Applying formal methods to software testing

Philip Alan Stocks

BSc (Honours)

A thesis submitted to

THE DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF QUEENSLAND

for the degree of

DOCTOR OF PHILOSOPHY

December 1993

I declare that the work presented in this thesis is, to the best of my knowledge, original and my own work, except as acknowledged in the text, and that it has not been submitted, either in whole or in part, for a degree at this, or any other, university.

P. A. Stocks

December 1993

Acknowledgements

I cannot adequately express my gratitude to my supervisor, Dr David Carrington, whose enthusiasm, encouragement, and wisdom have been invaluable to me. Moreover, I wish to thank him for letting me find my own path, being there when I needed help, and for always having the right words. Finally, I must thank him for his tireless and thorough efforts in proof-reading drafts of this thesis and other works. I also wish to thank Professor Andrew Lister for his efforts in assisting the funding of this project, and for his interim supervision while Dr Carrington was on sabbatical, where he always asked the difficult questions.

In 1992 I spent a valuable six months as a visiting scholar at the University of Massachusetts at Amherst. I wish to thank Professor Lori Clarke for making this visit possible, and for her careful supervision during this time. I also wish to thank Professor Debra Richardson of the University of California at Irvine for her comments and for arranging a visit to her department in mid-1993.

At the University of Queensland I owe thanks to everyone, drawing special attention to the comments, advice, and influences of Dr Paul Bailes, Dr Colin Fidge, Dr Iain Fogg, Dr Ian Hayes, Tim Mansfield, Professor Gordon Rose, Dr Paul Strooper, and Nigel Ward. Also, I wish to thank the members of CEDIS, especially Dr Alison Payne and Dr Kerry Raymond. Lastly, I'd like to thank my office-mates for their camaraderie and friendship.

This project received funding from an Australian Postgraduate Research Award and from Telecom Research Laboratories, Australia, under the CEDIS banner. For all this, I am extremely grateful.

With great pleasure, I acknowledge the vital role my family has played. For their continual love, encouragement, and support I am both extremely lucky and thankful. My parents, Alan and Miriam, provided me with much-needed determination and means to carry out this study. My brother, Graham, and his wife, Cindy, have helped more than they can know, and more than I can thank them for here. Finally, I wish to extend my deepest love to my marvelous wife, Karen, and my thanks for knowing me so well.

For my family

Abstract

This thesis examines applying formal methods to software testing. Software testing is a critical phase of the software life-cycle which can be very effective if performed rigorously. Formal specifications offer the bases for rigorous testing practices. Not surprisingly, the most immediate use of formal specifications in software testing is as sources of black-box test suites. However, formal specifications have more uses in software testing than merely being sources for test data. We examine these uses, and show how to get more assistance and benefit from formal methods in software testing.

At the core of this work is a flexible framework in which to conduct specification-based testing. The framework is founded on formal definitions of tests and test suites, which directly addresses important issues in managing software testing. This provides a uniform platform for other applications of formal methods to testing such as analysis and reification of tests, and also for applications beyond testing such as maintenance and specification validation.

The framework has to be flexible so that any testing strategies can be used. We examine the need to adapt certain strategies to work with the framework and formal specification. Our experiments showed some deficiencies that arise when using derivation strategies on abstract specifications. These deficiencies led us to develop two new specification-based testing strategies based on extensions to existing strategies.

We demonstrate the framework, strategies, and other applications of formal methods to software testing using three case studies. In each of these, the framework was easy to use. It provided an elegant and powerful means for defining and structuring tests, and a suitable staging ground for other applications of formal methods to software testing. This thesis demonstrates how formal specification techniques can systematise the application of testing strategies, and also how the concepts of software testing can be combined with formal specifications to extend the role of the formal specification in software development.

Contents

Chapter 1

Introduction

1.1 Overview and Motivation

The software engineering community is well-aware of the vital role that testing plays in software development. Testing is a practical means of detecting program errors, which can be highly effective if performed rigorously. Despite the major limitation of testing that it can only show the presence of errors and never their absence[1], it will always be a necessary verification technique. Lucid arguments to this effect can be found in [Tan76].

The community is also aware of the usefulness of formal methods for specifying and designing software. The accepted role of formal specifications in program verification is as the basis for proofs of correctness and rigorous transformation methodologies. However, formal specifications can play an important role in software testing. Of course, it is not surprising that specifications are important to software testing; it is impossible to test software without specifications of some kind. As Goodenough and Gerhart note, testing based only on program implementation is fundamentally flawed [GG75]. Despite this, only a small portion of the testing literature deals with specification-based testing issues. The only explanation we can offer is that informal specifications have limited usefulness (but are still required) in testing, and that the real benefits are to be gained from formal specifications, which are now reaching a

[1] Dijkstra, of course, paraphrased from [DDH72].

level of maturity and stability.

This thesis examines applications of formal methods to software testing, which offers many advantages for testing. The formal specification of a software product can be used as a guide for designing functional tests for the product. The specification precisely defines fundamental aspects of the software, while more detailed and structural information is omitted. Thus, the tester has the important information about the product's functionality without having to extract it from unnecessary detail. Testing from formal specifications offers a simpler, structured, and more rigorous approach to the development of functional tests than standard testing techniques. The strong relationship between specification and tests facilitates error pin-pointing and can simplify regression testing. An important application of specifications in testing is providing test oracles. The specification is an authoritative description of system behaviour and can be used to derive expected results for test data. Other benefits of specification-based testing include using the derived tests to validate the original specification, simplified auditing of the testing process, and developing tests concurrently with design and implementation. This latter is also useful for breaking 'code now/test later' practices in software engineering, and helping develop a parallel testing activity for all software life-cycle phases as advocated in [Het88].

Our rather general interest in using formal methods to assist software testing leads us towards developing a framework in which to conduct specification-based testing, which includes a formal model of test suites. The framework directly addresses some particular aspects of specification-based testing, but also has application to many other aspects.

1.2 Thesis outline

The structure of the thesis is as follows.

Chapter 2 reviews specification-based testing research. We offer a classification of this work, identify gaps, and explain the context and aims of this thesis.

Chapter 3 begins the discussion of our framework, examining input and output spaces and the sources of tests. It shows how to calculate these spaces and presents the formalisms which form the basis of the framework. Chapter 4 presents the formal mechanics of our defining framework for specification-based testing. Chapter 5 discusses specification-based testing strategies. It discusses how to adapt existing strategies to the framework and introduces two new strategies we developed.

Chapter 6 demonstrates the use of the framework and testing strategies on three case studies. Chapter 7 examines application of the framework in areas beyond deriving and defining tests, such as test plans, reification, analysis, and maintenance. The examples from chapter 6 are used to illustrate these ideas.

Chapter 8 discusses other possible applications of the framework which we have not explored in any depth, areas for future work, and discusses the contributions of this thesis.

The rest of this chapter is devoted to a brief overview of some background material in software testing and formal methods.

1.3 Background knowledge and terminology

This thesis draws together two well-studied areas: software testing and formal methods. This section introduces some background knowledge and the terminology used in this thesis.

1.3.1 Specification-based testing

The term specification-based testing, as used in the literature, usually refers to testing based solely on the specification, i.e., testing not using any information from the implementation. Our wider interest in maximising use of formal specifications in software testing leads us to different usage of the term specification-based testing; our usage would, perhaps, be better termed *specification-sourced testing*.

We are still conducting testing based on the specification, but we are also using external information and processes to maximise the benefit of this basis. The simple question of what to do with tests after they are derived from the specification shows the limits of not using information external to the specification. It is unlikely that the abstract tests so derived will be in a suitable form for actual testing, so information outside the specification must be used to transform the tests into a form suitable for execution. Specification-based testing strategies subtly depend on implementations. It does not make sense to think that the purpose of such strategies is not related to detecting errors, yet such errors are errors in the implementation. Therefore, some heuristic knowledge of implementation errors and specification elements is being used.

Our aim is to provide a general framework and so we do not discount external sources of information. Nevertheless, our main usage of the term corresponds to the standard usage, and it would be unnecessarily confusing to try to introduce new terminology.

1.3.2 Software testing

We assume understanding of the general principles of software testing (e.g., [Bei90, Mye79]). For the sake of clarity, the following explains how we will use some of the less strict terminology throughout this thesis.

Functional unit

Functional units are the elements of specification or code defining distinct pieces of system functionality.

Unit testing

Unit testing is the process of testing individual functional units of a system in isolation.

Success/failure

There are two contrasting views of a successful test. One view, the one which we adopt in this thesis, is that a test is successful if the system shows correct or acceptable behaviour for that input. The other view is that a test that finds no bugs is useless since it doesn't reveal anything about the system. Hence, a successful test is one which finds errors. We do not use 'successful' in this context.

Oracle

An oracle is a means to judge the success or failure of a test, that is, to judge the correctness of the system for some test. The simplest oracle is comparing actual results with expected results by hand. This can be very time consuming, so automated oracles are sought.

Test case

A test is useless if no expectations of behaviour are held. Hence, a test case must contain both test data and a test oracle for the data.

Input domain

The term input domain is applied rather loosely to describe the possible inputs to a program. Test data are drawn from a program's input domain. We often talk about dividing the input domain into smaller sub-domains.

Test selection criteria

Tests are chosen according to certain guidelines or properties they must satisfy, which are collectively called test selection criteria. Broadly, criteria can state how to choose tests, aspects which chosen tests must exercise, or properties of a system that passes the tests.

We also explain briefly some popular testing strategies from the literature. We offer some references to the literature for guidance, but they are not intended to be complete.

Random testing

Tests are chosen randomly from the space of possible inputs. No sub-division of input domains is undertaken. Random testing is a useful benchmark when considering other testing strategies (e.g., [DN84, HT88]). Random tests can be generated from a specification of the structure of the input.

Partitioning

There are many input partitioning approaches to software testing (e.g., [Bei90, OB88, RC85, WO80]). The common approach of each is to divide the input domain into sub-domains for which each element has the same error-detecting ability as any other. Guidelines for choosing these sub-domains are varied, and in general, one cannot guarantee that every element of a sub-domain will be as good as any other. One popular partitioning approach is cause-effect testing, where the various output classes are determined (the effects), and then the corresponding inputs required to generate the outputs (the causes) form the basis of the input partition.

Domain testing

Domain testing is a very successful technique for testing systems with linear input spaces [WC80, CHR82]. A program's control flow is analysed to partition the input into subdomains with linear domain boundaries. Errors in the control flow of the program will cause these boundaries to shift from their correct positions. With domain testing, the minimum number of points is selected to ensure detection of boundary shifts.

Fault-based testing

Fault-based testing (e.g., [How86b, Mor90a]) is not a single strategy but a broad categorisation of strategies. The common theme of fault-based strategies is demonstrating that known faults do not exist in programs. Known faults range from using relational operators incorrectly (like using $\leq$ where $<$ was intended), to using variables before they are declared. The appeal of fault-based strategies over other more intuitive testing strategies is that formal statements about the program's correctness can be made from the results of testing. Fault-based strategies have been shown to be at least as powerful as code coverage strategies such as executing all branches in a program [Gou83, How86b]. Mutation testing, which is described below, is an example of a fault-based testing strategy.

Mutation testing

Mutation testing and analysis (e.g., [Bud81, DLS78, Ham77]) is based on the 'competent programmer hypothesis' [DLS78], which states that a competent programmer produces programs which are 'nearly' correct. By corollary, the proposal of mutation techniques is that actual programs differ from correct programs by small, and perhaps syntactic, amounts. A program mutant is a program generated from the original by making a small change to part of the code, for example, changing relational operators or variable names. Mutants can be generated automatically. Test suites are analysed by determining how many of the mutant programs they can distinguish from the original. Any mutants not distinguished are potentially correct programs.

1.3.3 Formal methods

The concepts of formal methods and specification may not be as familiar as software testing. The goal of a formal specification is to define the behaviour of a system precisely, concisely, and unambiguously. Formal, mathematical, notations assist these goals. The distinguishing factor between specifications and implementation is abstractness. With specification, we restrict ourselves to defining what the system does, without going into the detail of how it is done. Specifications do not have to be executable, and making them executable often restricts our ability to write concise, abstract specifications.

Some general specification-related terms we use in this thesis are

Specification level

The term specification level refers to concepts at the level of an abstract specification rather than at the level of a concrete implementation.

Operation

An operation is a defined function of the system. At the implementation level, we can replace operation by a term such as 'procedure'.

Pre-condition

The pre-condition of an operation is a predicate over the operation's input domain describing input for which the operation is defined, or constraining the possible input to the operation.

Disjunctive normal form (DNF)

The disjunctive normal form of a statement in predicate calculus is obtained by reducing the statement to a combination of disjuncts. For example, in simple propositional calculus, the statement

$$(p \lor q) \land r$$

is made up of two conjuncts. The disjunctive normal form of this statement is

$$p \land r \lor q \land r$$

DNF is a useful concept for sub-dividing input domains.

Partition

A mathematical partition of some set, S, is a set of subsets of S, say $SS_1 \mathinner{\ldotp\ldotp} SS_n$, such that each SS_i and SS_j are disjoint, and the union of all SS_i's equals S.

There are three general styles of specification:

Process algebras

Process algebras describe systems in terms of behaviour and interaction of active agents. Examples of process algebra formal methods are CSP [Hoa85] and LOTOS [BB89].

Algebraic specifications

Algebraic specifications describe systems constructively in terms of applications of system operations. States are defined by construction from some defined basic structure, and passed as parameters. Examples of algebraic specification notations are Larch [GHW85], OBJ [Gog84], and RSL [RAI92].

Model-based specifications

> Model-based specifications construct explicit models of the system state and show how the state can be changed by various operations. Examples of model-based specification notations are RSL [RAI92], VDM [Jon90], and Z [Spi92, BN92].

In this thesis, we use the Z notation [Spi92, BN92]. Z is a model-based specification notation, which was developed by J.-R. Abrial and the Programming Research Group at Oxford University. Z is based on set theory and predicate calculus, and uses a schema calculus for defining states and operations. We assume the reader is familiar with such concepts as sets, relations, and predicate calculus. Appendix A presents excerpts from the Z glossary in [Hay93], explaining the Z notation we use that may not be familiar.

Chapter 2

Specification-based testing issues

To discuss work related to this thesis and to define the contribution of this thesis, we present a classification of the elements of specification-based software testing. This chapter describes the related work in the area in the context of the classification, and discusses how our work fits into the classification.

2.1 Classification of specification-based testing elements

The classification is based on a broad division of testing activities: driving activities, and supporting activities. That is, activities concerned with calculating test cases and assessing software, and activities supporting these driving activities. Most of the elements are applicable to any form of testing, but some are specification-specific, for example, reification of tests, and, perhaps, defining oracles.

The classification is

- ▹ Driving testing
    - Deriving tests
        - Informal specifications
        - Process algebra specifications
        - Algebraic specifications
        - Model-based specifications

  - Guidance
  - Oracles
  - Planning/sequencing
  - Reification

▷ Supporting testing

  - Analysis
    - Evaluation
    - Adequacy
  - Defining test suites
    - Defining tests
    - Structuring tests

2.2 Driving testing

The driving activities of testing are those that are actively concerned with determining tests and test results.

2.2.1 Deriving tests

The most important element of testing is the actual tests themselves, if for no other reason than they are the basis of almost every other testing concern. Accordingly, by far the most prominent use of formal methods in testing is for the derivation and generation of comprehensive black-box test sets. The specification is an authoritative description of functionality and is an obvious source for black-box tests.

The three different styles of formal specification support different test derivation methods. The key concepts of each approach are somewhat complementary. However, we first consider testing using informal specifications.

Informal specifications

Any specification is useful in software testing, and most specifications are informal, presented using natural language and sometimes augmented with diagrams and structure charts. Some work has been done on directly using such specifications in testing. These methods focus on identifying key elements in the specification.

The first realisation when considering deriving tests from informal specifications is the size and impreciseness of such specifications, along with (usually) poor ability to relate components of the specification. Clearly, tool support is a major consideration. Ostrand et al. [OSW86] describe a tool for managing specification-based testing from informal specifications. The major functions of the tool are to annotate parts of the specification for record keeping purposes (for example, highlighting functional units), and maintain relationships between parts of the specification and any test information derived from them.

Category partitioning [OB88, BHO89] is a more advanced method for natural language specification-based testing. Specifications are analysed to determine the various functional units. For each functional unit, the relevant characteristics of the parameters and environment objects are identified and classified in categories. Then, using experience, the tester decides significant choices of input for the categories. This information is the basis of the test suites. The strength of the method is the definition of a test specification language, TSL, used in automatic construction of test suites and test execution. This is described in more detail in section 2.3.2.

It is clear that most of the effort in these approaches is extracting information from informal specifications that is trivial to extract from formal specifications. An example is the work on category partitioning using Z specifications discussed in the section on test derivation from model-based specifications below.

Process algebra specifications

The body of literature on test derivation from process algebra specifications, chiefly concerned with protocol testing, is Brobdingnagian, to say the least. Trying to give pointers into this immense body, on a topic not central to this thesis, is futile. The key concept in test derivation from process algebras is trace analysis. Process algebra notations (e.g., Petri Nets [Pet81], CSP [Hoa85], LOTOS [BB89]) define labelled transition systems, showing how a system moves from one state to the next, without concentrating on the details of states. The simple nature of these systems, and formal syntax of the notations, facilitate automatic generation of traces for all paths through the specification. Thus, large test sets can be generated quite efficiently.

Algebraic specifications

An advantage of algebraic specifications in testing is the ease with which test data are constructed. The constructive nature of the specifications shows how to build various instances of the defined data types, which also implicitly include test sequencing information. There is also great potential for automatically generating test cases from specifications by steadily building up data from the base cases using the constructor operations. Deciding which data to choose is still an open conjecture, but the facility for automation allows more brute-force selection strategies. Research on deriving tests from algebraic specifications is quite advanced.

The basis of most work on testing using algebraic specifications is an ongoing effort involving the Universite de Paris and the Ecole Normale Supérieure, which has produced appealing results in testing theory and test derivation from algebraic specifications. Bougé [Bou85] introduces the concepts of projective reliability and asymptotic validity, which are collectively renamed the regularity hypothesis in later work [BCFG86]. The essence of the hypothesis is that algebraic data types can be tested by constructing all their instances (based on operation combinations) up to a certain complexity level. As this complexity level approaches infinity, the test set approaches the exhaustive test set, so theoretically more effective test sets can be constructed by increasing the complexity level used. Bougé et al. [BCFG86] also introduce the uniformity hypothesis, which is a statement of the well-known technique of inferring a program's correctness for a subdomain of input by testing its correctness on one element of the subdomain. An extension to the underlying theory is reported by Bernot et al. [BGM91], where the projective, asymptotic testing approach is replaced by the approach of starting with an ideal exhaustive test set, derived from the notion of specification satisfaction, and using hypotheses to reduce the size of the test set to practical levels, while maintaining as much of the error-revealing power as possible. A Prolog-based tool has been developed which automatically derives tests from the algebraic specification, requiring minimal human interaction. The specification elements are translated into Prolog clauses which are instantiated to various levels, deriving tests concordant with the regularity hypothesis. In the case of conditional expressions, case analysis is used to partition the input domains which are assumed to behave according to the uniformity hypothesis. The method and tool have been used successfully on a substantial case study [DM91].

In other work on testing based on algebraic specifications, Arkko et al. describe a tool which derives tests from algebraic specifications using methods similar to those outlined above, and assists in transforming them into an appropriate form for testing the implementation [AHKN90]. Gerrard et al. expound the benefits of using these methods in design time testing [GCG90].

Model-based specifications

In contrast to process algebras, the body of literature on deriving tests from model-based specifications is Lilliputian. In regard to test derivation, an advantage of model-based specifications is that the data involved and the relationships amongst them are clearly defined. Effects of data upon operations are usually explicit, which helps identify important sub-divisions of input and output spaces. Research into testing based on model-based specifications is newer than for algebraic specifications, and hasn't received the same focussed attention. One explanation for this is that model-based specifications are not executable and not necessarily constructive, which limits our ability to automatically generate meaningful tests. However, there are still some very good results in specification-based testing using model-based notations.

Hall uses a general approach to derive tests from Z specifications [Hal88]. Simple partitions of the input space are constructed by examining the obvious divisions of input defined in the predicates of operations. This case analysis is highly structured, but not rigorous.

Scullard uses a reduction technique to derive tests from VDM specifications [Scu88]. Function signatures are reduced to their basic types, and the input and output spaces are analysed to extract sets of typical values according to various domain partitions. This is essentially category partitioning [OB88] of both the input and output spaces.

Richardson et al. [ROT89] examine strategies for selecting tests by extending implementation-based testing techniques to apply to formal specifications. Testing strategies are classified as error-based or fault-based. Error-based testing attempts to detect errors in results of program execution, which usually involves some path or partition analysis of the input and selection of tests sensitive to certain types of errors. Fault-based testing attempts to detect faults in the source code, which usually involves applying rules to elements of source code to produce tests sensitive to commonly introduced code faults. Implementation-based strategies are also extended by actively using the specification as an oracle to be violated. This work does not define new ways to select tests; it defines, in general terms for each of these broad classes of testing approaches, the elements of the input and acceptance criteria.

In terms of adapting existing testing techniques to rely on model-based specifications, category partitioning [OB88] has received some attention [AA92, Lay92]. However, the only connection between using category partitioning with formal specifications and improved testing these papers draw is that a model-based specification clearly and unambiguously states information relevant to category partitioning, which is somewhat self-evident. It would be interesting to see how knowledge of model-based language constructs could be used to assist in choosing more informative tests using category partitioning.

Cusack and Wezeman [CW93] derive labelled transition systems from Object-Z[1] specifications. The labelled transition systems are concerned with the external behaviour of the specification and the CO-OP method [Wez90] is used to derive canonical testers from the transition systems which test conformance of the external behaviours of the implementations to the specifications. This is similar to methods used in conformance testing based on process algebras.

The work of Dick and Faivre is a major contribution to the use of model-based

[1] Object-Z is an object-oriented extension to Z developed at the University of Queensland [DKRS91].

specifications in software testing [DF92, DF93]. A standard heuristic in testing is partition testing, where the input space is partitioned into sub-domains, and one test is drawn from each sub-domain. The goal is to select sub-domains so that if the program displays an error for one input from the sub-domain, it will display an error for all inputs from the sub-domain. In simple terms, this generally means choosing input sub-domains which are 'treated the same way' by the program. The obvious way to partition the input space of an operation specified in a model-based notation is to reduce the input expression to disjunctive normal form (DNF), and this is exactly the thrust of Dick's and Faivre's work. The simplicity of this approach should not colour our judgement of its value. Reduction to DNF is a highly effective way of partitioning the input space of an operation. One of the most significant features of this work is that it has tool support. A Prolog-based system is used to transform VDM expressions into DNF (using a VDM grammar and 200 inference rules, and occasional human input). The tool-set also includes a VDM editor and type-checker.

Though all these approaches use a specific model-based notation, the ideas are common to all model-based notations.

2.2.2 Guidance

Many testing strategies exist, only some of which give guidelines as to their use and when to stop using them. The absence of a universally effective testing strategy means multiple strategies should be used in conjunction, but without guidelines, this relies heavily on judgement and experience.

Collofello et al. [CFR90] describe a framework for choosing and applying testing strategies. Their approach classifies four levels of testing (structural, functional, multifunctional, and system) and provides reasonable entry and exit criteria for advancing from one stage to the next. They also discuss a range of common testing strategies and how they fit into this framework. While this classification is useful, they concede experience and judgement is still a major influence on strategy use.

2.2.3 Oracles

Important as the test data are, they are useless if the performance of the system on the tests cannot be judged. The formal specification of a system offers an authoritative source of information about the correct behaviour of the system. In fact, the earliest uses of (formal) specifications in testing have been as sources of test oracles.

DAISTS (Data Abstraction, Implementation, Specification and Testing System) is a system which focuses on using specifications as oracles [GMH81, MG83]. The DAISTS approach is to annotate program code with algebraic specifications of data types and tests. The specification axioms are translated into code segments which call procedures in the implementation. The tests specify which axiom they are testing and provide instantiations for the free variables in the axiom. The translated specification axioms are executed using the test data with success or failure being based on whether both sides of the axiom return equal results. In some cases, this requires defining equality axioms for complex data types. Selection of test data is not a consideration of the system.

DAISTS checks that the program implements the specification for the cases defined in the tests section by constructing implementation drivers from the specification and using the tests as input. This notion of the specification driving the implementation is extended by Hayes [Hay86], who considers oracle issues for model-based specifications of abstract data types. Hayes shows how oracle procedures can be derived from Z specifications of abstract data types to check invariants, pre-conditions, and the input-output relationship. This is preliminary work on data refinement and is concerned with demonstrating that the more concrete specification of the data type implements the abstract specification of the data type.

The importance of oracles is being taken more and more seriously, as is demonstrated by Richardson et al. [RAO92], who argue 'test oracles should be derived from specifications in conjunction with testing criteria, represented in a common form, and their use made integral to the testing process'. The underlying approach is not limited to any particular specification language, though [RAO92] presents examples using the RTL and Z notations. The approach is to construct mappings from the name spaces of the specification and implementation to the name space of the oracle. Usually, the oracle name space is the same as the specification name space. Mappings between implementation and corresponding specification control points are established, as are data mappings between the implementation state and the specification state. The implementation state and state changes are monitored at determined control points, and the implementation state is checked using the data mapping to the corresponding specification state as an oracle. This approach assumes that operation sequencing information is available and requires substantial, but very worthwhile, extra development effort in defining the various mappings.

2.2.4 Planning/sequencing

It may not be possible to directly apply the tests derived from specifications. Part of each test involves a concept of system state, embodied in various variables. States involving complex data may need to be constructed. If there are no special operations external to the system under test to build states, then operations of the system must be used. This requires additional planning to determine how these states can be constructed, and consequently in what order system functions must be tested in order to ensure that an untested function is not being used to test another function. This information is also contained in formal specifications, though it is easier to extract in some cases.

In process algebra and algebraic specifications, this information is explicit, whereas in model-based specifications it is implicit. Hall, considering how to construct certain system states from operations in a Z specification, notes, 'All this leads us towards a far more algebraic interest in the specification, and the need to know the result of sequences of operations in terms of their visible effects for given input sequences.' [Hal88].

For example, for algebraic specifications, a derived test is defined in terms of the constructor operations required to build it from the basic type components; this embodies a direct mapping to the sequence of operations in the implementation required to construct this test. In model-based specifications, this information is implicit; the state component of a test is defined in terms of the state model used in the specification.

For model-based specifications, approaches taken to determine sequencing information for testing involve extracting a finite state automaton for the system from the information embodied in the specification. Dick and Faivre [DF92, DF93] use reduction to DNF, similar to their approach to deriving tests. The before and after state expressions for all the specified operations are disjoined and reduced to DNF. From the resulting expression, a finite state automaton representing the behaviour of the system can be derived and the relevant paths showing how to construct the tests are shown. Cusack and Wezeman [CW93] also take a state automata approach, deriving labelled transition systems from the specification.

2.2.5 Reification

The abstractness of the (formal) specification has two ramifications on specification-based testing. Firstly, only black-box testing can be achieved. Secondly, in their pure form, the derived tests are as abstract as the specification, and some, perhaps instinctive, transformation from abstract to concrete takes place in producing the final test data. Specification reification (e.g., [Jon90, Mor90b]) has interesting applications to specification-based testing.

Our previous work [SC91, SC93b] recognises the potential of rigorous reification methods for steadily increasing the white-box aspects of the test suite, and for ensuring accurate implementation-level representations of the tests. Discussion of these concepts is deferred to chapter 7. Dick and Faivre [DF93] also recognise the possibility of using refinements of specifications to introduce white-box tests into the test suite.

2.3 Supporting testing

Testing support concerns practices designed to improve and judge the overall quality of test suites. Without adequate support practices, the value of testing will always be dubious.

2.3.1 Analysis

Evaluation: what do test results mean?

The fundamental limitation of testing is that the correctness of a program that passes all tests is still undetermined. A successful test indicates the implementation works correctly for that case. The underlying theory of testing is based on inducing the correctness of the implementation on all input based on a sample of input. Hall analyses the relationship between specifications and testing, making this point [Hal91]. Tests represent the 'base case' of the inductive proof. The inductive step relies on generalising the results of one input to many. Mostly, this is done informally. Hall adds that the ultimate goal of specification-based testing is to be able to approximate how close the implementation function is to the specification function, based on measures of how often this generalisation fails, and how bad the results are in those cases.

The basis of these inductive approaches is defining properties of test suites selected according to test selection criteria. Such properties enable statements to be made about implementations tested on test data derived using the test criterion.

The pioneering work of Goodenough and Gerhart on testing theory [GG75] formalised this issue, and prompted much additional work in this area. Goodenough and Gerhart introduce the reliability and validity hypotheses [GG75]. A test suite is defined to be successful if every test is passed. A test criterion is reliable if every derived test suite is successful or every derived test suite is unsuccessful. A test suite is valid if it is possible for a derived test suite to detect every error in the implementation. If a reliable and valid test suite is successful the implementation is correct. As you might expect, it is extremely hard to meet these properties. It is, in fact, trivial to meet each property individually, but there is no general or algorithmic way to construct test suites that are both reliable and valid.

The more general notion of generalising the results of a test is the basis of revealing subdomains, introduced by Weyuker and Ostrand [WO80]. A subdomain of the input is revealing if the implementation passes every test input in the subdomain, or the implementation fails every test input in the subdomain. Revealing subdomains exhibit uniform testing behaviour. So, any input chosen from a revealing subdomain can be used as a base case in inferring implementation correctness for the whole subdomain. Similar to reliability and validity, there is no general or algorithmic way to partition an implementation's input into revealing subdomains.

In their work on deriving tests from algebraic specifications, Bougé et al. also developed hypotheses for test suites. They introduce the regularity hypothesis and uniformity hypothesis for algebraic specifications [Bou85, BCFG86]. The regularity hypothesis was developed from the notions of projective reliability and asymptotic validity, which correspond to Goodenough's and Gerhart's reliability and validity. Algebraic data types are built up from applications of operations on the data type. The number of operations used to build an instance of the data type is called the complexity of the instance. The regularity hypothesis states that for some complexity, k, if the implementation is correct for all data constructs of complexity less than or equal to k, then it is correct for all data constructs. The uniformity hypothesis corresponds to the notion of revealing subdomains. Not surprisingly, there is no general or algorithmic way to select tests satisfying the regularity or uniformity hypothesis.

Adequacy criteria

The general inability to choose tests satisfying the properties above prompted development of less exacting standards for test suites. These properties are usually called adequacy criteria. These simpler properties assist in deciding when an implementation has received sufficient testing. Parrish and Zweben analysed the eleven adequacy criteria for test sets selected using program structure [PZ91]. They demonstrated that these properties were interdependent and inconsistent in some cases, and refined them to a collection of seven independent and consistent criteria:

1. Applicability

   For every program, there is an adequate test set for criteria derived from the program.

2. Nonexhaustive applicability.

Such an adequate test set is not the exhaustive test set.

3. Monotonicity.

   If a test set satisfies some criteria, a superset of the test set satisfies the criteria.

4. Inadequate empty set.

   The empty set is never an adequate test set.

5. Renaming.

   Results are unchanged by isomorphic renaming.

6. Complexity.

   There is a point at which a subset of an adequate test set is not adequate.

7. Definition coverage.

   An adequate test set covers a path from every reachable definition to some reachable use in an implementation, that does not redefine any of the variables in the original definition.

The benchmark of any test selection criteria not meeting such standards as reliability and validity is to be more effective than random testing. The intuition that any structured approach to test selection must be better than random sampling of the input has come under attack [DN84], even to the point of random testing being superior to partition testing approaches in some cases [Ham87]. This calls into question any confidence gained in an implementation through partition testing strategies [HT88]. Weyuker and Jeng compared partition testing and random testing on an analytical basis and determined circumstances in which each strategy excels [JW89, WJ91]. For example, their fourth observation states that if all partitions are of the same size and the same number of tests for each partition is chosen, then the test set is at least as good as a random test set. Clearly, the observations described in this work should be used to make sure any test set derived using a partition strategy is superior to a random test set.

Mutation analysis [Ham77, DLS78, Bud81] is an evaluation method with a different approach. The properties in the previous section aim at demonstrating that the implementation is correct for certain input and inferring it is correct for all input. Conversely, mutation testing demonstrates that certain errors are not in the implementation. Mutation analysis as an adequacy criteria assigns a score to a selection criteria based on how many implementation mutants a test suite derived using the criteria can distinguish.

2.3.2 Defining test suites

Most research in testing focuses on the particular aspect under examination, and is usually not concerned with the full picture. It is very important to be able to collect the test information being produced in a uniform and useful way. Methods for defining and manipulating test suites are very useful, and are probably necessary for any realistic application. Apart from providing much needed structure to the overall testing process, such methods relate important components together, such as functional units, tests, and oracles, and establish dependencies. Effects of changes to the specification can be traced and appropriate areas updated, leading to enhanced regression testing. Broadly, test definition concentrates on two factors: defining tests, and structuring tests.

Defining tests

A test is more than some statement of input data. The functional unit under test, test oracle, and test purpose are all examples of additional considerations. Hence the need for some method of relating this information and defining tests.

TSL, Test Specification Language, is a notation used in defining tests derived using category partitioning [OB88, BHO89]. Test scripts are derived from (informal) specifications indicating relevant inputs to operations and possible choices of values for these parameters. A test frame is one combination of choices for categories. Test frames are automatically generated by essentially calculating the cross product of the choices in the categories. Originally, each test frame was translated by hand into a final test. Extensions to TSL include

- annotations indicating conditions under which to use certain choices because the full cross product generates cases with contradicting requirements,
- annotations of choices with possible final instances to alleviate some of the burden of translating test frames into tests, and
- including result descriptions in test frames for test assessment.

A contrasting approach to test specification is taken by Pachl [Pac90], who introduces notation for defining test criteria. Tests are defined by the criteria they must satisfy. Basic criteria such as being exhaustive and various orderings are defined along with two operators for constructing the union and intersection of test criteria.

Structuring tests

The formally defined tests must also be structured. That is, a test's place in the hierarchy of tests must be made clear, which involves expressing relationships between the test and other tests, the function being tested, the oracle, and possibly other considerations such as relative importance, responsibility, and so on.

An extremely undervalued area of research, this problem is addressed by Ostrand et al. [OSW86]. Their tool for specification-based testing, SPECMAN, assists in structuring tests, by acting as record keeping support for constructing two tables, called the Functional Test Table and the Test Case Table. The Functional Test Table relates tests to functional units described in the (informal) specification, while the Test Case Table relates tests to actual test data, some nebulous description of test purpose, and the tester responsible. A goal of using the tool is to derive a strong notion of the structure of the functional units in the specification, and accordingly, the tests are structured in the Functional Test Table.

2.4 Specification correctness

Specification-based tests are limited by the quality of the specification. The goal of specification-based testing is to demonstrate that an implementation conforms to the specification. Demonstrating that an implementation conforms to an incorrect specification is of dubious value. Incorrect specifications do not meet the user requirements; they can be nearly correct, displaying errors similar to common program faults such as 'off-by-one' errors, or they can be a well-formed statement of something entirely different to what the user requires.

Taking steps to ensure the specification is correct, specification validation, is perhaps the most crucial task in any software engineering project. Unfortunately, the specification can only be compared against the original requirements, which are not formal, so it is not possible to prove that the specification is correct. However, steps can be taken to improve our confidence in the specification.

Kemmerer [Kem85] discusses an approach whereby specifications are made executable, and run on some test cases. Kneuper [Kne89] presents specification animation using symbolic execution as a validation method. Both of these approaches are prototyping the product by making the specification live in some way. There are two major drawbacks of the prototyping approach. Firstly, the prototype has to be exercised on test data, which, similar to test data for implementations, may not reveal all the problems or provide a sufficient demonstration of the program's facilities. Secondly, not all specifications can be executed or animated, hence some translation is involved, which is essentially programming and a potential source of error.

As can be seen, the identified elements of specification-based testing are varied. Most have received considerable attention and many methods exist for tackling most of the problems raised. What is unclear is how to connect and relate the various approaches in a cohesive framework for specification-based testing. One area that has not received as much attention as the others is defining test suites. This is the key to understanding the aims of this thesis.

