach, and assumes that a literal beginning with a lower case letter indicates a response predicate, unless a directive specifically labels a literal as referring to either a stimulus or a response. This technique has been found to be adequate for system-level requirements-based testing, because the vocabulary used to specify responses is usually different from that of specifying stimuli.

An alternative is to base the distinction on whether variables within predicate arguments refer to the state of the system at the time the stimulus occurs, or whether they refer to the state at the time the system responds. For example, in Z a prime (') is used to distinguish post-operation values from pre-operation values. Thus, the specification $(z \neq g(x,5)) \vee (z' = g(x,10))$ has the test class normal form $(z = g(x,5)) \Rightarrow (z' = g(x,10))$. In this example, $z = g(x,5)$ is a stimulus because it does not refer to the state of the system at the time of the response. The presence of z' indicates that $z' = g(x,10)$ is a response. In this system of distinguishing stimuli and responses, the same predicate can appear as either a stimulus or a response, e.g., the predicate $\lambda\, a, b, c.c = g(a,b)$.

This latter approach may prove quite useful in situations where the same predicate is used to express different relationships within a specification. For example, assuming *CorrectForm* is defined, the specification

$$CorrectForm(i, f')$$

can be used to generate test frames for filling out a form correctly, while

$$\neg CorrectForm(i, f) \Leftrightarrow FlagError'$$

can be used to generate test frames for flagging errors when a given form is filled out incorrectly.

7.7 Algorithms for Coverage Schemes

This section describes algorithms to implement the Implicant, DNF, and Term coverage test frame selection schemes and examines the effect of infeasible test frames on these algorithms. Each of these algorithms selects members of a set of prime implicants which correspond to the antecedents of test frames. A test frame is constructed from a prime implicant and its corresponding test class. Thus, it is sufficient to describe coverage scheme algorithms in terms of selecting certain prime implicants of a Boolean expression. The problem of finding a minimal set of prime implicants that satisfies the given coverage is NP-hard in each case. A solution to this problem would also solve what Garey and Johnson refer to as "[SP5] MINIMUM COVER" [25]. To select a set of test frames with the desired coverage in polynomial time, each algorithm abandons the minimal set but attempts to keep the selected set as small as possible.

A set of prime implicants can be generated by constructing a disjunctive normal form of the Boolean expression and using Strzemecki's algorithm [63] for producing the prime implicants.

7.7.1 Implicant Coverage

One algorithm for an implicant coverage scheme simply uses the general decision procedure of Section 7.4.2 to eliminate any infeasible test frames, then simplifies those that remain.

Although infeasible test frames are theorems of the specification, they have no value as descriptions of tests because the stimulus can never be achieved. For example, one test frame of the test class $x \in \{1, 2\} \wedge x < 2 \Rightarrow r$ is $(x = 2) \wedge x < 2 \Rightarrow r$. This is an infeasible test frame; it does not describe a test where the specified system can be forced to produce r to be consistent with its specification. Infeasible test frames are common in non-trivial specifications, and do not necessarily indicate the presence of specification errors.

7.7.2 DNF Coverage

This selection scheme selects a set of prime implicants that represents a disjunctive normal form of the original logical expression. Since it is possible for a logical expression to have more than one disjunctive normal form, the algorithm for this selection scheme attempts to minimize the set by avoiding disjuncts that overlap where possible.

The algorithm proceeds as follows:

1. Select a most general prime implicant that does not overlap the set already selected, i.e., $\bigvee S \wedge p = \bot$ for a set S of already selected prime implicants and unselected prime implicant p. A most general prime implicant is one with the fewest frame stimuli, i.e., the shortest conjunction.

2. The selected prime implicant is tested to ensure that it is feasible in the context of specified condition dependencies. Any infeasible prime implicants are discarded from the selected set.

3. Repeat steps 1 and 2 until the disjunction represented by the set is logically equivalent to the original logical expression, or no more prime implicants are available that fit the description in step 1.

4. To fill in any gaps, repeatedly select feasible most general prime implicants not implied by those already selected, i.e., $\neg(\bigvee S \Rightarrow p)$, until logical equality with the original expression is achieved, or no other such prime implicant, p, remains.

5. Simplify the selected set of prime implicants. This step involves the use of decision procedures, such as one for simple arithmetic, together with defined

condition dependencies.

7.7.3 Term Coverage

The algorithm for Term Coverage selects prime implicants that cover as many frame stimuli as possible. The algorithm is as follows:

1. Select a prime implicant that contains the most frame stimuli that are not yet represented in the selection set.

2. The selected prime implicant is tested to ensure that it is feasible. Any infeasible prime implicants are discarded from the selected set.

3. Repeat steps 1 and 2 until no other prime implicants contain frame stimuli that are not represented in the selected set, or no unselected prime implicants remain.

4. Simplify the selected set of prime implicants.

7.7.4 Infeasible Test Frames and Coverage Schemes

The determination of an infeasible prime implicant raises an interesting issue. Should the fact that a prime implicant is infeasible be incorporated into the original logical expression? In other words, when an infeasible prime implicant, p, is found in a logical expression, E, should the selection algorithm be restarted with the new set of prime implicants of the logical expression $E' = E \wedge \neg p$?

This would ensure that, based on the given condition dependencies, no infeasible test steps could be derived from the test frames produced. This is certainly a desirable property. However, prime implicants are costly to compute, hence this is not generally a feasible approach. Furthermore, it is assumed that the given domain knowledge expresses the common sense of the test engineers. If an infeasible instance of a test frame did exist, the test engineer would not choose this instance by using their common sense. Thus, it is not critical for the test frame generator to do more with infeasible prime implicants than discard them.

It is also possible that a selection algorithm cannot satisfy the corresponding coverage scheme due to discarded infeasible prime implicants. This is a valid situation, and does not imply that the selected set is deficient.

7.8 Examples

This section presents examples of applying the general process described in this chapter to specifications written by other authors. The specification notation and the test frame generator described in this section are merely examples of a parseable notation and a particular implementation, respectively.

7.8.1 Steam Boiler

The following example is a more detailed S [39] translation of a portion of Schinagl's VDM [37] style RSL [27] steam boiler control specification [58]. This example illustrates the application of the general test frame generation process to a specification from the literature. The specification problem is to formally specify requirements for a control system responsible for maintaining the correct level of water in a boiler attached to a steam-driven turbine. One of the requirements of the system is to identify whether or not any inconsistencies exist in the sensor readings.

The specification below is interleaved with descriptions of points of interest. Since S is an ASCII-based specification language, the words `Exists_unique,` exists, forall, and In replace the symbols $\exists!, \exists, \forall$, and $\in$, respectively. The S expression \x.E is the ASCII version of the lambda calculus abstraction $\lambda x.E$.

```
%include startup.s


(:t) Exists_unique (P:t -> bool) :=
    (exists v.P v)
        /\ (forall v1.forall v2.P v1 /\ P v2 ==> (v1 = v2));


inmess_ok : bool;
```

The variable inmess_ok is the message consistency indicator. Since predicate names beginning with a lower case letter indicate responses, this is the only response predicate in this specification.

```
:PUMP;
:STATE;


:message :=
    PumpState :(PUMP # STATE)
  | PumpCtrState :(PUMP # STATE)
  | Level :num
  | Steam :num
  | SteamBoilerWaiting
  | PhysicalUnitsReady
  | PumpRep :PUMP
  | PumpCtrRep :PUMP
  | PumpFlrAck :PUMP
  | PumpCtrFlrAck :PUMP
  | LevelRep
```

```
  | SteamRep
  | LevelFlrAck
  | SteamFlrAck;

InMess : (message)set;
```

The type `message` represents the various messages that can be received by the boiler control unit. `InMess` represents the set of messages received.

```
Waiting,Ready : bool;
States [Waiting; Ready];

:MODE;
Working,Repairing,Broken : MODE;
forall P.States [P Working; P Repairing; P Broken];
```

The above portion of the specification defines domain knowledge for the states `Waiting` and `Reading` along with `Working, Broken,` and `Repairing`.

```
Mst,Pst : PUMP -> MODE -> bool;
Qst,Vst : MODE -> bool;
```

`Mst p m` indicates that the boiler control believes that the control unit for pump `p` is in mode `m`. `Pst p m` indicates that the boiler control believes that pump `p` is in mode `m`.[1] `Qst m` indicates that the boiler control believes that the water level indicator is in mode `m`. `Vst m` indicates that the boiler control believes that the steam indicator is in mode `m`.

[1]The original specification used the condition `Pump.pst(p) = Pump.repairing` to express the same semantics as `Pst p Repairing`. This translation was performed to demonstrate the use of state information.

```
MaxWater : num;
MaxSteam : num;

SetInMessOK :=
    inmess_ok <=>
        (forall p.
            (Exists_unique (\s.PumpState(p, s) In InMess)) /\
            (Exists_unique (\s.PumpCtrState(p, s) In InMess))) /\
        (Exists_unique (\l.Level l In InMess)) /\
        (select l.Level l In InMess) <= MaxWater /\
        (Exists_unique (\l.Steam l In InMess)) /\
        (select l.Steam l In InMess) <= MaxSteam /\
        (SteamBoilerWaiting In InMess ==> Waiting) /\
        (PhysicalUnitsReady In InMess ==> Ready) /\
        (forall p.
            (PumpRep p In InMess ==> Pst p Repairing) /\
            (PumpCtrRep p In InMess ==> Mst p Repairing) /\
            (PumpFlrAck p In InMess ==> Pst p Broken) /\
            (PumpCtrFlrAck p In InMess ==> Mst p Broken)) /\
        (LevelRep In InMess ==> Qst Repairing) /\
        (SteamRep In InMess ==> Vst Repairing) /\
        (LevelFlrAck In InMess ==> Qst Broken) /\
        (SteamFlrAck In InMess ==> Vst Broken);

%no_expand In
%tcg -t -S SetInMessOk
```

`SetInMessOK` specifies how the input message consistency flag is set. The specification for `SetInMessOk` is not in test class normal form, but is still a relationship between stimuli and responses.

The directive `%no_expand In` suppresses the expansion of the definition of `In`. The directive `%tcg -t -S SetInMessOk` directs the prototype test frame generator to produce test frames using the criterion (pure test classes, no response-response resolution, not a closed world, no frame stimuli simplification, Term Coverage, base test frames). The `-t` flag indicates that Term Coverage is to be used rather than the default DNF Coverage. The `-S` flag indicates that the output should be in the form of S expressions.

The condition dependency information regarding the states of the system, e.g. `Repairing, Broken`, is valuable. Without this information, it is possible that a test frame could include

$$\ldots\wedge \texttt{ Pst p Repairing } \wedge \texttt{ Pst p Broken } \wedge\ldots$$

within a test frame. If the Term Coverage scheme were to select such a prime implicant, the decision procedure would determine a match with `P = (Pst p)`. Thus, such infeasible test frames are avoided.

The test classes and associated test frames produced from this specification are listed in Appendix B. The number of test classes, prime implicants, and test frames for DNF and Term Coverage for this example are detailed in Table 7.1.

Test Class	Prime Implicants	DNF Coverage	Term Coverage
1	20	20	20
2	64	27	2

Table 7.1: Numbers of Prime Implicants and Test Frames

7.8.2 North Atlantic Separation Minima

This example, described in a separate technical report [20], demonstrates the semi-automatic generation of a set of 169 test frames from a formal specification of aircraft separation minima for the North Atlantic. The test frames were automatically generated by the prototype test frame generator from an S specification of the separation minima. Figure 7.2 provides a sample of the S specification. The specification is approximately 650 lines of S. Figure 7.3 provides a sample of one of the automatically generated test frames. The combined set of 169 test frames provides complete coverage of all conditions contained in the separation minima specification. 125 of the 169 test frames are instances of the "separation exists" condition. The remaining 44 test frames are instances of the "separation does not exist" condition.

```
LongitudinallySeparated(A,B) :=
    if (AngularDifferenceGreaterThan90Degrees
            (RouteSegment A, RouteSegment B))
    then     /* opposite direction */
        NOT (WithinOppDirNoLongSepPeriod(A,B))
    else     /* same direction */
        ABS(TimeAtPosition A - TimeAtPosition B)
            > LongSameDirSepRequired(A,B);
```

Figure 7.2: NATS S Specification Fragment.

This example demonstrates the capability of this test generation approach to produce test frames for a logically complex specification. It is expected that the 169 test frames could be used directly by test engineers in the development of test procedures for systems that monitor air traffic over the North Atlantic.

The separation minima were originally written in a formal table notation [14]. This specification was not authored with the intention of generating test frames. The formal specification of this separation minima is based on a description provided in

Stimuli	Response
1. AngularDifferenceGreaterThan90Degrees (RouteSegment A , RouteSegment B) 2. ¬ (IsSupersonic B) 3. IsTurbojet A 4. IsTurbojet B 5. ¬ (IsWestOf60W B) 6. ¬ (InWATRSAirspace B) 7. ReportedOverCommonPoint (A , B) 8. ept (A , B) + 10 < "separation check time"	1. "are separated" (A , B)

Figure 7.3: A NATS Test Frame.

a source document entitled "Application of Separation Minima for the NAT Region" (3rd edition, effective December 1992), published by Transport Canada on behalf of the ICAO North Atlantic Systems Planning Group. The table-based specification was algorithmically converted into an S specification by N. Day.

Although the S specification simply stated the conditions for separation and did not specify requirements for a system, it was easily transformed into the stimulus-response style system requirements specification

```
forall A B.AreSeparated (A,B) ⇔ "are separated" (A,B)
```

for the purpose of generating test frames. This specification requires that the system indicate that two aircraft are separated precisely when they are separated according to the requirements specified by `AreSeparated(A,B)`.

The following example provides a comparison between base and differentiated test frames. One of the base test frames is:

Stimuli	Response
1. AngularDifferenceGreaterThan90Degrees (RouteSegment A , RouteSegment B) 2. $\neg$ (IsSupersonic B) 3. IsTurbojet A 4. IsTurbojet B 5. $\neg$ (IsWestOf60W B) 6. $\neg$ (InWATRSAirspace B) 7. ReportedOverCommonPoint (A , B) 8. ept (A , B) + 10 $<$ "separation check time"	1. "are separated" (A , B)

The differentiated version of the same test frame is:

Stimuli	Response
1. AngularDifferenceGreaterThan90Degrees (RouteSegment A , RouteSegment B) 2. ¬ (IsSupersonic B) 3. IsTurbojet A 4. IsTurbojet B 5. ¬ (IsWestOf60W B) 6. ¬ (InWATRSAirspace B) 7. ReportedOverCommonPoint (A , B) 8. ept (A , B) + 10 < "separation check time" 9. ¬ (VerticallySeparated (A , B)) 10. ¬ (LaterallySeparated (A , B)) 11. EnterWATRSAirspaceAtSomeTime A 12. EnterWATRSAirspaceAtSomeTime B 13. IsWestOf60W A 14. MachTechniqueUsed A 15. MachTechniqueUsed B 16. OnPublishedRoute A 17. OnPublishedRoute B 18. "SameOr Diverging Tracks" (A , B) 19. ept (A , B) + 10 < EndTime ("WATRSOppDir NoLongSepPeriod" (A , B))	1. "are separated" (A , B)

Using an iterative approach, computing the base test frames required a total of three hours[2] on an Ultra-Sparc 60. Computing the differentiated test frames required five and a half hours on the same machine. Constructing an initial set of scripts for generating test frames took approximately one hour.

Since the S specification is large and complex, the particular test frame generator used in this example, TCG, does not have the capacity to process it in full detail. An iterative approach was used to overcome this problem.

In the first iteration, only the predicate `AreSeparated` was expanded. All other predicates and functions within the specification were treated as primitives. This resulted in the following expanded specification:

```
forall A.
  forall B.
    (~
       (VerticallySeparated (A , B) \/
          LaterallySeparated (A , B) \/
            LongitudinallySeparated (A , B)) \/
       "are separated" (A , B)) /\
      (~ ("are separated" (A , B)) \/
         VerticallySeparated (A , B) \/
           LaterallySeparated (A , B) \/
             LongitudinallySeparated (A , B))
```

From this expansion, two test classes were generated: one for each of the responses `"are separated" (A , B)` and ¬ `("are separated" (A , B))`. An initial set of test frames was generated along with the test classes.

Additional condition dependencies were added when infeasible test frames

[2] The times given are the elapsed time reported by the unix time utility.

were found in the TCG output, or when the TCG tool found no feasible test frames in a particular iteration. (Finding no feasible test frames implies that the input specification for that iteration was also infeasible.) This added a few days to the time required for the construction of scripts for generating feasible test frames. This was due to condition dependencies which exist between different levels of abstraction within the specification. This suggests that, although this iterative approach is capable of processing large, complex formal specifications, more work is required to allow this particular type of condition dependencies to be determined with less effort.

For this specification, the differentiated test frames are only slightly different from the base test frames. This is due to the table structure from which the S specification was generated.

In some iterations, some of the test frames were found to be redundant. This occurs when the stimuli for two or more test frames subsume the stimuli of another. There are 161 differentiated test frames compared with 169 base test frames. This demonstrates the value of differentiation in eliminating redundant test frames.