2.5 Contribution of this thesis

The major contribution of this thesis is a flexible formal framework for conducting specification-based testing. The framework consists of a formal model of tests and test suites, and a method for using the model in testing. It directly supports defining test suites, both tests and structure, in a concise, abstract, and formal manner. We believe that all these aspects are important and are the major failings of existing test definition methods. Neither TSL nor Pachl's method focus on the whole picture, and are not expressive enough to capture all possible situations formally. TSL is not fully formal, relying on natural language descriptions of tests, and is not sufficiently abstract, so that tests and expected results must be expressed in concrete terms. The SPECMAN tool does concentrate on the collection of tests, but does not consider oracle information or a formal uniform representation of information. Most of the specification-based test derivation methods have some implicit notion of test definition, but again, this is not concerned with the whole picture.

Our framework unifies test derivation, oracles, and reification, and is amenable to conducting analysis, deriving planning information, and perhaps expressing some properties of testing strategies useful for guidance. This is mostly a consequence of the formal and abstract test specification. While existing work on test strategies is satisfying, this thesis addresses how to adapt strategies, and introduces two new strategies developed by extending ideas of existing strategies. Applying reification to test suites is a novel contribution of this thesis.

It is possible to formalise the specification of test suites using a model-based specification language, which we do. We feel not having to introduce and force users to learn a new notation is an important feature of the framework. Also, being able to define specification-based test suites in the same notation as the specification is advantageous. All these issues are discussed more fully in later chapters.

Chapter 3

Input and output spaces

In essence, an operation or functional unit under test is a relation. It represents some transformation of input values into output values. This chapter is concerned with sources of test cases. At the basis of testing is a solid understanding of the data space from which tests are drawn. This understanding is commonly implicit and subconscious. However, these input and output spaces are the formal foundation of our framework; they are explicit and need definition.

The important concepts of this chapter are Input Space (IS), Output Space (OS), and Valid Input Space (VIS). The significance of these concepts to specification-based testing is explained in this chapter (particularly section 3.3.1), while their role in defining and using the framework is discussed in chapter 4.

3.1 Signatures

The *signature* of an operation defines the structure of the input to and output from the operation. The signature represents the type of the operation, or rather, the type of the operation defines the signature. Consider the function

$$\lambda\, x : \mathbb{N} \bullet x - 1$$

We can see that this function is transforming numbers into numbers. Its signature might be

$$\mathbb{N} \rightarrow\!\!\!\!\!\!\!+ \; \mathbb{N}$$

or more generally

$$\mathbb{N} \times \mathbb{N}$$

The *input signature* is just that component of the signature defining the inputs of the operation. Similarly, the *output signature* is just that component of the signature defining the outputs of the operation. So, the input signature of our decrement function is $\mathbb{N}$, and the output signature is also $\mathbb{N}$.

3.2 Input spaces and output spaces

The *input space* (IS) of an operation is the space from which input can be drawn, and is defined by the input signature. The input signature may have more than one dimension, i.e., the operation defines a relation of domain arity greater than 1. Generally, we want to reference dimensions of an operation's input using identifiers, as in the x-component, or simply x.

Hence, the **input space** is the set of all bindings of named inputs (identifiers) to values of named types. Since Z schemas define sets of bindings, we use Z schemas to define input spaces. The input signature of decrement is $\mathbb{N}$; the input space is

$$[x : \mathbb{N}]$$

The **output space** (OS) is similarly defined over the output signature.

If the components of input or output are not explicitly given names in the operation definition, names can be introduced without loss of generality. So, the output space of decrement could be

$$[y : \mathbb{N}]$$

Consider this further example, an operation for adding an element to a sequence:

$$\begin{array}{|l}
\multicolumn{1}{l}{Add[ELEMENT]} \\
\hline
e? : ELEMENT \\
s? : \operatorname{seq} ELEMENT \\
s! : \operatorname{seq} ELEMENT \\
\hline
s! = s? \frown \langle e? \rangle \\
\hline
\end{array}$$

A Z operation schema definition does not restrict the ordering of input or output components, so the input signature could be either

$$ELEMENT \times \operatorname{seq} ELEMENT$$

or

$$\operatorname{seq} ELEMENT \times ELEMENT$$

In general, this choice is not significant, and we will use the order present in the schema definition to define the unambiguous input signature.

The input space of Add is

$$[e? : ELEMENT; s? : \operatorname{seq} ELEMENT]$$

and the output space is

$$[s! : \operatorname{seq} ELEMENT]$$

More important than what the input and output spaces are, is what they mean. In essence, elements of an operation's input space are type-compatible inputs for the operation, and elements of an operation's output space are type-compatible outputs for the operation.

3.3 Valid input space

If data is type-compatible with an operation, i.e., is in the input space of the operation, it may still not be 'sensible' input for the operation. That is, the operation may not be defined over the entire input space. In other words, an operation may not relate every element of its input space to an element of the output space. Again, consider our decrement function. It is not defined when the input is 0, because the type of the range is the natural numbers. If the function were declared over integers

$$decrement : \mathbb{Z} \longrightarrow \mathbb{Z}$$

then something like

$$decrement\ 0 \qquad\qquad (= -1)$$

makes sense, but the ordered pair

$$(0, -1)$$

is not an element of the function

$$decrement : \mathbb{N} \nrightarrow \mathbb{N}$$

We introduce the notion of *valid input space* (VIS) to be the subset of the input space for which the operation is defined. The **valid input space** of an operation is the subset of the input space satisfying the pre-condition of the operation. The valid input space may equal the input space. The valid input space can be derived directly from the formal specification of an operation, and this is largely an automatic process. There is one valid input space for each operation.

For example, consider this operation which makes an iinteger value one closer to zero

$$\begin{array}{|l}
\textit{ToZero} \\
\hline
x?, x! : \mathbb{Z} \\
\hline
(x? < 0 \land x! = x? + 1) \lor (x? > 0 \land x! = x? - 1) \\
\hline
\end{array}$$

The input space is

$$[x? : \mathbb{Z}]$$

but the operation is not defined when

$$x? = 0 \qquad\qquad (\text{i.e., } \neg\,(x? < 0 \lor x? > 0))$$

so the valid input space is

$$[x? : \mathbb{Z} \mid x? < 0 \lor x? > 0]$$

Appendix B discusses calculating pre-conditions of operations specified using various notations. It is important that all implicit constraints on the pre-condition are considered.

3.3.1 Significance of the valid input space

The valid input space is very significant in specification-based testing; it is the source of all test data. No meaningful tests outside the valid input space can be derived from the specification.

This is because the specification defines exactly what happens for input in the valid input space. No statement is made about what the operation does when given input outside the valid input space. In the *ToZero* example, the specification does not define a value for the output when the input is 0. Thus, the operation could do anything and still satisfy the specification. This means an oracle can't be defined for input outside the valid input space, or rather that there is no need to define an oracle because there is no incorrect behaviour. So, nothing can be determined by exercising the operation on input outside the valid input space.

3.3.2 Exceptions and robustness

It may seem disturbing that we are restricting ourselves to testing only the defined behaviour of operations. People may criticise, 'what about exceptions?', or 'error inputs are important'. These are two separate issues: exception handling and robustness. The first realisation is that exception handling is still tested, since exceptions are defined behaviour of the operation.

Robustness is a more complex issue. We must remember we are conducting black-box testing, in which robustness testing does not take part. Again, this is because no behaviour is specified, so no judgement can be made. Robustness is a non-functional requirement. However, the valid input space concept can be useful in robustness testing. Elements of the input space of an operation that are not elements of its valid input space represent a potential source of robustness tests.

It is simple to derive an expression for the elements of the input space not contained in the valid input space using Z schema negation on the valid input space. This negation of the valid input space under the input space also has application to specification validation, but this discussion is deferred to chapter 7.

3.4 Valid output space

The valid output space of an operation can be defined similarly to the valid input space. However, it is not so prominent in our framework. In fact, it is the source of all expected output expressions, but the concept of output space suits our needs in this regard. The valid output space does have some application to specification validation, but, again, this discussion is deferred to chapter 7.

Chapter 4

Test template framework

This chapter describes the mechanics of the Test Template Framework (TTF), our defining and structuring formalism for specification-based testing. It discusses defining test inputs, test structures and suites, oracles, and test instances.

4.1 Test templates

The central concept of the framework is the Test Template (TT), which is the basic unit for defining data. The art of designing test data is determining the particular aspects of the implementation that are to be tested, and determining the distinguishing characteristics of input data that test these aspects. Once these classes of requirements are defined, any actual input satisfying them is appropriate test data. Most important is defining the classes of requirements that test data must satisfy.

A **test template** is a formal statement of a constrained data space, and thus can be a description of test data as input meeting certain requirements. The key features of a test template are that it is

- generic, i.e., it represents a class of input,
- abstract, i.e., it has the same level of implementation detail as the specification,
- instantiable, i.e., there is some representation of a single element of the defined class of input, and

- derivable from a formal specification.

Test templates constrain important features of data without placing unnecessary restrictions. That is, test templates can be expressed by constraints over the input variables defined in the specification. In this sense, test templates define sets of bindings of input variables to acceptable values. As with the various data spaces discussed in chapter 3, we use Z schemas to model test templates. For example,

$$A\_template \mathrel{\widehat=} [x, y : \mathbb{N} \mid x < y]$$

defines a set of tests having two values, x and y, such that x is less than y. This template can represent the input for a single test case, though it defines an infinite set of possible bindings. The point is that each binding satisfying the template is an acceptable test input exercising the requirements of the single test case.

We stress that a test template only defines sets of data. We use templates to represent test data, but there is nothing intrinsic in their definition that indicates they are defining test data for an operation using some criteria. This is done to preserve flexibility and structure in our framework. Later, we define a hierarchy of test templates, and this is where the connection between templates and test cases is made.

4.1.1 Valid input spaces

We see that test templates and valid input spaces have similar definitions as bindings of input variables to appropriate data values. Our definition of a test template is deliberately flexible, and clearly the valid input space of an operation is a test template for that operation.

As a test template, the valid input space of an operation is very coarse. It offers little in the way of defining classes of input which we believe to have similar error-detecting ability. In fact, the valid input space on its own is suitable only for deriving a suite of random tests, each a member of the valid input space. Nevertheless, the valid input space is a useful template to define and has an important role to play in the framework.

As mentioned at the end of chapter 3, the valid input space of an operation must be the source of all specification-based tests for the operation. This means that any test is an element of the valid input space. It also means that any test template must be a subset of the valid input space. So, we can define a Z type for test templates for a certain operation, *Op*:

$$TT_{Op} == \mathbb{P}\ VIS_{Op}$$

Note the subscripted use of the operation name. This is a practice we will adopt for the remainder of the thesis. This definition defines TT_{Op} to be the type of all test templates for *Op*.

4.1.2 Notes on the schema model of templates

Schemas vs sets: The significance of bindings

It has already been noted that templates describe sets of test data and that Z schemas are used to define templates. It may seem strange not to use sets to define templates. As mentioned in section 4.1, defining test data for an operation involves assigning values to the input components (both state and parameter) of the operation, that is, defining a binding between input component identifiers and values. Thus, a template intuitively defines a set of bindings, which is exactly what a Z schema defines. The set of bindings can be constrained by predicates in the same way as sets are defined using set comprehension. Schema types define generalised tuples, where ordering of components is not significant, and individual components can be referenced. Consider these alternatives

$$SchemaT \mathrel{\widehat{=}} [x, y : \mathbb{N} \mid x < y] \qquad\qquad SetT = \{x, y : \mathbb{N} \mid x < y\}$$

Used as a template, $SetT$ defines a set of ordered pairs, where individual components cannot be referenced. $SchemaT$ defines a set of bindings of values to the identifiers x and y. If B were such a binding ($B : SchemaT$), then $B.x$ and $B.y$ reference the components of the binding. The descriptive power of schemas fits the idea of describing test data.

What does using a schema to represent a template mean? As a test template, $SchemaT$ describes the set of test data consisting of two components, x and y, both natural numbers, satisfying the condition that x is less than y. Templates are, of course, types in the Z notation. An instance of a template is a particular binding of values to components, and represents an actual test.

Sets of templates: Z peculiarities

The particulars of the Z syntax and semantics raise two points in the usage of schemas as test templates. These do not restrict the use of templates, but must be made clear. A useful concept in the framework is reasoning with sets of templates, that is, sets of Z schemas. Both points relate to this usage.

Syntax of singleton sets of templates

The first point is one of Z syntax. Schemas are Z types. Defining objects with schema types (bindings) has the syntax $inst : Schema$. However, this is a shorthand for the syntax $inst : \{Schema\}$, which states that $inst$ is a member of the set of bindings defined by $Schema$. Because of this shorthand, a singleton set of schemas, containing only schema S, cannot be declared $\{S\}$, since this is merely the schema type "set of all bindings defined by S." Rather the singleton set is placed in parenthesis to unambiguously describe the correct set: $\{(S)\}$. Non-singleton sets of templates can be defined normally, since there is no ambiguity.

The difference between schema types and schemas

There is a subtle difference between schemas and schema types in Z, best illustrated with an example. Consider the following definitions with the type of the defined entity shown at its side. Bindings are described using the $\langle\!|\,..\,|\!\rangle$ notation from [Spi92].

$Schema \mathrel{\widehat=} [x, y : \mathbb{N}]$	$\text{type} : \mathbb{P}\langle\!| x : \mathbb{N}; y : \mathbb{N} |\!\rangle$
$Schema\_Set == \mathbb{P}\, Schema$	$\text{type} : \mathbb{P}(\mathbb{P}\langle\!| x : \mathbb{N}; y : \mathbb{N} |\!\rangle)$
$\mid S : Schema$	$\text{type} : \langle\!| x : \mathbb{N}; y : \mathbb{N} |\!\rangle$
$\mid SSet : Schema\_Set$	$\text{type} : \mathbb{P}\langle\!| x : \mathbb{N}; y : \mathbb{N} |\!\rangle$
$\mid SS : SSet$	$\text{type} : \langle\!| x : \mathbb{N}; y : \mathbb{N} |\!\rangle$

Both *Schema* and *SSet* define sets of bindings, and instances of each are as expected. However, they are not exactly the same. Despite the similarity, *Schema* is a Z schema, and *SSet* is only a set of bindings. This means that operations of the schema calculus cannot be applied to *SSet*: it is a set, not a schema. In every other regard *Schema* and *SSet* are identical. Instances of both are bindings (with no ordering of elements and component reference). Because the types of schemas and sets of bindings are so similar, schemas can be used in set expressions. Set operations require all sets in the expression to have the same signature. The resulting type of a set expression involving schemas is a set of bindings.

4.2 Test template hierarchy

We use a structured approach to build a hierarchy of test templates. Coarser templates are iteratively divided into smaller templates using testing strategies. Test data derivation is simplified by this structured approach involving the systematic application of various testing strategies.

Since all tests for an operation must be derived from the operation's valid input space, the valid input space is the starting point of a hierarchy. Once the valid input space of the functional unit is determined, the next step is to subdivide the valid input space into the desired subsets, or partitions, called *domains*. Choice of domains is not determined by the test template framework. Rather, testing strategies and heuristics are used to subdivide the valid input space. The goal is to derive domains which are equivalence classes of error-detecting ability for the function under test, and which cover the valid input space. That is, the goal is to choose domains so that each element of a domain has the same error-detecting ability. Some, but not all, strategies assume every element of a domain is equivalent to all the others for this purpose and so only one need be chosen. However, this assumption is often invalid. To preserve the flexibility to choose tests for domains selectively, the domain derivation step is used repeatedly, dividing domains into further sub-domains, until the tester is satisfied that the domains represent desired equivalence classes.

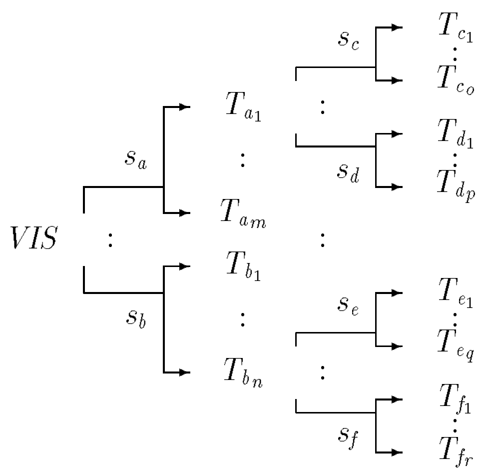


Figure 4.1: Typical TTF Hierarchy

This derivation results in a collection of test templates, related to each other by their derivation and the strategies used in their derivation. We construct a graph where nodes are templates and edges represent application of testing strategies. The edges are directed from parent templates to child templates. Typically, a template hierarchy looks something like Figure 4.1. A hierarchy can be considered as a tree of tests, with the valid input space at the root. In fact, in the general case, a hierarchy is a directed graph, because it is possible to derive the same template using different strategies (and hence different links in the graph). The significance of a template in the hierarchy is that it can be used as the source of test data. If it is too coarse for this, there should be sub-templates derived representing finer divisions of the parent template. The terminal nodes in a hierarchy represent the final input classes.

Some strategies do not advocate domain partitioning (e.g., random testing), in which final tests are derived directly from the valid input space. Some partitioning strategies assume each member of a domain is equivalent to all others, in which case only one level of derivation is required. Some strategies may advocate further subdividing of already derived templates. The framework is merely a defining structure, and doesn't enforce particular derivation approaches on the tester. Figure 4.1 shows a common hierarchy structure.

4.2.1 Hierarchy model

The hierarchy of templates for each operation is a directed graph. Notationally, all elements of the hierarchy relating directly to the particular operation or functional unit under test are subscripted with the operation's name. All templates in the hierarchy are sub-schemas of the valid input space. The hierarchy shows the derivation structure of the templates as a relationship between sets of templates derived from some other template using some testing strategy. The generic set of strategies is introduced and deliberately left abstract:

$$[STRATEGY]$$

The Test Template Hierarchy (TTH) graph for an operation is a set of mappings from parent template/strategy tuples to the set of child templates derived from the parent using the strategy:

$$TTH_{Op} : TT_{Op} \times STRATEGY \nrightarrow \mathbb{P}\ TT_{Op}$$

Templates are defined in terms of their parents and additional constraints. For example, a template, $T1$, derived from VIS_{Op} with the additional constraint cst is defined

$$T1 \mathrel{\widehat=} [VIS_{Op} \mid cst]$$

If the strategy used in this derivation was $strat$, then its position in the hierarchy can be described by

$$strat : STRATEGY$$

$$T1 \in TTH_{Op}(VIS_{Op}, strat)$$

If $T1$ is the only template derived from the valid input space using $strat$, then this section of the hierarchy can be completely defined by

$$\{(T1)\} = TTH_{Op}(VIS_{Op}, strat)$$

Useful relationships among templates, based on the structure of the hierarchy, can be defined. We define two standard functions over templates in a hierarchy: $children_{Op}$ and $descendants_{Op}$.

$$children_{Op} : TT_{Op} \longrightarrow \mathbb{P}(TT_{Op})$$

$$children_{Op} = (\lambda\, T : TT_{Op} \bullet \bigcup\{s : STRATEGY \bullet TTH_{Op}(T, s)\})$$

$$descendants_{Op} : TT_{Op} \longrightarrow \mathbb{P}(TT_{Op})$$

$$\begin{aligned} descendants_{Op} = (\lambda\, T : TT_{Op} \bullet \\ children_{Op}(T) \cup \bigcup\{T2 : children_{Op}(T) \bullet descendants_{Op}(T2)\}) \end{aligned}$$

The function $children_{Op}$ determines the set of templates directly derived from some template using any strategy. For example, given the hierarchy in figure 4.1

$$children_{Fig1}(VIS) = \{T_{a_1}, .., T_{a_m}, .., T_{b_1}, .., T_{b_n}\}$$

The function $descendants_{Op}$ determines the set of templates directly or indirectly derived from some template using any strategy. That is, the descendant templates from some template are all the templates in the sub-graph extending from that template. For example, given the hierarchy in figure 4.1

$$\begin{aligned} descendants_{Fig1}(VIS) = \{&T_{a_1}, .., T_{a_m}, .., T_{b_1}, .., T_{b_n}, \\ &T_{c_1}, .., T_{c_o}, .., T_{d_1}, .., T_{d_p}, .., T_{e_1}, .., T_{e_q}, .., T_{f_1}, .., T_{f_r}\} \end{aligned}$$

4.3 Instances

After applying all the desired strategies to derive test templates, the template hierarchy is considered complete. Instances of the templates in the hierarchy represent test data. If no further subdivision of templates is to be undertaken, each instance of a terminal template in the hierarchy graph is considered equivalent to all other instances of this template for testing purposes. For a complete description of the test data, the only remaining task is to instantiate the terminal templates in the hierarchy.

There are two ways to view the instantiation of templates. Before discussing these, however, it must be noted that an instance of a template is a precisely defined object, but it is still abstract. That is, it exists at the same level of abstraction as the templates. An instance of a template will most likely not serve as final test data because it probably has some data reification to undergo. For example, suppose one input class identified by a test template for queue operations involves a two element queue (of natural numbers, say) with duplicate elements. In Z, the queue would be represented by a sequence, so this template would be

$$QT1 \mathrel{\widehat=} [q : \mathrm{seq}\,\mathbb{N} \mid \#q = 2 \land \#(\mathrm{ran}\, q) = 1]$$

Any instances of this template expressed in Z describe specific Z sequences (e.g., $\langle 1, 1\rangle$), but if the final implementation refined the sequence representation of the queue to a linked list, the instances of templates would also have to be refined to suitable linked list equivalents.

The most straightforward way to describe instances of templates is to use schema instantiation. If $QT1$ is a template, then

$$Q : QT1$$

is an instance of the template—it is (abstract) test data. This form of instantiation is no more useful than the original template because no new information is presented. Constraints can be defined on instances, so this approach could be used to describe the test datum mentioned above:

$$\begin{array}{|l} Q : QT1 \\ \hline Q.q = \langle 1, 1\rangle \end{array}$$

The preferred approach to describing instances is to define instance templates. These are merely templates (schemas) with only one possible instantiation. This approach is more attractive in three ways. Firstly, it presents more information, as in the second example of using schema instantiation above. Secondly, uniform use of schemas and templates is made in the hierarchy, which is important when we consider making general expressions about all templates in a hierarchy. Thirdly, some templates derived using strategies may have only one instantiation so, again, the uniformity of the model is preserved. The instance template corresponding to the instance of $QT1$ described above is simply defined as

$$Q \mathrel{\widehat{=}} [QT1 \mid q = \langle 1, 1 \rangle]$$

Again, the final translation of instance templates to concrete test data is implementation dependent. Instance templates are incorporated into the hierarchy. The 'strategy' to derive instance templates is assumed in the framework:

$$\mid\ instantiation : STRATEGY$$

4.4 Oracles

The formal specification does more than describe conditions on the input. The relationship between input states and output states is precisely specified. This means that the specification can serve as a test oracle. An oracle is a means of determining the success or failure of a test. The conceptually simplest oracle is a comparison of the actual output for some input against a pre-calculated expected output for the same input. From the formal specification, it is simple to derive descriptions of expected output for given input.

Using a similar idea to test templates, we construct abstract specifications of expected output, which we call **oracle templates**. This is a simple model of oracles, but is flexible and can be easily extended with more complex and rigorous oracle models. We are limited at this stage to the abstract level when dealing with the formal specification, and so cannot include considerations of actual output in our basic oracle model. Chapter 7 discusses some ways to use oracle templates as the basis for assessing actual test runs.

Like test templates, oracle templates are essentially just descriptions of data sets, and it is our interpretation of them that lends them meaning. Oracle templates represent a precise description of the set of suitable output for certain input. An oracle template is derived for each test data template by using the input-output relationship of the operation to derive an expression for the output components given the input as described in the test template. The oracle template is defined over the output space of the operation. Note that an oracle can define sets of expected output in cases where the template from which it is derived is not an instance template, or where the operation is non-deterministic.

A general expression for the oracle template of any test template T derived from operation Op is

$$(Op \land T) \upharpoonright OS_{Op}$$

This describes the restriction of the operation's input to that defined in the test template projected onto the output space (OS) of the operation[1]. We use the expression $oracle_{Op}$ to represent this. For example, with the test template $T1$

$$oracle_{Op}(T1) == (Op \land T1) \upharpoonright OS_{Op}$$

Thus, a description of the expected output for each test template can be derived. Again, final concrete instantiation of oracle templates depends on the final implementation.

We can also describe general features of oracles in some cases. Firstly, distinguishing the state components and the parameter components of a schema is useful. We define $StateComp(Op)$ to extract the state components from a schema, and $ParamComp(Op)$ to extract the parameter components from a schema. For example, given the schema

$$\begin{array}{|l}
\hline
\textit{A_schema} \\
state, state' : STATE \\
in? : IN \\
out! : OUT \\
\hline
\end{array}$$

[1]The projection operation is defined in the Z glossary in appendix A.

$$StateComp(A\_schema) = (state, state')$$

$$ParamComp(A\_schema) = (in?, out!)$$

Hence we can restrict attention in a schema to the state and its constraints, and similarly the parameters and their constraints, using the hiding operator on schemas.

$$State(Op) \mathrel{\widehat{=}} Op \setminus ParamComp(Op)$$

$$Params(Op) \mathrel{\widehat{=}} Op \setminus StateComp(Op)$$

General oracle expressions use these extractors in cases where such global statements can be made. For example, a common application of this idea relates to operations that do not change the system state. For such an operation, say $NoChange$, we can define the state component of all the oracles with

$$\begin{array}{l} \forall\, T : descendants_{NoChange}(VIS_{NoChange}) \bullet \\ \qquad State(oracle_{NoChange}(T)) = State(T)' \end{array}$$

Oracle templates do not present any information not already in the specification. They do, however, present it in a more concise and usable form, especially if it is possible to define general oracle expressions. Again, final concrete instantiation of this oracle template depends on the final implementation.

4.5 Other models of test templates

We have presented one model of test templates and hierarchies, but it is not the only possible model. This model of templates arose from the evolving effects of several case studies, and has remained stable in its current form. We have experimented with eight other models, both similar and dissimilar to the current model, which for various reasons directed the development of the model described in this chapter.

It is very natural when using Z to construct an explicit model of what one is describing. This turned out to be somewhat of a pitfall for our template hierarchy.

Our initial models explicitly defined tests and what the relevant components of a test were. The first hurdle to be overcome was defining such a model using legal Z notation, which, indeed, our first attempt did not. The models that were legal Z still suffered from awkward and over-burdened notation, making them practically unusable for defining tests and test suites. Part of the problem was that templates did not receive uniform definition and treatment in the model. There were many different types of templates. This also restricted the flexibility of these models, and made them quite awkward to use.

We then adopted implicit modelling of tests using Z schemas. That is, using Z schemas as test templates (implicitly), rather than explicitly defining templates by defining the components of templates and assigning values to these components. We also abandoned the distinctions between templates. We experimented with various representations of testing strategies and the template hierarchy. These models are all very similar and any could be used; our experience suggests that the current model is the most suitable. Summarising the advantages of the model described in this chapter over other models we tried, the current model

- has a simple and elegant notation, which proved not to be awkward in our case studies,
- implicitly models templates and is legal Z, and
- provides uniform representation of templates, which also helps preserve flexibility.

4.6 In summary

This chapter has introduced the mechanics of the framework. We now have a formalism for

- defining test input classes with granularities ranging from the entire input space to individual elements of the input space,
- defining expected output classes for given input classes, thus extending our ability to define test inputs to the ability to define test cases (input and oracle),

- defining test suites by relating the above components in a hierarchy, and
- making generalised expressions over templates based on definitions in the hierarchy.

Later chapters show how the framework is used in specification-based testing (chapters 5 and 6), and also for use in various analyses and activities other than test derivation (chapter 7).

Chapter 5

Strategies

Some discussion of specification-based testing strategies is in order. Though strategies aren't central to this thesis, we use quite a range in demonstrating the framework. This chapter discusses using existing strategies with the framework (essentially, using strategies at the specification level), and two new strategies we developed.

5.1 Adapting existing strategies

We do not need to invent a gamut of new testing strategies for specification-based testing. Most existing strategies already use either generally applicable selection criteria or specification-level criteria. We can use these strategies with little or no adaptation to the specification-level.

There are two issues in adapting strategies for specification-based testing:

- how differences between the implementation and the specification affect the strategy, and
- how the strategy can make full use of the specification.

Dealing with a specification can affect a strategy due to the abstract nature of the specification, certain elements of specification style, or features of the particular specification language used. Certain implementation concepts are alien in a specification. For example, the concept of a path through an implementation does not transfer well to an abstract specification where the detailed steps in transforming input to output are not defined. So, a strategy like path testing does not adapt well to specification-based testing. Specification languages commonly use different standard data structures such as sets. Testing involving data types can only be concerned with a conceptual understanding of the data type, rather than some implementation representation such as linked lists.

However, there may be little or no adaptation required. Input partitioning, for example, is a concept perhaps more applicable at the specification level than at the implementation level. Using partitioning strategies on implementations usually requires deriving abstract expressions for conditions over the input. Such expressions are specifications, and if they are not already explicit in the specification they should be easier to derive from a specification.

Clearly, knowledge of the specification notation is required to extract relevant information such as condition expressions. Some strategies may be able to make use of details of the notation in the specification, particularly any pre-defined operators in the language.

We consider adapting some popular strategies to give the flavour of using strategies at the specification level.

Partition/Cause-effect testing

Partitioning strategies divide the input space into domains according to some criteria. The most commonly used criteria are branch conditions using variables of the input space. Domains of such a partition are determined by reducing the conditions in the input expression to disjunctive normal form such that each disjunct is disjoint. Each disjunct is a constraint over the input which defines an input domain. Other partitions can be just as easily defined, though not necessarily so easily derived.

Partitioning the input space based only on the input expression can be a pitfall in specification-based testing. Some partitioning strategies partition the input space using more information than contained in the input expression. An example is cause-effect mapping. With the cause-effect strategy, input 'causes' are mapped to output 'effects'. In terms of partitioning, this requires an output partition to be determined, and then the input partition is based on the input domains that map to the identified output domains. This output partition is determined by reducing the output expression to disjunctive normal form.

Domain testing

Domain testing [WC80] uses the control flow of a program to partition its input space. The path predicates form boundaries of the various input domains in the program's input space. The strategy tests for domain errors by checking whether the domain borders are in the correct position. A major pre-requisite of domain testing is that the path predicates have a linear representation in the program's input space, i.e., if a graph of the input space is constructed, the path predicates define domains with linear structures. The dimension of these structures depends on the number of variables in the path predicate. Domain testing also assumes that there is no coincidental correctness, there are no missing path errors, adjacent domains compute different functions, the correct border is also linear, the input space is continuous, and there are no loops in the code as this greatly increases the complexity of the path predicates.

The path predicates are easily determined from a specification by reducing the input expression to disjunctive normal form. The disjuncts are the path predicates and represent the domain boundaries. A much more significant problem with adapting domain testing to the specification level is finding linear representations for the path predicates. That is, finding a way to represent the predicate so that it forms a linear structure in the input space. It is common for path predicates to be high level expressions involving complex data types. In some cases, a linear representation suggests itself, but there is no guarantee that a linear representation exists. For example, sets and set operations defy linear representation[1]. It is probably more likely that a linear representation does not exist—it depends largely on the problem specified. If a linear representation can be found, however, domain testing is a very appealing strategy to use. Another consideration is that specifications commonly use discrete spaces. Numerically, the naturals and integers serve in most specifications,

[1]Except cardinality (#): see the file read example in chapter 6.

and data types are likely to be represented by discrete spaces if a representation can be found at all. This is not a problem per se; continuous input spaces are advantageous because they allow arbitrarily accurate testing. Testing with discrete spaces has limitations on accuracy.

Old favourites

Less rigorous strategies are practised widely in testing. Here we refer to such old favourites as boundary testing, testing zero, one, and many occurrences of some particular phenomenon, and other standard practices given some knowledge of the system specifics, data types, and operators. These adapt to the specification level very easily—the only transition required is working with the notation of the specification rather than that of the implementation as is usually done.

5.2 New specification-based strategies

Our experiments using testing strategies at the specification level led us to develop two new specification-based testing strategies. The first, domain propagation, is an extension of partition testing. The second, specification mutation, is an adaptation of the existing implementation-based mutation testing technique.

5.2.1 Domain propagation

Partition strategies are intuitively appealing strategies. It seems obvious that testing an element from each input class is a satisfactory way of testing which will almost certainly detect all errors. However, as shown by the comparative studies between partition testing and random testing discussed in chapter 2, this intuition is deceptive. In these cases, the domains of the partition are not revealing[2]. What this means is that the domains could/should have been further subdivided, hopefully into a collection of revealing sub-domains.

The problem of shallow partitioning is greater in specification-based testing, because everything is expressed at such a high level. The **domain propagation** strategy

[2] In the sense of Weyuker and Ostrand [WO80].

addresses this shallow partitioning, providing a simple, yet rigorous, technique for partitioning the input space in a more realistic way. The domains of the input partition cannot necessarily be shown to be revealing, but they are a more accurate representation of the various input classes.

In high-level specifications, complicated operations are expressed in relatively simple terms by combining many sub-operations to define the functionality. Each sub-operation has input domain partitions of its own, which are ignored by standard partition strategies. Using domain propagation, the input domain partitions of sub-operations are propagated to the higher level. That is, the domains of a sub-operation partition are expressed in terms of the variables of the higher level operation. The resulting expressions are reduced to disjunctive normal form, providing the more complete domains of the input partition. Note that reducing the expression to DNF also eliminates any contradictions that may arise. In practice, contradictions are eliminated as soon as they are detected to reduce the amount of redundancy.

The depth of domain propagation is not restricted. At the lowest level, pre-defined partitions for the standard operators of the specification language may be used.

Pre-defined partitions of standard operators

For each operator in the specification language, a collection of domains to propagate should be determined. How such a collection is determined is flexible. Also, different collections of domain propagations for the same operator can be maintained for different occasions.

For example, consider simple binary set operations such as union and intersection. A combination of common sense and experience leads us to the following collection of propagation domains. That is, for some basic operation on sets S and T, these expressions represent propagation domains.

1. $S = \{\} \wedge T = \{\}$
2. $S = \{\} \wedge T \neq \{\}$
3. $S \neq \{\} \wedge T = \{\}$

4. $S \neq \{\} \wedge T \neq \{\} \wedge S \cap T = \{\}$

5. $S \neq \{\} \wedge T \neq \{\} \wedge S \subset T$

6. $S \neq \{\} \wedge T \neq \{\} \wedge T \subset S$

7. $S \neq \{\} \wedge T \neq \{\} \wedge S = T$

8. $S \neq \{\} \wedge T \neq \{\} \wedge S \cap T \neq \{\} \wedge \neg (S \subset T) \wedge \neg (T \subset S) \wedge S \neq T$

We desire this collection firstly to be a partition and secondly to be maximal in the sense that there are no more domains required. The first condition is demonstrated by reducing the disjunctive union of the domain predicates to a tautology (appendix E). The second condition is somewhat subjective. We can continue to subdivide these domains until we achieve exhaustive testing. We believe this collection to be a thorough partition of relevant cases.

Since this thesis is largely concerned with the Z notation, we have accumulated standard propagation domains for some Z operators. These can be found in raw form in appendix C.

Discussion of domain propagation

Apart from improved test suites, domain propagation is attractive in two ways. Firstly, the potential to maintain standard collections of domains to propagate in some sort of library is appealing. Secondly, such a library, combined with parsers for the specification language would make automatic generation of domain propagations a reality. The flexibility should be preserved to change propagation domains in the library, use different propagations as the situation demands, or even assign priority to propagations should the need arise.

5.2.2 Specification mutation

Mutation analysis [Ham77, DLS78, Bud81, How82] is primarily a means of assessing test suites. When a program passes all tests in a suite, mutant programs are generated and the suite is assessed in terms of how many mutants it distinguishes from the original program. The major ramification of mutation analysis is that the postulated errors in the dead mutants are shown not to exist in the program. The live mutants are still a cause for concern and are potentially correct programs. The test suite may be updated accordingly based on the live mutants. Mutation relies on an external oracle.

Our approach in specification mutation has a different focus. That is, we are not proposing specification mutants as potentially correct specifications as in implementation mutation. Rather, specification mutation is a specification-based strategy for testing implementations. We assume that the specification we have is correct. We generate specification mutants similarly to implementation mutants and derive a test suite that distinguishes all of these mutants from the original specification. The results of specification mutation testing demonstrate that the implementation does not implement any of the mutant (incorrect) specifications, i.e., that none of the postulated errors are in the implementation. Specification mutation testing locates misinterpretations or misunderstandings of the specification in the implementation.

The relations specified by the original specification and the mutant specification are different. There are four possibilities:

1. Disjoint: $Mut \cap Orig = \{\}$
2. Subset: $Mut \subset Orig$
3. Superset: $Mut \supset Orig$
4. Overlap: $Mut \cap Orig \neq \{\} \wedge \neg\, Mut \subset Orig \wedge \neg\, Mut \supset Orig$

Mutants and originals are distinguished by tests from one of the relations and not the other. Since we maintain the assumption that the specification is correct, all test data must come from the original specification's valid input space. Consequently, any test distinguishes the first type of mutant, and we can never distinguish the third type of mutant because the distinguishing input will lie outside the VIS. The only way to distinguish a type 3 mutant is by human input where the specifier examines the specification and/or the distinguishing predicate. A useful technique in this examination is to construct the negation of the valid input space ($\neg\, VIS$), which is discussed further in chapter 7. So, we only need to choose special mutation tests for type 2 and type 4 mutants.

Choosing mutants

The same ideas used in implementation mutation are applied in specification mutation. That is, operators, types, and variable names are all good sources for mutants. Mutating operators, types, and variables is similar to implementation mutation and easy to envisage. However, abstract specification languages usually have powerful operators built in to the language, and these make for interesting mutations.

Keeping the set theme, consider set union, say $S \cup T$. We construct a mutant specification that uses intersection instead of union, thus $S \cap T$ (a type 2 subset mutant). We need to select a test (from the original valid input space) that distinguishes the mutant from the original. So,

original: $S \cup T$
mutant: $S \cap T$
distinguished by: $S \neq T$

Again, given our Z bias, we have accumulated sets of mutants and distinguishing predicates for the Z operators. These can be found in raw form in appendix D.

Discussion of specification mutation

An appealing factor of implementation mutation is that it can be almost entirely automated. Mutant programs can be generated automatically and exercised on test suites automatically. This automation is important, because any program of reasonable size generates a very large number of mutants. Specification mutation is not so automatable. Mutant specifications can be generated automatically from libraries of standard mutations, but we are not comparing mutants and original against a ready-made test suite. For operator mutants, the distinguishing test can be automatically generated, but for other mutants the distinguishing test needs to be derived by hand. As in implementation mutation, there will be a large number of mutants for any specification of reasonable size. There is a lot of effort involved in deriving a complete mutation test suite.

An open question in mutation analysis is whether it is worthwhile constructing mutants of mutants in the hopes of catching some 'devious' bugs. Practically, the chance of spotting extra errors with this approach does not warrant the costs of constructing a complete suite of second-order mutants. Research into the so-called 'coupling-effect' suggests that tests exposing simple errors also expose complex errors, and that second-order mutation is thus not required [Off92]. We cannot advocate either of these hypotheses, but on the whole we do not have enough evidence to suggest that second-order mutation is worthwhile. Our specification mutation strategy deals only with single mutations.

A problem with mutation is that one is left with little intuition about how effective the resulting test suite is. The suite does, however, verify that none of the proposed mutations are in the implementation. Specification mutants can still be used as a means of assessing other test suites, in the same manner as implementation mutants. As we will see later in this chapter, the descriptions of tests distinguishing mutants are quite general, and represent large subsets of the valid input space of the original specification. A mutation test is stating that correct behaviour for any input satisfying certain conditions guarantees that the mutation is not in the implementation. So, there are usually a wide collection of tests that can satisfy the criteria. Tests derived using other strategies often satisfy most of the mutation criteria as well. In this sense, suites can be rated based on how many mutants they detect.

Finally, though we assume that the original specification is correct, it may not be correct. If we suppose a 'competent specifier hypothesis' along similar lines to the competent programmer hypothesis, then one of the mutant specifications really may be the correct specification. Our experience is that specification mutation assists in specification validation, but there is little formal basis for this assertion. This is discussed further in chapter 7.

5.3 A toy example

Here is a simple example showing how the new strategies work. It also helps illustrate some issues in the next section. We use Z to specify a simple system and derive tests from the specification.

The specification is a simple accounting of a life as a record of deeds done. For each person there are good deeds, bad deeds, and deeds to be done.

We introduce the generic set of deeds which is not specified further in this example. We also introduce the notion of good deeds, being a proper subset of all the deeds. Bad deeds are assumed to be any deeds that are not good.

$$[DEED]$$

$$GoodDeeds : \mathbb{P}\, DEED$$

$$GoodDeeds \subset DEED$$

Each person's deed record keeps track of the good and bad deeds done, and accomplishments yet to be made.

$$DeedRecord$$

$$good, bad,$$
$$deeds\_to\_be\_done : \mathbb{P}\, DEED$$

$$good \cap bad = \{\}$$

When a deed is done, the deed record is updated accordingly.

$$Do$$

$$\Delta DeedRecord$$
$$deed? : DEED$$

$$(deed? \in GoodDeeds \wedge good' = good \cup \{deed?\} \wedge bad' = bad$$
$$\vee$$
$$deed? \notin GoodDeeds \wedge good' = good \wedge bad' = bad \cup \{deed?\})$$
$$deeds\_to\_be\_done' = deeds\_to\_be\_done \setminus \{deed?\}$$

St Peter watches over all, prepared to sentence everyone to eternal life or eternal damnation (accumulating some statistics along the way).

$$Eternity ::= HEAVEN \mid HELL$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\text{— } StPeter \text{ ————————————————}} \\
DeedRecord \\
angelhood! : 0 \mathrel{..} 100 \\
vocation! : Eternity \\
\hline
angelhood! = \#good \mathsf{~div~} (\#good + \#bad) * 100 \\
(angelhood! > 50 \land vocation! = HEAVEN \\
\lor \\
angelhood! < 50 \land vocation! = HELL) \\
\hline
\end{array}$$

The valid input spaces of the operations Do and $StPeter$ are

$$VIS_{Do} \mathrel{\widehat=} [good, bad, deeds\_to\_be\_done : \mathbb{P}\ DEED; deed? : DEED]$$

$$VIS_{StPeter} \mathrel{\widehat=} [good, bad, deeds\_to\_be\_done : \mathbb{P}\ DEED \mid \\ (\#good + \#bad) \neq 0 \land \#good \neq \#bad]$$

Domain propagation testing

Consider Do. Even a cursory examination of the specification reveals that reducing the input expression to DNF yields the following two partitioning templates.

$$P_{Do.1} \mathrel{\widehat=} [VIS_{Op} \mid deed? \in GoodDeeds]$$

$$P_{Do.2} \mathrel{\widehat=} [VIS_{Op} \mid deed? \notin GoodDeeds]$$

However, this does not consider the three set operations in Do:

1. $good \cup \{deed?\}$
2. $bad \cup \{deed?\}$

3. $deeds\_to\_be\_done \setminus \{deed?\}$

We use the propagation domains for basic set operations defined earlier and in appendix C. That is, we construct expressions for these domains using the variables from Do in place of S and T.

For $good \cup \{deed?\}$, this gives us

1. $good = \{\} \wedge \{deed?\} = \{\}$
2. $good = \{\} \wedge \{deed?\} \neq \{\}$
3. $good \neq \{\} \wedge \{deed?\} = \{\}$
4. $good \neq \{\} \wedge \{deed?\} \neq \{\} \wedge good \cap \{deed?\} = \{\}$
5. $good \neq \{\} \wedge \{deed?\} \neq \{\} \wedge good \subset \{deed?\}$
6. $good \neq \{\} \wedge \{deed?\} \neq \{\} \wedge \{deed?\} \subset good$
7. $good \neq \{\} \wedge \{deed?\} \neq \{\} \wedge good = \{deed?\}$
8. $good \neq \{\} \wedge \{deed?\} \neq \{\} \wedge good \cap \{deed?\} \neq \{\} \wedge \neg (good \subset \{deed?\}) \wedge \neg (\{deed?\} \subset good) \wedge good \neq \{deed?\}$

which, after eliminating contradictions and simplifying, reduces to

2. $good = \{\}$
4. $good \neq \{\} \wedge good \cap \{deed?\} = \{\}$
6. $\{deed?\} \subset good$
7. $good = \{deed?\}$

Similarly, for $bad \cup \{deed?\}$, we have

2. $bad = \{\}$
4. $bad \neq \{\} \wedge bad \cap \{deed?\} = \{\}$

6. $\{deed?\} \subset bad$

7. $bad = \{deed?\}$

and for $deeds\_to\_be\_done \setminus \{deed?\}$ we have

2. $deeds\_to\_be\_done = \{\}$

4. $deeds\_to\_be\_done \neq \{\} \wedge deeds\_to\_be\_done \cap \{deed?\} = \{\}$

6. $\{deed?\} \subset deeds\_to\_be\_done$

7. $deeds\_to\_be\_done = \{deed?\}$

In terms of partitioning the input space of Do, propagating these domains gives us this expression for the subdomains of Do

$$
\begin{array}{l}
(deed? \in GoodDeeds \wedge \\
\quad (good = \{\} \vee \\
\quad good \neq \{\} \wedge good \cap \{deed?\} = \{\} \vee \\
\quad \{deed?\} \subset good \vee \\
\quad good = \{deed?\}) \\
\vee \\
deed? \notin GoodDeeds \wedge \\
\quad (bad = \{\} \vee \\
\quad bad \neq \{\} \wedge bad \cap \{deed?\} = \{\} \vee \\
\quad \{deed?\} \subset bad \vee \\
\quad bad = \{deed?\}) \\
) \\
\wedge \\
(deeds\_to\_be\_done = \{\} \vee \\
deeds\_to\_be\_done \neq \{\} \wedge deeds\_to\_be\_done \cap \{deed?\} = \{\} \vee \\
\{deed?\} \subset deeds\_to\_be\_done \vee \\
deeds\_to\_be\_done = \{deed?\})
\end{array}
$$

This expression reduces to a DNF expression with thirty-two disjuncts. This case explosion results from distributing the *deeds_to_be_done* cases across the others. However, since the values for *deeds_to_be_done* do not depend on values for *good* or *bad*, there is no need for so many tests. That is, if *deeds_to_be_done* is correctly updated when $good = \{\}$, it will almost certainly be correctly updated when $good = \{deed?\}$. Of course, there is never certainty, and with sufficient time and/or patience, all thirty-two cases should be tested. It is reasonable to assign lower priority to these cases, however. The point of interest is that where a surface partitioning has two domains, domain propagation reveals a thirty-two domain partition.

Specification mutation

Again, consider *Do*. We will generate one of each of the broad classes of mutants discussed above to give the flavour of the approach. The mutated part of the specification is shown in bold face.

Type mutation.

There are two types defined in the specification: *DEED* and *GoodDeeds*. We construct mutant specifications by swapping occurrences of these types. For example,

$$
\begin{array}{|l}
\hline
TypeMutant_{Do.1} \\
\Delta DeedRecord \\
deed? : \mathbf{GoodDeeds} \\
\hline
\vdots \\
\hline
\end{array}
$$

Here, we've mutated the type of the input variable. This mutant is a subset of the original specification, and is easily distinguished by an input that is not in *GoodDeeds*:

$$MutantTest_{Do.Type.1} \mathrel{\widehat=} [VIS_{Do} \mid deed? \notin GoodDeeds]$$

Variable mutation.

Similarly to type mutation, we swap occurrences of variables for other variables that are still legal specifications. That is, we can't swap *good* for *deed?* because it is not legal Z. This example shows an occurrence of *good* replaced by *bad*:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ VarMutant_{Do.1} \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\vdots \\
\hline
(deed? \in GoodDeeds \wedge good' = \mathbf{bad} \cup \{deed?\} \wedge bad' = bad \\
\vdots \\
\hline
\end{array}
$$

This mutant overlaps the original and, again, is easily distinguished by input where *good* and *bad* are different:

$$MutantTest_{Do.Var.1} \mathrel{\widehat{=}} [VIS_{Do} \mid good \neq bad]$$

<u>Operator mutation.</u>

Here, operators are replaced by other operators maintaining type correctness of the specification. For example, we replace $\cup$ with $\cap$ in the first predicate:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ OpMutant_{Do.1} \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\vdots \\
\hline
(deed? \in GoodDeeds \wedge good' = good \boldsymbol{\cap} \{deed?\} \wedge bad' = bad \\
\vdots \\
\hline
\end{array}
$$

From appendix D we know this mutant is distinguished from the original when *good* is different from $\{deed?\}$. Thus,

$$MutantTest_{Do.Op.1} \mathrel{\widehat{=}} [VIS_{Do} \mid good \neq \{deed?\}]$$

Notes on partitioning domains

Here we examine the differences between input partitioning and cause-effect partitioning, and how specification style can affect cause-effect partitioning. Consider *StPeter* from the toy example. Using simple input partitioning based on disjunctive normal form, there are clearly two partitions:

$$P_{StP.1} \mathrel{\widehat{=}} [VIS_{StP} \mid (\#good \operatorname{\mathsf{div}} (\#good + \#bad) * 100) > 50]$$

$$P_{StP.2} \mathrel{\widehat{=}} [VIS_{StP} \mid (\#good \operatorname{\mathsf{div}} (\#good + \#bad) * 100) < 50]$$

A cause-effect strategy bases the input partition on partitions of the output domain. We do not consider each different value of $angelhood!$ to be significant enough to warrant a different input partition, but rather, partition based on values of $vocation!$. Given this criterion our cause-effect templates are the same as those for input partitioning:

$$CE_{StP.1} \mathrel{\widehat{=}} [VIS_{StP} \mid (\#good \operatorname{div} (\#good + \#bad) * 100) > 50]$$

$$CE_{StP.2} \mathrel{\widehat{=}} [VIS_{StP} \mid (\#good \operatorname{div} (\#good + \#bad) * 100) < 50]$$

These partitionings are very straightforward, but more subtle cases can arise. Certainly, the cause-effect partitions will not always be the same as input partitions based on DNF. Consider the very similar definition of *StPeter* below:

$$
\begin{array}{|l}
\hline
\textit{StPeter2} \\
\hline
DeedRecord \\
angelhood! : 0 \mathrel{..} 100 \\
vocation! : Eternity \\
\hline
angelhood! = \#good \operatorname{div} (\#good + \#bad) * 100 \\
angelhood! > 50 \Rightarrow vocation! = HEAVEN \\
angelhood! < 50 \Rightarrow vocation! = HELL \\
\hline
\end{array}
$$

The valid input space of *StPeter* had restrictions that *good* and *bad* weren't both empty to avoid division by zero, and that they didn't have the same number of elements, for then $angelhood!$ would equal 50 exactly and the operation isn't defined. Though seemingly very similar, *StPeter2* is less restrictive than *StPeter* because it doesn't fail when $angelhood! = 50$, it just isn't deterministic. Despite this, the cause-effect domain divisions established for *StPeter* stand, with an additional domain

$$CE_{StP2} \mathrel{\widehat{=}} [VIS_{StP} \mid (\#good \operatorname{div} (\#good + \#bad) * 100) = 50]$$

This is because we still need to restrict $angelhood!$ to strictly greater or less than 50 to *guarantee* certain output behaviours. Now consider the equivalent operation formed by expressing the implications as disjunctions

$$\begin{array}{|l}
\hline
\textit{StPeter3} \\
\textit{DeedRecord} \\
\textit{angelhood}! : 0 \mathinner{..} 100 \\
\textit{vocation}! : \textit{Eternity} \\
\hline
\textit{angelhood}! = \#\textit{good} \ \mathsf{div} \ (\#\textit{good} + \#\textit{bad}) * 100 \land \\
(\textit{angelhood}! \leq 50 \lor \textit{vocation}! = \textit{HEAVEN} \land \\
\textit{angelhood}! \geq 50 \lor \textit{vocation}! = \textit{HELL}) \\
\hline
\end{array}$$

If we didn't know *StPeter3* was equivalent to *StPeter2* we might be surprised to find that the same cause-effect input classes exist for *StPeter3* as for *StPeter2*. Again, this is because we cannot guarantee any behaviour when $angelhood! = 50$, we know only that the operation succeeds.

The point is that while partitioning approaches such as DNF partitioning and cause-effect mapping may seem simple to apply, particularly with Z specifications, we must be careful that our analysis of the classes is correct. Less intuitive constructions in specifications can lead to less obvious input partitions. For cause-effect mapping, care must be taken to ensure that we can determine that output for certain input is in the required output class. This also raises the question as to what is good specification style, for it seems that *StPeter3*, though equivalent to *StPeter2*, is harder to interpret.

5.4 In summary

This chapter has discussed using testing strategies with our framework and introduced two new testing strategies. This prepares the way for the next chapter which shows substantial examples of testing using these strategies with the framework. Clearly, the key issue of a testing strategy is expressing some criteria for test selection, but there are other issues to discuss. The formal descriptions of test suites enable us to check certain properties we expect of test strategies. We can also analyse the strategies with respect to various adequacy criteria. However, we need to collect some examples of testing with the framework before we look into these issues in more detail in chapter 7.

Chapter 6

Examples and case studies

We have conducted a number of case studies deriving tests from formal specifications in the course of developing the framework. Some of these are presented here as examples of the issues discussed in previous chapters. The discussion in this chapter focuses on using the framework and testing strategies. As such, we derive only test templates and oracle templates using various strategies. That is, we derive the abstract test suite. Other considerations such as creating actual tests from instance templates, reification, and maintenance in the TTF are discussed in chapter 7. These examples form the context for these later discussions.

The first case study is a very simple function. It shows a thorough derivation of tests using simple adapted strategies. It gives the feel of using the framework to define a test suite and for using adapted strategies. The second case study is similar in intent to the first. It is a more complex specification, however, introducing a state component to the specification. The main strategy used is domain testing; it is interesting to see that this fairly complex strategy is easily adapted and used. The last case study is quite detailed and much more complete. The simpler strategies used in the first two studies proved insufficient for this case study, which prompted the development of the two new strategies introduced in chapter 5. This example shows these new strategies in action. A further case study demonstrating the new strategies can be found in [SC93a]. While this case study also highlights the need for these new strategies, we feel that it does not add much to the discussion here, and so it is not included. The specifications are all presented in Z.

6.1 Triangle

Our first example is the very familiar triangle problem, with which we demonstrate some familiar testing strategies. A statement of the problem is drawn from [Mye79]:

> *The required program is to input three natural numbers and determine whether these values can be the sides of a triangle, and if so, what type of triangle (equilateral, isosceles or scalene).*

This case study was used to demonstrate applications of formal methods in software testing in [CS94]; the specification and testing below are drawn from this work.

6.1.1 Specification

We define the set of all valid integer representations of triangles:

$$
\begin{array}{|l}
\mathit{ValidTriangle} : \mathbb{P}(\mathbb{N} \times \mathbb{N} \times \mathbb{N}) \\
\hline
\forall x, y, z : \mathbb{N} \bullet \\
\quad (x, y, z) \in \mathit{ValidTriangle} \Leftrightarrow \\
\qquad (x < y + z \wedge \\
\qquad y < x + z \wedge \\
\qquad z < x + y)
\end{array}
$$

We define a free type *TRIANGLE* representing the possible classification outcomes, with an additional case for invalid triangles.

$$\mathit{TRIANGLE} ::= \mathit{EQUILATERAL} \mid \mathit{ISOSCELES} \mid \mathit{SCALENE} \mid \mathit{INVALID}$$

The classification operation is defined using schemas. Firstly, valid triangles are classified according to the number of sides of equal length:

$$
\begin{array}{|l}
\textit{ValidCase} \\
\hline
x?, y?, z? : \mathbb{N} \\
class! : \mathit{TRIANGLE} \\
\hline
(x?, y?, z?) \in \mathit{ValidTriangle} \\
\#\{x?, y?, z?\} = 1 \Rightarrow class! = \mathit{EQUILATERAL} \\
\#\{x?, y?, z?\} = 2 \Rightarrow class! = \mathit{ISOSCELES} \\
\#\{x?, y?, z?\} = 3 \Rightarrow class! = \mathit{SCALENE} \\
\hline
\end{array}
$$

Any input not representing a triangle is classified as invalid.

$$
\begin{array}{|l}
\textit{InvalidCase} \\
\hline
x?, y?, z? : \mathbb{N} \\
class! : \mathit{TRIANGLE} \\
\hline
(x?, y?, z?) \notin \mathit{ValidTriangle} \\
class! = \mathit{INVALID} \\
\hline
\end{array}
$$

The final classifying operation combines the valid and invalid case operations. The final schema definition is built from smaller components using schema disjunction.

$$\mathit{Classify} \mathrel{\widehat{=}} \mathit{ValidCase} \lor \mathit{InvalidCase}$$

6.1.2 Strategies

The following testing strategies are used in this example.

Cause-effect mapping

Maps causes (input domains) to effects (output domains). Effects are determined by possible output domains. Determine output domains making a disjunctive normal form partition of the output space. For each effect, determine the input domain mapping to the effect.

DNF partitioning

Determine input domains making a disjunctive normal form partition of the input space.

Zero, one, many occurrences

When dealing with occurrence of certain phenomena, consider zero, one, and many occurrences.

Permutation

Break a condition down into all possible permutations of values. Similar to DNF partitioning, but not so algorithmic.

6.1.3 TTF Testing

Preliminary definitions

Firstly, we deal with the preliminary definitions of the input spaces and template hierarchy for *Classify*.

The input space is defined by

$$IS_{Classify} \mathrel{\widehat{=}} [x?, y?, z? : \mathbb{N}]$$

Classify is defined over all possible inputs, so the valid input space is the same as the input space:

$$VIS_{Classify} \mathrel{\widehat{=}} \text{pre } Classify = [x?, y?, z? : \mathbb{N}]$$

The type of test templates for *Classify* is given by

$$TT_{Classify} == \mathbb{P}\ VIS_{Classify}$$

Finally, the hierarchy of templates for *Classify* is introduced:

$$TTH_{Classify} : TT_{Classify} \times STRATEGY \mathrel{+\!\!\!\rightarrow} \mathbb{P}\ TT_{Classify}$$

Cause-effect mapping

We begin testing using the cause-effect method that partitions the valid input space based on equivalence classes of the output space.

$$cause\_effect : STRATEGY$$

This is easy to apply to this specification because the equivalence classes are explicit. By analysing the formal specification, we derive four partitions of the valid input space corresponding to the four possible values of the output variable *class!*.

$$CE_{equ} \mathrel{\widehat=} [VIS_{Classify} \mid (x?, y?, z?) \in ValidTriangle \land \#\{x?, y?, z?\} = 1]$$
$$CE_{iso} \mathrel{\widehat=} [VIS_{Classify} \mid (x?, y?, z?) \in ValidTriangle \land \#\{x?, y?, z?\} = 2]$$
$$CE_{sca} \mathrel{\widehat=} [VIS_{Classify} \mid (x?, y?, z?) \in ValidTriangle \land \#\{x?, y?, z?\} = 3]$$
$$CE_{inv} \mathrel{\widehat=} [VIS_{Classify} \mid (x?, y?, z?) \notin ValidTriangle]$$

We define the derivation relationship between these templates and the valid input space in the template hierarchy.

$$\{CE_{equ}, CE_{iso}, CE_{sca}, CE_{inv}\} = TTH_{Classify}(VIS_{Classify}, cause\_effect)$$

DNF partitioning

Our next strategy uses the partition analysis of Dick and Faivre [DF93]. This reduces mathematical expressions representing operations or existing test templates to Disjunctive Normal Form (DNF). From the DNF expression, disjoint partitions can be extracted easily.

$$DNF\_partition : STRATEGY$$

We apply this strategy to sub-divide the cause-effect templates derived above. We expand and simplify these templates to a form suitable for DNF partitioning:

$$\begin{aligned}
CE_{equ} &= [VIS_{Classify} \mid x? = y? \land y? = z? \land x? \neq 0] \\
CE_{iso} &= [VIS_{Classify} \mid (z? \neq 0 \land x? = y? \land z? \neq x? \land z? < x? + y?) \lor \\
&\qquad (y? \neq 0 \land x? = z? \land y? \neq z? \land y? < x? + z?) \lor \\
&\qquad (x? \neq 0 \land y? = z? \land x? \neq y? \land x? < y? + z?)] \\
CE_{sca} &= [VIS_{Classify} \mid x? < y? + z? \land y? < x? + z? \land z? < x? + y? \land \\
&\qquad x? \neq y? \land y? \neq z? \land z? \neq x?] \\
CE_{inv} &= [VIS_{Classify} \mid x? \geq y? + z? \lor y? \geq x? + z? \lor z? \geq x? + y?]
\end{aligned}$$

CE_{equ} and CE_{sca} provide only one partition, CE_{iso} generates three, while from CE_{inv} we obtain three partitions (not yet disjoint). Thus from CE_{iso} we obtain

$$PA_{iso.1} \mathrel{\widehat=} [CE_{iso} \mid z? \neq 0 \land x? = y? \land z? \neq x? \land z? < x? + y?]$$
$$PA_{iso.2} \mathrel{\widehat=} [CE_{iso} \mid y? \neq 0 \land x? = z? \land y? \neq z? \land y? < x? + z?]$$
$$PA_{iso.3} \mathrel{\widehat=} [CE_{iso} \mid x? \neq 0 \land y? = z? \land x? \neq y? \land x? < y? + z?]$$

$$\{PA_{iso.1}, PA_{iso.2}, PA_{iso.3}\} = TTH_{Classify}(CE_{iso}, DNF\_partition)$$

Dick and Faivre generate disjoint partitions by transforming $A \lor B$ into $A \land B$, $A \land \neg B$ and $\neg A \land B$. Applying this to CE_{inv}, generates

$$\begin{aligned}
CE_{inv} = [\,VIS_{Classify} \mid\ & (x? \geq y? + z? \land y? \geq x? + z? \land z? \geq x? + y?) \lor \\
& (x? \geq y? + z? \land y? \geq x? + z? \land z? < x? + y?) \lor \\
& (x? \geq y? + z? \land y? < x? + z? \land z? \geq x? + y?) \lor \\
& (x? < y? + z? \land y? \geq x? + z? \land z? \geq x? + y?) \lor \\
& (x? \geq y? + z? \land y? < x? + z? \land z? < x? + y?) \lor \\
& (x? < y? + z? \land y? \geq x? + z? \land z? < x? + y?) \lor \\
& (x? < y? + z? \land y? < x? + z? \land z? \geq x? + y?) \\
]&
\end{aligned}$$

Knowledge of properties of arithmetic allow us to simplify this to

$$\begin{aligned}
CE_{inv} = [\,VIS_{Classify} \mid\ & (x? = 0 \land y? = 0 \land z? = 0) \lor \\
& (x? \neq 0 \land x? = y? \land z? = 0) \lor \\
& (x? \neq 0 \land y? = 0 \land x? = z?) \lor \\
& (x? = 0 \land y? \neq 0 \land y? = z?) \lor \\
& (x? \geq y? + z? \land y? < x? + z? \land z? < x? + y?) \lor \\
& (x? < y? + z? \land y? \geq x? + z? \land z? < x? + y?) \lor \\
& (x? < y? + z? \land y? < x? + z? \land z? \geq x? + y?) \\
]&
\end{aligned}$$

This generates these templates

$$PA_{inv.1} \mathrel{\widehat=} [CE_{inv} \mid x? = 0 \land y? = 0 \land z? = 0]$$
$$PA_{inv.2} \mathrel{\widehat=} [CE_{inv} \mid x? \neq 0 \land x? = y? \land z? = 0]$$
$$PA_{inv.3} \mathrel{\widehat=} [CE_{inv} \mid x? \neq 0 \land y? = 0 \land x? = z?]$$
$$PA_{inv.4} \mathrel{\widehat=} [CE_{inv} \mid x? = 0 \land y? \neq 0 \land y? = z?]$$
$$PA_{inv.5} \mathrel{\widehat=} [CE_{inv} \mid x? \geq y? + z? \land y? < x? + z? \land z? < x? + y?]$$
$$PA_{inv.6} \mathrel{\widehat=} [CE_{inv} \mid x? < y? + z? \land y? \geq x? + z? \land z? < x? + y?]$$
$$PA_{inv.7} \mathrel{\widehat=} [CE_{inv} \mid x? < y? + z? \land y? < x? + z? \land z? \geq x? + y?]$$

$$\{PA_{inv.1}, PA_{inv.2}, PA_{inv.3}, PA_{inv.4}, PA_{inv.5}, PA_{inv.6}, PA_{inv.7}\} = \\ TTH_{Classify}(CE_{inv}, DNF\_partition)$$

Note the interesting correspondence with the valid triangle categories: $PA_{inv.1}$ corresponds to an equilateral triangle with zero length sides while $PA_{inv.2}$, $PA_{inv.3}$ and $PA_{inv.4}$ all correspond to isosceles triangles with one side of length zero.

Zero, one, many occurrences

Noting that the above partitioning differs from the intuitive partitioning of the invalid template into cases with three zero values, two zero values, one zero value and no zero values, we choose to apply a 'zero, one, many' rule to $PA_{inv.5}$, $PA_{inv.6}$ and $PA_{inv.7}$ (this rule is not usefully applied to the first four partitions).

$$zero\_one\_many : STRATEGY$$

In this situation we are interested in the number of zero elements in $\{x?, y?, z?\}$ so 'many' is constrained to two. The result is nine further templates:

$$ZOM_{inv.5.0} \mathrel{\widehat=} [PA_{inv.5} \mid y? \neq 0 \land z? \neq 0]$$
$$ZOM_{inv.5.1} \mathrel{\widehat=} [PA_{inv.5} \mid (y? = 0 \land z? \neq 0) \lor (y? \neq 0 \land z? = 0)]$$
$$ZOM_{inv.5.2} \mathrel{\widehat=} [PA_{inv.5} \mid y? = 0 \land z? = 0]$$

$$ZOM_{inv.6.0} \mathrel{\widehat=} [PA_{inv.6} \mid x? \neq 0 \land z? \neq 0]$$
$$ZOM_{inv.6.1} \mathrel{\widehat=} [PA_{inv.6} \mid (x? = 0 \land z? \neq 0) \lor (x? \neq 0 \land z? = 0)]$$
$$ZOM_{inv.6.2} \mathrel{\widehat=} [PA_{inv.6} \mid x? = 0 \land z? = 0]$$

$$ZOM_{inv.7.0} \mathrel{\widehat=} [PA_{inv.7} \mid x? \neq 0 \land y? \neq 0]$$
$$ZOM_{inv.7.1} \mathrel{\widehat=} [PA_{inv.7} \mid (x? = 0 \land y? \neq 0) \lor (x? \neq 0 \land y? = 0)]$$
$$ZOM_{inv.7.2} \mathrel{\widehat=} [PA_{inv.7} \mid x? = 0 \land y? = 0]$$

$$\{ZOM_{inv.5.0}, ZOM_{inv.5.1}, ZOM_{inv.5.2}\} = TTH_{Classify}(PA_{inv.5}, zero\_one\_many)$$
$$\{ZOM_{inv.6.0}, ZOM_{inv.6.1}, ZOM_{inv.6.2}\} = TTH_{Classify}(PA_{inv.6}, zero\_one\_many)$$
$$\{ZOM_{inv.7.0}, ZOM_{inv.7.1}, ZOM_{inv.7.2}\} = TTH_{Classify}(PA_{inv.7}, zero\_one\_many)$$

Partitioning again

Partition analysis can be applied once more to $ZOM_{inv.5.1}$, $ZOM_{inv.6.1}$ and $ZOM_{inv.7.1}$ giving a total of sixteen test templates for the original invalid template.

$$PA_{inv.5.1.1} \mathrel{\widehat=} [ZOM_{inv.5.1} \mid y? = 0 \land z? \neq 0]$$
$$PA_{inv.5.1.2} \mathrel{\widehat=} [ZOM_{inv.5.1} \mid y? \neq 0 \land z? = 0]$$
$$PA_{inv.6.1.1} \mathrel{\widehat=} [ZOM_{inv.6.1} \mid x? = 0 \land z? \neq 0]$$
$$PA_{inv.6.1.2} \mathrel{\widehat=} [ZOM_{inv.6.1} \mid x? \neq 0 \land z? = 0]$$
$$PA_{inv.7.1.1} \mathrel{\widehat=} [ZOM_{inv.7.1} \mid x? = 0 \land y? \neq 0]$$
$$PA_{inv.7.1.2} \mathrel{\widehat=} [ZOM_{inv.7.1} \mid x? \neq 0 \land y? = 0]$$

$$\{PA_{inv.5.1.1}, PA_{inv.5.1.2}\} = TTH_{Classify}(ZOM_{inv.5.1}, DNF\_partition)$$
$$\{PA_{inv.6.1.1}, PA_{inv.6.1.2}\} = TTH_{Classify}(ZOM_{inv.6.1}, DNF\_partition)$$
$$\{PA_{inv.7.1.1}, PA_{inv.7.1.2}\} = TTH_{Classify}(ZOM_{inv.7.1}, DNF\_partition)$$

Permutation

We return to the CE_{sca} template and apply a permutation strategy to the values $x?$, $y?$ and $z?$.

$$\mid\; permutation : STRATEGY$$

Since there are three values, known to be unequal, we get six permutations.

$$PERM_{xyz} \mathrel{\widehat{=}} [CE_{sca} \mid x? < y? < z?]$$
$$PERM_{xzy} \mathrel{\widehat{=}} [CE_{sca} \mid x? < z? < y?]$$
$$PERM_{yxz} \mathrel{\widehat{=}} [CE_{sca} \mid y? < x? < z?]$$
$$PERM_{yzx} \mathrel{\widehat{=}} [CE_{sca} \mid y? < z? < x?]$$
$$PERM_{zxy} \mathrel{\widehat{=}} [CE_{sca} \mid z? < x? < y?]$$
$$PERM_{zyx} \mathrel{\widehat{=}} [CE_{sca} \mid z? < y? < x?]$$

$$\{PERM_{xyz}, PERM_{xzy}, PERM_{yxz}, PERM_{yzx}, PERM_{zxy}, PERM_{zyx}\} = \\ TTH_{Classify}(CE_{sca}, permutation)$$

Templates summary

We have generated twenty-six leaf test templates: one for the equilateral case, three for the isosceles, six for the scalene, and sixteen for invalid triangles. Figure 6.1 shows a diagrammatic representation of the derived test templates and their relationships.

Twenty-six test templates may seem excessive for this small function but the framework allows test generation to stop whenever the test developer feels that an adequate set of tests has been generated. This will depend on the context, the criticality of the software and the perceived need for testing. One option would be to generate only the four test templates corresponding to the cause-effect strategy. Instantiating these templates would produce test-cases that would check that all four outcomes are able to be generated by the software under test. Myers [Mye79] uses this example to demonstrate the complexities of software testing, indicating a wide range of cases that should be tested, and expecting readers not to guess many of them. This systematic testing using the framework covers all these cases and more.

Instances

Because there is no state component to the input or the output, and the parameter components are all simple data types, any instance templates we derive are directly suitable as test data. Constructing such instances is straightforward and requires little discussion. Here is an example instance template for the triangle case study:

$$T1 \mathrel{\widehat{=}} [CE_{equ} \mid x? = 1;\ y? = 1;\ z? = 1]$$

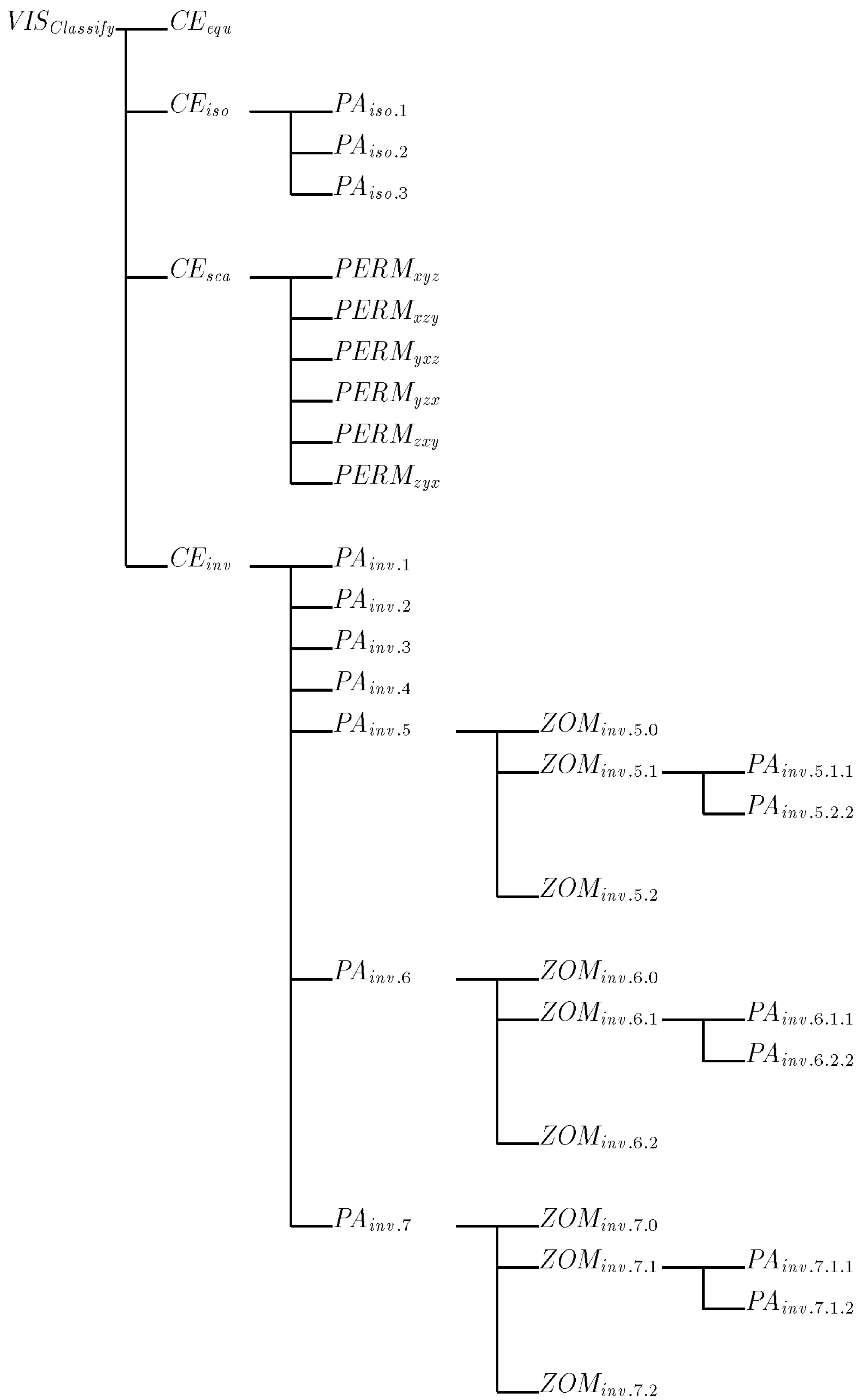


Figure 6.1: Test templates for the triangle classification problem.