7.9 Conclusion

This chapter has defined the general test frame generation process, and has presented aspects of one possible implementation of a test frame generator for this process. Although this chapter presents examples using a specific notation, S, and a particular implementation of a test frame generator, TCG, these are only examples of the possible notations and tools. The generality of this process allows it to be applied to specifications based on logics that are consistent with the logical manipulations described in Chapter 5. The next chapter presents a refinement of this general test frame generation process that can be applied to system-level requirements-based testing.

Chapter 8

System-Level Requirements-Based Testing

This chapter illustrates how the discipline of specification-based test derivation presented in this dissertation can be applied to system-level requirements-based testing. A practical approach to automating portions of system-level requirements-based testing requires special attention to issues of process integration. A primary issue is the choice of language to be used by requirements authors. Other issues include support for traceability, requirements validation, and measurements. This chapter examines these issues and presents a refinement of the general test frame generation process described in Chapter 7, which accounts for these issues. The resulting process provides a solution to the problems described in chapters 2 and 4.

8.1 Introduction

In the field of system-level requirements-based testing, a distinction is often made between those stimuli and responses that are externally visible, and those that only refer to the internal state of the system. In this chapter, *pre-conditions* are stimuli that either:

- are not externally visible, i.e., they refer to the internal state of the system and not the environment, or
- specify conditions on parameters to externally visible stimuli.

Similarly, *post-conditions* are responses that refer either to:

- the internal state of the system, or
- to parameters of externally visible responses.

In the remainder of this chapter, the terms stimulus and response refer to atoms that are not pre- or post-conditions.

The general test frame generation process of Chapter 7 requires an amount of formal structure in the specification. Integrating an automated test frame generator into a current system-level requirements-based test derivation process requires the use of a formal language for requirements specification that is readable by non-specialists. Specification language features were developed as part of this research in order to enhance readability by non-specialists, while providing the formal structure required for automated test frame generation. The Q specification language is the author's collection of these features.

An automated approach to test frame generation does not eliminate the need for traceability. For auditing purposes, it is necessary to be able to determine which requirements are represented in each of the test frames. This capability is provided by augmenting the rewrite system of Section 7.5.

In addition to generating test frames, this partially automated process provides additional benefits to software development processes. Test frames can be used by requirements authors for validating the requirements they have written. Also, the nomenclature from this thesis can be used for detailing how much system-level requirements-based testing is required, and how much has been completed.

Section 8.2 provides an overview of the test frame generation process refined for system-level requirements-based testing. Section 8.3 describes the Q requirements specification language. Section 8.4 describes how traceability is achieved. Section 8.5 describes examples of the application of this process to real world specifications. Section 8.6 describes additional benefits of this testing discipline.

8.2 Process Overview

Figure 8.1 illustrates a refinement of the general test frame generation process applicable to system-level requirements-based testing. The requirements are written in Q by requirements authors. It is likely that the resulting Q specification is easily read by other individuals for various other requirements-based processes. These other individuals can include other requirements authors and test engineers, domain experts, software designers, customers, and government regulators. Test engineers define the coverage criterion and any user mandated tests. Domain knowledge can come from several sources, such as the requirements authors, domain experts, and test engineers. Once the test frames have been generated, test engineers select the appropriate data to produce test steps. Requirements authors can also use the test frame generator to validate their requirements in a manner similar to that recommended by Somerville and Sawyer [59].

While the general test frame process accepts a formal specification in a general form, requirements authors and those who would typically read system-level requirements specifications are insufficiently familiar with the notation. The Q

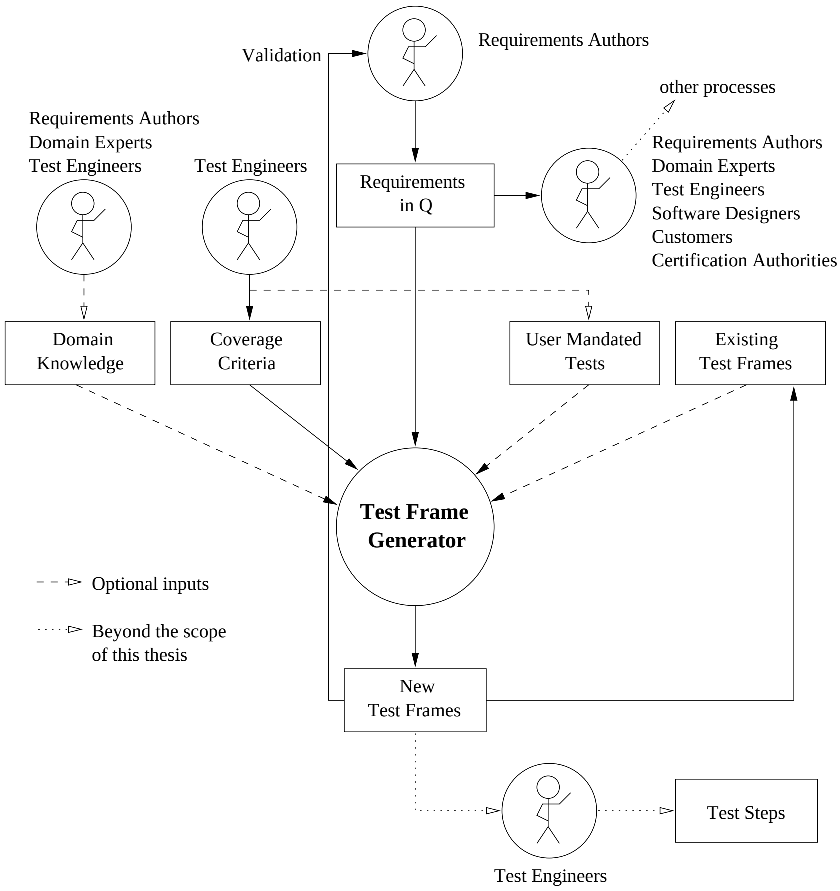

Figure 8.1: Integrating Automated Test Frame Generation

specification language is an attempt at solving this problem. Q provides a means of annotating requirements text so that the logical relationships relevant to test frame generation are made explicit and precise, while preserving readability.

The following process accomplishes system-level requirements-based testing.

1. The testable requirements, i.e., those that can be verified through testing, are specified in Q.

2. Any domain knowledge and user-defined tests are specified.

3. An appropriate coverage criterion is selected. Since most documents on system-level requirements-based testing specify that "there is at least one test for each requirement," this criterion will most commonly include a Term Coverage selection scheme.

4. Test frames are generated automatically from the Q specification.

5. Test engineers perform manual data selection to produce test steps and test procedures.

The use of a test frame generator for requirements validation is described in Section 8.6.

8.3 The Q Specification Language

This section describes the motivation for Q and defines the Q specification language.

8.3.1 Overview

The successful integration of an automated test frame generation process requires that the formal language for specifying requirements be accepted by the requirements authors. Traditional formal specification languages such as Z [61] and VDM-SL [37] impose formality, together with an amount of symbology that creates a language very foreign to requirements authors. In contrast, Q imposes as little symbology as possible, and allows the authors to use phrases of their own design.

The formal aspect of the Q language is required by the test frame generator. A formal specification that is also readable relieves the need for maintaining two specifications; one formal for input to the test frame generator, and another for non-specialists.

A specification written in a traditional formal language, in this case S [39], may appear as:

```
:flight;
leader, follower : flight;
Supersonic : flight -> bool;

Spec := Supersonic leader \/ Supersonic follower;
```

In Q, the specification may appear as:

```
:flight;
"the leading aircraft", "the following aircraft" : flight;
" * is supersonic" : flight -> bool;
" * or * " x y := x \/ y;

BEGIN_Q

{Spec} is true iff
{{{the leading aircraft} is supersonic} or
        {{the following aircraft} is supersonic}}.
```

```
END_Q
```

In the Q specification, it is likely that the definition of `Spec` is more readable by non-specialists.[1] The pre-amble above the keyword BEGIN_Q would normally be contained in the infrastructure for supporting a Q specification of the requirements.

The Q specification language provides a syntax for concisely denoting the logical relationships and alternatives within the requirements, while also providing a natural language style. For example, the requirement fragment,

Either the leading aircraft or the trailing aircraft is supersonic

is specified as

```
{any of {the leading aircraft, the trailing aircraft}} is supersonic.
```

The braces impose a parseable structure on the requirements. The semantics of the language constructs, such as "`any of`," allows the test frame generator to calculate the logically equivalent expression, which in this case is:

```
{{the leading aircraft} is supersonic} or {{the trailing aircraft}
is supersonic}.
```

Once these constructs are expanded into their logical equivalents, test frames can be calculated as in Chapter 7.

Q is implemented as an extension of the S specification language, and is used to formalize natural language stimulus-response style specifications for the purpose of requirements-based testing. Q can be used to define predicates within a requirements specification, but relies on S syntax for defining constants, types, and functions. Q statements are contained within the keywords `BEGIN_Q` and `END_Q`.

[1] Although multi-word variable names are supported by S, this style is rarely used in specification styles similar to S, Z, or VDM-SL. However, the use of multi-word variables is encouraged and is more natural in Q due to the flex-fix notation.

The lightweight simplicity of the Q language helps to preserve the readability and conciseness of the specification. The mathematical semantics of Q ensure that each statement has an unambiguous meaning. With these qualities, Q provides the mathematical link between a requirements specification and the test frame generation tool introduced in the previous chapter.

There are three essential features of Q. The first is the use of braces, {}, which delimit phrases and parameters within the specification. Injecting these braces into the specification effectively transforms the phrases of natural language into formal functions and arguments. This technique was first used by Joyce in his Test Case Element Language (TCEL) [38].

When formalizing the natural language phrase

the leading aircraft is supersonic or the following aircraft is supersonic

for the purpose of system-level requirements-based testing, only the choices need to be made explicit. Thus, the appropriate formalization for testing is to choose "`or`," as the predicate and the two adjoining phrases are conditions. The resulting Q version of the above phrase is:

```
{the leading aircraft is supersonic} or {the following aircraft
is supersonic}.
```

In this Q expression, " `* or *` " is the function, and "`the leading aircraft is supersonic`" and "`the following aircraft is supersonic`" are its arguments. The predicate logic equivalent, where function application is expressed by the juxtaposition of literals, is:

```
" * or * " "the leading aircraft is supersonic" "the following aircraft
is supersonic"
```

The predicate "`* or *`" has the type `bool` $\to$ `bool` $\to$ `bool`, as expected.

The "`*`" in the function name denotes positions in the text where arguments are placed. This type of notation is referred to as a flex-fix notation [13]. Flex-fix, the second Q feature, allows arguments to be distributed within a function name. This helps preserve readability. For example, the Q expression

`{aircraft A} and {aircraft B} are separated by at least {1000 feet}`

corresponds to the following predicate logic representation:

"`* and * are separated by at least *`" "`aircraft A`" "`aircraft B`" "`1000 feet.`"

The Q expression is more readable to requirements specification authors than, say, an, S, Z or VDM-SL expression, such as

$$ABS(Altitude(aircraft\_A) - Altitude(aircraft\_B)) \geqslant feet(1000).$$

The third feature of Q, due to the author, is the use of keywords that define multiple arguments for a function's parameter. These keywords are motivated by natural language phraseology such as "both aircraft are," and "either A or B is." For example, the requirement

either the leading aircraft or the following aircraft is supersonic

can be formalized in Q as

`{any of {the leading aircraft, the following aircraft}} is supersonic.`

A predicate containing an "`any of`" argument is equivalent to a disjunction of that predicate evaluated at each of the values in the "`any of`" set. In this case, the equivalent expression is

```
{{the leading aircraft} is supersonic} or {{the following aircraft}
is supersonic}.
```

This example contains more formal detail than the expression

```
{the leading aircraft is supersonic} or {the following aircraft is
supersonic}.
```

In the former expression, there are formal references to two aircraft. In the latter expression, there are only two conditions. The fact that these conditions were based on two aircraft was not made explicit in the latter expression. This latest example is referred to as a deeper specification, because it contains more formal detail. Test engineers decide how deep a specification should be by determining the conditions they wish to reveal to the test frame generator.

Another parameter mechanism is the "distinct choices" keyword. This keyword is used in encoding phrase structures such as:

all of the following are true:

1. *aircraft A is dumping fuel,*
2. *aircraft B is using standard altimeter setting,*
3. *if one aircraft is supersonic and the other is not then further conditions*

In this example, "one aircraft" and "the other" refer to either "aircraft A" or "aircraft B," interchangeably. They represent distinct choices of the two aircraft. The Q version is:

```
{all of {
```

1. `{aircraft A} is dumping fuel,`

2. `{aircraft B} is using standard altimeter setting,`

3. `if {{one aircraft, the other} are any distinct choices of {aircraft A, aircraft B} in`
   `{{{one aircraft} is supersonic} and {it is not the case that {{the other} is supersonic}}}} then {further conditions}`

The "distinct choices" phrase in this example is necessary in order to formally define the references "one aircraft" and "the other." However, this construction is still more concise and more readable than the full expansion of the distinct choice, which is:

```
{{{aircraft A} is supersonic} and
        {it is not the case that {{aircraft B} is supersonic}}} or
{{{aircraft B} is supersonic} and
        {it is not the case that {{aircraft A} is supersonic}}}
```

The formal semantics of "`any of`," its counterpart, "`each of`," and other parameter mechanisms are defined more precisely in later sections.

8.3.2 Expressions

A Q expression is a string of at least one word and any number of arguments separated by white-space characters. Arguments are expressions contained within a comma-delimited list surrounded by braces. In the following grammar, * and + refer to zero or more and one or more of the preceding symbol, respectively.

expression	:=	word+ "." primitive_expression
	|	primitive_expression

primitive_expression	:=	("{" expression ("," expression)* "}")+ primitive_expression+
	|	primitive_expression+ ("{" expression ("," expression)* "}")+
	|	word+

The optional prefix, word+ ".", for each expression allows specification authors to tag expressions for traceability purposes. These tags have no semantic value with respect to the logical meaning of the specification.

8.3.3 Predicate Definitions

A Q specification is a collection of predicate definitions. Predicates are defined using the "`* is true iff *` " statement.

definition	:=	"{" parm_expression "}" is true iff "{" expression "}" "."

parm_expression	:=	("{" word+ ("," word+)* "}")+ parm_expression+
	|	parm_expression+ ("{" word+ ("," word+)* "}")+
	|	word+

8.3.4 Conjunctive and Disjunctive Lists

Requirements specifications often provide lists of conditions which represent logical conjunction, e.g., "all of the following," or disjunction, e.g., "at least one of the following." Such a list format is provided by the predicates "`all of`" and "`any of`." The Q expression `all of {S}`, where `S` is a comma-separated list of predicates, is semantically equivalent to $\bigwedge \mathtt{S}$, where $\bigwedge(\{x\} \cup A) = x \wedge (\bigwedge A)$, and $\bigwedge\{\} = \top$.

Similarly, `any of` $\{$`S`$\}$ is semantically equivalent to $\bigvee$`S`, where $\bigvee(\{x\} \cup A) = x \vee (\bigwedge A)$, and $\bigvee\{\} = \bot$.

8.3.5 Argument-Based Conjunctions and Disjunctions

The keywords "`each of *` " and "`any of *` " are used to construct conjunctions and disjunctions, respectively, of a predicate over different arguments. These keywords both appear as functions having the type $(t)list \rightarrow t$. The semantics of these functions is defined in terms of predicates, i.e., predicate logic expressions that do not contain logical connectives (see Section 5.3). The equivalent logic expression is determined by evaluating the predicate logic expression $AE\_Ua\ P$ for "`any of`" or $AE\_Ue\ P$ for "`each of`" using the rules of Appendix A. These two functions map the application of a predicate to a list of arguments into a disjunction or conjunction, respectively, of the predicate applied to each argument of the list, separately.

Although multiple uses of one of these keywords can be used within a predicate, mixtures of "`any of`" and "`each of`" within arguments to a single reference of a predicate are problematic. This is because it is unclear whether the expression containing argument keywords represents a conjunction of disjunctions, or vice versa.

For example, the expression

```
{the {each of {apple, tomato}} is a {any of {vegetable, fruit}}}
```

may have been intended to mean either

```
{{{the {apple} is a {vegetable}} or {the {tomato} is a {vegetable}}}
and {{the {apple} is a {fruit}} or {the {tomato} is a {fruit}}}}
```

or, alternatively,

```
{{{the {apple} is a {vegetable}} and {the {tomato} is a {vegetable}}}
or {{the {apple} is a {fruit}} and {the {tomato} is a {fruit}}}}.
```

Clearly, these two semantic evaluations are logically different.

Although the rules of Appendix A disambiguate such a construction, this rule would need to be learned and would not be obvious to a non-specialist from the text alone. Since this is counter to the objective of Q, mixtures of "`any of`" and "`each of`" are not allowed within arguments to the same predicate. The order of semantic evaluation in these situations can be made more clear using expression aliasing.

8.3.6 Expression Aliasing

An expression alias is the same as the let statement found in functional programming languages such as ML [49]. The purpose of the alias is to assign a short name to a complex expression in order to make a portion of text more readable.