Oracles

For the *Classify* operation, the output space is simply

$$OS_{Classify} \mathrel{\widehat{=}} [class! : TRIANGLE]$$

Using the general expression for the oracle template corresponding to any test template derived from an operation:

$$oracle_{Op} = (Op \wedge T) \upharpoonright OS_{Op}$$

We have the following oracle templates for the cause-effect templates derived above.

$$oracle_{Classify}(CE_{equ}) = [OS_{Classify} \mid class! = EQUILATERAL]$$
$$oracle_{Classify}(CE_{iso}) = [OS_{Classify} \mid class! = ISOSCELES]$$
$$oracle_{Classify}(CE_{sca}) = [OS_{Classify} \mid class! = SCALENE]$$
$$oracle_{Classify}(CE_{inv}) = [OS_{Classify} \mid class! = INVALID]$$

It is useful to note that all test templates derived from the four cause-effect templates share the same oracle template as their parent making test validation easier. That is,

$$\forall\, T : descendents(CE_{equ}) \bullet$$
$$\quad oracle_{Classify}(T) = [OS_{Classify} \mid class! = EQUILATERAL]$$

$$\forall\, T : descendents(CE_{iso}) \bullet$$
$$\quad oracle_{Classify}(T) = [OS_{Classify} \mid class! = ISOSCELES]$$

$$\forall\, T : descendents(CE_{sca}) \bullet$$
$$\quad oracle_{Classify}(T) = [OS_{Classify} \mid class! = SCALENE]$$

$$\forall\, T : descendents(CE_{inv}) \bullet$$
$$\quad oracle_{Classify}(T) = [OS_{Classify} \mid class! = INVALID]$$

6.2 File read

The second example introduces a little more complexity of specification. The specification is of a simplified read operation on files, and is based on the specification of the UNIX read operation in [Hay87]. This case study was used as a primary example of the framework in [SC93c]; the specification and testing below are drawn from this work.

6.2.1 Specification

Files are modelled as bounded sequences of bytes. Bytes are introduced as a generic set and not defined further.

$$[BYTE]$$

$$\begin{array}{|l}
MaxFileSize : \mathbb{N} \\
\hline
MaxFileSize > 0
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{File} \\
\hline
file : \mathrm{seq}\, BYTE \\
\hline
\# file \leq MaxFileSize \\
\hline
\end{array}$$

The simplified read operation takes an input length and reads that many characters from the beginning of the file. We distinguish three classes of read operation: a successful read, a read on an empty file, and a request for too many bytes from a file. The *Read* operation does not alter the contents of the file (expressed through the inclusion of $\Xi File$).

$$ReadStatus ::= ok \mid file\_empty \mid file\_too\_short$$

$$
\begin{array}{|l}
\hline
\textit{Read\_0} \\
\Xi \textit{File} \\
\textit{len}? : \mathbb{N} \\
\textit{data}! : \operatorname{seq} \textit{BYTE} \\
\textit{stat}! : \textit{ReadStatus} \\
\hline
\# \textit{file} > 0 \\
\textit{len}? \leq \# \textit{file} \\
\textit{data}! = (1 \mathrel{..} \textit{len}?) \lhd \textit{file} \\
\textit{stat}! = \textit{ok} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{FileEmpty} \\
\Xi \textit{File} \\
\textit{data}! : \operatorname{seq} \textit{BYTE} \\
\textit{stat}! : \textit{ReadStatus} \\
\hline
\# \textit{file} = 0 \\
\textit{data}! = \langle \rangle \\
\textit{stat}! = \textit{file\_empty} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{FileTooShort} \\
\Xi \textit{File} \\
\textit{len}? : \mathbb{N} \\
\textit{data}! : \operatorname{seq} \textit{BYTE} \\
\textit{stat}! : \textit{ReadStatus} \\
\hline
\# \textit{file} > 0 \\
\textit{len}? > \# \textit{file} \\
\textit{data}! = \langle \rangle \\
\textit{stat}! = \textit{file\_too\_short} \\
\hline
\end{array}
$$

$$\textit{Read} \mathrel{\widehat{=}} \textit{Read\_0} \lor \textit{FileEmpty} \lor \textit{FileTooShort}$$

To illustrate how this combination of schemas works, we show the fully expanded schema defined by *Read*. This expansion is syntactic. The predicate of the schema may contain redundant information or may be simplified by other means, but that is not of interest to this discussion.

The fully expanded version of *Read* is

$$
\begin{array}{|l}
\textit{Read} \\
\hline
\mathit{file}, \mathit{file}' : \mathrm{seq}\, BYTE \\
len? : \mathbb{N} \\
data! : \mathrm{seq}\, BYTE \\
stat! : ReadStatus \\
\hline
\#\mathit{file} \leq MaxFileSize \\
\#\mathit{file}' \leq MaxFileSize \\
\mathit{file}' = \mathit{file} \\
((\#\mathit{file} > 0 \wedge len? \leq \#\mathit{file} \wedge data! = (1 \mathinner{..} len?) \lhd \mathit{file} \wedge stat! = ok) \vee \\
(\#\mathit{file} > 0 \wedge len? > \#\mathit{file} \wedge data! = \langle\rangle \wedge stat! = \mathit{file\_too\_short}) \vee \\
(\#\mathit{file} = 0 \wedge data! = \langle\rangle \wedge stat! = \mathit{file\_empty})) \\
\hline
\end{array}
$$

6.2.2 Strategies

The following testing strategies are used in this example.

DNF partitioning

Determine input domains making a disjunctive normal form partition of the input space.

Boundary analysis

Test the values occurring at the boundaries of ranges, particularly in bounded data structures and conditional expressions.

Domain testing

White's and Cohen's domain testing [WC80]. Construct linear representation for input domains, and test particular points on and near the domain boundaries. Of the restrictions to the applicability of domain testing, the only one relevant to this discussion is that the domain boundaries can be represented in a linear space. For this example, the effect of this restriction, and how it is dealt with, is explained later when the abstract tests are derived. The domain boundaries are determined by the simple predicates over the valid input space. Points to verify the boundaries are chosen in accordance with the strategy presented in [WC80]. These tests guarantee that the boundaries are correct (up to a small error margin), i.e., that the control flow is correct.

DNF partitioning and boundary analysis are used in conjunction, while domain testing is used separately, thus generating two test suites.

6.2.3 TTF Testing

Preliminary definitions

The input space of *Read* is given by

$$IS_{Read} \mathrel{\widehat{=}} [\mathit{file} : \operatorname{seq} BYTE;\ len? : \mathbb{N}]$$

The first step in determining the VIS is to determine the pre-condition of *Read*. For every output state, some input state can be found which satisfies the *Read* relation, so there are no implicit restrictions on the input space due to unreachable output states. An expression for the pre-condition is

$$\begin{aligned}&\#\mathit{file} \leq MaxFileSize \land \\ &((\#\mathit{file} > 0 \land len? \leq \#\mathit{file}) \lor (\#\mathit{file} > 0 \land len? > \#\mathit{file}) \lor (\#\mathit{file} = 0))\end{aligned}$$

which is the predicate describing the various combinations of input variables. This predicate simplifies to

$$\#\mathit{file} \leq MaxFileSize \land \#\mathit{file} \geq 0$$

which can be further simplified to

$$\#\mathit{file} \leq MaxFileSize$$

since the range of $\#$ is $\mathbb{N}$. The valid input space is thus

$$VIS_{Read} \mathrel{\widehat=} [file : \mathrm{seq}\, BYTE; len? : \mathbb{N} \mid \#file \leq MaxFileSize]$$

With the definition of the valid input space, the template hierarchy for *Read* can be defined

$$TT_{Read} == \mathbb{P}\ VIS_{Read}$$

$$TTH_{Read} : TT_{Read} \times STRATEGY \mathrel{+\!\!\!\rightarrow} \mathbb{P}(TT_{Read})$$

DNF partitioning and boundary analysis

In deriving the first collection of test data, DNF partitioning is used to partition the valid input space into sub-domains representing equivalence classes of input; test points within these domains are chosen using boundary analysis.

There are three partitions of the pre-condition, as shown by the three distinct classes of input.

$$PA_1 \mathrel{\widehat=} [VIS_{Read} \mid \#file > 0 \land len? \leq \#file]$$
$$PA_2 \mathrel{\widehat=} [VIS_{Read} \mid \#file > 0 \land len? > \#file]$$
$$PA_3 \mathrel{\widehat=} [VIS_{Read} \mid \#file = 0]$$

$$DNF\_partition : STRATEGY$$

$$\{PA_1, PA_2, PA_3\} = TTH_{Read}(VIS_{Read}, DNF\_partition)$$

Note that these same templates would be derived using cause-effect testing. Boundary value points based on each partition are now defined.

$$boundary : STRATEGY$$

For the first partition, data close to the file size limits are chosen in combination with length values close to, and less than, the file size.

$$BA_{1.1} \mathrel{\widehat=} [PA_1 \mid \#\mathit{file} = \mathit{MaxFileSize} \wedge \mathit{len?} = \#\mathit{file}]$$
$$BA_{1.2} \mathrel{\widehat=} [PA_1 \mid \#\mathit{file} = \mathit{MaxFileSize} \wedge \mathit{len?} = \#\mathit{file} - 1]$$
$$BA_{1.3} \mathrel{\widehat=} [PA_1 \mid \#\mathit{file} = \mathit{MaxFileSize} - 1 \wedge \mathit{len?} = \#\mathit{file}]$$
$$BA_{1.4} \mathrel{\widehat=} [PA_1 \mid \#\mathit{file} = \mathit{MaxFileSize} - 1 \wedge \mathit{len?} = \#\mathit{file} - 1]$$
$$BA_{1.5} \mathrel{\widehat=} [PA_1 \mid \#\mathit{file} = 1 \wedge \mathit{len?} = \#\mathit{file}]$$
$$BA_{1.6} \mathrel{\widehat=} [PA_1 \mid \#\mathit{file} = 1 \wedge \mathit{len?} = \#\mathit{file} - 1]$$

$$\{BA_{1.1}, BA_{1.2}, BA_{1.3}, BA_{1.4}, BA_{1.5}, BA_{1.6}\} = TTH_{Read}(PA_1, \mathit{boundary})$$

For the second partition, data close to the file size limits are chosen in combination with length values close to, and larger than, the file size.

$$BA_{2.1} \mathrel{\widehat=} [PA_2 \mid \#\mathit{file} = \mathit{MaxFileSize} \wedge \mathit{len?} = \#\mathit{file} + 1]$$
$$BA_{2.2} \mathrel{\widehat=} [PA_2 \mid \#\mathit{file} = \mathit{MaxFileSize} - 1 \wedge \mathit{len?} = \#\mathit{file} + 1]$$
$$BA_{2.3} \mathrel{\widehat=} [PA_2 \mid \#\mathit{file} = 1 \wedge \mathit{len?} = \#\mathit{file} + 1]$$

$$\{BA_{2.1}, BA_{2.1}, BA_{2.1}\} = TTH_{Read}(PA_2, \mathit{boundary})$$

The last partition varies the length of data read, while keeping the file size at zero.

$$BA_{3.1} \mathrel{\widehat=} [PA_3 \mid \#\mathit{file} = 0 \wedge \mathit{len?} = \#\mathit{file}]$$
$$BA_{3.2} \mathrel{\widehat=} [PA_3 \mid \#\mathit{file} = 0 \wedge \mathit{len?} = \#\mathit{file} + 1]$$

$$\{BA_{3.1}, BA_{3.2}\} = TTH_{Read}(PA_3, \mathit{boundary})$$

Domain testing

Domain testing is a testing heuristic designed to detect ‘shifts’ in the boundaries between input sub-domains caused by errors in the input specifications. There are a number of assumptions, one of which is that the boundaries can be represented in a linear space (as lines or planes, etc.). The simple predicates defining domains represent the domain boundaries. Given this, a finite number of test points for each line are selected so that if the boundary has shifted from the ‘correct’ position it will be detected.

The input space is represented using $\#\mathit{file}$ and $len?$ as the axes.

There are five domain borders in the valid input space, represented by the simple predicates in the specification:

$$DB_1 : \#\mathit{file} \leq MaxFileSize$$
$$DB_2 : \#\mathit{file} > 0$$
$$DB_3 : \#\mathit{file} = 0$$
$$DB_4 : len? \leq \#\mathit{file}$$
$$DB_5 : len? > \#\mathit{file}$$

These borders form only three input domains, corresponding to the partitions in the previous section. A graphical representation of the input space and these borders is shown in figure 6.2.

Points to test each border are derived.

Firstly, the error margin, ϵ, is defined; it is the maximum distance of the OFF points from the boundary being tested. In the discrete space of this example the minimum value for ϵ is 1.

$$\begin{array}{|l} \epsilon : \mathbb{N} \\ \hline \epsilon = 1 \end{array}$$

We introduce the domain testing strategy.

$$\begin{array}{|l} domain\_testing : STRATEGY \end{array}$$

Domain boundary DB_1 requires four test points: 2 ON and 2 OFF. [WC80] uses only 1 OFF point for this type of inequality, but, as noted in [Sco88], another OFF point is required to distinguish this inequality from an equality.

$$DB_{1.ON1} \mathrel{\widehat=} [VIS_{Read} \mid \#\mathit{file} = MaxFileSize \wedge len? = 0]$$
$$DB_{1.ON2} \mathrel{\widehat=} [VIS_{Read} \mid \#\mathit{file} = MaxFileSize \wedge len? = MaxFileSize]$$
$$DB_{1.OFF1} \mathrel{\widehat=} [VIS_{Read} \mid \#\mathit{file} = MaxFileSize - \epsilon \wedge len? = MaxFileSize \mathsf{\ div\ } 2]$$
$$DB_{1.OFF2} \mathrel{\widehat=} [VIS_{Read} \mid \#\mathit{file} = MaxFileSize + \epsilon \wedge len? = MaxFileSize \mathsf{\ div\ } 2]$$

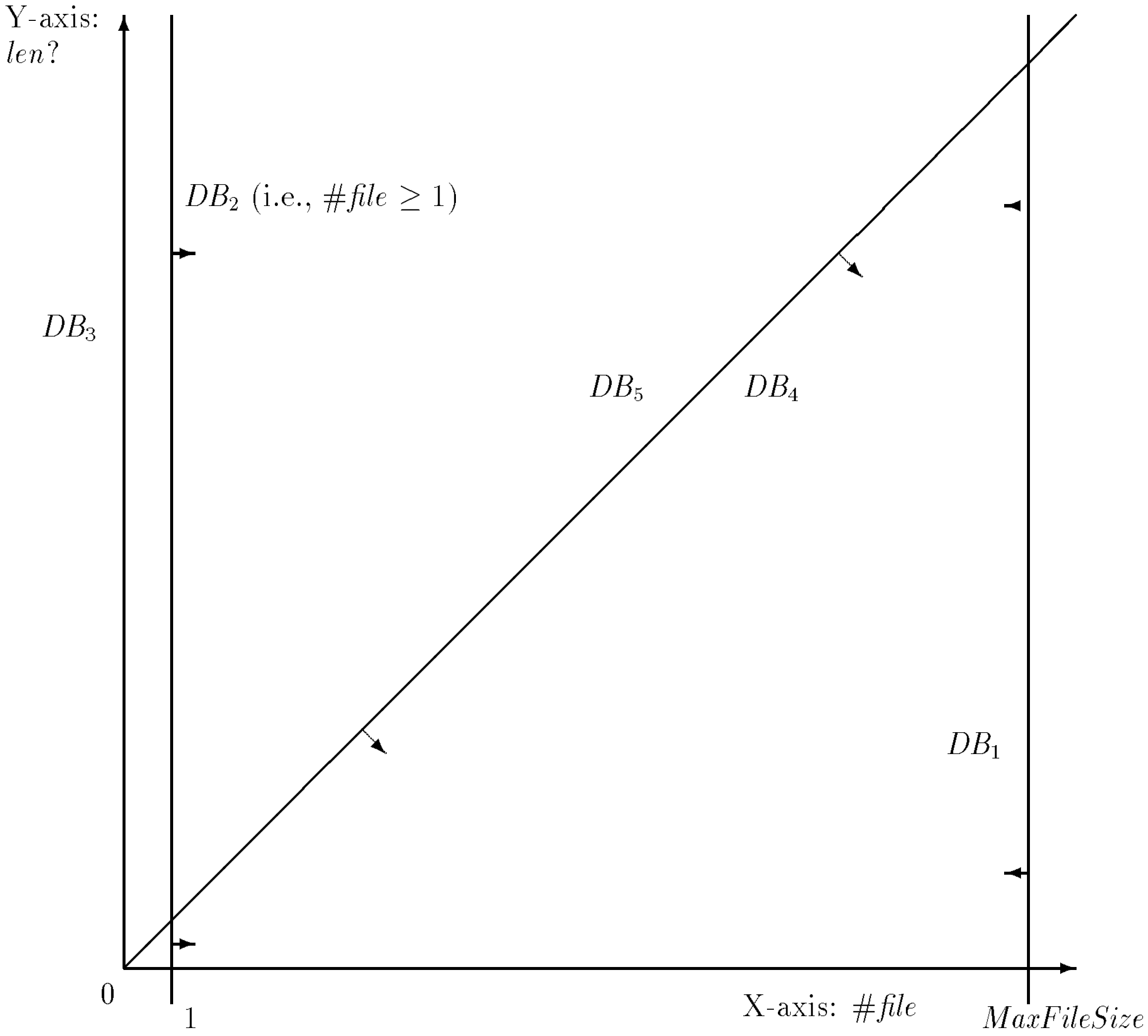


Figure 6.2: Domain boundaries for *Read*

$DB_{1.OFF2}$ falls outside the valid input space (because it contradicts the constraints of VIS_{Read}), and is ignored.

Domain boundary DB_2 requires three test points: 2 ON and 1 OFF.

$$DB_{2.ON1} \mathrel{\widehat{=}} [VIS_{Read} \mid \#file = 1 \wedge len? = 0]$$
$$DB_{2.ON2} \mathrel{\widehat{=}} [VIS_{Read} \mid \#file = 1 \wedge len? = MaxFileSize]$$
$$DB_{2.OFF} \mathrel{\widehat{=}} [VIS_{Read} \mid \#file = 0 \wedge len? = MaxFileSize \mathsf{~div~} 2]$$

Domain boundary DB_3 is an equality boundary requiring five test points: 3 ON and 2 OFF; only 1 OFF point is used because the other falls in the space where file sizes have negative values, which does not exist.

$$DB_{3.ON1} \mathrel{\widehat{=}} [VIS_{Read} \mid \#file = 0 \wedge len? = 0]$$
$$DB_{3.ON2} \mathrel{\widehat{=}} [VIS_{Read} \mid \#file = 0 \wedge len? = MaxFileSize]$$
$$DB_{3.ON3} \mathrel{\widehat{=}} [VIS_{Read} \mid \#file = 0 \wedge len? = MaxFileSize \mathsf{~div~} 2]$$
$$DB_{3.OFF} \mathrel{\widehat{=}} [VIS_{Read} \mid \#file = 1 \wedge len? = MaxFileSize \mathsf{~div~} 2]$$

Domain boundary DB_4 also requires four test points: 2 ON and 2 OFF.

$$DB_{4.ON1} \mathrel{\widehat{=}} [VIS_{Read} \mid \#file = 1 \wedge len? = 1]$$
$$DB_{4.ON2} \mathrel{\widehat{=}} [VIS_{Read} \mid \#file = MaxFileSize \wedge len? = MaxFileSize]$$
$$DB_{4.OFF1} \mathrel{\widehat{=}} [VIS_{Read} \mid \#file = (MaxFileSize \mathsf{~div~} 2) - \epsilon \wedge \\ len? = MaxFileSize \mathsf{~div~} 2]$$
$$DB_{4.OFF2} \mathrel{\widehat{=}} [VIS_{Read} \mid \#file = (MaxFileSize \mathsf{~div~} 2) + \epsilon \wedge \\ len? = MaxFileSize \mathsf{~div~} 2]$$

Domain boundary DB_5 requires three test points: 2 ON and 1 OFF.

$$DB_{5.ON1} \mathrel{\widehat{=}} [VIS_{Read} \mid \#file = 1 \wedge len? = 1]$$
$$DB_{5.ON2} \mathrel{\widehat{=}} [VIS_{Read} \mid \#file = MaxFileSize \wedge len? = MaxFileSize]$$
$$DB_{5.OFF} \mathrel{\widehat{=}} [VIS_{Read} \mid \#file = (MaxFileSize \mathsf{~div~} 2) - \epsilon \wedge \\ len? = MaxFileSize \mathsf{~div~} 2]$$

Because of the intersection of the boundaries, we also need to test the pathological case on the $\#file$ axis where the $\#file > 0$ boundary cuts the $\#file \leq length?$

boundary. Due to the discrete input space, this cut makes a 'hole' in the input space between the points $(0,0),(1,0),(0,1),(1,1)$.

$$DB_{2.PATH} \mathrel{\widehat{=}} [VIS_{Read} \mid \#file = 1 \land len? = 0]$$

In the template hierarchy we have

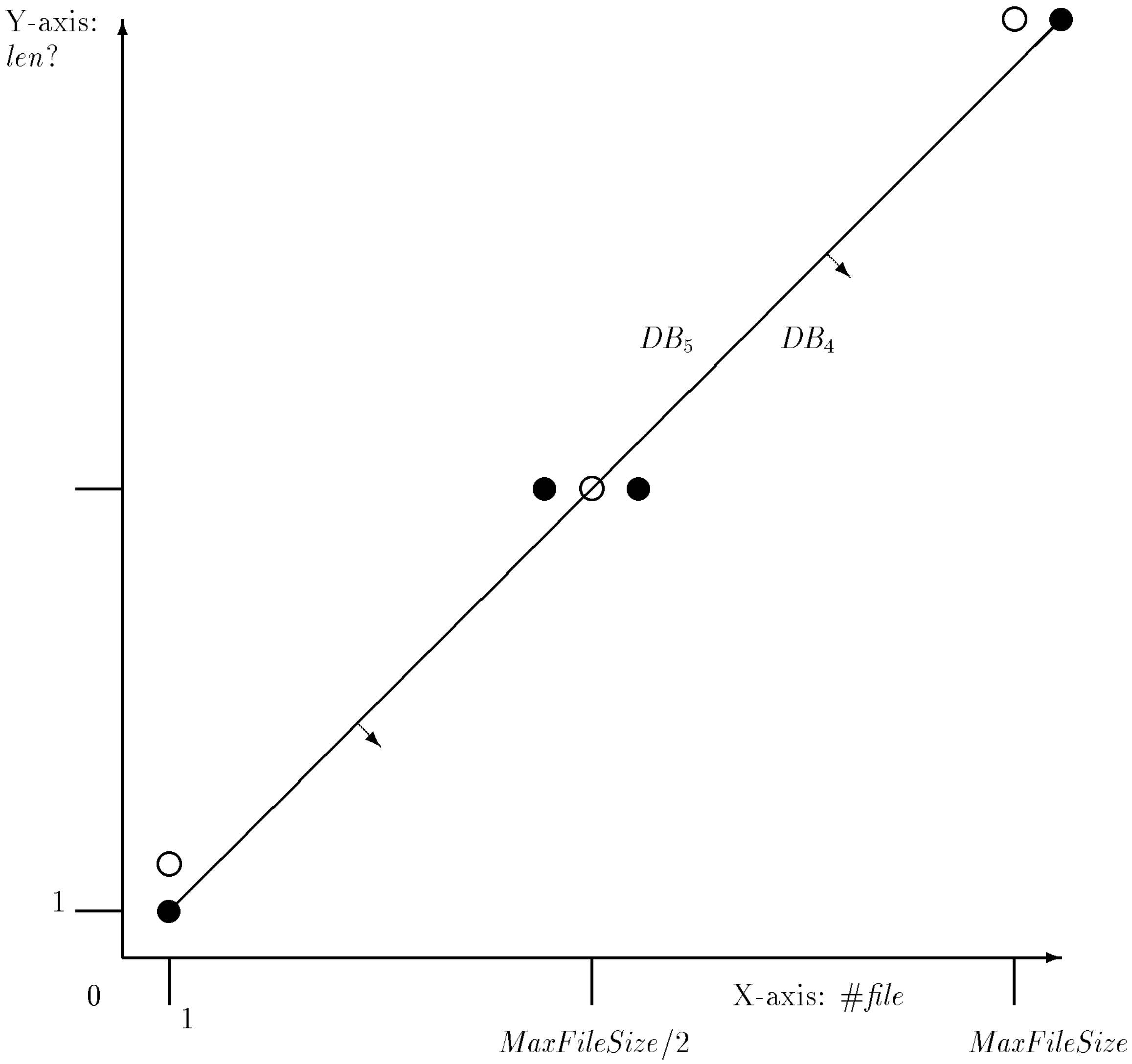
$$\begin{aligned}
&\{DB_{1.ON1}, DB_{1.ON2}, DB_{1.OFF1},\\
&DB_{2.ON1}, DB_{2.ON2}, DB_{2.OFF},\\
&DB_{3.ON1}, DB_{3.ON2}, DB_{3.ON3}, DB_{3.OFF},\\
&DB_{4.ON1}, DB_{4.ON2}, DB_{4.OFF1}, DB_{4.OFF2},\\
&DB_{5.ON1}, DB_{5.ON2}, DB_{5.OFF},\\
&DB_{2.PATH}\} = TTH_{Read}(VIS_{Read}, domain_testing)
\end{aligned}$$

The points derived for boundaries DB_4 and DB_5 are shown in figure 6.3.

Template summary

We have generated eleven leaf test templates in the first suite and seventeen leaf templates in the second suite. Figure 6.4 shows a diagrammatic representation of the derived test templates and their relationships.

Many of these templates are redundant however. Equivalent templates can be derived using one strategy (e.g., domain testing), or can arise due to using multiple strategies. A template is redundant if it is a terminal node in the template hierarchy and it is equivalent to another template in the hierarchy. Redundant templates can be ignored. None of the 11 partition-boundary templates are redundant amongst themselves. Of the 18 domain test points, 12 aren't redundant amongst themselves. A strength of domain testing is the relatively small number of test points generated to infer the correctness of the domain boundaries. The redundancies are



● tests for DB_4: $len? \leq \#file$

○ tests for DB_5: $len? > \#file$

Figure 6.3: Sample test data points for *Read*

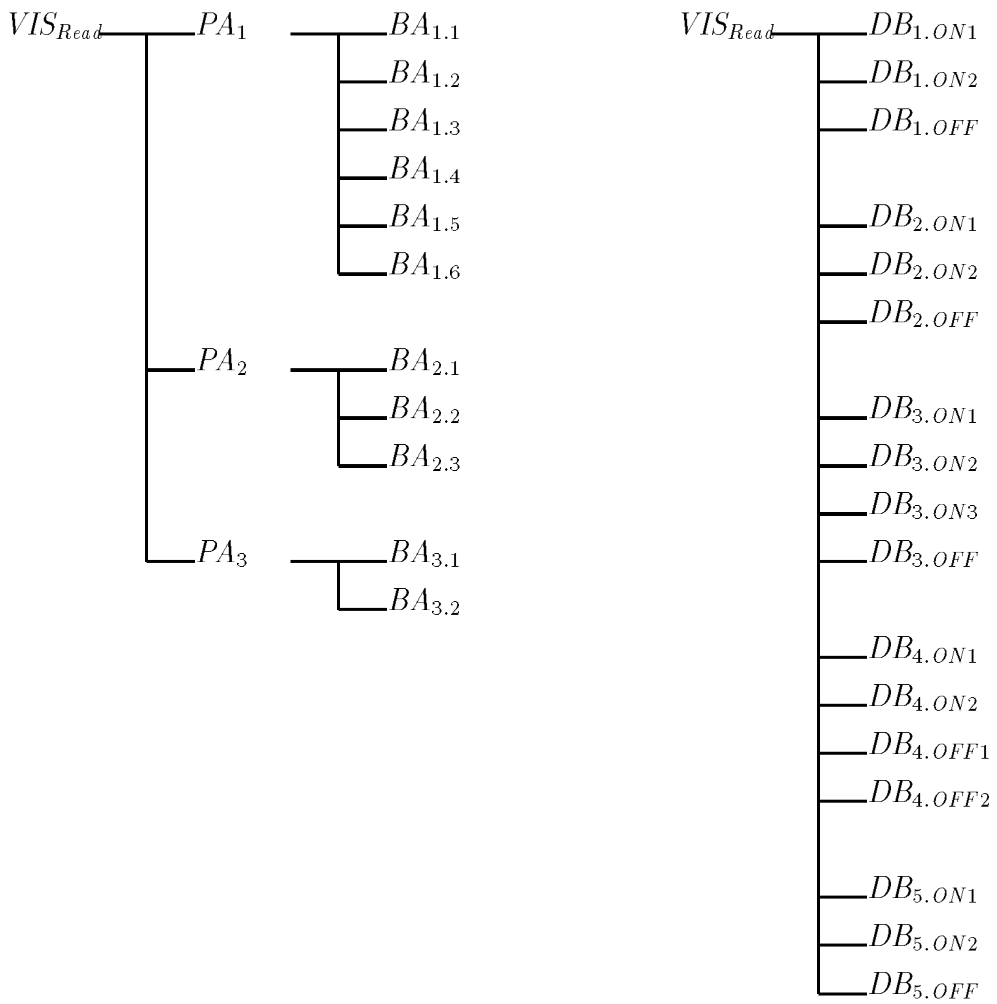


Figure 6.4: Test templates for the file read operation.

$BA_{1.1} \Leftrightarrow DB_{1.ON2} \Leftrightarrow DB_{4.ON2} \Leftrightarrow DB_{5.ON2}$

$BA_{1.5} \Leftrightarrow DB_{4.ON1} \Leftrightarrow DB_{5.ON1}$

$BA_{1.6} \Leftrightarrow DB_{1.PATH} \Leftrightarrow DB_{2.ON1}$

$BA_{3.1} \Leftrightarrow DB_{3.ON1}$

$DB_{2.OFF} \Leftrightarrow DB_{3.ON3}$

$DB_{4.OFF1} \Leftrightarrow DB_{5.OFF}$

It is not surprising that boundary analysis produces similar tests to domain testing because the methods are very similar. The redundancy amongst domain testing templates again is not surprising since test points are chosen on boundary intersections.

Overall, there are 18 non-redundant templates representing test data for *Read*:

1)$BA_{1.1}$	7)$BA_{2.1}$	10)$BA_{3.1}$	12)$DB_{1.ON1}$	15)$DB_{2.ON2}$	17)$DB_{3.ON2}$
2)$BA_{1.2}$	8)$BA_{2.2}$	11)$BA_{3.2}$	13)$DB_{1.OFF1}$	16)$DB_{2.OFF}$	18)$DB_{3.OFF}$
3)$BA_{1.3}$	9)$BA_{2.3}$		14)$DB_{4.OFF1}$		
4)$BA_{1.4}$					
5)$BA_{1.5}$					
6)$BA_{1.6}$					

Instances

Instance templates for *Read* are not as useful as those for the triangle problem, because there is state information to be modelled and this state information is a complex data type. For this case study, all the instance templates do is define the precise abstract input for each recognised input class. To make instance templates, we need to make assumptions about the generic type *BYTE*. The safest way to do this, remembering that we do not know the final form of the type *BYTE*, is to introduce unique values of type BYTE as necessary. For example

| ‘a’: $BYTE$

$T_{1.5} \mathrel{\widehat=} [BA_{1.5} \mid \mathit{file} = \langle \text{‘a’} \rangle;\ len? = 1]$

To make actual tests, we need some way of constructing complex data and setting the state of the system.

Oracles

The output space of *Read* is

$$OS_{Read} \mathrel{\widehat=} [\mathit{file}' : \mathrm{seq}\, BYTE;\ data! : \mathrm{seq}\, BYTE;\ stat! : ReadStatus]$$

and the generic expression for oracle templates for *Read* is

$$oracle_{Read}(T) = (Read \land T) \upharpoonright OS_{Read}$$

The read operation has more complex oracles than the triangle problem because we cannot construct a precise expression describing a general oracle for any templates derived from PA_1. This is because for any of these templates, the oracle expression depends on the state and parameters of the input. One generalisation we can make is that the state component of every oracle template is the same as that for the input template, since the *Read* operation does not change the state:

$$\forall\, T : descendants_{Read}(VIS_{Read}) \bullet State(oracle_{Read}(T)) = State(T')$$

We show the derivation of one oracle template from the first partition to give the flavour:

$$\begin{aligned} & oracle_{Read}(BA_{1.1}) \\ & \quad = [VIS_{Read} \mid \#\mathit{file} = MaxFileSize \land len? = \#\mathit{file} \land \\ & \qquad\qquad \mathit{file}' = \mathit{file} \land data! = (1 \mathrel{..} len?) \lhd \mathit{file} \land \\ & \qquad\qquad stat! = ok] \upharpoonright OS_{Read} \\ & \quad = [OS_{Read} \mid data! = \mathit{file}' \land stat! = ok] \end{aligned}$$

For templates derived from PA_2 and PA_3, we respectively can define all the oracle templates by

$$\begin{aligned} & \forall\, T : descendants_{Read}(PA_2) \bullet \\ & \quad oracle_{Read}(T) = [OS_{Read} \mid data! = \langle\rangle \land stat! = \mathit{file\_too\_short}] \end{aligned}$$

$$\begin{aligned} & \forall\, T : descendants_{Read}(PA_3) \bullet \\ & \quad oracle_{Read}(T) = [OS_{Read} \mid data! = \langle\rangle \land stat! = \mathit{file\_empty}] \end{aligned}$$

6.3 Dependency management system

The Dependency Management System (DMS) case study was a testbed in part of a larger project. This larger project was exploring software development methodologies, and the DMS was used as a common case study [Lin92].

The DMS is a critical component of a theorem proving tool, whose role is to keep track of dependencies between theorems and assertions in a proof, thus preventing circular reasoning. However, the concept of dependency management generalises to other problems (such as revision control systems), so the DMS is a reasonably generic component.

The DMS supports many operations relating to manipulating a dependency graph, from adding nodes and dependencies to calculating all the nodes dependent on some other node. To demonstrate our framework, we examine only one operation of the DMS. This operation, *CanAdd*, determines whether a new dependency can be added to the dependency graph without introducing circularities. An operational profile of the DMS shows that *CanAdd* is the most used operation, and it also has the most interesting specification.

6.3.1 Specification

The specifications of the DMS and *CanAdd* are drawn from [Ros92].

The dependency management system keeps track of dependencies between nodes. In the context of the surrounding proof system, nodes will represent axioms or theories. The basic DMS maintains a set of nodes, direct dependencies between nodes, and inferred transitive dependencies between nodes.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{DepManSys}[X]} \\
nodes : \mathbb{F}\, X \\
dir\_dep\_on : X \leftrightarrow X \\
\_ \succ \_ : X \leftrightarrow X \\
\hline
dir\_dep\_on \subseteq nodes \times nodes \\
(\_ \succ \_) = dir\_dep\_on^{+} \\
\neg\, (\exists\, x : X \bullet x \succ x) \\
\hline
\end{array}
$$

The operation of the DMS we are considering here is concerned with adding dependencies between nodes to the system. A dependency can be added from one node to a second if both are nodes of the system and no dependency (direct or indirect) exists from the second node to the first.

$$\textbf{Bool} ::= \mathit{True} \mid \mathit{False}$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CanAdd}[X]} \\
\Xi \textit{DepManSys}[X] \\
x?, y? : X \\
result! : \textbf{Bool} \\
\hline
\{x?, y?\} \subseteq nodes \\
result! = \mathit{True} \Leftrightarrow (\neg\, (y? \succ x?) \land x? \neq y?) \\
\hline
\end{array}
$$

6.3.2 Strategies

The following strategies are used in this case study.

DNF partition

Determine input domains making a disjunctive normal form partition of the input space.

Domain propagation

Iteratively propagate input domains of sub-operations to higher level, starting from the base level with no sub-operations.

Specification mutation

Postulate mutant specifications and choose tests to distinguish mutants from the original.

6.3.3 TTF Testing

Preliminary definitions

We assume the definition of the generic set for $CanAdd$.

$$[X]$$

The input space space of $CanAdd$ is given by

$$IS_{CanAdd} \mathrel{\widehat=} [nodes : \mathbb{F}\, X;\ dir\_dep\_on, \_ \succ \_ : X \leftrightarrow X;\ x?, y? : X]$$

There are some restrictions on input to $CanAdd$, mainly arising from restrictions on legal states of the DMS. The valid input space of $CanAdd$ is

$$
\begin{array}{|l}
\hline
VIS_{CanAdd} \\
nodes : \mathbb{F}\, X \\
dir\_dep\_on : X \leftrightarrow X \\
\_ \succ \_ : X \leftrightarrow X \\
x?, y? : X \\
\hline
dir\_dep\_on \subseteq nodes \times nodes \\
(\_ \succ \_) = dir\_dep\_on^{+} \\
\neg\, (\exists\, x : X \bullet x \succ x) \\
\{x?, y?\} \subseteq nodes \\
\hline
\end{array}
$$

The template hierarchy for $CanAdd$ is

$$TT_{CanAdd} == \mathbb{P}\, VIS_{CanAdd}$$

$$
\begin{array}{|l}
TTH_{CanAdd} : TT_{CanAdd} \times STRATEGY \leftrightarrow \mathbb{P}\, TT_{CanAdd}
\end{array}
$$

Some simple strategies

By a combination of the simple cause-effect and DNF partitioning we can see that the following is an obvious partition of the input space

$$CE_1 \mathrel{\widehat{=}} [VIS_{CanAdd} \mid \neg\,(y? \succ x?) \wedge x? \neq y?]$$
$$CE_{2PA_1} \mathrel{\widehat{=}} [VIS_{CanAdd} \mid y? \succ x?]$$
$$CE_{2PA_2} \mathrel{\widehat{=}} [VIS_{CanAdd} \mid x? = y?]$$

This is as far as such simple strategies take us with this example, though we are left with the intuition that this is not enough testing. The problem is that transitive closure is not a familiar concept in testing, and standard techniques are not applicable. Hence the need for more specification-specific strategies such as domain propagation and specification mutation.

Domain propagation

Domain propagation is concerned with the actual predicates in the specification. As such, a common set of domain propagations will be derived for each cause-effect template above based on the predicates from the valid input space. Thus, we first apply the domain propagation strategy to VIS_{CanAdd}, and extend this collection as appropriate for each cause-effect template.

$$dom\_prop : STRATEGY$$

Each simple predicate of VIS_{CanAdd} is examined for domain propagations, using the suggested propagations defined in appendix C:

1. $dir\_dep\_on \subseteq nodes \times nodes$
   (a) $dir\_dep\_on = \{\} \wedge nodes \times nodes = \{\}$
   (b) $dir\_dep\_on = \{\} \wedge nodes \times nodes \neq \{\}$
   (c) $dir\_dep\_on \neq \{\} \wedge nodes \times nodes \neq \{\} \wedge dir\_dep\_on \subset nodes \times nodes$
   (d) $dir\_dep\_on \neq \{\} \wedge nodes \times nodes \neq \{\} \wedge dir\_dep\_on = nodes \times nodes$
2. $(\_ \succ \_) = dir\_dep\_on^{+}$

   (a) $\text{dom } dir\_dep\_on \cap \text{ran } dir\_dep\_on = \{\}$

   (b) $\text{dom } dir\_dep\_on = \text{ran } dir\_dep\_on \wedge dir\_dep\_on = dir\_dep\_on^+$

   (c) $dir\_dep\_on \subset dir\_dep\_on^+$

3. $\neg (\exists x : X \bullet x \succ x)$ has no domain propagations to offer.

4. $\{x?, y?\} \subseteq nodes$

   (a) $\{x?, y?\} = \{\} \wedge nodes = \{\}$

   (b) $\{x?, y?\} = \{\} \wedge nodes \neq \{\}$

   (c) $\{x?, y?\} \neq \{\} \wedge nodes \neq \{\} \wedge \{x?, y?\} \subset nodes$

   (d) $\{x?, y?\} \neq \{\} \wedge nodes \neq \{\} \wedge \{x?, y?\} = nodes$

All combinations of the resulting possibilities are considered in conjunction with the whole predicate of the operation to rule out any contradictions. We see that cases 4(a) and 4(b) are unsatisfiable (as $\{x?, y?\}$ can never equal $\{\}$), so these cases are ignored. Predicates 1(b) and 2(c) contradict each other. Predicates 1(c) and 2(b) contradict the valid input space since they require there to be nodes dependent on themselves in $(\_ \succ \_)$. Predicates 2(c) and 4(d) contradict each other since it is impossible to have a pair in $dir\_dep\_on^+$ and not in dir_dep_on when there are only two nodes.

For a similar reason to cases 4(a) and 4(b), case 1(a) is eliminated: we know that $nodes$ must at least have two elements ($\{x?, y?\} \subseteq nodes$), so $nodes \times nodes$ cannot be empty. A more subtle contradiction arises from case 1(d). Combined with the valid input space this case is

$[VIS_{CanAdd} \mid dir\_dep\_on = nodes \times nodes]$

$\Leftrightarrow$

$[IS_{CanAdd} \mid (\_ \succ \_) = (nodes \times nodes)^{+} \wedge \neg (\exists x : X \bullet x \succ x) \wedge$
$\{x?, y?\} \subseteq nodes]$

$\Leftrightarrow$

$[IS_{CanAdd} \mid nodes = \{\} \wedge \{x?, y?\} \subseteq nodes]$

Since there can be no node related to itself by $\succ$.

$\Leftrightarrow$

$[IS_{CanAdd} \mid false]$

and hence, this possibility can never be true in the VIS. These contradictory cases are ignored, leaving the following combinations.

1. $1(b) \wedge 2(a) \wedge 4(c)$
2. $1(b) \wedge 2(a) \wedge 4(d)$
3. $1(b) \wedge 2(b) \wedge 4(c)$
4. $1(b) \wedge 2(b) \wedge 4(d)$
5. $1(c) \wedge 2(a) \wedge 4(c)$
6. $1(c) \wedge 2(a) \wedge 4(d)$
7. $1(c) \wedge 2(c) \wedge 4(c)$

These domain divisions are applied to each of the cause-effect templates derived earlier. No further domain propagations are derived from the remaining predicates in the cause-effect templates, because there are no appropriate domains to propagate. However, some of the domain propagation combinations above will contradict these extra predicates. From template CE_1 we derive the following templates.

$$DP_{CE_1.1} \widehat{=} [CE_1 \mid dir\_dep\_on = \{\} \wedge \{x?, y?\} \subset nodes]$$

$$DP_{CE_1.2} \widehat{=} [CE_1 \mid dir\_dep\_on = \{\} \wedge \{x?, y?\} = nodes]$$

$$DP_{CE_1.3} \mathrel{\widehat=} [CE_1 \mid dir\_dep\_on = \{\} \land \{x?, y?\} \subset nodes]$$

$$DP_{CE_1.4} \mathrel{\widehat=} [CE_1 \mid dir\_dep\_on = \{\} \land \{x?, y?\} = nodes]$$

$$\begin{aligned} DP_{CE_1.5} \mathrel{\widehat=} [CE_1 \mid\ & dir\_dep\_on \neq \{\} \land dir\_dep\_on \subset nodes \times nodes \land \\ & \operatorname{dom} dir\_dep\_on \cap \operatorname{ran} dir\_dep\_on = \{\} \land \\ & \{x?, y?\} \subset nodes] \end{aligned}$$

$$\begin{aligned} DP_{CE_1.6} \mathrel{\widehat=} [CE_1 \mid\ & dir\_dep\_on \neq \{\} \land dir\_dep\_on \subset nodes \times nodes \land \\ & \operatorname{dom} dir\_dep\_on \cap \operatorname{ran} dir\_dep\_on = \{\} \land \\ & \{x?, y?\} = nodes] \end{aligned}$$

$$\begin{aligned} DP_{CE_1.7} \mathrel{\widehat=} [CE_1 \mid\ & dir\_dep\_on \neq \{\} \land dir\_dep\_on \subset nodes \times nodes \land \\ & dir\_dep\_on \subset dir\_dep\_on^+ \land \{x?, y?\} \subset nodes] \end{aligned}$$

Templates $DP_{CE_1.3}$ and $DP_{CE_1.4}$ are equivalent to $DP_{CE_1.1}$ and $DP_{CE_1.2}$ respectively, and are ignored.

$$\{DP_{CE_1.1}, DP_{CE_1.2}, DP_{CE_1.5}, DP_{CE_1.6}, DP_{CE_1.7}\} = TTH_{CanAdd}(CE_1, dom\_prop)$$

From template CE_{2PA_1} we derive a reduced set of templates due to contradictions between the domain propagations and the final predicate of CE_{2PA_1}. The cases involving predicate 1(b) contradict the requirement of this cause-effect template that a dependency from $y?$ to $x?$ already exists, because 1(b) requires the dependency relation to be empty. We derive the following templates

$$\begin{aligned} DP_{CE_{2PA_1}.5} \mathrel{\widehat=} [CE_{2PA_1} \mid\ & dir\_dep\_on \neq \{\} \land dir\_dep\_on \subset nodes \times nodes \land \\ & \operatorname{dom} dir\_dep\_on \cap \operatorname{ran} dir\_dep\_on = \{\} \land \\ & \{x?, y?\} \subset nodes] \end{aligned}$$

$$\begin{aligned} DP_{CE_{2PA_1}.6} \mathrel{\widehat=} [CE_{2PA_1} \mid\ & dir\_dep\_on \neq \{\} \land dir\_dep\_on \subset nodes \times nodes \land \\ & \operatorname{dom} dir\_dep\_on \cap \operatorname{ran} dir\_dep\_on = \{\} \land \\ & \{x?, y?\} = nodes] \end{aligned}$$

$$DP_{CE_{2PA_1}}.7 \mathrel{\widehat{=}} [CE_{2PA_1} \mid dir\_dep\_on \neq \{\} \wedge dir\_dep\_on \subset nodes \times nodes \wedge \\ dir\_dep\_on \subset dir\_dep\_on^+ \wedge \{x?, y?\} \subset nodes]$$

$$\{DP_{CE_{2PA_1}}.5, DP_{CE_{2PA_1}}.6, DP_{CE_{2PA_1}}.7\} = TTH_{CanAdd}(CE_{2PA_1}, dom\_prop)$$

From template CE_{2PA_2} we also derive a reduced set of templates due to contradictions between the domain propagations and the final predicate of CE_{2PA_2}. The combination of 1(c) and 4(d) is a contradiction in this case where $x?$ is the same as $y?$ because $nodes$ is restricted to having only one element, and we cannot construct dir_dep_on to be a non-empty proper subset of $nodes \times nodes$. We derive the following templates

$$DP_{CE_{2PA_2}}.1 \mathrel{\widehat{=}} [CE_{2PA_2} \mid dir\_dep\_on = \{\} \wedge \{x?, y?\} \subset nodes]$$

$$DP_{CE_{2PA_2}}.2 \mathrel{\widehat{=}} [CE_{2PA_2} \mid dir\_dep\_on = \{\} \wedge \{x?, y?\} = nodes]$$

$$DP_{CE_{2PA_2}}.3 \mathrel{\widehat{=}} [CE_{2PA_2} \mid dir\_dep\_on = \{\} \wedge \{x?, y?\} \subset nodes]$$

$$DP_{CE_{2PA_2}}.4 \mathrel{\widehat{=}} [CE_{2PA_2} \mid dir\_dep\_on = \{\} \wedge \{x?, y?\} = nodes]$$

$$DP_{CE_{2PA_2}}.5 \mathrel{\widehat{=}} [CE_{2PA_2} \mid dir\_dep\_on \neq \{\} \wedge dir\_dep\_on \subset nodes \times nodes \wedge \\ \mathrm{dom}\, dir\_dep\_on \cap \mathrm{ran}\, dir\_dep\_on = \{\} \wedge \\ \{x?, y?\} \subset nodes]$$

$$DP_{CE_{2PA_2}}.7 \mathrel{\widehat{=}} [CE_{2PA_2} \mid dir\_dep\_on \neq \{\} \wedge dir\_dep\_on \subset nodes \times nodes \wedge \\ dir\_dep\_on \subset dir\_dep\_on^+ \wedge \{x?, y?\} \subset nodes]$$

As in the templates derived from CE_1, the third and fourth templates are equivalent to the first and second respectively, and are ignored.

$$\{DP_{CE_{2PA_2}}.1, DP_{CE_{2PA_2}}.2, DP_{CE_{2PA_2}}.5, DP_{CE_{2PA_2}}.7\} = \\ TTH_{CanAdd}(CE_{2PA_2}, dom\_prop)$$

The domains propagated for the transitive closure operator contain further sub-operators for which domains can be propagated, namely set intersection ($\cap$) in case 2(a) and subset ($\subset$) in case 2(c). However, none of these propagations add anything to our test set. Case 2(a) restricts the intersection of the domain and range of dir_dep_on to be empty, so we could only propagate the first and fourth domain propagations of intersection. The only factor of interest in these propagations is whether or not dir_dep_on is empty, which is regulated by other constraints. Thus adding these propagations either doesn't change the template or results in a contradiction. For case 2(c), the only interesting propagation is whether dir_dep_on is empty, and again, this is already regulated by previous propagations.

We can carry on the testing by using some further 'common-sense' techniques to ensure various sizes of sets and relations are tested, and that all permutations of inputs are tested. However, the purpose of this example is to demonstrate the new strategies introduced in the previous chapter, so we do not present this aspect of testing the DMS. This testing can be found in [SC92].

Specification mutation

We now apply the specification mutation strategy.

$$spec\_mut : STRATEGY$$

The mutants are constructed by substituting or swapping types and variables, and applying the various mutations shown in appendix D to the specification. We work from this fully expanded specification of $CanAdd$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_\ CanAdd\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta DepManSys \\
x?, y? : X \\
result! : \mathbf{Bool} \\
\hline
dir\_dep\_on \subseteq nodes \times nodes \\
(\_ \succ \_) = dir\_dep\_on^{+} \\
\neg\, (\exists\, x : X \bullet x \succ x) \\
\{x?, y?\} \subseteq nodes \\
result! = True \Leftrightarrow (\neg\, (y? \succ x?) \land x? \neq y?) \\
\hline
\end{array}
$$

Applying all the possible mutations generates many mutants. Many mutants are either incorrectly formed specifications (detectable by a type checker), or equivalent to the original specification. In these cases, the mutants are not shown. We also do not show type 1 mutants, and do not derive tests for type 3 mutants.

Type mutation.

No type mutations that are reasonable under the competent specifier hypothesis produce valid specifications for this example.

Variable mutation.

Similarly to type mutation, many variable mutations result in invalid specifications. Most of the valid mutants are equivalent to the original specification, or result in type 1 mutants, not requiring special cases. The only variable mutants of note result from swapping dir_dep_on for $(\_ \succ \_)$ and vice versa. However, all the valid mutants using these mutations result in equivalent mutants to those derived from unary relational operator mutation. This is because the predicate $(\_ \succ \_) = dir\_dep\_on^{+}$, which defines $(\_ \succ \_)$, means that the distinguishing cases are the same as for distinguishing dir_dep_on from $dir\_dep\_on^{+}$. The unary relational operator mutants are shown with other operator mutants below.

Operator mutation.

We use the following notation to name a mutation template:

$$M_{<label>:[<mutant\ operator>/<original\ operator>]}$$

That is, the subscript is first some label—usually the predicate number as used in previous examples—followed by a statement of the mutant operator that is being used in the stead of the original operator. In the case where an operator is being introduced where none was previously, or an operator is being omitted, a blank is used.

The operator mutations below do not include unary relational operator mutants where an operator is introduced where there was none originally. Most of these are type 3 mutants and must be distinguished by the specifier. The others result in two sets of equivalent tests, distinguishing the relations dir_dep_on and $(\_ \succ \_)$ from their inverses. Thus, we have these mutation test templates:

$$M_{dir\_dep\_on:[\sim/]} \mathrel{\widehat{=}} [VIS_{CanAdd} \mid \mathrm{dom}\ dir\_dep\_on \neq \mathrm{ran}\ dir\_dep\_on]$$
$$M_{(\_\succ\_):[\sim/]} \mathrel{\widehat{=}} [VIS_{CanAdd} \mid \mathrm{dom}(\_ \succ \_) \neq \mathrm{ran}(\_ \succ \_)]$$

The rest of the operator mutants are dealt with on a line by line basis.

predicate 1: $dir\_dep\_on \subseteq nodes \times nodes$

mutant: $dir\_dep\_on = nodes \times nodes$
distinguished by: $dir\_dep\_on \subset nodes \times nodes$

$$M_{1:[=/\subseteq]} \mathrel{\widehat{=}} [VIS_{CanAdd} \mid dir\_dep\_on \subset nodes \times nodes]$$

mutant: $dir\_dep\_on \neq nodes \times nodes$
distinguished by: $dir\_dep\_on = nodes \times nodes$

Note that, as seen in the previous section detailing the domain propagation testing, this distinguishing test contradicts the valid input space. So, no test distinguishes this mutant from the original specification.

mutant: $dir\_dep\_on \subset nodes \times nodes$

distinguished by: $dir\_dep\_on = nodes \times nodes$

As above, this mutant cannot be detected by testing.

predicate 2: $dir\_dep\_on^+$

mutant: dir_dep_on

distinguished by: $\mathrm{dom}\ dir\_dep\_on \cap \mathrm{ran}\ dir\_dep\_on \neq \{\}$

$$M_{2:[/+]} \mathrel{\widehat=} [VIS_{CanAdd} \mid \mathrm{dom}\ dir\_dep\_on \cap \mathrm{ran}\ dir\_dep\_on \neq \{\}]$$

mutant: $dir\_dep\_on^\sim$

distinguished by: $(\mathrm{dom}\ dir\_dep\_on \cup \mathrm{ran}\ dir\_dep\_on) \setminus$
$(\mathrm{dom}\ dir\_dep\_on \cap \mathrm{ran}\ dir\_dep\_on) \neq \{\}$

$$M_{2:[\sim/+]} \mathrel{\widehat=} [VIS_{CanAdd} \mid (\mathrm{dom}\ dir\_dep\_on \cup \mathrm{ran}\ dir\_dep\_on) \setminus (\mathrm{dom}\ dir\_dep\_on \cap \mathrm{ran}\ dir\_dep\_on) \neq \{\}]$$

mutant: $dir\_dep\_on^*$

distinguished by: This is a type 3 mutant and cannot be distinguished by specification-based tests.

mutant: $dir\_dep\_on^k$

distinguished by: $dir\_dep\_on^2 \neq dir\_dep\_on^+$

$$M_{2:[^k/+]} \mathrel{\widehat=} [VIS_{CanAdd} \mid dir\_dep\_on^2 \neq dir\_dep\_on^+]$$

predicate 3: $\neg\ (\exists\, x : X \bullet x \succ x)$

All mutations of the mutable operators in this predicate ($\neg$, $\exists$) result in type 1 mutants, which are distinguished by any test.

predicate 4: $\{x?, y?\} \subseteq nodes$

mutant: $\{x?, y?\} = nodes$

distinguished by: $\{x?, y?\} \subset nodes$

$$M_{4:[=/\subseteq]} \mathrel{\widehat=} [VIS_{CanAdd} \mid \{x?, y?\} \subset nodes]$$

mutant: $\{x?, y?\} \neq nodes$

distinguished by: $\{x?, y?\} = nodes$

$$M_{4:[\neq/\subseteq]} \mathrel{\widehat=} [VIS_{CanAdd} \mid \{x?, y?\} = nodes]$$

mutant: $\{x?, y?\} \subset nodes$

distinguished by: $\{x?, y?\} = nodes$

$$M_{4:[\subset/\subseteq]} \mathrel{\widehat=} [VIS_{CanAdd} \mid \{x?, y?\} = nodes]$$

predicate 5: $result! = True \Leftrightarrow (\neg (y? \succ x?) \wedge x? \neq y?)$

We consider mutations of $\Leftrightarrow$ and $\wedge$. All the mutations of $\wedge$ are type 3 mutants and cannot be distinguished by tests. For the mutations of $\Leftrightarrow$ note that the first operand is $result! = True$. We do not have control over this in the input, so the mutants do not include expressions using this operand.

mutant: $result! = True \wedge (\neg (y? \succ x?) \wedge x? \neq y?)$

distinguished by: $result! \neq True \wedge (y? \succ x? \vee x? = y?)$

$$M_{5:[\wedge/\Leftrightarrow]} \mathrel{\widehat=} [VIS_{CanAdd} \mid y? \succ x? \vee x? = y?]$$

mutant: $result! = True \vee (\neg (y? \succ x?) \wedge x? \neq y?)$

distinguished by: $result! \neq True \vee (y? \succ x? \vee x? = y?)$

$$M_{5:[\vee/\Leftrightarrow]} \mathrel{\widehat=} [VIS_{CanAdd} \mid y? \succ x? \vee x? = y?]$$

mutant: $result! = True \Rightarrow (\neg (y? \succ x?) \wedge x? \neq y?)$

distinguished by: This is a type 3 mutant and cannot be distinguished by specification-based tests.

Some of these templates are equivalent; we only keep track of non-redundant templates in the hierarchy.

$$
\begin{aligned}
&\{M_{dir\_dep\_on:[\sim/]}, M_{(\_\succ\_):[\sim/]},\\
&M_{1:[=/\subseteq]},\\
&M_{2:[/+]}, M_{2:[\sim/+]}, M_{2:[^k/+]},\\
&M_{4:[=/\subseteq]}, M_{4:[\neq/\subseteq]},\\
&M_{5:[\wedge/\Leftrightarrow]}\}\\
&= TTH_{CanAdd}(VIS_{CanAdd}, spec\_mut)
\end{aligned}
$$

Template summary

Overall, we have twelve templates derived using domain propagation in combination with other partitioning strategies, and thirteen mutation templates. Figure 6.5 shows a diagrammatic representation of the derived tests templates and their relationships.

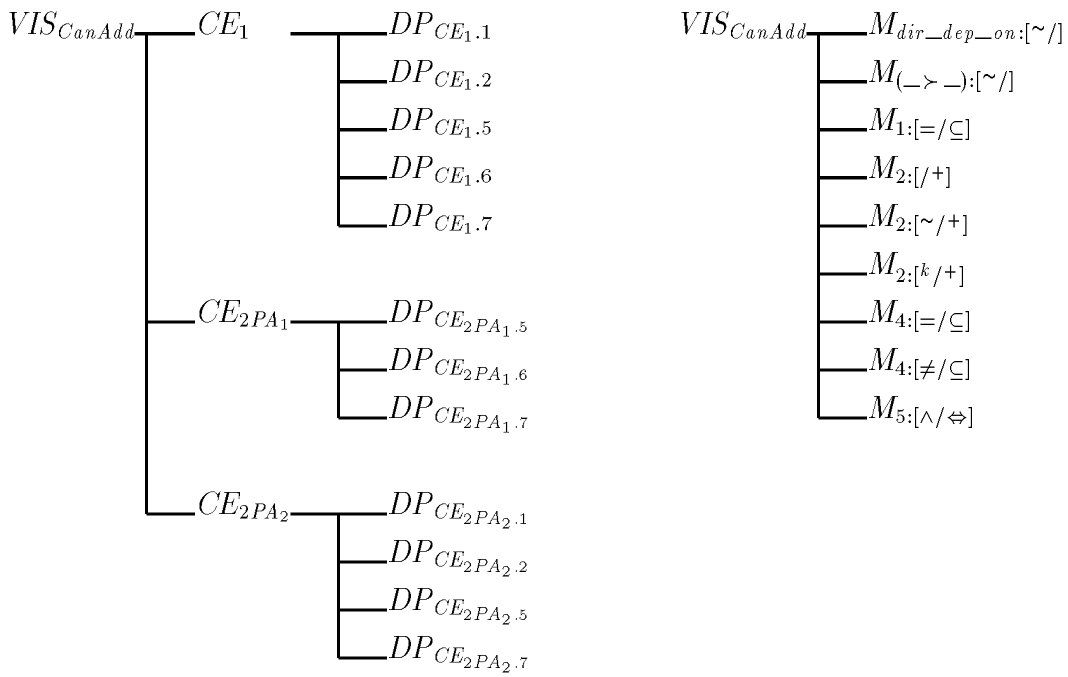


Figure 6.5: Test templates for the DMS *CanAdd* operation.

The number of domain partitions hidden by sub-operators, but revealed using domain propagation is intuitively pleasing. Again, we expect that many of the criteria set by the mutation templates will be met by the domain propagation suite. This can be used to assess the completeness of the strategies used. Further discussion on this aspect of specification mutation is deferred to chapter 7.

Instances

As in the *Read* example, instance templates for *CanAdd* are only precise descriptions of the abstract state because of the complex data types and state components of the input. An example instance template (introducing values of type X as appropriate is

$$A, B, C : X$$

$$T_{CE_1.1} \mathrel{\widehat=} [DP_{CE_1.1} \mid dir\_dep\_on = \{\}; nodes = \{A, B, C\}; x? = A; y? = B]$$

Again, to make actual tests we need to be able to construct complex data types and set the state of the system. Because the DMS case study is a more complete case study and was carried through to an implementation, we have the means to make actual tests. This is discussed in chapter 7.

Oracles

The output space of *CanAdd* is

$$OS_{CanAdd}[X] \mathrel{\widehat=} [DepManSys'[X]; result! : \mathbf{Bool}]$$

and the generic expression for oracle templates for *CanAdd* is

$$oracle_{CanAdd}(T) = (CanAdd \wedge T) \upharpoonright OS_{CanAdd}$$

The *CanAdd* operation has fairly simple oracles, since the state is never changed and there is only one output parameter which can only take one of two values. We can make the following statements about the oracles for *CanAdd*:

$$\begin{array}{l} \forall\, T : descendants_{CanAdd}(VIS_{CanAdd}) \bullet \\ \qquad State(oracle_{CanAdd}(T)) == State(T)' \end{array}$$

$$\begin{array}{l} \forall\, T : descendants_{CanAdd}(CE_1) \bullet \\ \qquad Params(oracle_{CanAdd}(T)) \mathrel{\widehat=} [result! : \mathbf{Bool} \mid result! = True] \end{array}$$

$$\forall\, T : descendants_{CanAdd}(CE_{2PA_1}) \cup descendants_{CanAdd}(CE_{2PA_2}) \bullet$$
$$Params(oracle_{CanAdd}(T)) \mathrel{\widehat=} [result! : \mathbf{Bool} \mid result! = False]$$

6.4 Discussion

The purpose of this chapter is to demonstrate how the framework is used to define tests, and how strategies are adapted to the framework. Using Z to provide concise, formal definitions of (abstract) test suites is successful. It is a flexible and expressive notation for defining the aspects of a test suite such as tests, oracles, and derivation histories. The definition of oracles shows this especially. Test strategies are easily applied to derive tests from the specification, either using standard practices such as boundary analysis, or strategies such as specification mutation that use knowledge of the specification language. We see that some strategies adapt easily to use at this abstract level, while others, such as domain testing, require careful consideration before they can be used. It is clear, however, that the skill of the tester cannot be replaced. An intuition guiding application of various strategies simplifies the testing process and helps cut short exploration of unfruitful avenues. Even in the more algorithmic strategies like domain propagation and specification mutation, the ability of the tester to determine when nothing will be gained by further application of the strategy is beneficial. It is not the place of a framework to direct a skilled user.

Discussion arising from these examples is deferred to the next chapter. It shows the effects of maintenance changes to the triangle study, reification in the file read study, and full derivation of test suites involving test planning and instantiation for the DMS study. At this point, we note that the new strategies are easy to apply, and are quite useful for uncovering input domains or special cases that might otherwise go unnoticed.

Chapter 7

The framework in the larger picture

With the framework we derive abstract descriptions of test suites as hierarchies of test cases derived using various strategies. This chapter examines the place of these abstract test suites in the larger picture of software testing and software development. A major consideration is how to construct or derive concrete test suites given the abstract test suites. This chapter also looks at how the formal basis of the framework assists in analysing the testing, and uses of the framework in other phases of the software lifecycle.

7.1 Constructing actual test suites

The framework generates abstract test hierarchies which are descriptions of test suites. At some stage in testing we need to produce actual tests that can be executed and evaluated. However, the important information in a test is embodied in its abstract specification, and a strength of the framework is the recognition of this. The following sections discuss techniques for deriving and constructing actual tests from our abstract test specifications.

7.1.1 Test planning/operation sequencing

Testing requires us to provide input to operations and check output from operations. There are two components in the input and the output. The first is the parameter component; for the input, this consists of the input variables or arguments, for the output this consists of the results. The second, and often ignored, component of both input and output is the state component. Operations often act on state information, and even if there is no change of state, the state components of the operation are still important and must be considered. To test individual operations we need to be able to set the inputs and check the outputs. At best, for the parameter components, this is a matter of providing some arguments and looking at the results. However, we may not have such direct access to, and control over, the system state. There are two courses of action available.

From a theoretical viewpoint, the ideal course is to have a way to set and check state components, say with operations external to the system, thus enabling complete access to the system state. Such operations need to be error-free, which may involve substantial extra effort in their development. In many cases, these operations may be almost identical to existing system operations.

The other, more practical, course is to use existing operations to set and check the state as much as possible. Operation functions and dependencies need to be considered to determine the order in which operations can be tested. The constructor/modifier/monitor classification of operations found in algebraic specifications is useful here. Constructor functions are used to build states, and monitor functions are used to examine states. Hence, to test an operation, the necessary constructor and monitor functions must be tested first.

Test planning in the DMS case study

Consider the Dependency Management System (DMS) case study from chapter 6. We need to examine the entire specification, though not in detail. The DMS has fifteen operations: *NoNodes, IsNode, AddNode, RemoveNode, NoDependencies, DependedUpon, IsDependency, CanAdd, AddDependence, RemoveDependence, Dependents, Supporters, UltSupporters, CandidateSupporters, SomeDirectDependent.*

Most of these, hopefully all the ones we will need, are self-explanatory, but their Z specifications can be found in appendix F. The DMS operations already include a number of operations which set and check state components. The DMS constructive operations would be virtually identical to any new operation defined to set the system state. The DMS checking operations take the form of interrogative operations: 'is x in the state?' Here, operations that display the whole state would be more useful, however such operations are unavailable so we make do with DMS operations to set and check the input and output during testing.

We need a test plan showing the operation dependencies so that the order of testing can be determined. For example, *AddDependence* cannot be tested until *AddNode* is shown to be correct so that nodes can be added to the system. The dependencies between other operations are not quite so straightforward, however, particularly when the operations needed to check the state are included. Figure 7.1 shows the test dependency graph for the DMS operations.

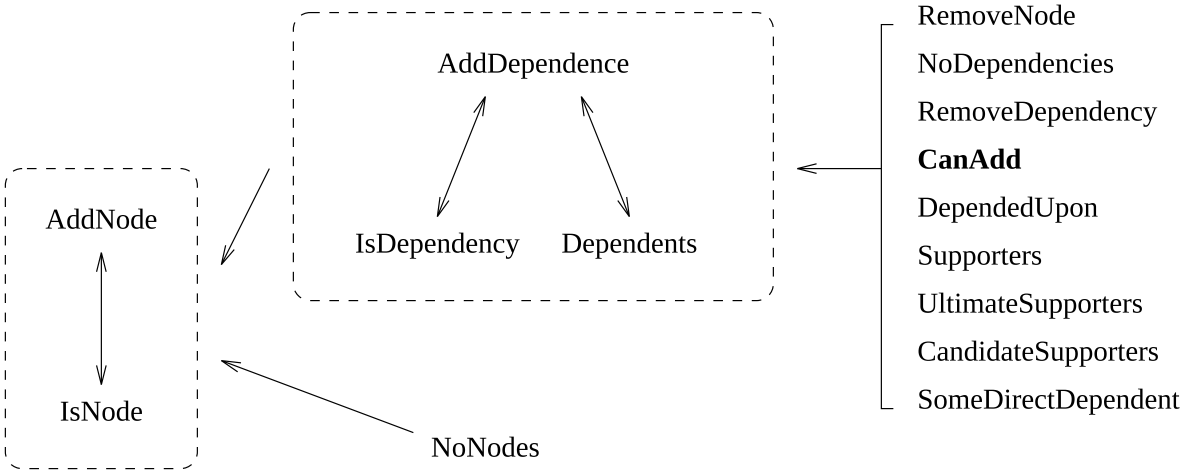


Figure 7.1: DMS operation test dependencies

Unfortunately there are operations which depend on each other. *AddNode* requires *IsNode* to check whether it has worked, whereas *IsNode* requires *AddNode* to generate a positive result case. A similar dependence exists between *AddDependence* and *IsDependency*, however *AddDependence* also requires *Dependents* to check that transitive dependencies are correctly inferred.

Much of the test dependency graph is derivable from the formal specification, by examining each operation and looking for simpler operations that affect relevant state variables. It is also helpful to have a notion of what operations are constructors and monitors in the algebraic specification sense. For example, testing all possibilities of *CanAdd* requires *nodes* and *dir_dep_on* to be non-empty, so we can see it depends on *AddNode* and *AddDependence*. Figure 7.1 shows that *AddNode* and *IsNode* must be tested first. Then *AddDependence, IsDependency*, and *Dependents* must be tested. Once these operations are validated we can test the operations on the right-hand side of figure 7.1 independently of each other and with confidence.

In our testing of *CanAdd*, we assume that the constructor operations *AddNode* and *AddDependence* work correctly, and that the monitor operations *IsNode*, *IsDependency*, and *Dependents* work correctly. *Dependents* lists all the nodes dependent on some other node and is used to check any transitive dependencies. This is not a complete check of the output state but is the best that can be done with the DMS operations. The parameter output of *CanAdd* can be checked externally. In this example, an even better checking operation would be one that determined whether or not there was any change in state caused by the operation under test.

Constructing actual tests from the DMS test plan

Given a test plan, and assuming that the required constructor and monitor operations have been tested, we can construct the actual test cases for the other operations. We show how the tests for *CanAdd* were developed.

The first step is to create the abstract operation sequences required for the tests. Given the DMS operations, these abstract operations sequences are generated algorithmically:

1. A call to *AddNode* is generated for every element of *nodes*.
2. A call to *AddDependence* is generated for every element of *dir_dep_on*.
3. A call to *CanAdd* is generated using $x?$ and $y?$ as arguments.
4. Each element of *nodes* is checked afterwards using *IsNode*.
5. Each element of *dir_dep_on* is checked afterwards using *IsDependency*.

6. Any elements of $(\_ \succ \_)$ that are not also in dir_dep_on are checked using *Dependents*, because *IsDependency* only displays direct dependencies, and so does not display any transitive dependencies.

The construction is based on the definition of the state in the test template; the checking is based on the definition of the state in the oracle template. Similarly, the arguments are determined by the test template and the expected results from the oracle template.

For example, consider the instance template derived in chapter 6

$$T_{CE_1.1} \mathrel{\widehat{=}} [DP_{CE_1.1} \mid dir\_dep\_on = \{\}; nodes = \{A, B, C\}; x? = A; y? = B]$$

Using the algorithm above, we can convert this abstract description into the following operation sequence.