The Q expression `{{x} is {y} in {E}}` is semantically equivalent to `{E}`, with `y` substituted for `x`. To encourage simpler specifications, the expression E must be a predicate logic expression rather than an arbitrary expression that might represent a non-Boolean value. The predicate `{{x} is {y} in {E}}` is syntactic sugar for the lambda calculus expression $(\lambda\, \mathtt{x}.\mathtt{E})\mathtt{y}$. Similarly, the tuple form `{{x,y} are {a,b} in {E}}` is syntactic sugar for the lambda calculus expression $(\lambda\, \mathtt{x}, \mathtt{y}.\mathtt{E})(\mathtt{a}, \mathtt{b})$.

Using expression aliasing, the earlier "`any of`" / "`each of`" example can be disambiguated as

```
{{item} is {each of {apple, tomato}} in
{the {item} is a {any of {vegetable, fruit}}}}
```

which results in a conjunction of disjunctions.

8.3.7 Argument Permutation

The predicates "`* are all distinct choices of * in *` " and "`* are any distinct choices of * in *` " are used to construct conjunctions and disjunctions involving permutations of arguments. An example of the use of this keyword was given earlier in Section 8.3.

`{{z} are all distinct choices of {A} in {E}}`

is semantically equivalent to

`{{z} are {each of {P(A)}} in {E}},`

where `z` is a tuple and `P(A)` is a list of all the permutations of tuples the same size as `z` uses elements of `A`. Similarly,

`{{z} are any distinct choices of {A} in {E}}`

is semantically equivalent to

`{{z} are {any of {P(A)}} in {E}}.`

8.3.8 Quantification

Universal and existential quantification are provided by the syntax `{for any {x} {E}}`, which is equivalent to $\forall$ `x.E`, and `{there exists {x} such that {E}}`, which is equivalent to $\exists$ `x.E`. Higher-order quantification is allowed. An example is:

```
{for any {separation of * and * rules} {separation of {target} and
{intruder} rules}}.
```

8.4 Traceability

As described in Section 2.6, traceability provides a means of mapping requirements to the tests that verify those requirements [16]. The traceability of test frames to requirements is automated in the following way by an augmented rewrite system. Authors tag the requirements in the Q specification with an identifier. When the Q specification is parsed, these tags are embedded in the atoms and arguments in the corresponding logical expressions. During test frame generation, the rewrite system maintains these tags. As logical expressions are rewritten, atoms and arguments from various places in the specification are brought together while the tags identify their origin.

Although this traceability mapping is generated in a test-frames-to-requirements manner, the desired inverse mapping can be easily computed.

8.5 Examples

This section describes examples of the application of the process described in this chapter.

8.5.1 CAATS SRS

To assess the practical usefulness of this process, the partially automated process described in this chapter was experimentally applied to a portion of the Software Requirements Specification for the Canadian Automated Air Traffic System (CAATS) being developed by Raytheon Systems of Canada Ltd. This example, presented in a conference paper [21], is taken from a portion of the CAATS software requirements which refers to separation rules. The separation rules form a set of complex conditions under which certain responses occur. The specification of the separation rules is composed of several subsections dealing with different aspects of separation. The portion of the specification used in this example contained 177 requirements designated as testable requirements[2].

When evaluating this process, it was decided that a test set with DNF Coverage would not be produced for this specification due to the large number of test frames which would have resulted. The specification refers to the separation rules in both a negative (the aircraft are not separated), and a positive (the aircraft are separated), context. This results in two corresponding test classes. The numbers of test frames constituting DNF Coverage are estimated to be approximately 1,000 for the positive case, and roughly 10^{24} for the negative case.

Test frames were generated using a Term Coverage scheme. This resulted in approximately 130 test frames for the positive case and approximately 230 test frames for the negative case.

Table 8.1 gives one of the test frames generated by our automated process. ROIDs are requirement object identifiers used to tag requirements statements.

It is important to note that the success of this example was due to the following essential qualities:

1. The consistency of the test frames, the assurance of proper coverage, and the accuracy of the tracing information are due to the mathematical underpinnings of the algorithms used.

2. The formal version of the software requirements fragment contained enough mathematical structure to facilitate test frame generation while still being

[2]In addition to requirements that can be verified through testing, requirements specifications often contain requirements that cannot be verified through a test program and must be addressed by other means, which are beyond the scope of this dissertation.

Stimulus	Conditions	Responses	ROIDs
{ACC operator} requests planned clearance	1. {planned clearance} exists for the flight 2. the source of the {planned clearance} is an aerodrome control tower with a tower method of operation of complex 3. the aircraft state is not AIRBORNE 4. {intruder} is using {altimeter setting} 5. {planned clearance} is using {altimeter setting} 6. the lowest altitude in the protected altitude band for {intruder} is at or below {FL 290} 7. the lowest altitude in the protected altitude band for {planned clearance} is at or below {FL 290} 8. the protected altitude band for {intruder} is vertically separated from the protected altitude band for {planned clearance} by {1000} feet or more 9. (NOT {planned clearance} is dumping fuel) 10. (NOT {intruder} is dumping fuel)	1. {ATA} shall commit {planned clearance}	84672 224215 226547 226549 226550

NOTE: This is only an example. This test frame was generated from a representation of only a portion of the CAATS software requirements which was used to evaluate the usefulness of this process. Any errors or omissions in this test frame are due to the way in which this portion was extracted by the author.

Table 8.1: An Automatically Generated Test Frame

readable.

3. Conditions were relatively independent, which allowed for a simple encoding of the existing condition dependencies.

8.5.2 ICAO Flight Plan

This example, described in a separate technical report [19], involved the semi-automatic generation of a set of 252 test frames from a portion of the ICAO instructions for filling out a flight plan as specified in Appendix 2, Subsection 2 of ICAO's Rules of the Air and Air Traffic Services [36]. The 252 test frames were automatically generated by the QTCG prototype tool from a Q representation of testable requirements. Figure 8.2 presents a portion of the 526 line Q specification. Figure 8.3 provides a sample of one of these automatically generated test frames. Two distinct sets of test frames were generated through different uses of the same requirements specification. Each set of test frames provides complete coverage of all the testable requirements relative to the context in which the requirements were used. 122 of the test frames are schemas for testing a system that automatically fills out a flight plan. The remaining 130 test frames are schemas for testing a system that validates a given flight plan.

It is expected that the 252 test frames could be used directly by test engineers in the development of test procedures for software that produces a filled-out flight plan and for software validating filled-out flight plans.

The ten pages of testable requirements were manually translated into a parseable representation of similar size. To produce the formal specification, text was translated directly from the ICAO flight plan instructions into a Q specification.

Computing the base test frames for filling out a flight plan required a total

```
I19ES4.
if {not {Dinghies are carried}} then {
  cross out {Item 19 D} - {each of {D, C}}}
else {all of {
  insert {Item 19 D} - {number of dinghies carried},

  insert {Item 19 D} -
    {total capacity in persons of all dinghies carried},

  if {not {Dinghies are covered}} then {
    cross out {Item 19 D} - {C}},

  insert {Item 19 D} - {colour of dinghies}
  }}
```

Figure 8.2: ICAO Flight Plan Specification Fragment.

ROIDs: I19ES4	
Stimuli	Response
1. Dinghies are carried	1. insert {Item 19 D} - {number of dinghies carried} 2. insert {Item 19 D} - {total capacity in persons of all dinghies carried} 3. insert {Item 19 D} - {colour of dinghies}

Figure 8.3: An ICAO Flight Plan Test Frame.

of one minute and 42 seconds[3] on an Ultra-Sparc 60. The base test frames for checking a filled-out flight plan required a total of two minutes and 39 seconds. Computing the differentiated versions of this latter set of test frames had to be done in pieces, and required approximately fifty minutes. Constructing the set of scripts for generating test frames took approximately half an hour.

From the author's exposure to industry practice, a very conservative estimate of the effort required to derive, review, and document a traceability map for a single test frame, on average, would be one hour.[4] By this estimate, the base test frames that were automatically generated in under three minutes would require approximately three person-weeks to prepare manually. This comparison does not include the translation time, since it is expected that requirements authors would produce original specifications in Q.

During the construction of the test class normal form, two potential specification anomalies were reported by the QTCG tool. Two of the test classes express facts implied by the specification.

Test class 59 is analogous to the test frame:

Stimuli	Response
1. NOT The flight is along a designated ATS route 2. ATS flight track points are required by the appropriate ATS authority 3. NOT Use ATS style track points	*false*

Since the response is false, this implies that the specification asserts that the stimuli can never occur. This poses a question to be answered by the requirements author,

[3]The times given are the elapsed time reported by the unix time utility.
[4]In many cases, a more conservative, and realistic estimate, is one day.

e.g., is it true that this combination of stimuli can never occur? An inconsistency would indicate an error in the specification.

Test class 87 is analogous to the test frame:

Stimuli	Response
true	1. insert {Item 19 E} - {the four digit fuel endurance in hours and minutes} 2. insert {Item 19 A} - {colour of aircraft and significant markings} 3. insert {Item 19 C} - {name of pilot in command}

This test frame indicates that the response will always occur. Thus, these response conditions can be appended to each of the other test frames, if desired. Again, this seems consistent with the importance of the information in these fields of the flight plan.

8.6 Additional Benefits

This section describes the use of test frames for requirements validation, and the use of the nomenclature of this discipline for describing measurements of complexity and progress of system-level requirements-based testing.

8.6.1 Validation

The process presented in Section 8.2 has the potential to improve requirements validation. The purpose of validation is to ensure that the requirements reflect what is actually intended. Although reviews are commonly used in software development processes to ensure that requirements are valid, a certain amount of requirements validation occurs during test development. This is because the activity of constructing tests from specifications provides an alternative perspective of the implications of the specification.

Unfortunately, test construction is performed after requirements authoring, and is typically performed by different individuals. Sommerville and Sawyer [59] recommend that requirements authors derive test steps as a means of validating the requirements they write. A test frame generator provides an automated means for requirements authors to leverage the gain of this testing perspective while they are writing the specification. Any anomalous test frames found during a review of those produced by the test frame generator, can be traced back to the offending requirements. This occurred during reviews of test frames produced for both the CAATS and ICAO Flight Plan examples.

Another benefit is that other non-specialists, such as domain experts, can participate in validating a formal requirements specification. This is difficult to achieve with traditional formal specification languages, which typically require a high degree of training to attain proficiency, such as with Z or VDM-SL.

The participation of a domain expert was illustrated during the authoring of a Q specification for Notices To Airmen [23]. A review of the Q specification by an individual with no training in Q realized that there were many assumptions held by the industry that were not made explicit in the specification. The ability of this individual to identify this problem shows that he was able to read and comprehend significant portions of the Q specification.

8.6.2 Complexity and Progress Measurement

The nomenclature of the discipline presented in this dissertation can be used as the basis for measurements. These measurements can provide an accurate picture of the progress of system-level requirements-based testing.

An upper bound on the number of test frames for Term Coverage can be computed in $O(n \log n)$ time. This is because test classes and frame stimuli can be determined in an amount of time that is $O(n \log n)$ in the length of the specification, and a Term Coverage scheme produces no more test frames than the number of frame stimuli in a test class.

A crude measure of the complexity of the system can be obtained from an approximation of the number of test frames. The number of test classes provides a measure of the number of operations required of a system, while the number of test frames per test class provides a crude measure of the complexity of each operation. A possible refinement is to assign weights to each frame stimulus, indicating the expected relative complexity of its detection.

The bound on the number of test frames can be used to estimate the progress of system-level requirements-based testing. As test frames and test steps are produced and their corresponding test procedures executed, the relationship between those completed and those not yet produced helps provide an assessment of progress.

8.7 Summary

This chapter has illustrated the application of the discipline of specification-based test derivation to system-level requirements-based testing. A particular refinement of the general test frame generation process from Chapter 7 was presented that addresses specific issues of system-level requirements-based testing. Other possible benefits of the discipline were also presented.

Chapter 9

Conclusions

This chapter reviews the results of this research, and outlines some possible avenues for further research.

9.1 Research Results

This dissertation has identified two major problems in the field of system-level requirements-based testing:

1. the lack of objective definitions of coverage criteria; and

2. the lack of automation.

This dissertation has also identified four challenges in automating the generation of test frames:

1. the structural independence of test frames from the specification;

2. dependencies between conditions within test frames;

3. existential and universal quantification within the specification; and

4. the delta problem.

This dissertation has presented a discipline of specification-based test derivation. It has also demonstrated that this discipline provides a scientific foundation for improving portions of software development processes, such as system-level requirements-based testing. This dissertation has defined a nomenclature for specification-based testing that forms a basis for objective specification-based coverage criteria definitions and test frame generation algorithms. These contributions satisfy the goals of this research.

Furthermore, the use of this discipline has several benefits. The discipline strongly encourages the development of a testable requirements specification. At the same time, the automation of test frame generation increases the value of producing a formal specification. Perhaps most importantly, the nomenclature can be used in future revisions of standards documents such as DO178B, DOD-STD-2167A, ANSI/IEEE 829-1983, and MIL-STD-498, to state objective testing requirements at the system level.

Moreover, system-level requirements-based testing is not the only application of the discipline of specification-based test derivation. For the purposes of test frame generation, this discipline can be applied to any stimulus-response style formal specification founded on a logic consistent with the algorithms given in Chapter 5. As illustrated in Section 8.6, automated test frame generation can contribute to the validation of the specification. This discipline can also be used for other purposes, such as complexity and progress measurements.

The application of this discipline to a broad range of specifications has been illustrated in four examples: one from the literature, one authored for purposes other than testing, one translated from an international public domain air traffic control authority, and one translated from a proprietary specification owned by a prominent company in the air traffic control industry.

Section 2.8 lists five characteristics for any solution to the problems of system-level requirements-based testing addressed by this thesis. The discipline presented in this dissertation satisfies each of these characteristics:

1. objective definitions of coverage criteria are based on the nomenclature;
2. test frame derivation is partially[1] automated;
3. test engineers have the control they require to exercise engineering judgement;
4. traceability is supported; and
5. the Delta Problem has been shown to be intractable (Appendix C), but heuristic solutions appear to be possible.

The algorithms presented in this dissertation produce test frames with the following properties.

1. *Conservative*: Each test frame is a logical consequence of the requirements.
2. *Tractable*: Test engineers have the control to exercise engineering judgement.
3. *Complete*: The set of test frames is produced according to a specified coverage criterion.
4. *Traceable*: The original elements can be determined from which a selected test frame was derived.

[1]The formalization of the requirements cannot be automated, in general.

Demonic specifications and those that do not have a test class normal form can be identified by the algorithms. These specification forms are suspect, and typically indicate specification errors.

9.2 Foundations for Future Work

The work presented in this dissertation can be extended in several areas. This research includes improvements to the test frame generation process; heuristics for the Delta Problem; incorporating the general test frame generation process into a testing methodology; a more mature language exploiting the main Q features; and the projection of test frames onto a design specification.

9.2.1 Test Frame Generation Process Improvements

There are opportunities for research in refining the process itself. An iterative application of a test frame generator allows the processing of large specifications. The application of this technique in situations where infeasible test frames are numerous requires further research.

Other research would focus on coverage criteria. While defining coverage criteria, Chapter 6 introduced several variations of test frame generation based on test classes, test frames, and frame stimuli. Although these terms form the core of the nomenclature for defining coverage criteria, it would be naive to expect that the criteria of Chapter 6 form an exhaustive list. In particular, the range between DNF and Term Coverages should be explored. It is quite probable that certain situations will require test engineers to develop other variations to suit their needs. However, these new variations will, in all likelihood, be described in terms of the basic nomenclature laid forth by this thesis.

9.2.2 Delta Heuristics

Although the Delta Problem is undecidable, in general, a process capable of integrating existing test frames with new test frames could potentially avoid wasteful rework. Appendix C shows that the Delta Problem is undecidable and outlines one possible heuristic solution to this problem.

9.2.3 Methodology

This section describes the extent to which the discipline of specification-based testing presented in this dissertation defines a methodology for system-level requirements-based testing. The following is a description of the basic contents of such a methodology. The methodology would specify:

1. the required properties of the specification language used to state the requirements;
2. precise definitions of test procedure and the contents of a test procedure, e.g. test steps;
3. what amount of detail is required in each test step;
4. algorithms for deriving and sequencing the contents of test procedures from the specified requirements;
5. how traceability is achieved;
6. what requirements coverage means and how to satisfy given coverage criteria;
7. under what circumstances particular coverage criteria should and should not be used; and

8. procedures to be performed when requirements changes occur after a set of test procedures has already been derived.

The discipline presented in this dissertation provides a basis for most of the above aspects of a methodology. The minimal required properties of the specification language, for the purpose of test frame generation, are given in Chapter 5. This discipline defines test procedures in terms of test frames and test steps. While it provides a formal definition of a test frame and a test step, this discipline does not prescribe precisely how to choose data values for test steps. Nor does it prescribe how to methodically sequence test frames within a test procedure. The discipline does provide some alternatives as to the amount of detail to be found in test frames, e.g., base vs. differentiated.

This discipline provides a precise mathematical definition of requirements coverage, and of how coverage criteria relate test frames to requirements. Algorithms that support traceability are provided for deriving a set of test frames to satisfy specific coverage criteria. However, no details are given as to when one coverage criterion should be used over another.

Regarding the effects of requirements changes, this dissertation has identified the Delta Problem and has shown that it is, in the worst case, undecidable. Although a heuristic partial solution is described, there is no evidence as to the applicability of this solution.

To summarize, the discipline presented in this dissertation provides a mathematical basis for a methodology for system-level requirements-based testing, and provides research opportunities for further development of a methodology.

9.2.4 Next Step for Q

The examples in Chapter 8 demonstrate that certain features of Q have been useful for formalizing natural language, stimulus-response requirements specifications for the purpose of system-level requirements-based testing. However, establishing that these features result in a language that can be easily learned and read by non-specialists will require further and controlled study.

The current implementation of Q, its reliance on S for underlying capabilities such as the definition of types, constraints, and functions other than predicates, makes it usable, but not terribly appealing, for general specification. Automating the declaration of predicates would improve the usefulness of this type of specification language. However, there must also be some mechanism to warn of situations where an author may have mis-spelled or mis-worded a predicate name. Future work should incorporate the concepts demonstrated in Q into a more generally applicable language.

9.2.5 Specification Projection

Watanabe and Sakamura [66] describe a manual test case generation strategy based on Z specifications that incorporates information about the implementation through a structure graph provided by programmers. Work such as this combines information from a specification, with information from the implementation. This is valuable for testing, since the combination allows the determination of additional input domain partitions. (The antecedent of a test frame is an example of such a partition.)

A similar idea is to project test frames from a formal requirements specification onto a formal design specification. This could serve to validate the design from a requirements perspective. Any test frames that did not fit onto the design indicate a deficiency in the design or an invalid test frame, which indicates an error in the requirements. Traceability would also provide a means of measuring progress during the design phase.

9.3 Epilogue

In an article published in the April 1996 issue of IEEE Computer, Hall [29] proposed the question: "What can formal methods contribute to improve the quality and decrease the cost of our systems?" This dissertation contributes part of an answer to this question. The discipline presented in this dissertation, which draws heavily from many aspects of formal methods, can improve the quality of our systems by ensuring that test procedures are developed according to objectively defined coverage criteria. The cost of developing our systems can be reduced in a variety of ways described in this dissertation including, but not limited to, the automation of some aspects of the task of deriving test procedures from requirements.

Bibliography

[1] Jean-Raymond Abrial. Steam boiler control specification problem. In Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors, *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, volume 1165 of *Lecture Notes in Computer Science*, pages 500–509, October 1996. http://www.informatik.uni-kiel.de/~procos/dag9523/dag9523.html.

[2] Paul Ammann and Jeff Offutt. Using formal methods to derive test frames in category-partition testing. In *Compass'94: 9th Annual Conference on Computer Assurance*, pages 69–80, Gaithersburg, MD, 1994. IEEE Computer Society Press. National Institute of Standards and Technology.

[3] Aonix. *Product Overview: Validator/Req*, June 1998. http://www.aonix.com/Pdfs/SQAS/Validator_PBr.ME.pdf (Figure 3).

[4] G. Battani and M. Meloni. Interpreteur du language de programmation PROLOG. Technical report, Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille II, 1973.

[5] Boris Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, 1995.

[6] G. Bernot, M-C. Gaudel, and B. Marre. Software testing based on formal specifications. *Software Engineering Journal*, 6(6), November 1991.

[7] Mark R. Blackburn and Robert D. Busser. T-VEC: A tool for developing critical systems. In *Compass'96: Eleventh Annual Conference on Computer Assurance*, pages 237–249, Gaithersburg, Maryland, 1996. National Institute of Standards and Technology.

[8] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[9] Alan Bundy. *The Computer Modelling of Mathematical Reasoning.* Academic Press, 1983.

[10] Juei Chang, Debra J. Richardson, and Sriram Sankar. Structural specification-based testing with ADL. In Steven J. Zeil, editor, *Proceedings of the 1996 International Symposium on Software Testing and analysis*, pages 62–70, New York, January 1996. ACM Press.

[11] T. Cheung and S. Ren. Executable test sequences and operational coverage for LOTOS specifications. In Jim Weeldreyer, editor, *Proceedings of the 12th Annual International Phoenix Conference on Computers and Communications*, pages 245–253, Tempe, AR, March 1993. IEEE Computer Society Press.

[12] John Joseph Chilenski and Philip H. Newcomb. Formal specification tools for test coverage analysis. *KBSE'94 Knowledge-Based Software Engineering*, pages 59–68, 1994.

[13] Kendra Cooper. Flex-fix predicates. conversation, June 1997.

[14] Nancy A. Day, Jeffrey J. Joyce, and Gerry Pelletier. Formalization and analysis of the separation minima for aircraft in the north atlantic: Complete specification and analysis results. Technical Report 97-12, Department of Computer Science, University of British Columbia, October 1997.

[15] Department of Defense, Washington D.C. *MIL-STD-498 Military Standard, Software Developement and Documentation*, December 1994.

[16] Michael S. Deutsch and Ronald R. Willis. *Software Quality Engineering.* Prentice-Hall, 1988.

[17] Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Formal Methods Europe '93*, volume 670 of *Lecture Notes in Computer Science*, pages 268–284. Springer-Verlag, 1993.

[18] Michael R. Donat. Automating formal specification-based testing. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97:Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE*, volume 1214 of *Lecture Notes in Computer Science*, pages 833–847. Springer-Verlag, April 1997.

[19] Michael R. Donat. Automatically generated test frames from a Q specification of ICAO flight plan form instructions. Technical Report TR-98-05, Department

of Computer Science, University of British Columbia, Vancouver, B.C., Canada, April 1998.

[20] Michael R. Donat. Automatically generated test frames from an S specification of separation minima for the North Atlantic region. Technical Report TR-98-04, Department of Computer Science, University of British Columbia, Vancouver, B.C., Canada, April 1998.

[21] Michael R. Donat and Jeffrey J. Joyce. Applying an automated test description tool to testing based on system level requirements. In *8th Annual Symposium of the International Council on Systems Engineering*, Vancouver, July 1998. International Council on Systems Engineering. http://www.incose.org.

[22] N. S. Eickelmann and Debra J. Richardson. An evaluation of software test environment architectures. In *18th International Conference on Software Engineering*, pages 353–365, Berlin - Heidelberg - New York, March 1996. Springer.

[23] Eurocontrol. *Operating Procedures for AIS Dynamic Data*, May 1997. EATCHIP draft AIS.ET1.ST05.1000-DEL-01.

[24] R. Ferguson and B. Korel. Software test data generation using the chaining approach. In *International Test Conference*, pages 703–709, Altoona, Pa., USA, October 1995. IEEE Computer Society Press.

[25] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H.Freeman and Company, San Francisco, 1979.

[26] Marie-Claude Gaudel. Testing can be formal, too. In *TAPSOFT: 6th International Joint Conference on Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96, 1995.

[27] C. George, P. Haff, K. Havelund, A.E. Haxthausen, R. Milne, C. Bendix Nielson, S. Prehn, and K.R. Wagner. *The Raise Specification Language.* Prentice Hall, New York, 1992.

[28] M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[29] Anthony Hall. Industrial practice: What is the formal methods debate anyway? *IEEE Computer*, 29(4):22–23, April 1996.

[30] I. Hayes and C. Jones. Specifications are not (necessarily) executable. *IEE Software Engineering Journal*, 4(6):330–338, November 1989.

[31] Ian Hayes. Specification directed module testing. *IEEE Transactions on Software Engineering*, 12(1):124–133, 1986.

[32] Steffen Helke, Thomas Neustupny, and Thomas Santen. Automating test case generation from Z specifications with Isabelle. In Jonathan Bowen, Mike Hinchey, and David Till, editors, *ZUM'97: The Z Formal Specification Notation, 10th International Conference of Z Users*, volume 1212 of *Lecture Notes in Computer Science*. Springer-Verlag, April 1997.

[33] R. M. Hierons. Extending test sequence overlap by invertibility. *The Computer Journal*, 39(4):325–330, 1996.

[34] Hans-Martin Hörcher. Improving software tests using Z specifications. In Jonathan P. Bowen and Michael G. Hinchey, editors, *ZUM '95 9th International Conference of Z Users, The Z Formal Specification Notation*, volume 967 of *Lecture Notes in Computer Science*, pages 152–166. Springer-Verlag, 1995.

[35] IEEE Standards Association, Washington D.C. *829-1983 (R1991) IEEE Standard for Software Test Documentation*, 1991.

[36] International Civil Aviation Organization, Montréal, Canada. *Rules of the Air and Air Traffic Services (PANS-RAC Doc 4444)*, November 1994. http://www.icao.int.

[37] C. B. Jones. *Systematic Software Development Using VDM (2nd edition)*. Prentice Hall, 1990.

[38] Jeffrey J. Joyce. TCEL. Proprietary document, April 1997.

[39] Jeffrey J. Joyce, Nancy Day, and Michael R. Donat. S: A machine readable specification notation based on higher order logic. In Thomas F. Melham and Juanito Camilleri, editors, *Higher Order Logic Theorem Proving and Its Applications, 7th International Workshop*, volume 859 of *Lecture Notes in Computer Science*, pages 285–299. Springer-Verlag, 1994.

[40] M. Karnaugh. The map method for synthesis of combinational logic circuits. *AIEE Transactions, Part I Communication and Electronics*, 72:593–599, November 1953.

[41] Gilbert Laycock. Formal specification and testing: A case study. *Software Testing, Verification and Reliability*, 2(1):7–23, May 1992.

[42] Gilbert Laycock. *The Theory and Practice of Specification Based Software Testing*. PhD thesis, University of Sheffield, April 1993.

[43] G. Luo, A. Das, and G. v. Bochmann. Software testing based on SDL specifications with save. *IEEE Transactions on Software Engineering*, 20(1):72–87, January 1994.

[44] Ian MacColl, David Carrington, and Philip Stocks. An experiment in specification-based testing. In K. Ramamohanarao, editor, *19th Australasian Computer Science Conference Proceedings (ACSC'96)*, pages 159–168, 1996.

[45] Brian Marick. *The Craft of Software Testing*. Prentice Hall, Englewood Cliffs, NJ, 1995.

[46] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[47] Lawrence C. Paulson. Designing a theorem prover. Technical Report 192, University of Cambridge, University of Cambridge, Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, England, May 1990.

[48] Lawrence C. Paulson. Designing a theorem prover. In *Handbook of Logic in Computer Science*, volume 2. Clarendon, 1992.

[49] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, second, paperback edition, 1992.

[50] Lawrence C. Paulson. Isabelle: A generic theorem prover. *Lecture Notes in Computer Science*, 828, 1994.

[51] David Poole, Alan Mackworth, and Randy Goebel. *Computational Intelligence*. Oxford University Press, January 1998.

[52] Debra J. Richardson, S. Leif-Aha, and T. O. OMalley. Specification-based Test Oracles for Reactive Systems. In *Proceedings of the 14th International Conference on Software Engineering*, pages 105–118, May 1992.

[53] Debra J. Richardson, T. O. O'Malley, C. T. Moore, and S. L. Aha. Developing and Integrating PRODAG in the Arcadia Environment. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, pages 109–119, December 1992.

[54] Debra J. Richardson and M. C. Thompson. An analysis of test data selection criteria using the RELAY model of fault detection. *IEEE Transactions on Software Engineering*, 19(6):533–553, June 1993.

[55] Debra J. Richardson and Alexander L. Wolf. Software testing at the architectural level. In *Joint Proceedings of the SIGSOFT '96 Workshops, Part 1*, pages 68–71, New York, October 1996. ACM Press.

[56] RTCA, Inc. and EUROCAE. *DO-178B, Software Considerations in Airbourne Systems and Equipment Certification*, 12B edition, December 1992.

[57] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, USA, July/August 1996. Springer-Verlag.

[58] Christian P. Schinagl. VDM specification of the steam-boiler control using RSL notation. In Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors, *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, volume 1165 of *Lecture Notes in Computer Science*, pages 428–452, October 1996.

[59] Ian Sommerville and Peter Sawyer. *Requirements Engineering*. John Wiley & Sons, Inc., Chichester, 1997.

[60] Space and Naval Warfare Systems Command, Washington D.C. *DOD-STD-2167A Military Standard, Defense System Software Developement*, February 1988.

[61] J. Michael Spivey. *Understanding Z: A Specification language and its formal semantics*. Cambridge University Press, 1988.

[62] Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, November 1996.

[63] Tadeusz Strzemecki. Polynomial-time algorithms for generation of prime implicants. *COMPLEXITY: Journal of Complexity*, 8:37–63, 1992.

[64] M. C. Thompson, Debra J. Richardson, and L. Clarke. An information flow model of fault detection. In Thomas Ostrand and Elaine Weyuker, editors,

Proceedings of the International Symposium on Software Testing and Analysis, pages 182–192, New York, NY, USA, June 1993. ACM Press.

[65] J. Voas, K. Miller, and J. Payne. Automating test case generation for coverages required by FAA standard DO-178B. In *Computers in Aerospace 9*, San Diego, CA, October 1993. American Institute of Aeronautics and Astronautics.

[66] A. Watanabe and K. Sakamura. A specification-based adaptive test case generation strategy for open operating system standards. In *18th International Conference on Software Engineering*, pages 81–89, Berlin - Heidelberg - New York, March 1996. Springer.

[67] Elaine Weyuker, Tarak Goradia, and Ashutosh Singh. Automatically generating test data from a Boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, May 1994.

Appendix A

Rules for Argument-Based Conjunctions and Disjunctions

The axioms below define the semantics of the Q phrases ``each of'' and ``any of.''

$$\forall\, x, l_1, l_2. Append\ (CONS\ x\ l_1)\ l_2\ =\ CONS\ x\ (Append\ l_1\ l_2)$$
$$\forall\, l. Append\ NIL\ l\ =\ l$$

$$\forall\, x, l, fn. Map\ (CONS\ x\ l)\ fn\ =\ CONS\ (fn\ x)\ (Map\ l\ fn)$$
$$\forall\, fn. Map\ NIL\ fn\ =\ NIL$$

$$AE\_Mx\ :\ (t1 \rightarrow t2)list\ \rightarrow\ (t1)list\ \rightarrow\ (t2)list$$
$$\forall\, x, l_1, l_2. AE\_Mx\ (CONS\ x\ l_1)\ l_2\ =\ Append\ (Map\ l_2\ x)\ (AE\_Mx\ l_1\ l_2)$$
$$\forall\, l. AE\_Mx\ NIL\ l\ =\ NIL$$

$AE\_Ue : t \rightarrow (t)list$

$\forall P, l. AE\_Ue\ (P\ (EACH\_OF\ l))\ =\ AE\_Mx\ (AE\_Ue\ P)\ l$

$\forall P, Q. AE\_Ue\ (P\ Q)\ =\ AE\_Mx\ (AE\_Ue\ P)\ (AE\_Ue\ Q)$

$\forall P. AE\_Ue\ P\ =\ [P]$, where P is an atom

$AE\_Ua : t \rightarrow (t)list$

$\forall P, l. AE\_Ua\ (P\ (ANY\_OF\ l))\ =\ AE\_Mx\ (AE\_Ua\ P)\ l$

$\forall P, Q. AE\_Ua\ (P\ Q)\ =\ AE\_Mx\ (AE\_Ua\ P)\ (AE\_Ua\ Q)$

$\forall P. AE\_Ua\ P\ =\ [P]$, where P is an atom

Appendix B

Automatically Generated Test Frames for the Steam Boiler Control

The test frames presented in this appendix were automatically generated from an S specification of a portion of Schinagl's VDM style RSL specification for Abrial's steam boiler specification problem.