$AddNode[A/x?]$
$AddNode[B/x?]$
$AddNode[C/x?]$
$CanAdd[A, B/x?, y?]$
$IsNode[A/x?]$
$IsNode[B/x?]$
$IsNode[C/x?]$

This constructs the state required for the test, calls *CanAdd*, and examines the state afterwards to see that all the nodes are still there. Note, that other nodes could be introduced erroneously and not detected—this is the limitation of monitor operations that do not display the whole state.

Abstract operation sequences for tests can be converted into concrete operation sequences, and hence actual test cases, when details about the implementation and test drivers are known.

Test drivers for the DMS prototypes were developed, with a common interface. We show how the tests were developed for these drivers. The test drivers for the various prototypes use the following interface. Integers represent nodes. Operations are invoked by a two letter abbreviation: *NoNodes* (`nn`), *IsNode* (`in`), *AddNode* (`an`), *RemoveNode* (`rn`), *NoDependencies* (`nd`), *DependedUpon* (`du`), *IsDependency* (`id`), *CanAdd* (`ca`), *AddDependence* (`ad`), *RemoveDependence* (`rd`), *Dependents* (`dp`), *Supporters* (`sp`), *UltSupporters* (`us`), *CandidateSupporters* (`cs`), and *SomeDirectDependent* (`sd`). Operations are invoked using these codes followed by any parameters separated by spaces. For example

```
an 1
```

adds the node '1' to the node set.

We assume the test driver outputs the name of the operation call invoked, and displays any results. For example, the above call to *AddNode* would result in the response

```
AddNode(1)
```

whereas a call to *CanAdd* might result in

```
CanAdd(1 2) true
```

This can lead to potential problems with regard to how the test driver orders output from operations. We assume output is displayed in ascending order of nodes.

From the abstract operation sequences of the tests and the interface to a test driver, we can develop actual test files. We can convert the abstract operation sequence derived for template $T_{CE_1.1}$ into the following test file, remembering that nodes are identified by integers. The file shows inputs on the left of the '@' symbol and expected output on the right of the '@' symbol.

```
an 1    @ AddNode(1)
an 2    @ AddNode(2)
an 3    @ AddNode(3)
ca 1 2  @ CanAdd(1 2) true
in 1    @ IsNode(1) true
in 2    @ IsNode(2) true
in 3    @ IsNode(3) true
```

Translating an abstract operation sequence into an actual test is a straightforward process, but depends on information not available at the specification stage of development.

7.1.2 Reification and structural testing

Reification (also called refinement) is the rigorous transformation of abstract specifications into concrete implementations. The aim of a reifying transformation is to derive or construct a more concrete specification that is at least as good as the original specification. An implementation can be viewed as a very concrete specification that happens to be executable. Transforming an abstract specification into a final implementation in one step is very difficult. Usually, a series of specifications are developed leading to a final implementation.

The test templates in our framework are as abstract as the specification. There are many possible implementations of a specification, and correspondingly there are many concrete representations of the abstract test information. If the specification is reified to an implementation, corresponding reifications can be made to the test templates to describe (more concretely) the test data and test information for the reified specification. There are two reasons this reification is important in connection with our framework.

Firstly, it provides us with an approach to transforming our abstract test specifications into actual tests. This means we can work comfortably at the abstract level, which simplifies our testing task.

Secondly, at each reification stage, more detail is introduced, thus more structural information about the final implementation is known. We can add to our test set accordingly, taking into account new variables and types, branches, paths, etc., in the specification. This allows incremental development of white-box test cases, which is easier than selecting white-box tests based on the whole implementation, and ensures that the structural tests are developed in an appropriate context.

Though there are many formal approaches to reification, we do not use them here. Our treatment of reification of test suites is not complete. This section is intended to introduce the concept and some of the basic formalisms involved, and mainly to whet our appetites for future research in this area. As such, we use the fundamental model of reification used in [Spi92]: between a specification (at the abstract level) and a reification of the specification (at the concrete level, though not necessarily the final implementation) there exists an abstraction relation, say Abs.

Reified test templates are derived from their abstract counterparts by conjoining the abstract template and the abstraction relation, and then hiding the components of the abstract data representation. To illustrate these ideas, we conduct a data reification on the file read example of chapter 6 (section 6.2). The original specification is reproduced in figure 7.2.

Consider implementing the data type $File$ by a combination of an array of bytes (represented as a mapping from $\mathbb{N}$ to $BYTE$) and a size variable representing the size of the array:

$$
\begin{array}{|l}
\multicolumn{1}{l}{File^R} \\
\hline
elts : \mathbb{N} \mathrel{+\!\!\!\twoheadrightarrow} BYTE \\
size : \mathbb{N} \\
\hline
size \leq MaxFileSize \\
\hline
\end{array}
$$

The relation between abstract and concrete files in this case is

$$
\begin{array}{|l}
\multicolumn{1}{l}{Abs} \\
\hline
File \\
File^R \\
\hline
\#file = size \\
\forall\, i : \operatorname{dom} file \bullet file(i) = elts(i) \\
\hline
\end{array}
$$

To see how this reification can be applied to the test templates, consider $BA_{1.1}$ derived in section 6.2.3 of chapter 6:

$$BA_{1.1} \mathrel{\widehat{=}} [PA_1 \mid \#file = MaxFileSize \land len? = \#file]$$

Using Abs, the template reified from the abstract template $BA_{1.1}$ (also represented by the superscript R on the name) is

$[BYTE]$

$$MaxFileSize : \mathbb{N}$$
$$MaxFileSize > 0$$

File

$$file : \operatorname{seq} BYTE$$
$$\#file \leq MaxFileSize$$

$$ReadStatus ::= ok \mid file\_empty \mid file\_too\_short$$

Read

$$file, file' : \operatorname{seq} BYTE$$
$$len? : \mathbb{N}$$
$$data! : \operatorname{seq} BYTE$$
$$stat! : ReadStatus$$

$$\#file \leq MaxFileSize$$
$$\#file' \leq MaxFileSize$$
$$file' = file$$
$$((\#file > 0 \land len? \leq \#file \land data! = (1 \mathrel{..} len?) \lhd file \land stat! = ok) \lor$$
$$(\#file > 0 \land len? > \#file \land data! = \langle\rangle \land stat! = file\_too\_short) \lor$$
$$(\#file = 0 \land data! = \langle\rangle \land stat! = file\_empty))$$

Figure 7.2: Z specification of simplified read operation from chapter 6.

$$
\begin{tikzcd}
\text{SPEC.} & \text{TT} & \text{ORACLE} \\
Read \arrow[r] \arrow[d, "\begin{array}{c}\text{reify}\\ \text{(using } Abs\text{)}\end{array}"'] & BA_{1.1} \arrow[r] \arrow[d] & oracle_{CanAdd}(BA_{1.1}) \arrow[d] \\
Read^R \arrow[r, "\begin{array}{c}\text{derive}\\ \text{template}\end{array}"'] & \begin{array}{l} BA^R_{1.1} \\ \Leftrightarrow \\ Read^R\_BA_{1.1} \end{array} \arrow[r, "\begin{array}{c}\text{derive}\\ \text{oracle}\end{array}"'] & \begin{array}{l} oracle_{CanAdd}(BA_{1.1})^R \\ \Leftrightarrow \\ oracle_{CanAdd}(Read^R\_BA_{1.1}) \end{array}
\end{tikzcd}
$$

Figure 7.3: Graphical relationship between specifications, test templates, and reifications.

$$BA^R_{1.1} \mathrel{\widehat{=}} (BA_{1.1} \land Abs) \setminus (file)$$

That is,

$$\begin{array}{l} BA^R_{1.1} \mathrel{\widehat{=}} \\ \qquad [elts : \mathbb{N} \mathrel{+\!\!\!\!\to\!\!\!\!\to} BYTE;\ size : \mathbb{N};\ len? : \mathbb{N} \mid size = MaxFileSize \land len? = size] \end{array}$$

If test derivation is conducted separately on the reified specification ($Read^R$), one of the templates produced using boundary analysis is

$$\begin{array}{l} Read^R\_BA_{1.1} \mathrel{\widehat{=}} \\ \qquad [elts : \mathbb{N} \mathrel{+\!\!\!\!\to\!\!\!\!\to} BYTE;\ size : \mathbb{N};\ len? : \mathbb{N} \mid size = MaxFileSize \land len? = size] \end{array}$$

As expected

$$BA^R_{1.1} \Leftrightarrow Read^R\_BA_{1.1}$$

The same is true of oracle templates. Graphically, these relationships are shown in figure 7.3.

Another simple example of reification demonstrates how more detailed information can require additional tests. At some point the type $\mathbb{N}$ will probably be replaced by the type integer ($\mathbb{Z}$). When this step is made, the implicit boundary $len? \geq 0$ must be made explicit.

The new valid input space derived after this reification is applied to *Read* is

$$VIS_{Read^{R2}} \mathrel{\widehat{=}} [file : \operatorname{seq} BYTE;\ len? : \mathbb{Z} \mid \#file \leq MaxFileSize \wedge len? \geq 0]$$

which adds a new domain boundary to the valid input space. More tests must be derived. For example, using domain testing, four more points must be derived:

$$\begin{aligned}
DB_{6.ON1} &\mathrel{\widehat{=}} [VIS_{Read^{R2}} \mid \#file = 0 \wedge len? = 0] \\
DB_{6.ON2} &\mathrel{\widehat{=}} [VIS_{Read^{R2}} \mid \#file = MaxFileSize \wedge len? = 0] \\
DB_{6.OFF1} &\mathrel{\widehat{=}} [VIS_{Read^{R2}} \mid \#file = MaxFileSize \ \mathsf{div}\ 2 \wedge len? = 1] \\
DB_{6.OFF2} &\mathrel{\widehat{=}} [VIS_{Read^{R2}} \mid \#file = MaxFileSize \ \mathsf{div}\ 2 \wedge len? = -1]
\end{aligned}$$

Template $DB_{6.OFF2}$ contradicts the valid input space and is ignored. Templates $DB_{6.ON1}$ and $DB_{6.ON2}$ are redundant, as they are equivalent to templates $DB_{3.ON1}$ and $DB_{1.ON1}$ respectively.

Reification offers a rigorous approach for transforming abstract test suites into concrete test suites, including systematic development of structural tests. Formal development methods often include some form of reification method. For example, VDM and RAISE offer guidelines and proof obligations for rigorous transformations of specifications in their design languages (VDM-SL and RSL, respectively) to executable code. The test template framework could easily use VDM-SL or RSL as the underlying description language instead of Z, meaning these reification methods could be directly applied to the abstract tests to develop concrete tests.

7.1.3 Notes on abstract test suites

These examples show two strengths of the framework's abstract test specifications. Firstly, there is a significant difference between how state and parameter input needs to be handled. The framework explicitly specifies which inputs are state components and which inputs are parameters. Secondly, we see that the final form of the tests is implementation dependent. The same specification could be implemented in many ways. Since the final tests are implementation dependent, the implementation strategy affects the test suites. However, the abstract specification of the tests can be used as the common starting point for testing regardless of the implementation strategy. In fact, converting the abstract tests into actual tests is at worst a well-structured process and at best a rigorous process.

7.2 Analysis

7.2.1 Test suites

Expressing the test data using a formal notation facilitates analysis of test sets. We consider how our definition of templates facilitates making assertions about, and imposing criteria, on the tests.

Properties of templates from strategies

Firstly, we expect tests derived using certain strategies to exhibit certain properties. We can add to our definitions of strategies by expressing formal relationships amongst the derived tests. Verification of these properties increases our confidence in the test set.

For example, we expect that input domains derived using some partitioning strategy do actually partition the input space for which they are derived. This can be formally expressed. Test templates derived from some space using a partitioning strategy must both cover the space and be disjoint to be a partition of the space. We define these relations on templates:

$$\_\,\underline{covers}\,\_ : \mathbb{P}\ TT_{Op} \leftrightarrow TT_{Op}$$

$$\forall\, SetofTT : \mathbb{P}\ TT_{Op};\ T : TT_{Op} \bullet SetofTT\ \underline{covers}\ T \Leftrightarrow \bigcup SetofTT = T$$

$$\_\,\underline{disjoint}\,\_ : TT_{Op} \leftrightarrow TT_{Op}$$

$$\forall\, T1, T2 : TT_{Op} \bullet T1\ \underline{disjoint}\ T2 \Leftrightarrow T1 \cap T2 = \{\}$$

So, we can assert that any templates derived using some partitioning strategy, say,

$$\mid \; partitioning : STRATEGY$$

must partition the template (or space) from which they are derived:

$$\forall\, Space : TT_{Op} \bullet TTH_{Op}(Space, partitioning) \;\underline{covers}\; Space$$

$$\forall\, Space : TT_{Op} \bullet$$
$$(\forall\, T1, T2 : TTH_{Op}(Space, partitioning) \mid T1 \neq T2 \bullet T1 \;\underline{disjoint}\; T2)$$

Consider the triangle classification case study from chapter 6. From the cause-effect template

$$CE_{iso} \mathrel{\widehat=} [VIS_{Classify} \mid (x?, y?, z?) \in ValidTriangle \wedge \#\{x?, y?, z?\} = 2]$$

These partition templates were derived

$$PA_{iso.1} \mathrel{\widehat=} [CE_{iso} \mid z? \neq 0 \wedge x? = y? \wedge z? \neq x? \wedge z? < x? + y?]$$
$$PA_{iso.2} \mathrel{\widehat=} [CE_{iso} \mid y? \neq 0 \wedge x? = z? \wedge y? \neq z? \wedge y? < x? + z?]$$
$$PA_{iso.3} \mathrel{\widehat=} [CE_{iso} \mid x? \neq 0 \wedge y? = z? \wedge x? \neq y? \wedge x? < y? + z?]$$

$$\{PA_{iso.1}, PA_{iso.2}, PA_{iso.3}\} = TTH_{Classify}(CE_{iso}, DNF\_partition)$$

To check that the DNF partitioning templates do indeed partition the cause-effect template, we need to check the following properties

$$TTH_{Classify}(CE_{iso}, DNF\_partition) \;\underline{covers}\; CE_{iso}$$

$$\forall\, T1, T2 : TTH_{Classify}(CE_{iso}, DNF\_partition) \mid T1 \neq T2 \bullet T1 \;\underline{disjoint}\; T2$$

That is, we need to show that

$$\bigcup\{PA_{iso.1}, PA_{iso.2}, PA_{iso.3}\} = CE_{iso} \wedge$$
$$PA_{iso.1} \cap PA_{iso.2} = \{\} \wedge$$
$$PA_{iso.1} \cap PA_{iso.3} = \{\} \wedge$$
$$PA_{iso.2} \cap PA_{iso.3} = \{\}$$

Required to show:

$$PA_{iso.1} \vee PA_{iso.2} \vee PA_{iso.3} = CE_{iso}$$

[Union of sets defined by schemas is equivalent to schema disjunction if signatures are the same.]
That is, show

$$\begin{array}{l}
z? \neq 0 \wedge x? = y? \wedge z? \neq x? \wedge z? < x? + y? \vee \\
y? \neq 0 \wedge x? = z? \wedge y? \neq z? \wedge y? < x? + z? \vee \\
x? \neq 0 \wedge y? = z? \wedge x? \neq y? \wedge x? < y? + z? \\
\Leftrightarrow \\
(x?, y?, z?) \in \mathit{ValidTriangle} \wedge \#\{x?, y?, z?\} = 2
\end{array}$$

[Expand.]

$$\begin{array}{l}
z? \neq 0 \wedge x? = y? \wedge z? \neq x? \wedge z? < x? + y? \vee \\
y? \neq 0 \wedge x? = z? \wedge y? \neq z? \wedge y? < x? + z? \vee \\
x? \neq 0 \wedge y? = z? \wedge x? \neq y? \wedge x? < y? + z? \\
\Leftrightarrow \\
x? < y? + z? \wedge y? < x? + z? \wedge z? < x? + y? \wedge \\
(x? = y? \wedge z? \neq x? \vee x? = z? \wedge y? \neq z? \vee y? = z? \wedge x? \neq y?)
\end{array}$$

[Distribute.]

$$\begin{array}{l}
z? \neq 0 \wedge x? = y? \wedge z? \neq x? \wedge z? < x? + y? \vee \\
y? \neq 0 \wedge x? = z? \wedge y? \neq z? \wedge y? < x? + z? \vee \\
x? \neq 0 \wedge y? = z? \wedge x? \neq y? \wedge x? < y? + z? \\
\Leftrightarrow \\
x? < y? + z? \wedge y? < x? + z? \wedge z? < x? + y? \wedge x? = y? \wedge z? \neq x? \vee \\
x? < y? + z? \wedge y? < x? + z? \wedge z? < x? + y? \wedge x? = z? \wedge y? \neq z? \vee \\
x? < y? + z? \wedge y? < x? + z? \wedge z? < x? + y? \wedge y? = z? \wedge x? \neq y?
\end{array}$$

[Simplify, $a < b + c \wedge a = b \Rightarrow c > 0$, $c \in \mathbb{N} \wedge c > 0 \Rightarrow c \neq 0$.]

$$\begin{array}{l}
z? \neq 0 \wedge x? = y? \wedge z? \neq x? \wedge z? < x? + y? \vee \\
y? \neq 0 \wedge x? = z? \wedge y? \neq z? \wedge y? < x? + z? \vee \\
x? \neq 0 \wedge y? = z? \wedge x? \neq y? \wedge x? < y? + z? \\
\Leftrightarrow \\
z? \neq 0 \wedge z? \neq 0 \wedge z? < x? + y? \wedge x? = y? \wedge z? \neq x? \vee \\
y? \neq 0 \wedge y? < x? + z? \wedge y? \neq 0 \wedge x? = z? \wedge y? \neq z? \vee \\
x? < y? + z? \wedge x? \neq 0 \wedge x? \neq 0 \wedge y? = z? \wedge x? \neq y?
\end{array}$$

[True, $p \Leftrightarrow p$.]

Figure 7.4: Coverage proof for partition templates of CE_{iso}

These proofs are quite straightforward. Mutual exclusion is easy to show as each DNF partition template has a predicate that contradicts the others. The proof of coverage is only slightly more complicated and is shown in figure 7.4.

These properties can be used to analyse other strategies. Consider cause-effect testing. Certainly, cause-effect templates should be disjoint, unless the operation is intended to be non-deterministic. If the templates derived using cause-effect mapping do not partition the valid input space, it means that there are causes with no effects. That is, that the operation is under-specified. Again, this could be intentional, but if it is not, we should check that our templates cover the input space. Since we believe *Classify* to be deterministic and fully specified, we desire the following properties to hold.

$$TTH_{Classify}(VIS_{Classify}, cause\_effect)\ \underline{covers}\ VIS_{Classify}$$

$$\forall\, T1, T2 : TTH_{Classify}(VIS_{Classify}, cause\_effect) \mid T1 \neq T2 \bullet T1\ \underline{disjoint}\ T2$$

That is, for coverage we desire

$$\begin{array}{l}
[VIS_{Classify} \mid (x?, y?, z?) \in ValidTriangle \wedge \#\{x?, y?, z?\} = 1] \vee \\
[VIS_{Classify} \mid (x?, y?, z?) \in ValidTriangle \wedge \#\{x?, y?, z?\} = 2] \vee \\
[VIS_{Classify} \mid (x?, y?, z?) \in ValidTriangle \wedge \#\{x?, y?, z?\} = 3] \vee \\
[VIS_{Classify} \mid (x?, y?, z?) \notin ValidTriangle] \\
= \\
VIS_{Classify}
\end{array}$$

and for mutual exclusion we desire

$$\begin{array}{l}
([VIS_{Classify} \mid (x?, y?, z?) \in ValidTriangle \wedge \#\{x?, y?, z?\} = 1] \wedge \\
[VIS_{Classify} \mid (x?, y?, z?) \in ValidTriangle \wedge \#\{x?, y?, z?\} = 2] \\
\qquad = [VIS_{Classify} \mid false]) \\
\wedge \\
\\
([VIS_{Classify} \mid (x?, y?, z?) \in ValidTriangle \wedge \#\{x?, y?, z?\} = 1] \wedge \\
[VIS_{Classify} \mid (x?, y?, z?) \in ValidTriangle \wedge \#\{x?, y?, z?\} = 3] \\
\qquad = [VIS_{Classify} \mid false]) \\
\wedge
\end{array}$$

$$([\mathit{VIS}_{\mathit{Classify}} \mid (x?, y?, z?) \in \mathit{ValidTriangle} \land \#\{x?, y?, z?\} = 1] \land$$
$$[\mathit{VIS}_{\mathit{Classify}} \mid (x?, y?, z?) \notin \mathit{ValidTriangle}]$$
$$= [\mathit{VIS}_{\mathit{Classify}} \mid \mathit{false}])$$
$$\land$$

$$([\mathit{VIS}_{\mathit{Classify}} \mid (x?, y?, z?) \in \mathit{ValidTriangle} \land \#\{x?, y?, z?\} = 2] \land$$
$$[\mathit{VIS}_{\mathit{Classify}} \mid (x?, y?, z?) \in \mathit{ValidTriangle} \land \#\{x?, y?, z?\} = 3]$$
$$= [\mathit{VIS}_{\mathit{Classify}} \mid \mathit{false}])$$
$$\land$$

$$([\mathit{VIS}_{\mathit{Classify}} \mid (x?, y?, z?) \in \mathit{ValidTriangle} \land \#\{x?, y?, z?\} = 2] \land$$
$$[\mathit{VIS}_{\mathit{Classify}} \mid (x?, y?, z?) \notin \mathit{ValidTriangle}]$$
$$= [\mathit{VIS}_{\mathit{Classify}} \mid \mathit{false}])$$
$$\land$$

$$([\mathit{VIS}_{\mathit{Classify}} \mid (x?, y?, z?) \in \mathit{ValidTriangle} \land \#\{x?, y?, z?\} = 3] \land$$
$$[\mathit{VIS}_{\mathit{Classify}} \mid (x?, y?, z?) \notin \mathit{ValidTriangle}]$$
$$= [\mathit{VIS}_{\mathit{Classify}} \mid \mathit{false}])$$

Both of these properties obviously hold, so our cause-effect tests partition the valid input space of *Classify*.

A notion of template equivalence is useful when using multiple strategies in test development, since some of the templates derived may be equivalent, and can thus be discarded. Schemas are equivalent when they describe the same collection of bindings. This can be represented in Z notation using the equality operator (=). In the file read example from chapter 6 we derived ten templates equivalent to others already derived. For example, for these templates

$$DB_{1.ON2} \mathrel{\widehat=} [\mathit{VIS}_{\mathit{Read}} \mid \#\mathit{file} = \mathit{MaxFileSize} \land \mathit{len}? = \mathit{MaxFileSize}]$$
$$BA_{1.1} \mathrel{\widehat=} [PA_1 \mid \#\mathit{file} = \mathit{MaxFileSize} \land \mathit{len}? = \#\mathit{file}]$$

$$DB_{1.ON2} = BA_{1.1}$$

because they both describe the same set of bindings. Note that syntactic equivalence is not required for templates to be equivalent.

Another potentially useful function describes the subset of a template not covered by its children:

$$\underline{notcovered} : TT_{Op} \longrightarrow TT_{Op}$$

$$\underline{notcovered} = (\lambda\, T : TT_{Op} \bullet T \setminus \bigcup(children_{Op}(T)))$$

This identifies regions of a domain for which tests are not derived, and can aid static checking of the application of strategies in test development. Not all domain subdivisions will enforce the entire input domain to be covered. For example, with an ideal, revealing domain-partitioning [WO80], where the input is partitioned into the set of all error-causing inputs and the set of all correct inputs, one need only consider the error-causing domains.

Checking these properties can be useful in detecting incorrect use of strategies when defining templates. Expression of such properties also helps increase understanding of strategies. It is possible to build a library of common properties of templates, such as the relations defined above, which can be used as desired to show properties of templates derived using certain strategies.

Adequacy criteria

Given that we can define various useful properties of test templates, can we also formally define adequacy criteria for test sets and check them? This is largely dependent on the particular adequacy criteria. Many criteria make statements about tests in terms of the results of executing them; we cannot define such criteria in the formalism of the framework. For example, validity and reliability as defined by Goodenough and Gerhart [GG75], depend on analysing the success of a test suite, where a test suite is defined to be successful if every test in the suite is passed. Because we cannot describe such phenomena of our abstract test suites using Z, we cannot define these criteria using the framework. Some criteria establish the relationship between tests and code (specification) exercised, which again we cannot define in the framework since we can't refer to code/specification elements. This is not to say that these criteria cannot be checked for tests derived using the framework, we just cannot make a formal statement of the criteria. However, some criteria are expressible. For example, [WJ91] lists a number of properties of partition testing (using random selection of tests within partitions) useful for comparing the test set with a collection of random tests. For example, Observation 4 is that if all partitions are of the same size and the same number of tests for each partition is chosen, then the test set is at least as good as a random test set. Since templates are Z schemas, and Z schemas are sets, we use the cardinality operator on sets (#) to represent the size of a template. Observation 4 may be expressed

$$
\begin{array}{l}
\forall T : TT_{Op} \bullet \\
\quad (\forall c1, c2 : children_{Op}(T) \bullet \\
\qquad \#c1 = \#c2 \land \\
\qquad \#(TTH_{Op}(c1, instantiation)) = \\
\qquad \#(TTH_{Op}(c2, instantiation)))
\end{array}
$$

This statement of the property is slightly dubious because Z's cardinality operator is only defined over finite sets. In most cases, templates will represent infinite sets (except for the instance templates, of course). However, if the templates are finite, they can be analysed by these observations.

Mutation analysis

The final form of analysis we discuss arises from our specification mutation strategy. Implementation mutation testing is an analysis technique for test suites. Mutant programs are generated and the test suite is given a mutation score based on the number of mutants detected by the tests. We can conduct the same analysis using our specification mutation technique. We consider the test derivation for the *CanAdd* operation of the DMS case study from chapter 6. Figure 7.5 shows the mutant suite required to detect the proposed specification mutants for *CanAdd*, which was derived in chapter 6, section 6.3.3. Figure 7.6 shows the test suite derived for *CanAdd* using domain propagation. It is this suite we analyse with regard to its mutant detecting ability. Figure 7.7 shows a table indicating which domain propagation templates satisfy the criteria set by the mutant templates.

$$M_{dir\_dep\_on:[\sim/]} \mathrel{\widehat=} [\, VIS_{Op} \mid \mathrm{dom}\, dir\_dep\_on \neq \mathrm{ran}\, dir\_dep\_on]$$

$$M_{(\_\succ\_):[\sim/]} \mathrel{\widehat=} [\, VIS_{Op} \mid \mathrm{dom}(\_ \succ \_) \neq \mathrm{ran}(\_ \succ \_)]$$

$$M_{1:[=/\subseteq]} \mathrel{\widehat=} [\, VIS_{Op} \mid dir\_dep\_on \subset nodes \times nodes]$$

$$M_{2:[/+]} \mathrel{\widehat=} [\, VIS_{Op} \mid \mathrm{dom}\, dir\_dep\_on \cap \mathrm{ran}\, dir\_dep\_on \neq \{\}]$$

$$\begin{aligned} M_{2:[\sim/+]} \mathrel{\widehat=} [\, VIS_{Op} \mid &(\mathrm{dom}\, dir\_dep\_on \cup \mathrm{ran}\, dir\_dep\_on) \setminus \\ &(\mathrm{dom}\, dir\_dep\_on \cap \mathrm{ran}\, dir\_dep\_on) \neq \{\}] \end{aligned}$$

$$M_{2:[^k/+]} \mathrel{\widehat=} [\, VIS_{Op} \mid dir\_dep\_on^2 \neq dir\_dep\_on^+]$$

$$M_{4:[=/\subseteq]} \mathrel{\widehat=} [\, VIS_{Op} \mid \{x?, y?\} \subset nodes]$$

$$M_{4:[\neq/\subseteq]} \mathrel{\widehat=} [\, VIS_{Op} \mid \{x?, y?\} = nodes]$$

$$M_{5:[\wedge/\Leftrightarrow]} \mathrel{\widehat=} [\, VIS_{Op} \mid y? \succ x? \vee x? = y?]$$

Figure 7.5: Specification mutation test suite for *CanAdd*.

We can see that the analysis seems favourable. Only one mutant is not detected by the domain propagation templates:

$$M_{2:[^k/+]} \mathrel{\widehat=} [\, VIS_{Op} \mid dir\_dep\_on^2 \neq dir\_dep\_on^+]$$

This test presents an input containing a transitive dependency from one node to another involving three direct dependency links. The domain propagation templates do not force this depth of transitivity. Naturally, we should include a special test to distinguish any mutants not detected by other tests.

7.2.2 Strategies

The test template framework provides common ground for comparing and contrasting testing strategies. For example, an aim of domain testing [WC80, CHR82] is to derive the fewest test points possible to test each boundary. We saw in the fileread example from chapter 6 that domain testing did, indeed, derive significantly fewer

$$DP_{CE_1}.1 \mathrel{\widehat=} [CE_1 \mid dir\_dep\_on = \{\} \land \{x?, y?\} \subset nodes]$$

$$DP_{CE_1}.2 \mathrel{\widehat=} [CE_1 \mid dir\_dep\_on = \{\} \land \{x?, y?\} = nodes]$$

$$\begin{aligned} DP_{CE_1}.5 \mathrel{\widehat=} [CE_1 \mid\ & dir\_dep\_on \neq \{\} \land dir\_dep\_on \subset nodes \times nodes \land \\ & \operatorname{dom} dir\_dep\_on \cap \operatorname{ran} dir\_dep\_on = \{\} \land \\ & \{x?, y?\} \subset nodes] \end{aligned}$$

$$\begin{aligned} DP_{CE_1}.6 \mathrel{\widehat=} [CE_1 \mid\ & dir\_dep\_on \neq \{\} \land dir\_dep\_on \subset nodes \times nodes \land \\ & \operatorname{dom} dir\_dep\_on \cap \operatorname{ran} dir\_dep\_on = \{\} \land \\ & \{x?, y?\} = nodes] \end{aligned}$$

$$\begin{aligned} DP_{CE_1}.7 \mathrel{\widehat=} [CE_1 \mid\ & dir\_dep\_on \neq \{\} \land dir\_dep\_on \subset nodes \times nodes \land \\ & dir\_dep\_on \subset dir\_dep\_on^+ \land \{x?, y?\} \subset nodes] \end{aligned}$$

$$\begin{aligned} DP_{CE_{2PA_1}}.5 \mathrel{\widehat=} [CE_{2PA_1} \mid\ & dir\_dep\_on \neq \{\} \land dir\_dep\_on \subset nodes \times nodes \land \\ & \operatorname{dom} dir\_dep\_on \cap \operatorname{ran} dir\_dep\_on = \{\} \land \\ & \{x?, y?\} \subset nodes] \end{aligned}$$

$$\begin{aligned} DP_{CE_{2PA_1}}.6 \mathrel{\widehat=} [CE_{2PA_1} \mid\ & dir\_dep\_on \neq \{\} \land dir\_dep\_on \subset nodes \times nodes \land \\ & \operatorname{dom} dir\_dep\_on \cap \operatorname{ran} dir\_dep\_on = \{\} \land \\ & \{x?, y?\} = nodes] \end{aligned}$$

$$\begin{aligned} DP_{CE_{2PA_1}}.7 \mathrel{\widehat=} [CE_{2PA_1} \mid\ & dir\_dep\_on \neq \{\} \land dir\_dep\_on \subset nodes \times nodes \land \\ & dir\_dep\_on \subset dir\_dep\_on^+ \land \{x?, y?\} \subset nodes] \end{aligned}$$

$$DP_{CE_{2PA_2}}.1 \mathrel{\widehat=} [CE_{2PA_2} \mid dir\_dep\_on = \{\} \land \{x?, y?\} \subset nodes]$$

$$DP_{CE_{2PA_2}}.2 \mathrel{\widehat=} [CE_{2PA_2} \mid dir\_dep\_on = \{\} \land \{x?, y?\} = nodes]$$

$$\begin{aligned} DP_{CE_{2PA_2}}.5 \mathrel{\widehat=} [CE_{2PA_2} \mid\ & dir\_dep\_on \neq \{\} \land dir\_dep\_on \subset nodes \times nodes \land \\ & \operatorname{dom} dir\_dep\_on \cap \operatorname{ran} dir\_dep\_on = \{\} \land \\ & \{x?, y?\} \subset nodes] \end{aligned}$$

$$\begin{aligned} DP_{CE_{2PA_2}}.7 \mathrel{\widehat=} [CE_{2PA_2} \mid\ & dir\_dep\_on \neq \{\} \land dir\_dep\_on \subset nodes \times nodes \land \\ & dir\_dep\_on \subset dir\_dep\_on^+ \land \{x?, y?\} \subset nodes] \end{aligned}$$

Figure 7.6: Domain propagation test suite for *CanAdd*.

Mutant template	Satisfied by
$M_{dir\_dep\_on:[\sim/]}$	$DP_{CE_1}.5$, $DP_{CE_1}.6$, $DP_{CE_2PA_1}.5$, $DP_{CE_2PA_1}.6$, $DP_{CE_2PA_2}.5$
$M_{(\_\succ\_):[\sim/]}$	$DP_{CE_1}.5$, $DP_{CE_1}.6$, $DP_{CE_2PA_1}.5$, $DP_{CE_2PA_1}.6$, $DP_{CE_2PA_2}.5$
$M_{1:[=/\subseteq]}$	$DP_{CE_1}.1$, $DP_{CE_1}.2$, $DP_{CE_1}.5$, $DP_{CE_1}.6$, $DP_{CE_1}.7$, $DP_{CE_2PA_1}.5$, $DP_{CE_2PA_1}.6$, $DP_{CE_2PA_1}.7$, $DP_{CE_2PA_2}.1$, $DP_{CE_2PA_2}.2$, $DP_{CE_2PA_2}.5$, $DP_{CE_2PA_2}.7$
$M_{2:[/+]}$	$DP_{CE_1}.7$, $DP_{CE_2PA_1}.7$, $DP_{CE_2PA_2}.7$
$M_{2:[\sim/+]}$	$DP_{CE_1}.5$, $DP_{CE_1}.6$, $DP_{CE_2PA_1}.5$, $DP_{CE_2PA_1}.6$, $DP_{CE_2PA_2}.5$
$M_{2:[^k/+]}$	
$M_{4:[=/\subseteq]}$	$DP_{CE_1}.1$, $DP_{CE_1}.5$, $DP_{CE_1}.7$, $DP_{CE_2PA_1}.5$, $DP_{CE_2PA_1}.7$, $DP_{CE_2PA_2}.1$, $DP_{CE_2PA_2}.5$, $DP_{CE_2PA_2}.7$
$M_{4:[\neq/\subseteq]}$	$DP_{CE_1}.2$, $DP_{CE_1}.6$, $DP_{CE_2PA_1}.6$, $DP_{CE_2PA_2}.2$
$M_{5:[\wedge/\Leftrightarrow]}$	$DP_{CE_2PA_1}.5$, $DP_{CE_2PA_1}.6$, $DP_{CE_2PA_1}.7$, $DP_{CE_2PA_2}.1$, $DP_{CE_2PA_2}.2$, $DP_{CE_2PA_2}.5$, $DP_{CE_2PA_2}.7$

Figure 7.7: Satisfaction of mutant template criteria by domain propagation templates.

tests than the other strategies, but we also saw that it is a more difficult strategy to apply, and it is not applicable in many cases.

It is interesting to note which test data are common in the derivations from different strategies, and whether any TTs not equivalent could satisfy criteria of other strategies. For example, instantiation templates derived using any strategy are also valid templates for random testing.

Templates derived independently using different strategies might have other, more general, relationships among them. One template might be a subset of another, or a collection of templates might partition another. If so, the strategies and templates should be examined to determine any redundancy or to consider using the strategies in conjunction.

Naturally, the heuristics can be compared based on error detection when run on implementations, which makes the common templates particularly interesting.

Specification mutation analysis also provides a means of comparing strategies based on their mutant detecting ability. As a simple example consider the cause-effect templates from the DMS case study in chapter 6, from which subsequent domain propagation templates were derived:

$$CE_1 \mathrel{\widehat{=}} [VIS_{Op} \mid \neg\,(y? \succ x?) \wedge x? \neq y?]$$
$$CE_{2PA_1} \mathrel{\widehat{=}} [VIS_{Op} \mid y? \succ x?]$$
$$CE_{2PA_2} \mathrel{\widehat{=}} [VIS_{Op} \mid x? = y?]$$

These templates alone only guarantee detection of one mutant from figure 7.5, by satisfying the requirements of $M_{5:[\wedge/\Leftrightarrow]}$. So, we see that extending our testing using domain propagation increases the number of mutant specifications distinguished from the original specification. Note that we are not analysing the test suite derived using specification mutation. The domain propagation test suite contains many tests that the mutation suite does not enforce. Mutation can be used for analysis because the idea of determining that certain errors are not in the specification gives some concrete basis for analysis.