B.1 S Specification

```
%include startup.s

(:t) Exists_unique (P:t -> bool) :=
    (exists v.P v)
        /\ (forall v1.forall v2.P v1 /\ P v2 ==> (v1 = v2));

inmess_ok : bool;
```

```
:PUMP;
:STATE;

:message :=
    PumpState :(PUMP # STATE)
  | PumpCtrState :(PUMP # STATE)
  | Level :num
  | Steam :num
  | SteamBoilerWaiting
  | PhysicalUnitsReady
  | PumpRep :PUMP
  | PumpCtrRep :PUMP
  | PumpFlrAck :PUMP
  | PumpCtrFlrAck :PUMP
  | LevelRep
  | SteamRep
  | LevelFlrAck
  | SteamFlrAck;

InMess : (message)set;

Waiting,Ready : bool;
States [Waiting; Ready];

:MODE;
Working,Repairing,Broken : MODE;
forall P.States [P Working; P Repairing; P Broken];

% Mst = software opinion of the state of the control unit
% Pst = software opinion of the state of the pump
```

```
Mst,Pst : PUMP -> MODE -> bool;

% Qst = software opinion of the state of the water level indicator
% Vst = software opinion of the state of the steam indicator
Qst,Vst : MODE -> bool;

MaxWater : num;
MaxSteam : num;

SetInMessOK :=
    inmess_ok <=>
        (forall p.
            (Exists_unique (\s.PumpState(p, s) In InMess)) /\
            (Exists_unique (\s.PumpCtrState(p, s) In InMess))) /\
        (Exists_unique (\l.Level l In InMess)) /\
        (select l.Level l In InMess) <= MaxWater /\
        (Exists_unique (\l.Steam l In InMess)) /\
        (select l.Steam l In InMess) <= MaxSteam /\
        (SteamBoilerWaiting In InMess ==> Waiting) /\
        (PhysicalUnitsReady In InMess ==> Ready) /\
        (forall p.
            (PumpRep p In InMess ==> Pst p Repairing) /\
            (PumpCtrRep p In InMess ==> Mst p Repairing) /\
            (PumpFlrAck p In InMess ==> Pst p Broken) /\
            (PumpCtrFlrAck p In InMess ==> Mst p Broken)) /\
        (LevelRep In InMess ==> Qst Repairing) /\
        (SteamRep In InMess ==> Vst Repairing) /\
        (LevelFlrAck In InMess ==> Qst Broken) /\
        (SteamFlrAck In InMess ==> Vst Broken);

%no_expand In
```

B.2 Base Test Frames

–Test Frame 1.1:

Stimuli	Response
1. Level v1 $\in$ InMess 2. Level v2 $\in$ InMess 3. $\neg$ (v1 = v2)	1. $\neg$ inmess_ok

–Test Frame 1.2:

Stimuli	Response
1. PumpState (p , v1') $\in$ InMess 2. PumpState (p , v2') $\in$ InMess 3. $\neg$ (v1' = v2')	1. $\neg$ inmess_ok

–Test Frame 1.3:

Stimuli	Response
1. SteamFlrAck $\in$ InMess 2. $\neg$ (Vst Broken)	1. $\neg$ inmess_ok

–Test Frame 1.4:

Stimuli	Response
1. LevelFlrAck $\in$ InMess 2. $\neg$ (Qst Broken)	1. $\neg$ inmess_ok

–Test Frame 1.5:

Stimuli	Response
1. SteamRep $\in$ InMess 2. $\neg$ (Vst Repairing)	1. $\neg$ inmess_ok

–Test Frame 1.6:

Stimuli	Response
1. LevelRep $\in$ InMess 2. $\neg$ (Qst Repairing)	1. $\neg$ inmess_ok

–Test Frame 1.7:

Stimuli	Response
1. PumpCtrFlrAck p' $\in$ InMess 2. $\neg$ (Mst p' Broken)	1. $\neg$ inmess_ok

–Test Frame 1.8:

Stimuli	Response
1. PumpFlrAck p' $\in$ InMess 2. $\neg$ (Pst p' Broken)	1. $\neg$ inmess_ok

–Test Frame 1.9:

Stimuli	Response
1. PumpCtrRep p' $\in$ InMess 2. $\neg$ (Mst p' Repairing)	1. $\neg$ inmess_ok

–Test Frame 1.10:

Stimuli	Response
1. PumpRep p' $\in$ InMess 2. $\neg$ (Pst p' Repairing)	1. $\neg$ inmess_ok

–Test Frame 1.11:

Stimuli	Response
1. PhysicalUnitsReady $\in$ InMess 2. $\neg$ Ready	1. $\neg$ inmess_ok

–Test Frame 1.12:

Stimuli	Response
1. SteamBoilerWaiting $\in$ InMess 2. $\neg$ Waiting	1. $\neg$ inmess_ok

–Test Frame 1.13:

Stimuli	Response
1. $\neg$ (v1 = v2) 2. Steam v1 $\in$ InMess 3. Steam v2 $\in$ InMess	1. $\neg$ inmess_ok

–Test Frame 1.14:

Stimuli	Response
1. $\neg$ (v1' = v2') 2. PumpCtrState (p , v1') $\in$ InMess 3. PumpCtrState (p , v2') $\in$ InMess	1. $\neg$ inmess_ok

–Test Frame 1.15:

Stimuli	Response
1. $\forall$ v.$\neg$ (PumpState (p , v) $\in$ InMess)	1. $\neg$ inmess_ok

–Test Frame 1.16:

Stimuli	Response
1. $\forall$ v.$\neg$ (PumpCtrState (p , v) $\in$ InMess)	1. $\neg$ inmess_ok

–Test Frame 1.17:

Stimuli	Response
1. $\forall$ v.$\neg$ (Level v $\in$ InMess)	1. $\neg$ inmess_ok

–Test Frame 1.18:

Stimuli	Response
1. $\forall$ v.$\neg$ (Steam v $\in$ InMess)	1. $\neg$ inmess_ok

–Test Frame 1.19:

Stimuli	Response
1. $\neg$ ((select l.Steam l $\in$ InMess) $\leqslant$ MaxSteam)	1. $\neg$ inmess_ok

–Test Frame 1.20:

Stimuli	Response
1. $\neg$ ((select l.Level l $\in$ InMess) $\leqslant$ MaxWater)	1. $\neg$ inmess_ok

–Test Frame 2.1:

Stimuli	Response
1. $\forall$ p.$\exists$ v.PumpState (p , v) $\in$ InMess 2. $\forall$ v1. $\forall$ v2. ($\forall$ p. $\neg$ (PumpState (p , v1) $\in$ InMess) $\vee$ $\neg$ (PumpState (p , v2) $\in$ InMess)) $\vee$ (v1 = v2) 3. $\forall$ p.$\exists$ v.PumpCtrState (p , v) $\in$ InMess 4. $\forall$ v1. $\forall$ v2. ($\forall$ p. $\neg$ (PumpCtrState (p , v1) $\in$ InMess) $\vee$ $\neg$ (PumpCtrState (p , v2) $\in$ InMess)) $\vee$ (v1 = v2) 5. Level v $\in$ InMess 6. $\forall$ v1. $\forall$ v2. $\neg$ (Level v1 $\in$ InMess) $\vee$ $\neg$ (Level v2 $\in$ InMess) $\vee$ (v1 = v2) 7. (select l.Level l $\in$ InMess) $\leqslant$ MaxWater 8. Steam v' $\in$ InMess 9. $\forall$ v1. $\forall$ v2. $\neg$ (Steam v1 $\in$ InMess) $\vee$ $\neg$ (Steam v2 $\in$ InMess) $\vee$ (v1 = v2) 10. (select l.Steam l $\in$ InMess) $\leqslant$ MaxSteam 11. Waiting 12. $\neg$ (PhysicalUnitsReady $\in$ InMess) 13. $\forall$ p.$\neg$ (PumpRep p $\in$ InMess) $\vee$ Pst p Repairing 14. $\forall$ p.$\neg$ (PumpCtrRep p $\in$ InMess) $\vee$ Mst p Repairing 15. $\forall$ p.$\neg$ (PumpFlrAck p $\in$ InMess) $\vee$ Pst p Broken 16. $\forall$ p.$\neg$ (PumpCtrFlrAck p $\in$ InMess) $\vee$ Mst p Broken 17. Qst Repairing 18. Vst Repairing 19. $\neg$ (LevelFlrAck $\in$ InMess) 20. $\neg$ (SteamFlrAck $\in$ InMess)	1. inmess_ok

–Test Frame 2.2:

Stimuli	Response
1. $\forall$ p.$\exists$ v.PumpState (p , v) $\in$ InMess 2. $\forall$ v1. $\forall$ v2. ($\forall$ p. $\neg$ (PumpState (p , v1) $\in$ InMess) $\vee$ $\neg$ (PumpState (p , v2) $\in$ InMess)) $\vee$ (v1 = v2) 3. $\forall$ p.$\exists$ v.PumpCtrState (p , v) $\in$ InMess 4. $\forall$ v1. $\forall$ v2. ($\forall$ p. $\neg$ (PumpCtrState (p , v1) $\in$ InMess) $\vee$ $\neg$ (PumpCtrState (p , v2) $\in$ InMess)) $\vee$ (v1 = v2) 5. Level v $\in$ InMess 6. $\forall$ v1. $\forall$ v2. $\neg$ (Level v1 $\in$ InMess) $\vee$ $\neg$ (Level v2 $\in$ InMess) $\vee$ (v1 = v2) 7. (select l.Level l $\in$ InMess) $\leqslant$ MaxWater 8. Steam v' $\in$ InMess 9. $\forall$ v1. $\forall$ v2. $\neg$ (Steam v1 $\in$ InMess) $\vee$ $\neg$ (Steam v2 $\in$ InMess) $\vee$ (v1 = v2) 10. (select l.Steam l $\in$ InMess) $\leqslant$ MaxSteam 11. $\neg$ (SteamBoilerWaiting $\in$ InMess) 12. Ready 13. $\forall$ p.$\neg$ (PumpRep p $\in$ InMess) $\vee$ Pst p Repairing 14. $\forall$ p.$\neg$ (PumpCtrRep p $\in$ InMess) $\vee$ Mst p Repairing 15. $\forall$ p.$\neg$ (PumpFlrAck p $\in$ InMess) $\vee$ Pst p Broken 16. $\forall$ p.$\neg$ (PumpCtrFlrAck p $\in$ InMess) $\vee$ Mst p Broken 17. $\neg$ (LevelRep $\in$ InMess) 18. $\neg$ (SteamRep $\in$ InMess) 19. Qst Broken 20. Vst Broken	1. inmess_ok

B.3 Differentiated Test Frames

–Test Frame 1.1:

Stimuli	Response
1. Level $v'' \in$ InMess 2. Level $v''' \in$ InMess 3. $\neg (v'' = v''')$ 4. (select l.Level l $\in$ InMess) $\leqslant$ MaxWater 5. (select l.Steam l $\in$ InMess) $\leqslant$ MaxSteam 6. Waiting 7. $\neg$ (PhysicalUnitsReady $\in$ InMess) 8. Qst Repairing 9. Vst Repairing 10. $\neg$ (LevelFlrAck $\in$ InMess) 11. $\neg$ (SteamFlrAck $\in$ InMess) 12. $\forall$ p. $\forall$ v1$'$. $\neg$ (PumpState (p , v1$'$) $\in$ InMess) $\vee$ ($\forall$ v2$'$. $\neg$ (PumpState (p , v2$'$) $\in$ InMess) $\vee$ (v1$'$ = v2$'$)) 13. $\forall$ p$'$.$\neg$ (PumpCtrFlrAck p$'$ $\in$ InMess) $\vee$ Mst p$'$ Broken 14. $\forall$ p$'$.$\neg$ (PumpFlrAck p$'$ $\in$ InMess) $\vee$ Pst p$'$ Broken 15. $\forall$ p$'$.$\neg$ (PumpCtrRep p$'$ $\in$ InMess) $\vee$ Mst p$'$ Repairing 16. $\forall$ p$'$.$\neg$ (PumpRep p$'$ $\in$ InMess) $\vee$ Pst p$'$ Repairing 17. $\forall$ v1. $\forall$ v2. (v1 = v2) $\vee$ $\neg$ (Steam v1 $\in$ InMess) $\vee$ $\neg$ (Steam v2 $\in$ InMess) 18. $\forall$ v1$'$. $\forall$ v2$'$. (v1$'$ = v2$'$) $\vee$ ($\forall$ p. $\neg$ (PumpCtrState (p , v1$'$) $\in$ InMess) $\vee$ $\neg$ (PumpCtrState (p , v2$'$) $\in$ InMess)) 19. $\forall$ p.$\exists$ v.PumpState (p , v) $\in$ InMess 20. $\forall$ p.$\exists$ v.PumpCtrState (p , v) $\in$ InMess 21. Level $v'''' \in$ InMess 22. Steam $v' \in$ InMess	1. $\neg$ inmess_ok

–Test Frame 1.2:

Stimuli	Response
1. PumpState (p''' , $v1''$) $\in$ InMess 2. PumpState (p''' , $v2''$) $\in$ InMess 3. $\neg$ ($v1'' = v2''$) 4. (select l.Level l $\in$ InMess) $\leqslant$ MaxWater 5. (select l.Steam l $\in$ InMess) $\leqslant$ MaxSteam 6. Waiting 7. $\neg$ (PhysicalUnitsReady $\in$ InMess) 8. Qst Repairing 9. Vst Repairing 10. $\neg$ (LevelFlrAck $\in$ InMess) 11. $\neg$ (SteamFlrAck $\in$ InMess) 12. $\forall$ p'.$\neg$ (PumpCtrFlrAck p' $\in$ InMess) $\vee$ Mst p' Broken 13. $\forall$ p'.$\neg$ (PumpFlrAck p' $\in$ InMess) $\vee$ Pst p' Broken 14. $\forall$ p'.$\neg$ (PumpCtrRep p' $\in$ InMess) $\vee$ Mst p' Repairing 15. $\forall$ p'.$\neg$ (PumpRep p' $\in$ InMess) $\vee$ Pst p' Repairing 16. $\forall$ v1. $\forall$ v2. (v1 = v2) $\vee$ $\neg$ (Steam v1 $\in$ InMess) $\vee$ $\neg$ (Steam v2 $\in$ InMess) 17. $\forall$ $v1'$. $\forall$ $v2'$. ($v1' = v2'$) $\vee$ ($\forall$ p. $\neg$ (PumpCtrState (p , $v1'$) $\in$ InMess) $\vee$ $\neg$ (PumpCtrState (p , $v2'$) $\in$ InMess)) 18. $\forall$ p.$\exists$ v.PumpState (p , v) $\in$ InMess 19. $\forall$ p.$\exists$ v.PumpCtrState (p , v) $\in$ InMess 20. Level v'''' $\in$ InMess 21. Steam v' $\in$ InMess 22. $\forall$ v1. $\neg$ (Level v1 $\in$ InMess) $\vee$ ($\forall$ v2.$\neg$ (Level v2 $\in$ InMess) $\vee$ (v1 = v2))	1. $\neg$ inmess_ok

–Test Frame 1.3:

Stimuli	Response
1. SteamFlrAck $\in$ InMess 2. (select l.Level l $\in$ InMess) $\leqslant$ MaxWater 3. (select l.Steam l $\in$ InMess) $\leqslant$ MaxSteam 4. Waiting 5. $\neg$ (PhysicalUnitsReady $\in$ InMess) 6. Qst Repairing 7. Vst Repairing 8. $\neg$ (LevelFlrAck $\in$ InMess) 9. $\forall$ p. $\forall$ v1$'$. $\neg$ (PumpState (p , v1$'$) $\in$ InMess) $\vee$ ($\forall$ v2$'$. $\neg$ (PumpState (p , v2$'$) $\in$ InMess) $\vee$ (v1$'$ = v2$'$)) 10. $\forall$ p$'$.$\neg$ (PumpCtrFlrAck p$'$ $\in$ InMess) $\vee$ Mst p$'$ Broken 11. $\forall$ p$'$.$\neg$ (PumpFlrAck p$'$ $\in$ InMess) $\vee$ Pst p$'$ Broken 12. $\forall$ p$'$.$\neg$ (PumpCtrRep p$'$ $\in$ InMess) $\vee$ Mst p$'$ Repairing 13. $\forall$ p$'$.$\neg$ (PumpRep p$'$ $\in$ InMess) $\vee$ Pst p$'$ Repairing 14. $\forall$ v1. $\forall$ v2. (v1 = v2) $\vee$ $\neg$ (Steam v1 $\in$ InMess) $\vee$ $\neg$ (Steam v2 $\in$ InMess) 15. $\forall$ v1$'$. $\forall$ v2$'$. (v1$'$ = v2$'$) $\vee$ ($\forall$ p. $\neg$ (PumpCtrState (p , v1$'$) $\in$ InMess) $\vee$ $\neg$ (PumpCtrState (p , v2$'$) $\in$ InMess)) 16. $\forall$ p.$\exists$ v.PumpState (p , v) $\in$ InMess 17. $\forall$ p.$\exists$ v.PumpCtrState (p , v) $\in$ InMess 18. Level v$''$ $\in$ InMess 19. Steam v$'''$ $\in$ InMess 20. $\forall$ v1. $\neg$ (Level v1 $\in$ InMess) $\vee$ ($\forall$ v2.$\neg$ (Level v2 $\in$ InMess) $\vee$ (v1 = v2))	1. $\neg$ inmess_ok

–Test Frame 1.4:

Stimuli	Response
1. LevelFlrAck $\in$ InMess 2. (select l.Level l $\in$ InMess) $\leqslant$ MaxWater 3. (select l.Steam l $\in$ InMess) $\leqslant$ MaxSteam 4. Waiting 5. $\neg$ (PhysicalUnitsReady $\in$ InMess) 6. Qst Repairing 7. Vst Repairing 8. $\neg$ (SteamFlrAck $\in$ InMess) 9. $\forall$ p. $\forall$ v1$'$. $\neg$ (PumpState (p , v1$'$) $\in$ InMess) $\vee$ ($\forall$ v2$'$. $\neg$ (PumpState (p , v2$'$) $\in$ InMess) $\vee$ (v1$'$ = v2$'$)) 10. $\forall$ p$'$.$\neg$ (PumpCtrFlrAck p$'$ $\in$ InMess) $\vee$ Mst p$'$ Broken 11. $\forall$ p$'$.$\neg$ (PumpFlrAck p$'$ $\in$ InMess) $\vee$ Pst p$'$ Broken 12. $\forall$ p$'$.$\neg$ (PumpCtrRep p$'$ $\in$ InMess) $\vee$ Mst p$'$ Repairing 13. $\forall$ p$'$.$\neg$ (PumpRep p$'$ $\in$ InMess) $\vee$ Pst p$'$ Repairing 14. $\forall$ v1. $\forall$ v2. (v1 = v2) $\vee$ $\neg$ (Steam v1 $\in$ InMess) $\vee$ $\neg$ (Steam v2 $\in$ InMess) 15. $\forall$ v1$'$. $\forall$ v2$'$. (v1$'$ = v2$'$) $\vee$ ($\forall$ p. $\neg$ (PumpCtrState (p , v1$'$) $\in$ InMess) $\vee$ $\neg$ (PumpCtrState (p , v2$'$) $\in$ InMess)) 16. $\forall$ p.$\exists$ v.PumpState (p , v) $\in$ InMess 17. $\forall$ p.$\exists$ v.PumpCtrState (p , v) $\in$ InMess 18. Level v$''$ $\in$ InMess 19. Steam v$'''$ $\in$ InMess 20. $\forall$ v1. $\neg$ (Level v1 $\in$ InMess) $\vee$ ($\forall$ v2.$\neg$ (Level v2 $\in$ InMess) $\vee$ (v1 = v2))	1. $\neg$ inmess_ok