7.2.3 New strategies

The ability to analyse test suites and testing strategies in the framework facilitates development of new specification-based testing strategies, particularly in the partitioning of the input space into domains. We saw this when failures of existing adapted strategies prompted the development of the domain partitioning and specification mutation strategies. We needed these strategies because the adapted strategies did not have enough focus on issues in abstract specifications. Any deficiency of a strategy due to any analysis (described above or not) can prompt additional developments to testing strategies.

7.3 Elsewhere in the software lifecycle

7.3.1 Specification validation

As mentioned in chapter 2, the fundamental limitation of specification-based testing is the correctness of the specification. Here we discuss how using our framework and testing strategies impacts on specification validation. In essence, these are side-effects of using the framework in test development. There is no formal basis with which to assess the possible advantages gained. These ideas are not meant to replace a disciplined and well-structured validation approach, if such exists. Nevertheless, we feel that any input in such a nebulous area as specification validation is useful, especially since most of the work has already been done during test derivation.

The first impact on validation comes from the test cases and test derivation. Test inputs and outputs are very similar to a state-based specification; they are a very simple prototype. Thus it is easier to spot potential discrepancies between the actual specification and what was intended. The act of deriving the tests brings these discrepancies to light. The test cases can also be used as a very primitive form of validation by presenting the user with simple scenarios: 'given this input you get this result, is that correct?'.

The second impact, which is somewhat more structured than the first, involves examining the input and output spaces of operations. The first, simple, check is to look at the valid input space of an operation. Any operation whose pre-condition is not true (i.e., whose valid input space does not equal its input space) is not defined over some input. This is always a potential source of error in specifications, indicating some cases of input may not have been fully considered. Such operations should be checked to see whether the excluded inputs are legitimately excluded. A second check is to construct the valid output space similarly to the valid input space. This summarises all the legal output from the operation and may help the specifier spot some inconsistencies between the requirements and the specification. For any operations whose valid input or output spaces do not equal their input or output spaces respectively, a third check is to consider what is not legal input or output of the operation by constructing the negations of the valid input space and valid output space[1]. Again, the hope is that this summary of invalid behaviour will help the specifier detect errors. This is especially effective if the specifier attempts to construct instances of invalid cases. Simple and common mistakes like off-by-one errors can remain hidden in large specifications that aren't well tested, and still be detected this way [SC91]. Part of the reason this is useful is that the invalid spaces present a different perspective on the specification and can often help the specifier overcome preconceptions and misconceptions if done carefully. Essentially this examination of input and output spaces is a focussed form of review.

The third impact is a result of specification mutation testing. Again, this is essentially a highly focussed form of review. Specification mutants are very similar specifications to the original that are postulated as being correct. Care must be taken in assuring that they are not correct. Specification mutation not only presents the specification in a different light, it proposes an alternative specification. Two aspects of specification mutation with a more formal basis in specification validation are type 3 mutants and other mutants whose distinguishing test leads to a contradiction. Type 3 mutants cannot be distinguished by tests drawn from the valid input space of the operation. Special care should be taken by the specifier or reviewer in determining that these nearly functionally equivalent mutants are not correct specifications. That is, the specifier's validation efforts are focussed on likely sources of specification error. It is also possible that the distinguishing condition for

[1] This negation is the set difference of the space and the valid space.

a mutant specification of another type leads to a contradiction. In this case, there is a mutant supposedly distinguishable by testing that is not distinguished by testing, which means a potential specification error. For example, consider the specification mutation of $CanAdd$ from chapter 6. A predicate from the original specification is

$$dir\_dep\_on \subseteq nodes \times nodes$$

For this mutant of the predicate

$$dir\_dep\_on \subset nodes \times nodes$$

the distinguishing predicate is

$$dir\_dep\_on = nodes \times nodes$$

which contradicts VIS_{CanAdd} as discussed in chapter 6. Therefore, this mutant cannot be distinguished by testing. The mutant represents a possibly correct specification, and since the contradiction lies with the state schema for the DMS many operations could be affected. Since the mutant is more constrained, the original specification should, perhaps, be changed. As it turned out for the DMS case study, not every operation's valid input space was contradicted by the distinguishing predicate above. In fact, some operations required it to be possible that dir_dep_on was equal to $nodes \times nodes$ for cases involving empty sets. So, the mutation only results in a correct specification for most of the DMS operations. One can imagine, however, that serious specification errors could be detected by the validation requirements suggested by specification mutation. In summary, specification mutation defines validation obligations for the specifier, i.e., situations for which the specifier is obliged to show that the original specification really specifies what is intended.

7.3.2 Testability and design

Freedman defines properties of software procedures that determine their 'testability' [Fre91]. These ideas, *controllability* and *observability*, are borrowed from established hardware verification processes. A controllable and observable procedure exhibits high testability. Essentially, a controllable and observable procedure defines an onto function, i.e., a function in the mathematical sense that maps input to its entire range.

In chapter 4, we saw simple examples of how different specification constructs, even equivalent specifications, affect the valid input spaces of operations and partition analysis of the valid input spaces. We argued that some specification styles are easier to derive tests for, and hence easier to test in some sense.

To what extent should testability be a concern during design? In simple cases like those from chapter 4, it seems reasonable to write specifications in a style that is easier to test. Testability by using onto functions is not so black-and-white an issue. Advantages of specifications are abstractness and non-determinism. Using only (onto) functions can compromise expressiveness in the specification. However, it may be reasonable to consider introducing testable elements during later design phases where more determinism is appropriate.

7.3.3 Maintenance

Here, we consider the benefits of the test template framework when (inevitably) the specification is changed to accommodate new user requirements. The test hierarchy structure and strong connection to the specification make it is easy to determine the extent of required changes to a test suite for a maintenance change. This is similar to introducing new tests during specification reification discussed above.

For example, consider the triangle case study from chapter 6. We postulate that the user becomes interested in knowing about right-angle triangles as well. The *TRIANGLE* type is revised to

$$\begin{aligned} \mathit{TRIANGLE} ::= {} & \mathit{EQUILATERAL} \mid \mathit{RA\_ISOSCELES} \mid \mathit{ISOSCELES} \mid \\ & \mathit{RA\_SCALENE} \mid \mathit{SCALENE} \mid \mathit{INVALID} \end{aligned}$$

A new relation is added for determining whether the input represents a right-angle triangle.

$$\begin{array}{|l}
RATriangle : \mathbb{P}(\mathbb{N} \times \mathbb{N} \times \mathbb{N}) \\
\hline
\forall x, y, z : \mathbb{N} \bullet \\
\quad x^2 = y^2 + z^2 \lor \\
\quad y^2 = x^2 + z^2 \lor \\
\quad z^2 = x^2 + y^2
\end{array}$$

The new specification for *ValidCase* is

$$\begin{array}{|l}
\hline
\textit{ValidCase} \\
x?, y?, z? : \mathbb{N} \\
class! : TRIANGLE \\
\hline
(x?, y?, z?) \in ValidTriangle \\
\#\{x?, y?, z?\} = 1 \Rightarrow class! = EQUILATERAL \\
\#\{x?, y?, z?\} = 2 \Rightarrow \\
\quad ((x?, y?, z?) \in RATriangle \land class! = RA_ISOSCELES \lor \\
\quad (x?, y?, z?) \notin RATriangle \land class! = ISOCELES) \\
\#\{x?, y?, z?\} = 3 \Rightarrow \\
\quad ((x?, y?, z?) \in RATriangle \land class! = RA_SCALENE \lor \\
\quad (x?, y?, z?) \notin RATriangle \land class! = SCALENE) \\
\hline
\end{array}$$

The effect on the test template derivation is localised; the area within the dotted box in figure 7.8 shows the affected parts of the template hierarchy for *Classify*.

The cause-effect strategy generates two new templates for *RA_ISOSCELES* and *RA_SCALENE*, and modifies those for *ISOSCELES* and *SCALENE*.

$$\begin{aligned}
CE_{isoR} \mathrel{\widehat=} [VIS_{Classify} \mid\ & (x?, y?, z?) \in ValidTriangle \land \\
& (x?, y?, z?) \in RATriangle \land \\
& \#\{x?, y?, z?\} = 2]
\end{aligned}$$

$$\begin{aligned}
CE_{iso} \mathrel{\widehat=} [VIS_{Classify} \mid\ & (x?, y?, z?) \in ValidTriangle \land \\
& (x?, y?, z?) \notin RATriangle \land \\
& \#\{x?, y?, z?\} = 2]
\end{aligned}$$

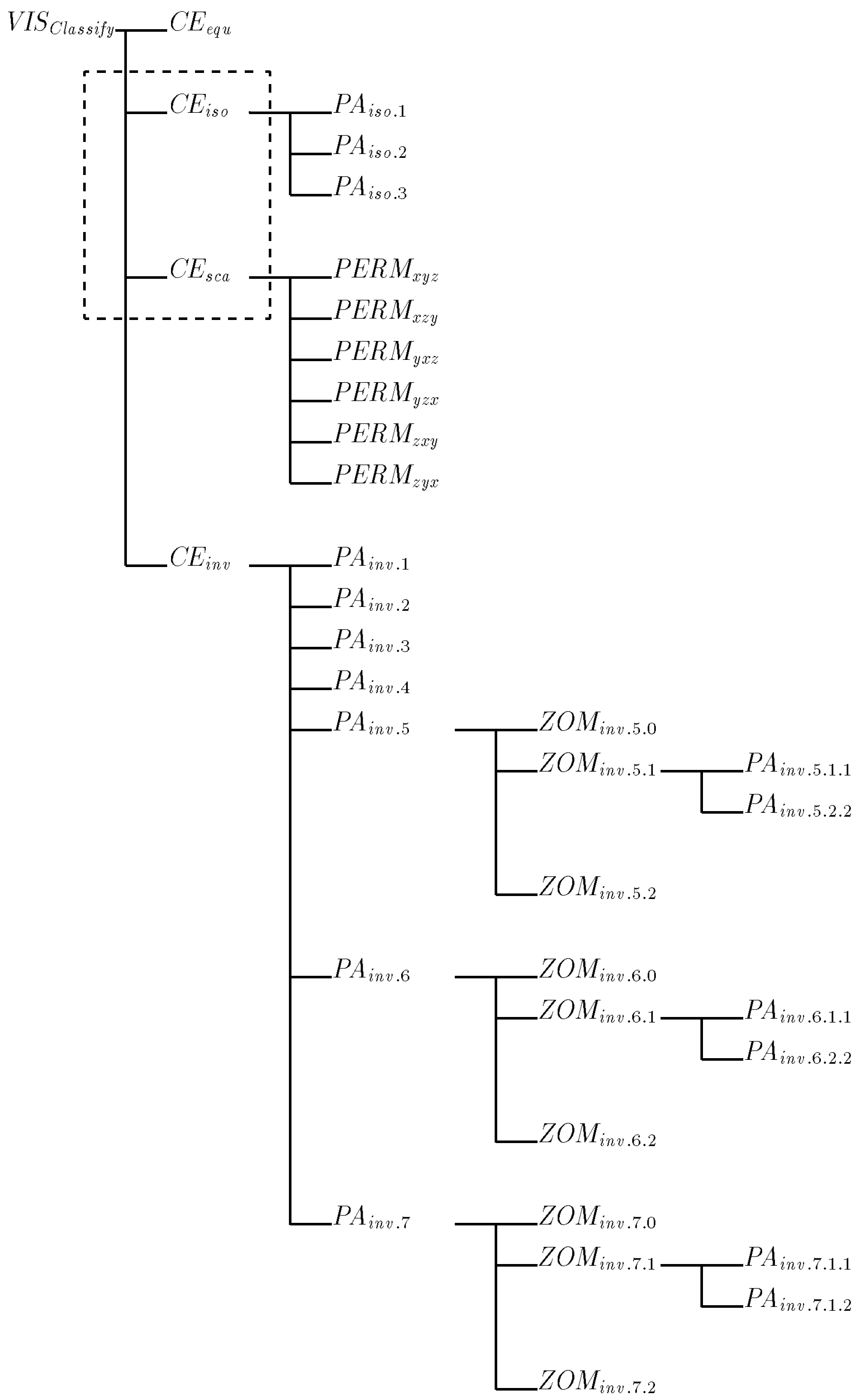


Figure 7.8: Test hierarchy for the triangle classification problem showing templates affected by maintenance changes to the specification.

$$CE_{scaR} \mathrel{\widehat=} [VIS_{Classify} \mid (x?, y?, z?) \in ValidTriangle \land \\ (x?, y?, z?) \in RATriangle \land \\ \#\{x?, y?, z?\} = 3]$$

$$CE_{sca} \mathrel{\widehat=} [VIS_{Classify} \mid (x?, y?, z?) \in ValidTriangle \land \\ (x?, y?, z?) \notin RATriangle \land \\ \#\{x?, y?, z?\} = 3]$$

The partition analysis and permutation templates are derived for each of the new cause-effect templates exactly as they were in the original test derivation. In fact, no new derivation is necessary, only a re-specification using the new cause-effect template as the inherited template. For example,

$$PA_{isoR.1} \mathrel{\widehat=} [CE_{isoR} \mid z? \neq 0 \land x? = y? \land z? \neq x? \land z? < x? + y?]$$

The definition of templates derived from CE_{iso} can stand as they are because CE_{iso} has been redefined. Thus, the new tests are introduced with minimal change to existing templates. Naturally, the hierarchy specification must be updated also:

$$\{CE_{equ}, CE_{isoR}, CE_{iso}, CE_{scaR}, CE_{sca}, CE_{inv}\} = \\ TTH_{Classify}(VIS_{Classify}, cause\_effect)$$

$$\{PA_{isoR.1}, PA_{isoR.2}, PA_{isoR.3}\} = TTH_{Classify}(CE_{isoR}, DNF\_partition)$$

$$\{PERM_{xyzR}, PERM_{xzyR}, PERM_{yxzR}, PERM_{yzxR}, PERM_{zxyR}, PERM_{zyxR}\} = \\ TTH_{Classify}(CE_{scaR}, permutation)$$

This new hierarchy is shown in figure 7.9. The oracle templates must also be updated suitably.

Another possible way to alter the template hierarchy for *Classify* would be to leave the definitions of CE_{iso} and CE_{sca} unchanged and introduce sub-templates for the two possibilities. For example,

$$CE_{isoR} \mathrel{\widehat=} [CE_{iso} \mid (x?, y?, z?) \in RATriangle]$$

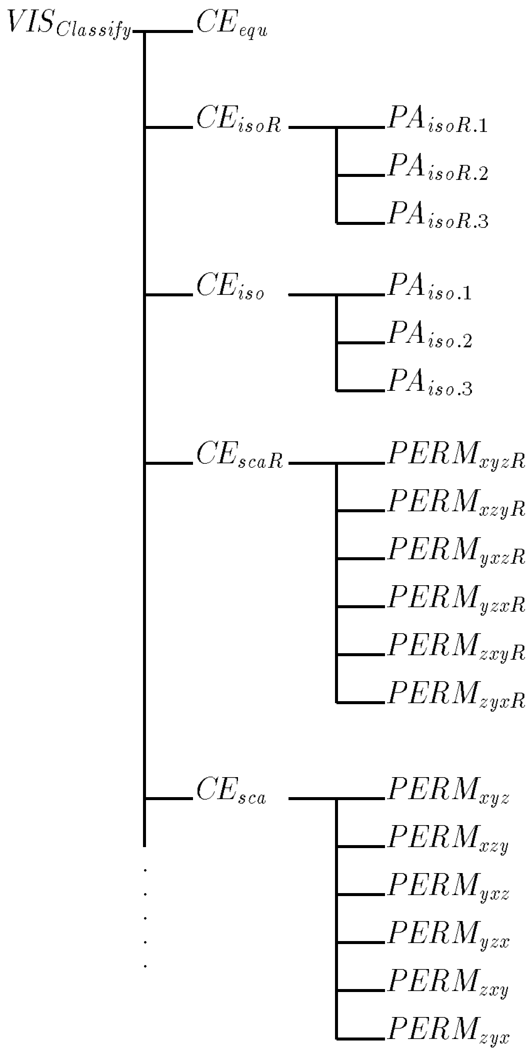


Figure 7.9: Test hierarchy for the triangle classification problem showing new templates derived after maintenance changes to the specification.

$$CE_{isoNotR} \mathrel{\widehat{=}} [CE_{iso} \mid (x?, y?, z?) \notin RATriangle]$$

The partition analysis templates and permutation templates would then be shown as derived from these new templates. This requires more changes to the hierarchy than the first alteration.

This example shows how the hierarchy specification is useful in simplifying regression testing for maintenance changes. The effects of the changes are easily shown and new tests can be re-derived. Derivation of test-cases is not intended to be automatic, rather the test template framework makes it more systematic and more likely to generate test-cases that "cover" the functionality in the specification. In many cases, only syntactic changes need to be made, such as changing the inherited template. Here, tool support would make regression test template derivation very efficient.

Chapter 8

Discussion

8.1 Main contributions

The major contribution of this work is the notion of test templates and the test template framework. We have defined and demonstrated a formal framework for specification-based testing. The framework offers a simple and elegant means of defining test cases and structuring tests in hierarchies. It also provides a uniform and formal basis for considering other specification-based testing issues such as reification, analysis, and maintenance. Abstractness is a useful concept in test definition, just as it is in system specification. Tests can be defined according to their abstract requirements and later transformed into actual tests by various means as discussed in chapter 7. The framework is not restrictive in its use, which allows many strategies to be used, as seen in chapters 5 and 6.

However, adapted testing strategies were not always effective. The next significant contribution of this work is the development of two novel testing strategies. These strategies address the failings of adapted strategies with which we experimented. These failings were due mainly to the 'flatness' of the specifications, that is, the high level statements of what must happen without the detail of how it must happen. This can hide important information from essentially implementation-based strategies. Domain propagation and specification mutation help bring this information to the surface. Because these strategies have a strong connection to the syntax of the specification notation used, we can construct libraries of standard test information for particular specification notations. Appendices C and D indicate how such libraries can be constructed.

The final significant and novel contribution of this work is the notion of rigorous reification of test specifications similar to specification reification. This is an effective approach for converting the abstract tests into concrete tests because we can reuse the transformations used in the development of the program implementation from its specification. Reification also provides a vehicle for structured development of white-box tests in manageable stages.

8.1.1 Some remarks on the framework model

All the benefits of the framework are results of the formal, abstract model of tests. Using an expressive specification notation is a powerful means to define tests, and this formal definition is the basis of other specification-based testing considerations. An important lesson learned in developing the model was to model tests implicitly rather than explicitly. Our explicit models of tests defined the components of a test, such as signature, constraints, parent, etc.; to specify a test, values had to be assigned to these components, which was awkward and notationally dense. The approach of simply defining constraints over bindings with the implicit understanding that they are tests improved both the elegance and expressiveness of the model.

The core of the framework focuses on modelling tests using a formal notation. We have used the Z notation, but other model-based notations could be used. Using the same notation for specifying the tests as is used in the original specification simplifies the process of defining test cases and increases the effectiveness of aspects which involve the formal specification, such as reification of tests in parallel with reification of the specification, and various analyses. This being said, we note that in essence the framework is using formal notations to model tests. That is, we are using the formal notation (in this case Z) as a test description language. This use of formal methods is independent of a formal specification. We can use the framework to define any test suites, not only those derived from formal specifications.

8.2 Future work

Perhaps the most important future work area is tool support. A simple tool interface to the template collection would greatly assist using the framework. Such a tool would keep track of the derivation structure so that only new information would need to be entered. Other useful tools that are more complicated are a pre-/post-condition analyser that derives pre- or post-conditions from operations' specifications, a theorem prover that can assist in verifying properties of templates, and a test data generator that automatically instantiates templates given the template definition. Fully automating test derivation may be impossible, and is discouraged, because of the insight and experience human testers bring to the process. The tool support should take care of the mundane tasks and record-keeping.

Also very important is empirical analysis of the effectiveness of testing strategies. Assessing testing strategies is a very difficult area and it is unclear what the best methods for assessing strategies are [Ham89], though the arguments for a statistical approach to reliability measurement presented in, for example, [Ham89, HV93] are compelling. We have proposed two new strategies with great intuitive appeal and some success in detecting errors undetected by other approaches. However, some form of empirical analysis is in order to justify our intuitions.

Finally, another aspect for consideration is applying the framework beyond unit testing. The specification defines operation interfaces, which is useful in module and integration testing. As it stands, the framework is useful for deriving unit test data, but means of testing module interfaces and interaction are required. In analogue to applying fault-based techniques for unit testing at the specification level (mutation testing), it would be interesting to bring fault-based techniques for interface testing to the specification-level, such as Howden's work on functional testing, which discusses fault models of interfaces [How86a, How86b].

8.2.1 Further applications

We saw in chapter 7 that the framework has many applications beyond software test definition and derivation. Here, we outline some ideas for some interesting applications and extensions that we have not examined in detail.

Debugging

There is a large gap between detecting an error in software and then finding the cause of the error. Debugging can be quite challenging, especially for large systems. Specification-derived tests may be of some assistance in debugging because the relationship between the test and the specification makes it clear which parts of the specification aren't implemented correctly. There can be a simple mapping from specification structure to implementation structure, but this need not necessarily be true [CDHW93]. If the implementation structure does correspond closely to the specification structure, it will be easier to find where the fault lies. Even if there isn't a close correspondence between the specification and implementation structures, knowing which parts of the specification aren't implemented correctly can still offer clues as to where in the implementation the fault lies.

The error-pinpointing ability of specification-derived tests would increase in cases where the implementation has been rigorously derived from the specification using some reification methodology, especially if the concrete tests were constructed similarly and enhanced appropriately as more structural information was introduced. The tests derived later in the reification process are effective for debugging because they are derived from a specification (albeit less abstract than the original) to which the implementation structure corresponds closely.

Software design: Components

Cox describes a software engineering ideal where software development involves constructing programs from pre-defined components, similar to conventional engineering's use of nuts and bolts [Cox90]. A software engineering revolution similar to the industrial revolution is discussed which will move software development from a 'build everything from scratch' approach to a 'build from re-usable components where possible' approach. Real, hardware components are defined by a specification of their purpose/parameters and some sort of gauge to determine whether a proposed piece of hardware is satisfactory. Cox advocates a similar approach for software. The framework is useful in component definition because the set of test cases derived from the specification can act as the component gauge. The test cases can be used to determine whether implementations of the component conform to the specification.

8.2.2 Extensions to the framework

There is room for extension to existing aspects of the framework. Certainly, our libraries of standard domain propagations and operator mutations should be completed. The number of specification mutants undetectable by testing, particularly mutants of logic operators, prompts us to consider other means of detecting these mutants.

Our model of oracles is simply to specify expected results; checking that the actual results match the expected results is not addressed. It could be done manually, of course, but there is room for applying and experimenting with other techniques, such as those discussed in [GMH81, Hay86, RAO92].

The reification model also requires extension. At present, it only deals with data reification, and is incomplete. Procedural reification also needs to be considered, especially the effects of procedural reification on specification/implementation structure and white-box tests.

A potential extension to the framework is suggested by our maintenance experiments. We saw in chapter 7 that when we 'split' a template high in the hierarchy, we usually derive a similar set of sub-templates from the new template as we derived from the original. The only difference between these two sets of templates is the parent template. So it seems that there is unnecessary syntactic work involved. A possible alteration to our template model which addresses this is for templates to define only constraints and have the parent (and hence signature) implicit from the specification of the hierarchy. Now, when a template is split, we can update the hierarchy with a mapping from the new template to the original constraint templates. We have to update the hierarchy specification anyway, so there is no extra work in doing this. There are other issues that need to be considered before we can decide whether such a model is superior to the current model. Making this new model legal

Z would require other changes in the model. Also, some naming convention for templates would have to be adopted which took account of the templates higher in the hierarchy. Finally, there is the trade-off of the reduction of syntactic burden (which tools could easily handle) against the stylistic issue that each template definition in our model is complete and can stand alone.

8.3 Conclusion

We have examined and demonstrated applications of formal methods to software testing. Our framework addresses the key issues in specification-based testing that we identified in chapter 2, providing a flexible and formal method of defining and structuring tests. The framework is a uniform basis from which to take advantage of the many beneficial applications of formal methods to testing.

Bibliography

[AA92] N. Amla and P. Ammann. Using Z specifications in category partition testing. In *Proceedings of COMPASS 1992, the Seventh Annual Conference on Computer Assurance*, pages 3–10, June 1992.

[AHKN90] J. Arkko, V. Hirvisalo, J. Kuusela, and E. Nuutila. Supporting testing of specifications and implementations. *EUROMICRO Journal*, 30(1-5):297–302, August 1990. EUROMICRO'90.

[BB89] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. In P. H. J. van Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 23–76. North Holland, 1989.

[BCFG86] L. Bougé, N. Choquet, L. Fribourg, and M.-C. Gaudel. Test sets generation from algebraic specifications using logic programming. *Journal of Systems and Software*, 6(4):343–360, November 1986.

[Bei90] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, New York, second edition, 1990.

[BGM91] G. Bernot, M.-C. Gaudel, and B. Marre. Software testing based on formal specifications: A theory and a tool. *Software Engineering Journal*, 6(6):387–405, November 1991.

[BHO89] M. J. Balcer, W. M. Hasling, and T. J. Ostrand. Automatic generation of test scripts from formal test specifications. *Software Engineering*

Notes, 14(8):210–218, December 1989. Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3).

[BN92] S. M. Brien and J. E. Nicholls. Z base standard version 1.0. Technical report, Programming Research Group, Oxford University Computing Laboratory, Oxford University, 1992.

[Bou85] L. Bougé. A contribution to the theory of program testing. *Theoretical Computer Science*, 37:151–181, 1985.

[Bud81] T. A. Budd. Mutation analysis: Ideas, examples, problems, and prospects. In B. Chandrasekaran and S. Radicchi, editors, *Computer Program Testing*. North-Holland, 1981.

[CDHW93] D. Carrington, D. Duke, I. Hayes, and J. Welsh. Deriving modular designs from formal specifications. *Software Engineering Notes*, 18(5):89–98, December 1993. Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering.

[CFR90] J. Collofello, T. Fisher, and M. Rees. A testing methodology framework. In G. J. Knafl, editor, *Proceedings of the 15th Annual International Computer Software and Applications Conference*. IEEE Computer Society, 1990.

[CHR82] L. A. Clarke, J. Hassell, and D. J. Richardson. A close look at domain testing. *IEEE Transactions on Software Engineering*, 8(4):380–390, July 1982.

[Cox90] B. J. Cox. Planning the software industrial revolution. *IEEE Software*, 7(6):25–33, November 1990.

[CS94] D. Carrington and P. Stocks. A tale of two paradigms: Formal methods and software testing. In *Proceedings of the 8th Z User Meeting*. Springer-Verlag, 1994.

[CW93] E. Cusack and C. Wezeman. Deriving tests for objects specified in Z. In J. P. Bowen and J. E. Nicholls, editors, *Z User Workshop*. Springer-Verlag, 1993.

[DDH72] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured programming*. APIC Studies in Data Processing, number 8. Academic Press, 1972.

[DF92] J. Dick and A. Faivre. Automatic partition analysis of VDM specifications. Technical Report RAD/DMA/92027, Research and Advanced Development, Bull Systems Products, BULL S.A., Rue Jean Jaurès, 78340 Les Clayes-sous-Bois, France, October 1992.

[DF93] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In J. C. P. Woodcock and P. G. Larsen, editors, *FME'93 Industrial Strength Formal Methods, Lecture Notes in Computer Science 670*, pages 268–284. Springer-Verlag, 1993.

[DKRS91] R. Duke, P. King, G. Rose, and G. Smith. The Object-Z specification language version 1. Technical Report 91-1, Software Verification Research Centre, The University of Queensland, Queensland 4072, Australia, May 1991.

[DLS78] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.

[DM91] P. Dauchy and B. Marre. Test data selection from algebraic specifications: Application to an automatic subway module. In A. van Lamsweerde and A. Fugetta, editors, *Lecture Notes in Computer Science 550*. Springer-Verlag, 1991. 3rd European Software Engineering Conference, ESEC '91.

[DN84] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10(4):438–444, July 1984.

[Fre91] R. S. Freedman. Testability of software components. *IEEE Transactions on Software Engineering*, 17(6):553–564, June 1991.

[GCG90] C. P. Gerrard, D. Coleman, and R. M. Gallimore. Formal specification and design time testing. *IEEE Transactions on Software Engineering*, 16(1):1–12, January 1990.

[GG75] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, 1(2):156–173, June 1975.

[GHW85] J. V. Guttag, J. J. Horning, and J. M. Wing. Larch in five easy pieces. Technical Report 5, Digital Equipment Corporation Systems Research Center, Palo Alto, California, USA, 1985.

[GMH81] J. Gannon, P. McMullin, and R. Hamlet. Data-Abstraction implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems*, 3(2):211–223, July 1981.

[Gog84] J. A. Goguen. Parameterized programming. *IEEE Transactions on Software Engineering*, 10(5):528–544, 1984.

[Gou83] John S. Gourlay. A mathematical framework for the investigation of testing. *IEEE Transactions on Software Engineering*, 9(6):686–709, November 1983.

[Hal88] P. A. V. Hall. Towards testing with respect to formal specifications. In *Second IEE/BCS Conference on Software Engineering 88*, pages 159–163. IEE, July 1988.

[Hal91] P. A. V. Hall. Relationship between specifications and testing. *Information and Software Technology*, 33(1):47–52, 1991.

[Ham77] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.

[Ham87] R. Hamlet. Probable correctness theory. *Information Processing Letters*, 25:17–25, April 1987.

[Ham89] R. Hamlet. Theoretical comparison of testing methods. *Software Engineering Notes*, 14(8):28–37, December 1989. Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3).

[Hay86] I. J. Hayes. Specification directed module testing. *IEEE Transactions on Software Engineering*, 12(1):124–133, January 1986.

[Hay87] I. Hayes, editor. *Specification Case Studies*. Series in Computer Science. Prentice Hall International, 1987. Later edition available.

[Hay93] I. Hayes, editor. *Specification Case Studies*. Series in Computer Science. Prentice Hall International, second edition, 1993.

[Het88] B. Hetzel. *The Complete Guide to Software Testing*. QED Information Sciences, Wellesley, Massachusetts, second edition, 1988.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall International, 1985.

[How82] W. E. Howden. Weak mutation testing and completeness of program test sets. *IEEE Transactions on Software Engineering*, SE-8, 1982.

[How86a] W. E. Howden. A functional approach to program testing and analysis. *IEEE Transactions on Software Engineering*, 12(10):997–1005, October 1986.

[How86b] W. E. Howden. *Functional Program Testing and Analysis*. McGraw Hill, 1986.

[HT88] R. Hamlet and R. Taylor. Partition testing does not inspire confidence. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 206–215, Banff, Canada, July 1988.

[HV93] R. Hamlet and J. Voas. Faults on its sleeve: Amplifying software reliability testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'93)*, pages 89–98, June 1993.

[Jon90] C. B. Jones. *Systematic Software Development Using VDM.* Series in Computer Science. Prentice Hall International, 1990. Second Edition.

[JW89] B. Jeng and E. J. Weyuker. Some observations on partition testing. *Software Engineering Notes*, 14(8):38–47, December 1989. Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3).

[Kem85] R. A. Kemmerer. Testing formal specifications to detect design errors. *IEEE Transactions on Software Engineering*, 11(1):32–43, January 1985.

[Kne89] Ralf Kneuper. Symbolic execution as a tool for validation of specifications. Technical Report UMCS-89-7-1, Department of Computer Science, University of Manchester, 1989. Ph.D. dissertation.

[Lay92] G. Laycock. Formal specification and testing: A case study. *Journal of Software Testing, Verification and Reliability*, 2(1):7–23, May 1992.

[Lin92] P. A. Lindsay. The ISDM case study: A dependency management system (inception paper). SVRC working paper. Software Verification Research Centre, The University of Queensland, 1992.

[MG83] Paul R. McMullin and John D. Gannon. Combining testing with formal specification: A case study. *IEEE Transactions on Software Engineering*, 9(3):328–335, May 1983.

[Mor90a] Larry J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, August 1990.

[Mor90b] C. Morgan. *Programming from Specifications.* Series in Computer Science. Prentice-Hall International, 1990.

[Mye79] G. J. Myers. *The Art of Software Testing.* Business data processing. Wiley-Interscience, 1979.

[OB88] 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.

[Off92] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology*, 1(1):5–20, January 1992.

[OSW86] T. J. Ostrand, R. Sigal, and E. J. Weyuker. Design for a tool to manage specification-based testing. In *Workshop on Software Testing*, pages 41–50, Banff, Canada, July 1986. IEEE Computer Society Press.

[Pac90] J. Pachl. A notation for specifying test selection criteria. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Protocol Specification, Testing, and Verification, X.* North-Holland, 1990.

[Pet81] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, 1981.

[PZ91] A. Parrish and S. H. Zweben. Analysis and refinement of software test data adequacy properties. *IEEE Transactions on Software Engineering*, 17(6):565–581, June 1991.

[RAI92] The RAISE Language Group. *The RAISE Specification Language*, 1992.

[RAO92] D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th International Conference on Software Engineering*, pages 105–118, May 1992.

[RC85] D. J. Richardson and L. A. Clarke. Partition analysis: A method combining testing and verification. *IEEE Transactions on Software Engineering*, 11(12):1447–1490, December 1985.

[Ros92] G. A. Rose. The ISDM case study: A dependency management system (Z and Object-Z specification). SVRC working paper. Software Verification Research Centre, The University of Queensland, 1992.

[ROT89] D. J. Richardson, O. O'Malley, and C. Tittle. Approaches to specification-based testing. *Software Engineering Notes*, 14(8):86–96, December 1989. Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3).

[SC91] P. Stocks and D. A. Carrington. Deriving software test cases from formal specifications. In *6th Australian Software Engineering Conference*, pages 327–340, July 1991.

[SC92] P. Stocks and D. A. Carrington. The ISDM case study: A dependency management system (specification-based testing). SVRC working paper. Software Verification Research Centre, The University of Queensland, 1992.

[SC93a] P. Stocks and D. A. Carrington. Test template framework: A specification-based testing case study. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'93)*, pages 11–18, June 1993.

[SC93b] P. Stocks and D. A. Carrington. Test templates: A specification-based testing framework. In *Proceedings of the 15th International Conference on Software Engineering*, pages 405–414, May 1993.

[SC93c] P. Stocks and D. A. Carrington. Test templates: A specification-based testing framework. Technical Report 243, Key Centre for Software Technology, Department of Computer Science, The University of Queensland, 1993.

[Sco88] L. T. Scott. On the problem of software testing and the generation of test data. Master's thesis, The University of Queensland, Queensland 4072, Australia, 1988.

[Scu88] G. T. Scullard. Test case selection using VDM. In R. Bloomfield, L. Marshall, and R. Jones, editors, *VDM '88 VDM- The Way Ahead, Lecture Notes in Computer Science 328*, pages 178–186. Springer-Verlag, 1988.

[Spi89] J. M. Spivey. *The Z Notation: A Reference Manual.* Series in Computer Science. Prentice Hall International, 1989. Later edition available.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual.* Series in Computer Science. Prentice Hall International, second edition, 1992.

[Tan76] A. S. Tanenbaum. In defense of program testing or correctness proofs considered harmful. *ACM SIGPLAN Notices*, 11(5):64–68, May 1976.

[WC80] L. J. White and E. I. Cohen. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, 6(3):247–257, May 1980.

[Wez90] C. D. Wezeman. The CO-OP method for compositional derivation of canonical testers. In E. Brinksma, G. Scollo, and C. A. Vissers, editors, *Protocol Specification, Testing and Verification IX.* North Holland, 1990.

[WJ91] E. J. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, July 1991.

[WO80] E. J. Weyuker and T. J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, 6(3):236–246, May 1980.

Appendix A

Z Glossary

This appendix is a glossary of some of the less familiar Z notation used in this thesis. These descriptions are drawn with permission (and thanks) directly from the full glossary of Z notation in [Hay93].

A.1 Definitions and declarations

Let $x, x_1, x_2, \ldots, x_n, X, X_1, X_2, \ldots, X_n$ be identifiers and let T, T_1, T_2, $\ldots$, T_n be set-valued expressions.