–Test Frame 1.5:

Stimuli	Response
1. SteamRep $\in$ InMess 2. (select l.Level l $\in$ InMess) $\leqslant$ MaxWater 3. (select l.Steam l $\in$ InMess) $\leqslant$ MaxSteam 4. Waiting 5. $\neg$ (PhysicalUnitsReady $\in$ InMess) 6. Qst Repairing 7. $\neg$ (LevelFlrAck $\in$ InMess) 8. Vst Broken 9. $\forall$ p. $\forall$ v1$'$. $\neg$ (PumpState (p , v1$'$) $\in$ InMess) $\vee$ ($\forall$ v2$'$. $\neg$ (PumpState (p , v2$'$) $\in$ InMess) $\vee$ (v1$'$ = v2$'$)) 10. $\forall$ p$'$.$\neg$ (PumpCtrFlrAck p$'$ $\in$ InMess) $\vee$ Mst p$'$ Broken 11. $\forall$ p$'$.$\neg$ (PumpFlrAck p$'$ $\in$ InMess) $\vee$ Pst p$'$ Broken 12. $\forall$ p$'$.$\neg$ (PumpCtrRep p$'$ $\in$ InMess) $\vee$ Mst p$'$ Repairing 13. $\forall$ p$'$.$\neg$ (PumpRep p$'$ $\in$ InMess) $\vee$ Pst p$'$ Repairing 14. $\forall$ v1. $\forall$ v2. (v1 = v2) $\vee$ $\neg$ (Steam v1 $\in$ InMess) $\vee$ $\neg$ (Steam v2 $\in$ InMess) 15. $\forall$ v1$'$. $\forall$ v2$'$. (v1$'$ = v2$'$) $\vee$ ($\forall$ p. $\neg$ (PumpCtrState (p , v1$'$) $\in$ InMess) $\vee$ $\neg$ (PumpCtrState (p , v2$'$) $\in$ InMess)) 16. $\forall$ p.$\exists$ v.PumpState (p , v) $\in$ InMess 17. $\forall$ p.$\exists$ v.PumpCtrState (p , v) $\in$ InMess 18. Level v$''$ $\in$ InMess 19. Steam v$'''$ $\in$ InMess 20. $\forall$ v1. $\neg$ (Level v1 $\in$ InMess) $\vee$ ($\forall$ v2.$\neg$ (Level v2 $\in$ InMess) $\vee$ (v1 = v2))	1. $\neg$ inmess_ok

–Test Frame 1.6:

Stimuli	Response
1. LevelRep $\in$ InMess 2. (select l.Level l $\in$ InMess) $\leqslant$ MaxWater 3. (select l.Steam l $\in$ InMess) $\leqslant$ MaxSteam 4. Waiting 5. $\neg$ (PhysicalUnitsReady $\in$ InMess) 6. Vst Repairing 7. Qst Broken 8. $\neg$ (SteamFlrAck $\in$ InMess) 9. $\forall$ p. $\forall$ v1$'$. $\neg$ (PumpState (p , v1$'$) $\in$ InMess) $\vee$ ($\forall$ v2$'$. $\neg$ (PumpState (p , v2$'$) $\in$ InMess) $\vee$ (v1$'$ = v2$'$)) 10. $\forall$ p$'$.$\neg$ (PumpCtrFlrAck p$'$ $\in$ InMess) $\vee$ Mst p$'$ Broken 11. $\forall$ p$'$.$\neg$ (PumpFlrAck p$'$ $\in$ InMess) $\vee$ Pst p$'$ Broken 12. $\forall$ p$'$.$\neg$ (PumpCtrRep p$'$ $\in$ InMess) $\vee$ Mst p$'$ Repairing 13. $\forall$ p$'$.$\neg$ (PumpRep p$'$ $\in$ InMess) $\vee$ Pst p$'$ Repairing 14. $\forall$ v1. $\forall$ v2. (v1 = v2) $\vee$ $\neg$ (Steam v1 $\in$ InMess) $\vee$ $\neg$ (Steam v2 $\in$ InMess) 15. $\forall$ v1$'$. $\forall$ v2$'$. (v1$'$ = v2$'$) $\vee$ ($\forall$ p. $\neg$ (PumpCtrState (p , v1$'$) $\in$ InMess) $\vee$ $\neg$ (PumpCtrState (p , v2$'$) $\in$ InMess)) 16. $\forall$ p.$\exists$ v.PumpState (p , v) $\in$ InMess 17. $\forall$ p.$\exists$ v.PumpCtrState (p , v) $\in$ InMess 18. Level v$''$ $\in$ InMess 19. Steam v$'''$ $\in$ InMess 20. $\forall$ v1. $\neg$ (Level v1 $\in$ InMess) $\vee$ ($\forall$ v2.$\neg$ (Level v2 $\in$ InMess) $\vee$ (v1 = v2))	1. $\neg$ inmess_ok

–Test Frame 1.7:

Stimuli	Response
1. PumpCtrFlrAck p''' $\in$ InMess 2. $\neg$ (Mst p''' Broken) 3. (select l.Level l $\in$ InMess) $\leqslant$ MaxWater 4. (select l.Steam l $\in$ InMess) $\leqslant$ MaxSteam 5. Waiting 6. $\neg$ (PhysicalUnitsReady $\in$ InMess) 7. Qst Repairing 8. Vst Repairing 9. $\neg$ (LevelFlrAck $\in$ InMess) 10. $\neg$ (SteamFlrAck $\in$ InMess) 11. $\forall$ p. $\forall$ $v1'$. $\neg$ (PumpState (p , $v1'$) $\in$ InMess) $\vee$ ($\forall$ $v2'$. $\neg$ (PumpState (p , $v2'$) $\in$ InMess) $\vee$ ($v1' = v2'$)) 12. $\forall$ p'.$\neg$ (PumpFlrAck p' $\in$ InMess) $\vee$ Pst p' Broken 13. $\forall$ p'.$\neg$ (PumpCtrRep p' $\in$ InMess) $\vee$ Mst p' Repairing 14. $\forall$ p'.$\neg$ (PumpRep p' $\in$ InMess) $\vee$ Pst p' Repairing 15. $\forall$ v1. $\forall$ v2. (v1 = v2) $\vee$ $\neg$ (Steam v1 $\in$ InMess) $\vee$ $\neg$ (Steam v2 $\in$ InMess) 16. $\forall$ $v1'$. $\forall$ $v2'$. ($v1' = v2'$) $\vee$ ($\forall$ p. $\neg$ (PumpCtrState (p , $v1'$) $\in$ InMess) $\vee$ $\neg$ (PumpCtrState (p , $v2'$) $\in$ InMess)) 17. $\forall$ p.$\exists$ v.PumpState (p , v) $\in$ InMess 18. $\forall$ p.$\exists$ v.PumpCtrState (p , v) $\in$ InMess 19. Level v'' $\in$ InMess 20. Steam v''' $\in$ InMess 21. $\forall$ v1. $\neg$ (Level v1 $\in$ InMess) $\vee$ ($\forall$ v2.$\neg$ (Level v2 $\in$ InMess) $\vee$ (v1 = v2))	1. $\neg$ inmess_ok

–Test Frame 1.8:

Stimuli	Response
1. PumpFlrAck $p''' \in$ InMess 2. $\neg$ (Pst p''' Broken) 3. (select l.Level l $\in$ InMess) $\leqslant$ MaxWater 4. (select l.Steam l $\in$ InMess) $\leqslant$ MaxSteam 5. Waiting 6. $\neg$ (PhysicalUnitsReady $\in$ InMess) 7. Qst Repairing 8. Vst Repairing 9. $\neg$ (LevelFlrAck $\in$ InMess) 10. $\neg$ (SteamFlrAck $\in$ InMess) 11. $\forall$ p. $\forall$ v1$'$. $\neg$ (PumpState (p , v1$'$) $\in$ InMess) $\vee$ ($\forall$ v2$'$. $\neg$ (PumpState (p , v2$'$) $\in$ InMess) $\vee$ (v1$'$ = v2$'$)) 12. $\forall$ p$'$.$\neg$ (PumpCtrFlrAck p$'$ $\in$ InMess) $\vee$ Mst p$'$ Broken 13. $\forall$ p$'$.$\neg$ (PumpCtrRep p$'$ $\in$ InMess) $\vee$ Mst p$'$ Repairing 14. $\forall$ p$'$.$\neg$ (PumpRep p$'$ $\in$ InMess) $\vee$ Pst p$'$ Repairing 15. $\forall$ v1. $\forall$ v2. (v1 = v2) $\vee$ $\neg$ (Steam v1 $\in$ InMess) $\vee$ $\neg$ (Steam v2 $\in$ InMess) 16. $\forall$ v1$'$. $\forall$ v2$'$. (v1$'$ = v2$'$) $\vee$ ($\forall$ p. $\neg$ (PumpCtrState (p , v1$'$) $\in$ InMess) $\vee$ $\neg$ (PumpCtrState (p , v2$'$) $\in$ InMess)) 17. $\forall$ p.$\exists$ v.PumpState (p , v) $\in$ InMess 18. $\forall$ p.$\exists$ v.PumpCtrState (p , v) $\in$ InMess 19. Level v$''$ $\in$ InMess 20. Steam v$'''$ $\in$ InMess 21. $\forall$ v1. $\neg$ (Level v1 $\in$ InMess) $\vee$ ($\forall$ v2.$\neg$ (Level v2 $\in$ InMess) $\vee$ (v1 = v2))	1. $\neg$ inmess_ok

–Test Frame 1.9:

Stimuli	Response
1. PumpCtrRep $p''' \in$ InMess 2. $\neg$ (Mst p''' Repairing) 3. (select l.Level l $\in$ InMess) $\leqslant$ MaxWater 4. (select l.Steam l $\in$ InMess) $\leqslant$ MaxSteam 5. Waiting 6. $\neg$ (PhysicalUnitsReady $\in$ InMess) 7. Qst Repairing 8. Vst Repairing 9. $\neg$ (LevelFlrAck $\in$ InMess) 10. $\neg$ (SteamFlrAck $\in$ InMess) 11. $\forall$ p. $\forall$ v1$'$. $\neg$ (PumpState (p , v1$'$) $\in$ InMess) $\vee$ ($\forall$ v2$'$. $\neg$ (PumpState (p , v2$'$) $\in$ InMess) $\vee$ (v1$'$ = v2$'$)) 12. $\forall$ p$'$.$\neg$ (PumpCtrFlrAck p$'$ $\in$ InMess) $\vee$ Mst p$'$ Broken 13. $\forall$ p$'$.$\neg$ (PumpFlrAck p$'$ $\in$ InMess) $\vee$ Pst p$'$ Broken 14. $\forall$ p$'$.$\neg$ (PumpRep p$'$ $\in$ InMess) $\vee$ Pst p$'$ Repairing 15. $\forall$ v1. $\forall$ v2. (v1 = v2) $\vee$ $\neg$ (Steam v1 $\in$ InMess) $\vee$ $\neg$ (Steam v2 $\in$ InMess) 16. $\forall$ v1$'$. $\forall$ v2$'$. (v1$'$ = v2$'$) $\vee$ ($\forall$ p. $\neg$ (PumpCtrState (p , v1$'$) $\in$ InMess) $\vee$ $\neg$ (PumpCtrState (p , v2$'$) $\in$ InMess)) 17. $\forall$ p.$\exists$ v.PumpState (p , v) $\in$ InMess 18. $\forall$ p.$\exists$ v.PumpCtrState (p , v) $\in$ InMess 19. Level v$''$ $\in$ InMess 20. Steam v$'''$ $\in$ InMess 21. $\forall$ v1. $\neg$ (Level v1 $\in$ InMess) $\vee$ ($\forall$ v2.$\neg$ (Level v2 $\in$ InMess) $\vee$ (v1 = v2))	1. $\neg$ inmess_ok

–Test Frame 1.10:

Stimuli	Response
1. PumpRep p''' ∈ InMess 2. ¬ (Pst p''' Repairing) 3. (select l.Level l ∈ InMess) $\leqslant$ MaxWater 4. (select l.Steam l ∈ InMess) $\leqslant$ MaxSteam 5. Waiting 6. ¬ (PhysicalUnitsReady ∈ InMess) 7. Qst Repairing 8. Vst Repairing 9. ¬ (LevelFlrAck ∈ InMess) 10. ¬ (SteamFlrAck ∈ InMess) 11. ∀ p. ∀ $v1'$. ¬ (PumpState (p , $v1'$) ∈ InMess) ∨ (∀ $v2'$. ¬ (PumpState (p , $v2'$) ∈ InMess) ∨ ($v1' = v2'$)) 12. ∀ p'.¬ (PumpCtrFlrAck p' ∈ InMess) ∨ Mst p' Broken 13. ∀ p'.¬ (PumpFlrAck p' ∈ InMess) ∨ Pst p' Broken 14. ∀ p'.¬ (PumpCtrRep p' ∈ InMess) ∨ Mst p' Repairing 15. ∀ v1. ∀ v2. (v1 = v2) ∨ ¬ (Steam v1 ∈ InMess) ∨ ¬ (Steam v2 ∈ InMess) 16. ∀ $v1'$. ∀ $v2'$. ($v1' = v2'$) ∨ (∀ p. ¬ (PumpCtrState (p , $v1'$) ∈ InMess) ∨ ¬ (PumpCtrState (p , $v2'$) ∈ InMess)) 17. ∀ p.∃ v.PumpState (p , v) ∈ InMess 18. ∀ p.∃ v.PumpCtrState (p , v) ∈ InMess 19. Level v'' ∈ InMess 20. Steam v''' ∈ InMess 21. ∀ v1. ¬ (Level v1 ∈ InMess) ∨ (∀ v2.¬ (Level v2 ∈ InMess) ∨ (v1 = v2))	1. ¬ inmess_ok

–Test Frame 1.11:

Stimuli	Response
1. PhysicalUnitsReady $\in$ InMess 2. (select l.Level l $\in$ InMess) $\leqslant$ MaxWater 3. (select l.Steam l $\in$ InMess) $\leqslant$ MaxSteam 4. Waiting 5. Qst Repairing 6. Vst Repairing 7. $\neg$ (LevelFlrAck $\in$ InMess) 8. $\neg$ (SteamFlrAck $\in$ InMess) 9. $\forall$ p. $\forall$ v1$'$. $\neg$ (PumpState (p , v1$'$) $\in$ InMess) $\vee$ ($\forall$ v2$'$. $\neg$ (PumpState (p , v2$'$) $\in$ InMess) $\vee$ (v1$'$ = v2$'$)) 10. $\forall$ p$'$.$\neg$ (PumpCtrFlrAck p$'$ $\in$ InMess) $\vee$ Mst p$'$ Broken 11. $\forall$ p$'$.$\neg$ (PumpFlrAck p$'$ $\in$ InMess) $\vee$ Pst p$'$ Broken 12. $\forall$ p$'$.$\neg$ (PumpCtrRep p$'$ $\in$ InMess) $\vee$ Mst p$'$ Repairing 13. $\forall$ p$'$.$\neg$ (PumpRep p$'$ $\in$ InMess) $\vee$ Pst p$'$ Repairing 14. $\forall$ v1. $\forall$ v2. (v1 = v2) $\vee$ $\neg$ (Steam v1 $\in$ InMess) $\vee$ $\neg$ (Steam v2 $\in$ InMess) 15. $\forall$ v1$'$. $\forall$ v2$'$. (v1$'$ = v2$'$) $\vee$ ($\forall$ p. $\neg$ (PumpCtrState (p , v1$'$) $\in$ InMess) $\vee$ $\neg$ (PumpCtrState (p , v2$'$) $\in$ InMess)) 16. $\forall$ p.$\exists$ v.PumpState (p , v) $\in$ InMess 17. $\forall$ p.$\exists$ v.PumpCtrState (p , v) $\in$ InMess 18. Level v$''$ $\in$ InMess 19. Steam v$'''$ $\in$ InMess 20. $\forall$ v1. $\neg$ (Level v1 $\in$ InMess) $\vee$ ($\forall$ v2.$\neg$ (Level v2 $\in$ InMess) $\vee$ (v1 = v2))	1. $\neg$ inmess_ok

–Test Frame 1.12:

Stimuli	Response
1. SteamBoilerWaiting $\in$ InMess 2. (select l.Level l $\in$ InMess) $\leqslant$ MaxWater 3. (select l.Steam l $\in$ InMess) $\leqslant$ MaxSteam 4. Ready 5. Qst Repairing 6. Vst Repairing 7. $\neg$ (LevelFlrAck $\in$ InMess) 8. $\neg$ (SteamFlrAck $\in$ InMess) 9. $\forall$ p. $\forall$ v1$'$. $\neg$ (PumpState (p , v1$'$) $\in$ InMess) $\vee$ ($\forall$ v2$'$. $\neg$ (PumpState (p , v2$'$) $\in$ InMess) $\vee$ (v1$'$ = v2$'$)) 10. $\forall$ p$'$.$\neg$ (PumpCtrFlrAck p$'$ $\in$ InMess) $\vee$ Mst p$'$ Broken 11. $\forall$ p$'$.$\neg$ (PumpFlrAck p$'$ $\in$ InMess) $\vee$ Pst p$'$ Broken 12. $\forall$ p$'$.$\neg$ (PumpCtrRep p$'$ $\in$ InMess) $\vee$ Mst p$'$ Repairing 13. $\forall$ p$'$.$\neg$ (PumpRep p$'$ $\in$ InMess) $\vee$ Pst p$'$ Repairing 14. $\forall$ v1. $\forall$ v2. (v1 = v2) $\vee$ $\neg$ (Steam v1 $\in$ InMess) $\vee$ $\neg$ (Steam v2 $\in$ InMess) 15. $\forall$ v1$'$. $\forall$ v2$'$. (v1$'$ = v2$'$) $\vee$ ($\forall$ p. $\neg$ (PumpCtrState (p , v1$'$) $\in$ InMess) $\vee$ $\neg$ (PumpCtrState (p , v2$'$) $\in$ InMess)) 16. $\forall$ p.$\exists$ v.PumpState (p , v) $\in$ InMess 17. $\forall$ p.$\exists$ v.PumpCtrState (p , v) $\in$ InMess 18. Level v$''$ $\in$ InMess 19. Steam v$'''$ $\in$ InMess 20. $\forall$ v1. $\neg$ (Level v1 $\in$ InMess) $\vee$ ($\forall$ v2.$\neg$ (Level v2 $\in$ InMess) $\vee$ (v1 = v2))	1. $\neg$ inmess_ok

–Test Frame 1.13:

Stimuli	Response
1. $\neg$ (v'' = v''') 2. Steam v'' $\in$ InMess 3. Steam v''' $\in$ InMess 4. (select l.Level l $\in$ InMess) $\leqslant$ MaxWater 5. (select l.Steam l $\in$ InMess) $\leqslant$ MaxSteam 6. Waiting 7. $\neg$ (PhysicalUnitsReady $\in$ InMess) 8. Qst Repairing 9. Vst Repairing 10. $\neg$ (LevelFlrAck $\in$ InMess) 11. $\neg$ (SteamFlrAck $\in$ InMess) 12. $\forall$ p. $\forall$ v1'. $\neg$ (PumpState (p , v1') $\in$ InMess) $\vee$ ($\forall$ v2'. $\neg$ (PumpState (p , v2') $\in$ InMess) $\vee$ (v1' = v2')) 13. $\forall$ p'.$\neg$ (PumpCtrFlrAck p' $\in$ InMess) $\vee$ Mst p' Broken 14. $\forall$ p'.$\neg$ (PumpFlrAck p' $\in$ InMess) $\vee$ Pst p' Broken 15. $\forall$ p'.$\neg$ (PumpCtrRep p' $\in$ InMess) $\vee$ Mst p' Repairing 16. $\forall$ p'.$\neg$ (PumpRep p' $\in$ InMess) $\vee$ Pst p' Repairing 17. $\forall$ v1'. $\forall$ v2'. (v1' = v2') $\vee$ ($\forall$ p. $\neg$ (PumpCtrState (p , v1') $\in$ InMess) $\vee$ $\neg$ (PumpCtrState (p , v2') $\in$ InMess)) 18. $\forall$ p.$\exists$ v.PumpState (p , v) $\in$ InMess 19. $\forall$ p.$\exists$ v.PumpCtrState (p , v) $\in$ InMess 20. Level v'''' $\in$ InMess 21. Steam v' $\in$ InMess 22. $\forall$ v1. $\neg$ (Level v1 $\in$ InMess) $\vee$ ($\forall$ v2.$\neg$ (Level v2 $\in$ InMess) $\vee$ (v1 = v2))	1. $\neg$ inmess_ok

–Test Frame 1.14:

Stimuli	Response
1. $\neg$ (v1$''$ = v2$''$) 2. PumpCtrState (p$''$, v1$''$) $\in$ InMess 3. PumpCtrState (p$''$, v2$''$) $\in$ InMess 4. (select l.Level l $\in$ InMess) $\leqslant$ MaxWater 5. (select l.Steam l $\in$ InMess) $\leqslant$ MaxSteam 6. Waiting 7. $\neg$ (PhysicalUnitsReady $\in$ InMess) 8. Qst Repairing 9. Vst Repairing 10. $\neg$ (LevelFlrAck $\in$ InMess) 11. $\neg$ (SteamFlrAck $\in$ InMess) 12. $\forall$ p. $\forall$ v1$'$. $\neg$ (PumpState (p , v1$'$) $\in$ InMess) $\vee$ ($\forall$ v2$'$. $\neg$ (PumpState (p , v2$'$) $\in$ InMess) $\vee$ (v1$'$ = v2$'$)) 13. $\forall$ p$'$.$\neg$ (PumpCtrFlrAck p$'$ $\in$ InMess) $\vee$ Mst p$'$ Broken 14. $\forall$ p$'$.$\neg$ (PumpFlrAck p$'$ $\in$ InMess) $\vee$ Pst p$'$ Broken 15. $\forall$ p$'$.$\neg$ (PumpCtrRep p$'$ $\in$ InMess) $\vee$ Mst p$'$ Repairing 16. $\forall$ p$'$.$\neg$ (PumpRep p$'$ $\in$ InMess) $\vee$ Pst p$'$ Repairing 17. $\forall$ v1. $\forall$ v2. (v1 = v2) $\vee$ $\neg$ (Steam v1 $\in$ InMess) $\vee$ $\neg$ (Steam v2 $\in$ InMess) 18. $\forall$ p.$\exists$ v.PumpState (p , v) $\in$ InMess 19. $\forall$ p.$\exists$ v.PumpCtrState (p , v) $\in$ InMess 20. Level v$''''$ $\in$ InMess 21. Steam v$'$ $\in$ InMess 22. $\forall$ v1. $\neg$ (Level v1 $\in$ InMess) $\vee$ ($\forall$ v2.$\neg$ (Level v2 $\in$ InMess) $\vee$ (v1 = v2))	1. $\neg$ inmess_ok

–Test Frame 1.15:

Stimuli	Response
1. $\forall$ v.$\neg$ (PumpState (p''' , v) $\in$ InMess) 2. (select l.Level l $\in$ InMess) $\leqslant$ MaxWater 3. (select l.Steam l $\in$ InMess) $\leqslant$ MaxSteam 4. Waiting 5. $\neg$ (PhysicalUnitsReady $\in$ InMess) 6. Qst Repairing 7. Vst Repairing 8. $\neg$ (LevelFlrAck $\in$ InMess) 9. $\neg$ (SteamFlrAck $\in$ InMess) 10. $\forall$ p. $\forall$ v1'. $\neg$ (PumpState (p , v1') $\in$ InMess) $\vee$ ($\forall$ v2'. $\neg$ (PumpState (p , v2') $\in$ InMess) $\vee$ (v1' = v2')) 11. $\forall$ p'.$\neg$ (PumpCtrFlrAck p' $\in$ InMess) $\vee$ Mst p' Broken 12. $\forall$ p'.$\neg$ (PumpFlrAck p' $\in$ InMess) $\vee$ Pst p' Broken 13. $\forall$ p'.$\neg$ (PumpCtrRep p' $\in$ InMess) $\vee$ Mst p' Repairing 14. $\forall$ p'.$\neg$ (PumpRep p' $\in$ InMess) $\vee$ Pst p' Repairing 15. $\forall$ v1. $\forall$ v2. (v1 = v2) $\vee$ $\neg$ (Steam v1 $\in$ InMess) $\vee$ $\neg$ (Steam v2 $\in$ InMess) 16. $\forall$ v1'. $\forall$ v2'. (v1' = v2') $\vee$ ($\forall$ p. $\neg$ (PumpCtrState (p , v1') $\in$ InMess) $\vee$ $\neg$ (PumpCtrState (p , v2') $\in$ InMess)) 17. $\forall$ p.$\exists$ v.PumpCtrState (p , v) $\in$ InMess 18. Level v'''' $\in$ InMess 19. Steam v' $\in$ InMess 20. $\forall$ v1. $\neg$ (Level v1 $\in$ InMess) $\vee$ ($\forall$ v2.$\neg$ (Level v2 $\in$ InMess) $\vee$ (v1 = v2))	1. $\neg$ inmess_ok

–Test Frame 1.16:

Stimuli	Response
1. $\forall$ v.$\neg$ (PumpCtrState (p''' , v) $\in$ InMess) 2. (select l.Level l $\in$ InMess) $\leqslant$ MaxWater 3. (select l.Steam l $\in$ InMess) $\leqslant$ MaxSteam 4. Waiting 5. $\neg$ (PhysicalUnitsReady $\in$ InMess) 6. Qst Repairing 7. Vst Repairing 8. $\neg$ (LevelFlrAck $\in$ InMess) 9. $\neg$ (SteamFlrAck $\in$ InMess) 10. $\forall$ p. $\forall$ v1'. $\neg$ (PumpState (p , v1') $\in$ InMess) $\vee$ ($\forall$ v2'. $\neg$ (PumpState (p , v2') $\in$ InMess) $\vee$ (v1' = v2')) 11. $\forall$ p'.$\neg$ (PumpCtrFlrAck p' $\in$ InMess) $\vee$ Mst p' Broken 12. $\forall$ p'.$\neg$ (PumpFlrAck p' $\in$ InMess) $\vee$ Pst p' Broken 13. $\forall$ p'.$\neg$ (PumpCtrRep p' $\in$ InMess) $\vee$ Mst p' Repairing 14. $\forall$ p'.$\neg$ (PumpRep p' $\in$ InMess) $\vee$ Pst p' Repairing 15. $\forall$ v1. $\forall$ v2. (v1 = v2) $\vee$ $\neg$ (Steam v1 $\in$ InMess) $\vee$ $\neg$ (Steam v2 $\in$ InMess) 16. $\forall$ v1'. $\forall$ v2'. (v1' = v2') $\vee$ ($\forall$ p. $\neg$ (PumpCtrState (p , v1') $\in$ InMess) $\vee$ $\neg$ (PumpCtrState (p , v2') $\in$ InMess)) 17. $\forall$ p.$\exists$ v.PumpState (p , v) $\in$ InMess 18. Level v'''' $\in$ InMess 19. Steam v' $\in$ InMess 20. $\forall$ v1. $\neg$ (Level v1 $\in$ InMess) $\vee$ ($\forall$ v2.$\neg$ (Level v2 $\in$ InMess) $\vee$ (v1 = v2))	1. $\neg$ inmess_ok

–Test Frame 1.17:

Stimuli	Response
1. $\forall$ v.$\neg$ (Level v $\in$ InMess) 2. (select l.Level l $\in$ InMess) $\leqslant$ MaxWater 3. (select l.Steam l $\in$ InMess) $\leqslant$ MaxSteam 4. Waiting 5. $\neg$ (PhysicalUnitsReady $\in$ InMess) 6. Qst Repairing 7. Vst Repairing 8. $\neg$ (LevelFlrAck $\in$ InMess) 9. $\neg$ (SteamFlrAck $\in$ InMess) 10. $\forall$ p. $\forall$ v1$'$. $\neg$ (PumpState (p , v1$'$) $\in$ InMess) $\vee$ ($\forall$ v2$'$. $\neg$ (PumpState (p , v2$'$) $\in$ InMess) $\vee$ (v1$'$ = v2$'$)) 11. $\forall$ p$'$.$\neg$ (PumpCtrFlrAck p$'$ $\in$ InMess) $\vee$ Mst p$'$ Broken 12. $\forall$ p$'$.$\neg$ (PumpFlrAck p$'$ $\in$ InMess) $\vee$ Pst p$'$ Broken 13. $\forall$ p$'$.$\neg$ (PumpCtrRep p$'$ $\in$ InMess) $\vee$ Mst p$'$ Repairing 14. $\forall$ p$'$.$\neg$ (PumpRep p$'$ $\in$ InMess) $\vee$ Pst p$'$ Repairing 15. $\forall$ v1. $\forall$ v2. (v1 = v2) $\vee$ $\neg$ (Steam v1 $\in$ InMess) $\vee$ $\neg$ (Steam v2 $\in$ InMess) 16. $\forall$ v1$'$. $\forall$ v2$'$. (v1$'$ = v2$'$) $\vee$ ($\forall$ p. $\neg$ (PumpCtrState (p , v1$'$) $\in$ InMess) $\vee$ $\neg$ (PumpCtrState (p , v2$'$) $\in$ InMess)) 17. $\forall$ p.$\exists$ v.PumpState (p , v) $\in$ InMess 18. $\forall$ p.$\exists$ v.PumpCtrState (p , v) $\in$ InMess 19. $\forall$ v1. $\neg$ (Level v1 $\in$ InMess) $\vee$ ($\forall$ v2.$\neg$ (Level v2 $\in$ InMess) $\vee$ (v1 = v2)) 20. Steam v$'$ $\in$ InMess	1. $\neg$ inmess_ok

–Test Frame 1.18:

Stimuli	Response
1. $\forall$ v.$\neg$ (Steam v $\in$ InMess) 2. (select l.Level l $\in$ InMess) $\leqslant$ MaxWater 3. (select l.Steam l $\in$ InMess) $\leqslant$ MaxSteam 4. Waiting 5. $\neg$ (PhysicalUnitsReady $\in$ InMess) 6. Qst Repairing 7. Vst Repairing 8. $\neg$ (LevelFlrAck $\in$ InMess) 9. $\neg$ (SteamFlrAck $\in$ InMess) 10. $\forall$ p. $\forall$ v1$'$. $\neg$ (PumpState (p , v1$'$) $\in$ InMess) $\vee$ ($\forall$ v2$'$. $\neg$ (PumpState (p , v2$'$) $\in$ InMess) $\vee$ (v1$'$ = v2$'$)) 11. $\forall$ p$'$.$\neg$ (PumpCtrFlrAck p$'$ $\in$ InMess) $\vee$ Mst p$'$ Broken 12. $\forall$ p$'$.$\neg$ (PumpFlrAck p$'$ $\in$ InMess) $\vee$ Pst p$'$ Broken 13. $\forall$ p$'$.$\neg$ (PumpCtrRep p$'$ $\in$ InMess) $\vee$ Mst p$'$ Repairing 14. $\forall$ p$'$.$\neg$ (PumpRep p$'$ $\in$ InMess) $\vee$ Pst p$'$ Repairing 15. $\forall$ v1. $\forall$ v2. (v1 = v2) $\vee$ $\neg$ (Steam v1 $\in$ InMess) $\vee$ $\neg$ (Steam v2 $\in$ InMess) 16. $\forall$ v1$'$. $\forall$ v2$'$. (v1$'$ = v2$'$) $\vee$ ($\forall$ p. $\neg$ (PumpCtrState (p , v1$'$) $\in$ InMess) $\vee$ $\neg$ (PumpCtrState (p , v2$'$) $\in$ InMess)) 17. $\forall$ p.$\exists$ v.PumpState (p , v) $\in$ InMess 18. $\forall$ p.$\exists$ v.PumpCtrState (p , v) $\in$ InMess 19. Level v$'$ $\in$ InMess 20. $\forall$ v1. $\neg$ (Level v1 $\in$ InMess) $\vee$ ($\forall$ v2.$\neg$ (Level v2 $\in$ InMess) $\vee$ (v1 = v2))	1. $\neg$ inmess_ok

–Test Frame 1.19:

Stimuli	Response
1. $\neg$ ((select l.Steam l $\in$ InMess) $\leqslant$ MaxSteam) 2. (select l.Level l $\in$ InMess) $\leqslant$ MaxWater 3. Waiting 4. $\neg$ (PhysicalUnitsReady $\in$ InMess) 5. Qst Repairing 6. Vst Repairing 7. $\neg$ (LevelFlrAck $\in$ InMess) 8. $\neg$ (SteamFlrAck $\in$ InMess) 9. $\forall$ p. $\forall$ v1$'$. $\neg$ (PumpState (p , v1$'$) $\in$ InMess) $\vee$ ($\forall$ v2$'$. $\neg$ (PumpState (p , v2$'$) $\in$ InMess) $\vee$ (v1$'$ = v2$'$)) 10. $\forall$ p$'$.$\neg$ (PumpCtrFlrAck p$'$ $\in$ InMess) $\vee$ Mst p$'$ Broken 11. $\forall$ p$'$.$\neg$ (PumpFlrAck p$'$ $\in$ InMess) $\vee$ Pst p$'$ Broken 12. $\forall$ p$'$.$\neg$ (PumpCtrRep p$'$ $\in$ InMess) $\vee$ Mst p$'$ Repairing 13. $\forall$ p$'$.$\neg$ (PumpRep p$'$ $\in$ InMess) $\vee$ Pst p$'$ Repairing 14. $\forall$ v1. $\forall$ v2. (v1 = v2) $\vee$ $\neg$ (Steam v1 $\in$ InMess) $\vee$ $\neg$ (Steam v2 $\in$ InMess) 15. $\forall$ v1$'$. $\forall$ v2$'$. (v1$'$ = v2$'$) $\vee$ ($\forall$ p. $\neg$ (PumpCtrState (p , v1$'$) $\in$ InMess) $\vee$ $\neg$ (PumpCtrState (p , v2$'$) $\in$ InMess)) 16. $\forall$ p.$\exists$ v.PumpState (p , v) $\in$ InMess 17. $\forall$ p.$\exists$ v.PumpCtrState (p , v) $\in$ InMess 18. Level v$''$ $\in$ InMess 19. Steam v$'''$ $\in$ InMess 20. $\forall$ v1. $\neg$ (Level v1 $\in$ InMess) $\vee$ ($\forall$ v2.$\neg$ (Level v2 $\in$ InMess) $\vee$ (v1 = v2))	1. $\neg$ inmess_ok

–Test Frame 1.20:

Stimuli	Response
1. $\neg$ ((select l.Level l $\in$ InMess) $\leqslant$ MaxWater) 2. (select l.Steam l $\in$ InMess) $\leqslant$ MaxSteam 3. Waiting 4. $\neg$ (PhysicalUnitsReady $\in$ InMess) 5. Qst Repairing 6. Vst Repairing 7. $\neg$ (LevelFlrAck $\in$ InMess) 8. $\neg$ (SteamFlrAck $\in$ InMess) 9. $\forall$ p. $\forall$ v1$'$. $\neg$ (PumpState (p , v1$'$) $\in$ InMess) $\vee$ ($\forall$ v2$'$. $\neg$ (PumpState (p , v2$'$) $\in$ InMess) $\vee$ (v1$'$ = v2$'$)) 10. $\forall$ p$'$.$\neg$ (PumpCtrFlrAck p$'$ $\in$ InMess) $\vee$ Mst p$'$ Broken 11. $\forall$ p$'$.$\neg$ (PumpFlrAck p$'$ $\in$ InMess) $\vee$ Pst p$'$ Broken 12. $\forall$ p$'$.$\neg$ (PumpCtrRep p$'$ $\in$ InMess) $\vee$ Mst p$'$ Repairing 13. $\forall$ p$'$.$\neg$ (PumpRep p$'$ $\in$ InMess) $\vee$ Pst p$'$ Repairing 14. $\forall$ v1. $\forall$ v2. (v1 = v2) $\vee$ $\neg$ (Steam v1 $\in$ InMess) $\vee$ $\neg$ (Steam v2 $\in$ InMess) 15. $\forall$ v1$'$. $\forall$ v2$'$. (v1$'$ = v2$'$) $\vee$ ($\forall$ p. $\neg$ (PumpCtrState (p , v1$'$) $\in$ InMess) $\vee$ $\neg$ (PumpCtrState (p , v2$'$) $\in$ InMess)) 16. $\forall$ p.$\exists$ v.PumpState (p , v) $\in$ InMess 17. $\forall$ p.$\exists$ v.PumpCtrState (p , v) $\in$ InMess 18. Level v$''$ $\in$ InMess 19. Steam v$'''$ $\in$ InMess 20. $\forall$ v1. $\neg$ (Level v1 $\in$ InMess) $\vee$ ($\forall$ v2.$\neg$ (Level v2 $\in$ InMess) $\vee$ (v1 = v2))	1. $\neg$ inmess_ok

–Test Frame 2.1:

Stimuli	Response
1. $\forall$ p.$\exists$ v.PumpState (p , v) $\in$ InMess 2. $\forall$ v1. $\forall$ v2. ($\forall$ p. $\neg$ (PumpState (p , v1) $\in$ InMess) $\vee$ $\neg$ (PumpState (p , v2) $\in$ InMess)) $\vee$ (v1 = v2) 3. $\forall$ p.$\exists$ v.PumpCtrState (p , v) $\in$ InMess 4. $\forall$ v1. $\forall$ v2. ($\forall$ p. $\neg$ (PumpCtrState (p , v1) $\in$ InMess) $\vee$ $\neg$ (PumpCtrState (p , v2) $\in$ InMess)) $\vee$ (v1 = v2) 5. Level v $\in$ InMess 6. $\forall$ v1. $\forall$ v2. $\neg$ (Level v1 $\in$ InMess) $\vee$ $\neg$ (Level v2 $\in$ InMess) $\vee$ (v1 = v2) 7. (select l.Level l $\in$ InMess) $\leqslant$ MaxWater 8. Steam v$'$ $\in$ InMess 9. $\forall$ v1. $\forall$ v2. $\neg$ (Steam v1 $\in$ InMess) $\vee$ $\neg$ (Steam v2 $\in$ InMess) $\vee$ (v1 = v2) 10. (select l.Steam l $\in$ InMess) $\leqslant$ MaxSteam 11. Waiting 12. $\neg$ (PhysicalUnitsReady $\in$ InMess) 13. $\forall$ p.$\neg$ (PumpRep p $\in$ InMess) $\vee$ Pst p Repairing 14. $\forall$ p.$\neg$ (PumpCtrRep p $\in$ InMess) $\vee$ Mst p Repairing 15. $\forall$ p.$\neg$ (PumpFlrAck p $\in$ InMess) $\vee$ Pst p Broken 16. $\forall$ p.$\neg$ (PumpCtrFlrAck p $\in$ InMess) $\vee$ Mst p Broken 17. Qst Repairing 18. Vst Repairing 19. $\neg$ (LevelFlrAck $\in$ InMess) 20. $\neg$ (SteamFlrAck $\in$ InMess) 21. SteamBoilerWaiting $\in$ InMess	1. inmess_ok

–Test Frame 2.2:

Stimuli	Response
1. $\forall$ p.$\exists$ v.PumpState (p , v) $\in$ InMess 2. $\forall$ v1. $\forall$ v2. ($\forall$ p. $\neg$ (PumpState (p , v1) $\in$ InMess) $\vee$ $\neg$ (PumpState (p , v2) $\in$ InMess)) $\vee$ (v1 = v2) 3. $\forall$ p.$\exists$ v.PumpCtrState (p , v) $\in$ InMess 4. $\forall$ v1. $\forall$ v2. ($\forall$ p. $\neg$ (PumpCtrState (p , v1) $\in$ InMess) $\vee$ $\neg$ (PumpCtrState (p , v2) $\in$ InMess)) $\vee$ (v1 = v2) 5. Level v $\in$ InMess 6. $\forall$ v1. $\forall$ v2. $\neg$ (Level v1 $\in$ InMess) $\vee$ $\neg$ (Level v2 $\in$ InMess) $\vee$ (v1 = v2) 7. (select l.Level l $\in$ InMess) $\leqslant$ MaxWater 8. Steam v$'$ $\in$ InMess 9. $\forall$ v1. $\forall$ v2. $\neg$ (Steam v1 $\in$ InMess) $\vee$ $\neg$ (Steam v2 $\in$ InMess) $\vee$ (v1 = v2) 10. (select l.Steam l $\in$ InMess) $\leqslant$ MaxSteam 11. $\neg$ (SteamBoilerWaiting $\in$ InMess) 12. Ready 13. $\forall$ p.$\neg$ (PumpRep p $\in$ InMess) $\vee$ Pst p Repairing 14. $\forall$ p.$\neg$ (PumpCtrRep p $\in$ InMess) $\vee$ Mst p Repairing 15. $\forall$ p.$\neg$ (PumpFlrAck p $\in$ InMess) $\vee$ Pst p Broken 16. $\forall$ p.$\neg$ (PumpCtrFlrAck p $\in$ InMess) $\vee$ Mst p Broken 17. $\neg$ (LevelRep $\in$ InMess) 18. $\neg$ (SteamRep $\in$ InMess) 19. Qst Broken 20. Vst Broken 21. PhysicalUnitsReady $\in$ InMess	1. inmess_ok

Appendix C

A Heuristic for the Delta Problem

This appendix presents a mathematical definition of the Delta Problem from Section 4.6 and outlines a proposed heuristic test frame delta algorithm. The proposed partial solution to the Delta Problem also allows the test frame generation process to accept user mandated tests, thereby providing further control by test engineers. An additional capability of this algorithm is the identification of some test frames which span multiple test classes while allowing for appropriate coverage.

As described in Section 4.6, the Delta Problem is to integrate test frames previously generated from a specification with new test frames generated from a changed version of the same specification while satisfying the specified coverage criterion. The original motivation for this problem is the reuse of existing test frames after specification changes have occurred. However, if the structural difference between the two versions of the specification is ignored, then whether the existing test frames were generated automatically or specified manually is of little consequence.

Thus, the integration of user mandated test frames with those test frames produced from a specification is an instance of the Delta Problem.

In the context of test frame generation, the Delta Problem is defined as follows. A prime ($'$) is used to distinguish new literals resulting from a requirements change from literals corresponding to the previous version of the requirements.

> Let $Q.f \Rightarrow R$ be an existing test frame derived from requirements specification A, and let $P'.S' \Rightarrow R'$ be a test class of A', where Q and P' represent the outer quantifiers of the test frame and test class, respectively. The antecedent of the existing test frame is represented by f. R represents the consequent of the existing test frame. S' represents the antecedent of the test class to which the existing test frame might belong. R' represents the consequent of this test class. The Delta Problem can be expressed as the following two questions:
>
> 1. Is $Q.f \Rightarrow R$ still a valid test? i.e., Does A' imply $Q.f \Rightarrow R$?
> 2. Can $Q.f \Rightarrow R$ be incorporated into a new set of test frames? More precisely, if $Q.f \Rightarrow R$ is implied by the test class $P'.S' \Rightarrow R'$, which of the prime implicants of S' is represented by f?

Since test class normal form is not canonical, it is possible that the existing test frame is valid, but is not implied by any one test class. In this situation, Question 2 above is irrelevant and the existing test frame cannot be incorporated into the new set of test frames. Thus, although confirmation of Question 1 is valuable, it does not assist in integrating an existing test frame with a new test frame set.

The first part of Question 2 is represented formally as the conjecture:

$$(P'.S' \Rightarrow R') \Rightarrow (Q.f \Rightarrow R) \tag{C.1}$$

If Conjecture (C.1) is a theorem, then the test frame is still valid and should be added to the initial set used by the coverage scheme. This conjecture is undecidable, in general. However, if a reasonable proportion of the instances of this conjecture that are true could be proven automatically, then this would provide a partial solution to the Delta Problem.

The following theorem hints at a partial solution.

$$\vdash (Q.\hat{P}'.(f \Rightarrow S') \wedge (R' \Rightarrow R)) \Rightarrow ((P'.S' \Rightarrow R') \Rightarrow (Q.f \Rightarrow R)) \tag{C.2}$$

where P and Q are sequences of quantifiers and $\hat{P}'$ is the logical dual of quantification P'. i.e., $\hat{P}'.X = \neg(P'.\neg X)$.

Thus, a proof of Conjecture (C.1) can be achieved by proving the following conjecture:

$$Q.\hat{P}'.(f \Rightarrow S') \wedge (R' \Rightarrow R) \tag{C.3}$$

A heuristic algorithm for attempting a proof of (C.3) is based on the assumption that most changes are small and that variables quantified by Q are similarly quantified by P'. The heuristic is suggested by examining a particular proof by contradiction of the trivial theorem $(\forall\, x. \exists\, y. E(x, y)) \Rightarrow (\forall\, x. \exists\, y. E(x, y))$.

$$\begin{aligned}
& \neg(\forall\, x. \exists\, y. E(x, y)) \Rightarrow (\forall\, x. \exists\, y. E(x, y)) \\
= \quad & (\forall\, x. \exists\, y. E(x, y)) \wedge \neg(\forall\, x. \exists\, y. E(x, y)) \\
= \quad & (\forall\, x. \exists\, y. E(x, y)) \wedge (\underline{\exists\, x.}\, \forall\, y. \neg E(x, y)) \\
= \quad & (\underline{\forall\, x.}\, \exists\, y. E(x, y)) \wedge (\forall\, y. \neg E(x, y)) \\
\Rightarrow \quad & (\underline{\exists\, y.}\, E(x, y)) \wedge (\forall\, y. \neg E(x, y)) \\
= \quad & E(x, y) \wedge (\underline{\forall\, y.}\, \neg E(x, y)) \\
\Rightarrow \quad & E(x, y) \wedge \neg E(x, y)
\end{aligned}$$

$$= \perp$$

This illustrates that when the expressions within the quantifiers, represented by E above, are similar, an appropriate matching of existentially quantified variables against universally quantified variables can result in a proof. This matching is referred to below as an appropriate set of bindings. Assuming the frame stimuli of S' and f are replaced by variables, this heuristic transforms a conjecture involving quantifiers into a conjecture in predicate calculus, which is decidable.

This approach is guaranteed to find a proof if the antecedent is a test class and the consequent is a test frame derived from that test class. There are also situations where small differences between the antecedent and consequent which do not affect the validity of the conjecture will still result in the heuristic being successful. Therefore, this heuristic will provide a partial solution to the Delta Problem.

The above analysis leads to the following test frame delta algorithm:

1. Find the set of bindings suggested by $Q.\hat{P}'.r' \Rightarrow R$.
2. Fail if no viable binding results in a proof.
3. Further constrain the set of bindings by comparing the frame stimuli of f to the frame stimuli within S'.
4. Fail if no viable bindings remain.
5. Scan the set of prime implicants of S' for those which match f.

This heuristic algorithm has the useful property that it will be able to prove conjectures such as

$$\vdash (\forall\, x.(A\; x \vee B\; x) \Rightarrow R) \Rightarrow (A\, 1 \Rightarrow R) \tag{C.4}$$

This means that mandated test frames do not need to be specified in their most general form for them to be matched to the corresponding test class.

So what does it mean to match f and a prime implicant, s', of S'? As an example, let $s' = A \wedge B$. There are three cases:

1. $f = s'$: The test frame matches perfectly.
2. $f \Rightarrow s'$: f is more specific, e.g. $f = A \wedge B \wedge C$.
3. $s' \Rightarrow f$: f is too vague, e.g. $f = A$, and the corresponding existing test frame is no longer valid.

Case 1) is the normal case. The change in specification A' has not affected $Q.f \Rightarrow R$. Case 2): if f is more specific than required, the existing test frame is still valid, but may be too restrictive. This should be reported so test engineers can adjust the existing test frame as desired. Case 3): if this is the most specific match for f (i.e., for all prime implicants, p', from all test classes which have a matching response, $(p' \Rightarrow f) \Rightarrow (s' \Rightarrow p')$), then s' is included in the new test set. This allows a test engineer to mandate a vague test frame and have the test frame generator determine the most general test class and "flesh out" the test frame. In all cases, $Q.f \Rightarrow R$ is tagged as being matched to this test class. Any existing test frames which are not matched should be reported.

These new test frame sets form the initial selections from the test class. Once the existing frames have been processed, the selected coverage scheme augments these sets as necessary, marking any redundant test frames as described in Section 5.6.2. This allows user mandated tests and existing tests to guide test frame selection. This approach also allows user mandated and existing test frame sets to be evaluated according to a selected coverage scheme.

Regarding case 2) above, it may be the case that f implies several different implicants from one or more test classes. In this case, the corresponding test classes should be used in combination to ensure that the given frame follows logically from the specification. For example, assuming $f \Rightarrow s'_{11} \wedge s'_{12} \wedge s'_2$, the theorem $\vdash A' \Rightarrow (Q.f \Rightarrow R)$ follows from the theorems:

$$\vdash A' \Rightarrow (P'_1.S'_1 \Rightarrow R'_1) \wedge (P'_2.S'_2 \Rightarrow R'_2) \tag{C.5}$$

and

$$\vdash Q.\hat{P}'_1.\hat{P}'_2.(f \Rightarrow s'_{11}) \wedge (f \Rightarrow s'_{12}) \wedge (f \Rightarrow s'_2) \wedge (R'_1 \wedge R'_2 \Rightarrow R) \tag{C.6}$$

where $s_{11} \Rightarrow S'_1$, $s_{12} \Rightarrow S'_1$, $s'_2 \Rightarrow S'_2$, Theorem (C.5) is produced by the test class algorithm, and Theorem (C.6) is produced by the test frame delta algorithm. It also follows that the use of prime implicants s'_{11} and s'_{12} from S'_1 and implicant s'_2 from S'_2 are subsumed by using f. This allows one user mandated test frame to account for partial coverage of more than one test class.

Although the Delta Problem is undecidable, in the worst case, it may be possible to solve this problem for many of the small changes that are made to a requirements specification during the course of system development. By treating these case automatically when it is possible to do so would result in reducing the amount of involvement required by test engineers to make the necessary adjustments to existing test infrastructure.