$LHS == RHS$

Definition of LHS as equivalent to RHS. A definition is distinguished from an equality ('=') syntactically by the use of the symbol '=='. A definition *defines* the left side to be equivalent to the right side, while an equality is a predicate that is either true or false.

$x : T$ — A declaration, $x : T$, introduces a new variable x of type T. This should be distinguished from the membership test, $x \in T$, which is a predicate that is either true or false.

$x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n$

List of declarations.

$x_1, x_2, \ldots, x_n : T$

$$== x_1 : T; \; x_2 : T; \; \ldots; \; x_n : T$$

$[X_1, X_2, \ldots, X_n]$

Introduction of basic types named $X_1, X_2, \ldots, X_n$. These are distinct new types whose structure is not constrained by this introduction, but may be constrained by predicates in the remainder of a specification.

A.2 Axiomatic definitions

Let D be a list of declarations and P a predicate.

The following axiomatic definition introduces the variables in D with the types as declared in D. These variables must also satisfy the predicate P. The scope of the variables is the whole specification.

$$\begin{array}{|l} D \\ \hline P \end{array}$$

For example,

$$\begin{array}{|l} small, large : \mathbb{N} \\ \hline small < large \end{array}$$

introduces two natural number variables *small* and *large* such that the value of *small* is less than the value of *large*.

The predicate part of an axiomatic definition is optional. For example, a global variable *MaxSize* may be introduced by the following axiomatic definition.

$$\begin{array}{|l} MaxSize : \mathbb{N} \end{array}$$

This variable may later be constrained by (global) predicates occurring in the specification.

$$MaxSize < 100$$

A.3 Binary relations

A binary relation is modelled by a set of ordered pairs. Hence operators defined for sets can be used on relations. Let X, Y and Z be sets; $x, x_1, \ldots, x_n : X$; $y, y_1, y_2, \ldots, y_n : Y$; S be a subset of X; T be a subset of Y; and R a relation between X and Y.

$x \underline{R} y$ $\quad == (x, y) \in R$

x is related by R to y. The name of a relation may either be an identifier or an infix operator symbol. A relation with an identifier name may be used as an infix operator by underlining it. For an infix relation operator, the whole relation may be referred to by placing underscores either side of the symbol and enclosing that in parentheses. For example, the whole relation corresponding to the infix operator '$<$' is referred to by '$(\_ < \_)$', so $(x < y) \Leftrightarrow (x, y) \in (\_ < \_)$.

$\mathrm{dom}\, R$ $\quad == \{x : X \mid (\exists\, y : Y \bullet x \underline{R} y)\}$

The domain of a relation: the set of x components that are related to some y.

$\mathrm{ran}\, R$ $\quad == \{y : Y \mid (\exists\, x : X \bullet x \underline{R} y)\}$

The range of a relation: the set of y components that some x is related to.

$R_1 \mathbin{⨾} R_2$ $\quad == \{x : X; z : Z \mid (\exists\, y : Y \bullet x \underline{R_1} y \wedge y \underline{R_2} z)\}$

Forward relational composition; $R_1 : X \leftrightarrow Y$; $R_2 : Y \leftrightarrow Z$. The composition relates x to z if there is some y such that x is related to y by R_1 and y is related to z by R_2.

$R^\sim$ $\quad == \{y : Y; x : X \mid x \underline{R} y\}$

Transpose of a relation R. $R^\sim$ relates y to x if and only if R relates x to y.

$\mathrm{id}\, S$ $\quad == \{x : S \bullet x \mapsto x\}$

Identity function on the set S.

R^k	The relation R composed with itself k times. This operator (sometimes called *iteration*) is only defined for homogeneous relations: relations that have the same source and destination sets. Given a homogeneous relation $R : X \leftrightarrow X$ and $k : \mathbb{N}$ $R^0 = \mathrm{id}\, X$ and $R^{k+1} = R^k \mathbin{;} R$.
R^+	$== \bigcup\{n : \mathbb{N}_1 \bullet R^n\}$ $= \bigcap\{Q : X \leftrightarrow X \mid R \subseteq Q \land Q \mathbin{;} Q \subseteq Q\}$ Transitive closure of relation R. A pair (x_1, x_n) is in the relation R^+ if and only if there exists a finite sequence of values $x_1, x_2, ..., x_n$, where $n \geq 2$, such that $(x_1, x_2) \in R$, $(x_2, x_3) \in R$, ... and $(x_{n-1}, x_n) \in R$.
R^*	$== \bigcup\{n : \mathbb{N} \bullet R^n\}$ $= R^+ \cup \mathrm{id}\, X$ $= \bigcap\{Q : X \leftrightarrow X \mid \mathrm{id}\, X \subseteq Q \land R \subseteq Q \land Q \mathbin{;} Q \subseteq Q\}$ Reflexive transitive closure.
$S \lhd R$	$== \{x : X; y : Y \mid x \in S \land x \underline{R}\, y\}$ Domain restriction: the relation R with its domain restricted to the set S.
$R \rhd T$	$== \{x : X; y : Y \mid x \underline{R}\, y \land y \in T\}$ Range restriction to T.

A.4 Free type definitions

$$X ::= ident1 \mid ident2\langle\!\langle S \rangle\!\rangle$$

Free types allow a new free set X to be introduced as well as defining constructors to generate elements of the type. The constructors may either be an identifier (*ident1*), which is an element of the new type, or a constructor function (*ident2*), which is a function taking an argument of type S and returning an element of the new type. Distinct values of arguments to constructor functions return distinct elements of the free type, and distinct constructors generate distinct elements. The constructors generate all the elements of the type.

Free types are useful for defining recursive structures, such as trees. The following example defines an arithmetic expression tree:

$$
\begin{aligned}
OP &::= plus \mid minus \mid times \mid divide \\
EXP &::= const\langle\!\langle \mathbb{N} \rangle\!\rangle \\
&\quad\;\; \mid \;\; binop\langle\!\langle OP \times EXP \times EXP \rangle\!\rangle
\end{aligned}
$$

Type OP contains four distinct elements which may be referenced by the identifiers *plus*, *minus*, *times* and *divide*. The type EXP describes an expression tree which is either a constant, natural number, or an expression consisting of a binary operator and two sub-expression trees.

A.5 Schema definition

A schema groups together a set of declarations of variables and a predicate relating the variables. If the predicate is omitted it is taken to be true, i.e. the variables are not further restricted. There are two ways of writing schemas: vertically, for example,

$$
\begin{array}{|l}
\multicolumn{1}{l}{S} \\ \hline
x : \mathbb{Z} \\
y : \mathbb{P}\,\mathbb{Z} \\ \hline
x \leq \#y \\ \hline
\end{array}
$$

and horizontally, for the same example,

$$S \mathrel{\widehat=} [x : \mathbb{Z};\ y : \mathbb{P}\,\mathbb{Z} \mid x \leq \#y]$$

As well as explicit declarations of variables, we allow schemas to be used in declarations as a shorthand for the declaration of the the variables in the schema constrained by the predicate of the schema. For example, schemas may be used in the declaration part of $\forall$, λ, $\{...\}$, etc.:

$$(\forall\, S \bullet y \neq \{\}) \Leftrightarrow (\forall\, x : \mathbb{Z};\ y : \mathbb{P}\,\mathbb{Z} \mid x \leq \#y \bullet y \neq \{\})$$

$\{S\}$ — Stands for the set of objects described by schema S. In declarations we usually write $w : S$ as an abbreviation for $w : \{S\}$, e.g. $w : S$ declares a variable w with components x (an integer) and y (a set of integers) such that $x \leq \#y$.

A.6 Schema operators

Let S be defined as above and $w : S$.

$w.x$ — $== (\lambda\, S \bullet x)(w)$
Projection functions: the component names of a schema may be used as projection (or selector) functions, e.g. $w.x$ is w's x component and $w.y$ is its y component; of course, the predicate '$w.x \leq \#w.y$' holds.

Compatibility

Two schemas are compatible if the declared sets of each variable common to the declaration parts of both the schemas are equal. In addition, any global variables referenced in the predicate part of one of the schemas must not have the same name as a variable declared in the other schema; this restriction is to avoid global variables being *captured* by the declarations.

Inclusion — A schema S may be included within the declarations of a schema R, in which case the declarations of S are merged with the other declarations of R (variables declared in both S and R must be compatible) and the predicates of S and R are conjoined. For example,

$$\begin{array}{|l}
\hline
R \\
S \\
z : \mathbb{Z} \\
\hline
z < x \\
\hline
\end{array}$$

is equivalent to

$$\begin{array}{|l}
\multicolumn{1}{l}{R} \\ \hline
x, z : \mathbb{Z} \\
y : \mathbb{P}\,\mathbb{Z} \\ \hline
x \leq \#y \land z < x \\ \hline
\end{array}$$

The included schema (S) may not refer to global variables that have the same name as one of the declared variables of the including schema (R).

Decoration Systematic renaming of the variables declared in a schema. Decoration with subscript, superscript, prime, etc. For example, S' is $[x' : \mathbb{Z};\ y' : \mathbb{P}\,\mathbb{Z} \mid x' \leq \#y']$. Multiple decorations of a single schema are allowed, e.g. S''.

$\neg\, S$ The schema S with its predicate part negated. For example, $\neg\, S$ is $[x : \mathbb{Z};\ y : \mathbb{P}\,\mathbb{Z} \mid \neg\,(x \leq \#y)]$.

$S \land T$ The schema formed from schemas S and T by merging their declarations and conjoining (and-ing) their predicates. The two schemas must be compatible (see above). Given

$$\begin{array}{|l}
\multicolumn{1}{l}{T} \\ \hline
x : \mathbb{Z} \\
z : \mathbb{P}\,\mathbb{Z} \\ \hline
x \in z \\ \hline
\end{array}$$

$S \land T$ is

$$\begin{array}{|l}
\multicolumn{1}{l}{S \land T} \\ \hline
x : \mathbb{Z} \\
y : \mathbb{P}\,\mathbb{Z} \\
z : \mathbb{P}\,\mathbb{Z} \\ \hline
x \leq \#y \land x \in z \\ \hline
\end{array}$$

$S \lor T$ The schema formed from schemas S and T by merging their declarations and disjoining (or-ing) their predicates. The two schemas must be compatible (see above). For example, $S \lor T$ is

$$\begin{array}{|l}
\hline
S \lor T \\
x : \mathbb{Z} \\
y : \mathbb{P}\,\mathbb{Z} \\
z : \mathbb{P}\,\mathbb{Z} \\
\hline
x \leq \#y \lor x \in z \\
\hline
\end{array}$$

$S \setminus (v_1, v_2, \ldots, v_n)$

Hiding: the schema S with variables $v_1, v_2, \ldots, v_n$ hidden – the variables listed are removed from the declarations and are existentially quantified in the predicate. For example, $S \setminus (x)$ is

$$\begin{array}{|l}
\hline
S \setminus (x) \\
y : \mathbb{P}\,\mathbb{Z} \\
\hline
(\exists\, x : \mathbb{Z} \bullet x \leq \#y) \\
\hline
\end{array}$$

$S \upharpoonright (v_1, v_2, \ldots, v_n)$

Projection: The schema S with any variables that do not occur in the list $v_1, v_2, \ldots, v_n$ hidden – the variables are removed from the declarations and are existentially qualified in the predicate. For example, $(S \land T) \upharpoonright (x, y)$ is

$$\begin{array}{|l}
\hline
(S \land T) \upharpoonright (x, y) \\
x : \mathbb{Z} \\
y : \mathbb{P}\,\mathbb{Z} \\
\hline
(\exists\, z : \mathbb{P}\,\mathbb{Z} \bullet x \leq \#y \land x \in z) \\
\hline
\end{array}$$

The list of variables may be replaced by a schema; the variables declared in the schema are used for projection.

A.7 Operation schemas

The following conventions are used for variable names in those schemas that represent operations, i.e. are written as descriptions of operations on some state:

undashed – state before the operation;

dashed – state after the operation;

ending in '?' – inputs to (arguments for) the operation; and

ending in '!' – outputs from (results of) the operation.

The basename of a name is the name with all decorations removed.

ΔS $\quad \widehat{=}\ S \land S'$

Change of state schema: this is a default definition for ΔS. In some specifications it is useful to have additional constraints on the change of state schema. In these cases ΔS can be explicitly defined.

ΞS $\quad \widehat{=}\ [\Delta S \mid \theta S' = \theta S]$

No change of state schema.

A.8 Operation schema operators

pre S Precondition: the after-state components (dashed) and the outputs (ending in '!') are hidden, e.g. given,

$$
\begin{array}{|l}
\hline
S \\
x?, s, s', y! : \mathbb{N} \\
\hline
s' = s - x? \land y! = s' \\
\hline
\end{array}
$$

pre S is

$$
\begin{array}{|l}
\hline
\text{pre } S \\
x?, s : \mathbb{N} \\
\hline
\exists\, s', y! : \mathbb{N} \bullet \\
\qquad s' = s - x? \land y! = s' \\
\hline
\end{array}
$$

Because, given the declarations above,

$$(\exists\, s', y! : \mathbb{N} \bullet s' = s - x? \land y! = s') \Leftrightarrow s \geq x?$$

the predicate can be simplified.

$$\begin{array}{|l}
\hline
\text{pre } S \\
x?, s : \mathbb{N} \\
\hline
s \geq x? \\
\hline
\end{array}$$

Appendix B

Calculating pre-conditions

To use the framework, we need to calculate pre-conditions of operations from their formal specifications. In some notations (e.g., VDM-SL and RSL), there is language support for explicitly defining the pre-condition of an operation, in which case 'calculating' the pre-condition is trivial. Otherwise, the pre-condition is implicit in the relationship between input and output states, which is the case in Z specifications where there is no language support for explicitly defining pre-conditions, but see the entry on schema pre-conditions in the Z glossary in appendix A.

B.1 Calculating implicit pre-conditions

Implicit pre-conditions are calculated by postulating the existence of an output state in the operation's predicate and then reducing this predicate to its simplest form. Consider the simple example from chapter 3:

ToZero

$x?, x! : \mathbb{Z}$

$(x? < 0 \land x! = x? + 1) \lor (x? > 0 \land x! = x? - 1)$

Postulating the existence of the output states gives us this expression for the pre-condition of *ToZero*

pre $ToZero \mathrel{\widehat{=}}$

$$
\begin{array}{|l}
x? : \mathbb{Z} \\
\hline
\exists\, x! : \mathbb{Z} \bullet \\
\quad (x? < 0 \land x! = x? + 1) \lor (x? > 0 \land x! = x? - 1)
\end{array}
$$

We can rewrite this expression as

$$
\begin{array}{|l}
x? : \mathbb{Z} \\
\hline
(x? < 0 \land (\exists\, x! : \mathbb{Z} \bullet x! = x? + 1)) \lor \\
(x? > 0 \land (\exists\, x! : \mathbb{Z} \bullet x! = x? - 1))
\end{array}
$$

which, of course, simplifies to

$$
\begin{array}{|l}
x? : \mathbb{Z} \\
\hline
x? < 0 \lor x? > 0
\end{array}
$$

because we can always find an $x!$ satisfying the right conjunct of each disjunct.

B.2 Pre-condition propagation

We must take care when simplifying predicates that some constraints are not overlooked. This is very similar to ideas behind our domain propagation strategy introduced in chapter 5. Some sub-operations in a specification may impose constraints over their input spaces, which must be taken into account when calculating the pre-condition of the operation. For example, consider this operation calculating percentages

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Percent}} \\
\hline
x?, y? : \mathbb{Z} \\
z! : \mathbb{Z} \\
\hline
z! = (x? \mathsf{div}\ y?) * 100 \\
\hline
\end{array}$$

A casual glance indicates that the pre-condition of *Percent* is true, since it seems we can always satisfy

$$\exists z! : \mathbb{Z} \bullet z! = (x? \mathsf{div}\ y?) * 100$$

But, of course, we have to consider the constraints over the input space of *Percent* imposed by the sub-operation of integer division. The pre-condition of div states that the divisor cannot be zero. This constraint propagates to the pre-condition of *Percent* because we cannot find a $z!$ satisfying the predicate above when $y? = 0$. So,

$$\text{pre } \textit{Percent} \mathrel{\widehat{=}} [x?, y? : \mathbb{Z} \mid y? \neq 0]$$

Note that the pre-condition that propagates to the higher level must be expressed in terms of the variables at the higher level. The pre-condition of div is that $divisor \neq 0$, which is expressed in terms of $y?$ at the top level. For another example, were the specification of *Percent* changed to

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Percent2}} \\
\hline
x?, y? : \mathbb{Z} \\
z! : \mathbb{Z} \\
\hline
z! = (x? \mathsf{div}\ (x? + y?)) * 100 \\
\hline
\end{array}$$

then the restriction imposed by the pre-condition of div is that $x? + y? \neq 0$, because $x? + y?$ is the expression representing *divisor* at the operation level. Integer division is a simple example because we are all familiar with this problem, but the principle is the same for all operations and not every sub-operation has pre-conditions that are so obvious.

Appendix C

Standard domains for Z operators

This appendix defines standard domain partitions for some of the Z operators defined in the Z Reference Manual (ZRM) [Spi89]. We only show this information for the operators requiring domain propagation in the case studies in chapter 6, but this demonstrates how a library of domain propagations for all Z operators could be constructed.

These domain propagations have been chosen based on a combination of experience and common sense. There are other possible sub-domain collections. The criteria for selecting domain propagations are to cover important input divisions and to partition the input space of which the propagations are sub-divisions. For example, appendix E shows that the domain divisions for set union, intersection, and difference form a partition.

C.1 Basic set operators

ZRM section 4.1, page 90.

$S \subseteq T$

1. $S = \{\} \land T = \{\}$
2. $S = \{\} \land T \neq \{\}$
3. $S \neq \{\} \land T \neq \{\} \land S \subset T$

4. $S \neq \{\} \wedge T \neq \{\} \wedge S = T$

$S \subset T$

1. $S = \{\} \wedge T \neq \{\}$
2. $S \neq \{\} \wedge T \neq \{\} \wedge S \subset T$

ZRM section 4.1, page 91.

$S \cup T$

$S \cap T$

$S \setminus T$

1. $S = \{\} \wedge T = \{\}$
2. $S = \{\} \wedge T \neq \{\}$
3. $S \neq \{\} \wedge T = \{\}$
4. $S \neq \{\} \wedge T \neq \{\} \wedge S \cap T = \{\}$
5. $S \neq \{\} \wedge T \neq \{\} \wedge S \subset T$
6. $S \neq \{\} \wedge T \neq \{\} \wedge T \subset S$
7. $S \neq \{\} \wedge T \neq \{\} \wedge S = T$
8. $S \neq \{\} \wedge T \neq \{\} \wedge S \cap T \neq \{\} \wedge \neg (S \subset T) \wedge \neg (T \subset S) \wedge S \neq T$

C.2 Relational operators

ZRM section 4.2, page 102.

R^+

1. $\mathrm{dom}\, R \cap \mathrm{ran}\, R = \{\}$
2. $\mathrm{dom}\, R = \mathrm{ran}\, R \wedge R = R^+$
3. $R \subset R^+$

Appendix D

Specification mutants for Z operators

This appendix defines mutants and distinguishing predicates for some of the Z operators defined in the Z Reference Manual (ZRM) [Spi89]. We only show this information for the operators mutated in the case studies in chapter 6, but this demonstrates how a library of mutants and distinguishing predicates for all Z operators could be constructed.

The type classification of mutants is

1. Disjoint: $Mut \cap Orig = \{\}$
2. Subset: $Mut \subset Orig$
3. Superset: $Mut \supset Orig$
4. Overlap: $Mut \cap Orig \neq \{\} \wedge \neg\, Mut \subset Orig \wedge \neg\, Mut \supset Orig$

Mutants for an operator are determined by substituting every type-compatible operator from the ZRM for the original operator. The mutants are classified, and distinguishing predicates calculated for all but type 3 mutants. Type 3 mutants lie outside the valid input space of the operation and cannot be distinguished by tests. The distinguishing predicates are intended to be as general as possible. In some cases the only general expression is an assertion that the mutant and original are not equal, which may not be helpful. In these cases, a less general distinguishing predicate involving a stronger constraint on the operands is offered. It is less general in the sense that any values satisfying this predicate will distinguish the mutant from the original, but there could be other combinations of values not satisfying the predicate that also would distinguish the mutant and the original.

D.1 Predicates

D.1.1 Logical connectives

ZRM section 3.7, page 70.

$p \land q$

$\lor$ - mutant: $p \lor q$
type: 3

$\Rightarrow$ - mutant: $p \Rightarrow q$
type: 3

$\Leftrightarrow$ - mutant: $p \Leftrightarrow q$
type: 3

$p \Leftrightarrow q$

$\land$ - mutant: $p \land q$
type: 2
distinguished by: $\neg p \land \neg q$

$\lor$ - mutant: $p \lor q$
type: 4
distinguished by: $\neg p \lor \neg q$

$\Rightarrow$ - mutant: $p \Rightarrow q$
type: 3

D.2 Mathematical toolkit

D.2.1 Sets

$\subseteq$ - subset

ZRM section 4.1, page 90.

$S \subseteq T$

$=$ - mutant: $S = T$

type: 2

distinguished by: $S \subset T$

$\neq$ - mutant: $S \neq T$

type: 4

distinguished by: $S = T$

$\subset$ - mutant: $S \subset T$

type: 2

distinguished by: $S = T$

$\cup$ - set union

ZRM section 4.1, page 91.

$S \cup T$

$\cap$ - mutant: $S \cap T$

type: 2

distinguished by: $S \neq T$

$\setminus$ - mutant: $S \setminus T$

type: 2

distinguished by: $T \neq \{\}$

D.2.2 Relations

Note that all the mutations based on unary relational operators such as relational inverse and transitive closure are only applicable to relations whose domain and range have the same type.

No unary operator

R

$^{\sim}$ - mutant: $R^{\sim}$

type: 4

distinguished by: $R \neq R^{\sim}$ ($\sqsubseteq$ $\mathrm{dom}\, R \neq \mathrm{ran}\, R$)

$^{+}$ - mutant: R^{+}

type: 3

* - mutant: R^{*}

type: 3

^{k} - mutant: R^{k}

type: 3

$^{+}$ - transitive closure

ZRM section 4.2, page 103.

R^{+}

^{1} - mutant: R^{1} or R

type: 2

distinguished by: $\mathrm{dom}\, R \cap \mathrm{ran}\, R \neq \{\}$

$^{\sim}$ - mutant: $R^{\sim}$

type: 4

distinguished by: $(\mathrm{dom}\, R \cup \mathrm{ran}\, R) \setminus (\mathrm{dom}\, R \cap \mathrm{ran}\, R) \neq \{\}$

* - mutant: R^{*}

type: 3

^{k} - mutant: R^{k}

type: 2

distinguished by: $R^{k} \neq R^{+}$ ($\sqsubseteq$ $R^{2} \neq R^{+}$)

k must be greater than 1 to distinguish R^{k} from R. For practical purposes, we can use the value 2 for k in a less general distinguishing predicate.

Appendix E

Proof of partitioning

The proposed standard partitions for basic set operations union, intersection, and difference on sets S and T are:

1. $S = \{\} \wedge T = \{\}$
2. $S = \{\} \wedge T \neq \{\}$
3. $S \neq \{\} \wedge T = \{\}$
4. $S \neq \{\} \wedge T \neq \{\} \wedge S \cap T = \{\}$
5. $S \neq \{\} \wedge T \neq \{\} \wedge S \subset T$
6. $S \neq \{\} \wedge T \neq \{\} \wedge T \subset S$
7. $S \neq \{\} \wedge T \neq \{\} \wedge S = T$
8. $S \neq \{\} \wedge T \neq \{\} \wedge S \cap T \neq \{\} \wedge \neg (S \subset T) \wedge \neg (T \subset S) \wedge S \neq T$

To prove that these domains are a partition we need to show that the input space is covered and that the domains are disjoint. If these domains cover the input space, their disjunction should simplify to true:

$(S = \{\} \wedge T = \{\}) \vee (S = \{\} \wedge T \neq \{\}) \vee (S \neq \{\} \wedge T = \{\}) \vee (S \neq \{\} \wedge T \neq \{\} \wedge S \cap T = \{\}) \vee$
$(S \neq \{\} \wedge T \neq \{\} \wedge S \subset T) \vee (S \neq \{\} \wedge T \neq \{\} \wedge T \subset S) \vee (S \neq \{\} \wedge T \neq \{\} \wedge$

$S = T) \vee$

$(S \neq \{\} \wedge T \neq \{\} \wedge \neg\, S \subset T \wedge \neg\, T \subset S \wedge S \cap T \neq \{\} \wedge S \neq T)$

$\Leftrightarrow$

[distribute 1st and 2nd]

$((S = \{\} \vee S = \{\}) \wedge (S = \{\} \vee T \neq \{\}) \wedge (T = \{\} \vee S = \{\}) \wedge (T = \{\} \vee T \neq$

$\{\}) \vee$

$(S \neq \{\} \wedge T = \{\}) \vee (S \neq \{\} \wedge T \neq \{\} \wedge S \cap T = \{\}) \vee$

$(S \neq \{\} \wedge T \neq \{\} \wedge S \subset T) \vee (S \neq \{\} \wedge T \neq \{\} \wedge T \subset S) \vee (S \neq \{\} \wedge T \neq \{\} \wedge$

$S = T) \vee$

$(S \neq \{\} \wedge T \neq \{\} \wedge \neg\, S \subset T \wedge \neg\, T \subset S \wedge S \cap T \neq \{\} \wedge S \neq T)$

$\Leftrightarrow$

[reduce, $p \wedge (p \vee q) = p$, $p \vee \neg\, p$]

$S = \{\} \vee$

$(S \neq \{\} \wedge T = \{\}) \vee (S \neq \{\} \wedge T \neq \{\} \wedge S \cap T = \{\}) \vee$

$(S \neq \{\} \wedge T \neq \{\} \wedge S \subset T) \vee (S \neq \{\} \wedge T \neq \{\} \wedge T \subset S) \vee (S \neq \{\} \wedge T \neq \{\} \wedge$

$S = T) \vee$

$(S \neq \{\} \wedge T \neq \{\} \wedge \neg\, S \subset T \wedge \neg\, T \subset S \wedge S \cap T \neq \{\} \wedge S \neq T)$

$\Leftrightarrow$

[$p \vee (\neg\, p \wedge q) = p \vee q$]

$S = \{\} \vee$

$T = \{\} \vee$

$(S \neq \{\} \wedge T \neq \{\} \wedge S \cap T = \{\}) \vee$

$(S \neq \{\} \wedge T \neq \{\} \wedge S \subset T) \vee (S \neq \{\} \wedge T \neq \{\} \wedge T \subset S) \vee (S \neq \{\} \wedge T \neq \{\} \wedge$

$S = T) \vee$

$(S \neq \{\} \wedge T \neq \{\} \wedge \neg\, S \subset T \wedge \neg\, T \subset S \wedge S \cap T \neq \{\} \wedge S \neq T)$

$\Leftrightarrow$

[let $p = (S = \{\} \vee T = \{\})$, $p \vee (\neg\, p \wedge q) = p \vee q$]

$S = \{\} \vee$

$T = \{\} \vee$

$S \cap T = \{\} \vee$

$(S \neq \{\} \wedge T \neq \{\} \wedge S \subset T) \vee (S \neq \{\} \wedge T \neq \{\} \wedge T \subset S) \vee (S \neq \{\} \wedge T \neq \{\} \wedge$
$S = T) \vee$

$(S \neq \{\} \wedge T \neq \{\} \wedge \neg S \subset T \wedge \neg T \subset S \wedge S \cap T \neq \{\} \wedge S \neq T)$

$\Leftrightarrow$

[ad inf.]

$S = \{\} \vee$

$T = \{\} \vee$

$S \cap T = \{\} \vee$

$S \subset T \vee$

$T \subset S \vee$

$S = T \vee$

$(S \neq \{\} \wedge T \neq \{\} \wedge \neg S \subset T \wedge \neg T \subset S \wedge S \cap T \neq \{\} \wedge S \neq T)$

$\Leftrightarrow$

[De Morgan, $p \vee \neg p$]

true

The domains cover the input space, and it should be obvious that the domains are also disjoint.

Appendix F

Z specification of the DMS

The complete Z specification of the Dependency Management System (DMS) is shown here. This specification was done as part of the Integrated Specification and Development Methodology (ISDM) project conducted by the Software Verification Research Centre at the University of Queensland. This specification is drawn from [Ros92].

F.1 The state schema

The state schema for the DMS, namely $DepManSys[X]$, is shown below. It has a generic parameter X to accommodate a variety of node types.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\, DepManSys[X] \,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
nodes : \mathbb{F}\, X \\
dir\_dep\_on : X \leftrightarrow X \\
\_ \succ \_ : X \leftrightarrow X \\
\hline
dir\_dep\_on \subseteq nodes \times nodes \\
(\_ \succ \_) = dir\_dep\_on^{+} \\
\neg\, (\exists\, x : X \bullet x \succ x) \\
\hline
\end{array}
$$

State variable *nodes* represents the finite set of nodes. State variables *dir_dep_on* and $\succ$ represent the direct dependency relationship and its transitive closure respectively: both are relations on X. (An aid to reading the specification is to think of the domain of dir_dep_on as dependents and the range as supporters.)

The state schema predicate is a system invariant which is true initially and remains true after every applicable operation. The predicate requires that only nodes are related by dir_dep_on, that $\succ$ is the transitive closure, and that the transitivity does not induce a circularity. The predicate is a conjunction of three conjuncts each occupying a line – Z notation elides the conjunction operator '$\wedge$' in such formats.

F.2 Initialisation

$$
\begin{array}{|l}
\hline
\textit{Init}[X] \\
\hline
\textit{DepManSys}[X] \\
\hline
\textit{nodes} = \{\} \\
\hline
\end{array}
$$

Initially, the DMS has no nodes and, by the state invariant, the direct dependency relation and its transitive closure are empty.

F.3 The operational schemas

Following are the fifteen operation schemas, only some of which have commentary for the sake of brevity.

$$\mathbf{Bool} ::= \textit{True} \mid \textit{False}$$

$$
\begin{array}{|l}
\hline
\textit{NoNodes}[X] \\
\hline
\Xi \textit{DepManSys}[X] \\
\textit{result}! : \mathbf{Bool} \\
\hline
\textit{result}! = \textit{True} \Leftrightarrow \textit{nodes} = \{\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{IsNode}[X] \\
\Xi \textit{DepManSys}[X] \\
x? : X \\
\textit{result}! : \mathbf{Bool} \\
\hline
\textit{result}! = \textit{True} \Leftrightarrow x? \in \textit{nodes} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{AddNode}[X] \\
\Delta \textit{DepManSys}[X] \\
x? : X \\
\hline
x? \notin \textit{nodes} \\
\textit{nodes}' = \textit{nodes} \cup \{x?\} \\
\textit{dir\_dep\_on}' = \textit{dir\_dep\_on} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{RemoveNode}[X] \\
\Delta \textit{DepManSys}[X] \\
x? : X \\
\hline
x? \in \textit{nodes} \setminus \operatorname{ran} \textit{dir\_dep\_on} \\
\textit{nodes}' = \textit{nodes} \setminus \{x?\} \\
\textit{dir\_dep\_on}' = \{x?\} \lhd \textit{dir\_dep\_on} \\
\hline
\end{array}
$$

Operation *RemoveNode* specifies the removal of a node from the system. It requires as input which node ($x?$) is to be considered for removal. The precondition is that the X value given must be an existing node and that no node depends on it (i.e. it is not a supporter). The remaining two conjuncts of the predicate specify the removal of the designated node from the set of nodes and from the direct dependency relation respectively. Z uses '\' for set subtraction and $\lhd$ for 'domain subtraction', i.e. in this application, removal of all pairs in $\textit{dir\_dep\_on}$ with domain value $x?$. (There is no need to also include range subtraction, denoted by $\rhd$, because the precondition excludes $x?$ from being a range value.) The value of the transitive closure after the operation is not explicitly stated – it is deducible from the new direct dependency relation according to its definition in the state invariant.

$NoDependencies[X]$

$\Xi DepManSys[X]$

$result! : \mathbf{Bool}$

$result! = True \Leftrightarrow dir\_dep\_on = \{\}$

$DependedUpon[X]$

$\Xi DepManSys[X]$

$x? : X$

$result! : \mathbf{Bool}$

$x? \in nodes$

$result! = True \Leftrightarrow x? \in \operatorname{ran} dir\_dep\_on$

$IsDependency[X]$

$\Xi DepManSys[X]$

$x?, y? : X$

$result! : \mathbf{Bool}$

$\{x?, y?\} \subseteq nodes$

$result! = True \Leftrightarrow (x?, y?) \in dir\_dep\_on$

$CanAdd[X]$

$\Xi DepManSys[X]$

$x?, y? : X$

$result! : \mathbf{Bool}$

$\{x?, y?\} \subseteq nodes$

$result! = True \Leftrightarrow (\neg\, (y? \succ x?) \wedge x? \neq y?)$

Operation *CanAdd* is interrogative. It is satisfied if the pair $(x?, y?)$ could be added to the direct dependency relation. The requirements are that $x?$ and $y?$ must be existing nodes and that $x?$ depending on $y?$ would not cause a circularity, i.e. that $y?$ does not already depend on $x?$ (directly or transitively) and that $x?$ and $y?$ are distinct.

$$
\begin{array}{|l}
\multicolumn{1}{l}{AddDependence[X]} \\
\hline
\Delta DepManSys[X] \\
x?, y? : X \\
\hline
\{x?, y?\} \subseteq nodes \\
\neg\,(y? \succ x?) \land x? \neq y? \\
nodes' = nodes \\
dir\_dep\_on' = dir\_dep\_on \cup \{(x?, y?)\} \\
\hline
\end{array}
$$

AddDependence specifies the addition of the nominated dependency $(x?, y?)$ to *dir_dep_on*. If successful, the state will change, hence the Δ prefix of *DepManSys*. The first and second predicates (preconditions) establish the requirements of *CanAdd*. The third conjunct of *AddDependence*'s predicate ensures that the operation does not change the node set and the fourth conjunct specifies the insertion of the input dependency. The conjunction of the third and fourth conjuncts with the primed version of the state invariant is the operation's postcondition. As the postcondition includes the requirements that only nodes are dependency related and that there is no circularity, the precondition could have been deduced and therefore elided. However, explicit inclusion is often used in Z specifications to assist comprehension.

Δ*DepManSys* opens all state variables for change. The third conjunct cannot be deduced from the other conjuncts and the state invariant and so must be explicit.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad RemoveDependence[X] \quad}} \\
\Delta DepManSys[X] \\
x?, y? : X \\
\hline
(x?, y?) \in dir\_dep\_on \\
nodes' = nodes \\
dir\_dep\_on' = dir\_dep\_on \setminus \{(x?, y?)\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Dependents[X] \quad}} \\
\Xi DepManSys[X] \\
x? : X \\
nodes! : \mathbb{F}\ X \\
\hline
x? \in nodes \\
nodes! = \{n : nodes \mid n \succ x?\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Supporters[X] \quad}} \\
\Xi DepManSys[X] \\
x? : X \\
nodes! : \mathbb{F}\ X \\
\hline
x? \in nodes \\
nodes! = \{n : nodes \mid x? \succ n\} \\
\hline
\end{array}
$$

Supporters is an interrogation on the state and outputs the set of nodes ($nodes!$) which support the nominated node ($x?$). A precondition is that the input value $x?$ is an existing node. Support is interpreted as either direct or transitive support.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{UltSupporters}[X]} \\
\Xi \textit{DepManSys}[X] \\
x? : X \\
\textit{nodes}! : \mathbb{F}\ X \\
\hline
x? \in \textit{nodes} \\
\textit{nodes}! = \{ n : \textit{nodes} \mid x? \succ n \land n \notin \mathrm{dom}\ \textit{dir\_dep\_on} \} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CandidateSupporters}[X]} \\
\Xi \textit{DepManSys}[X] \\
x? : X \\
\textit{nodes}! : \mathbb{F}\ X \\
\hline
x? \in \textit{nodes} \\
\textit{nodes}! = \{ n : \textit{nodes} \mid \neg\ (n \succ x?) \land n \neq x? \} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{SomeDirectDependent}[X]} \\
\Xi \textit{DepManSys}[X] \\
x? : X \\
\textit{node}! : X \\
\hline
x? \in \mathrm{ran}\ \textit{dir\_dep\_on} \\
(\textit{node}!, x?) \in \textit{dir\_dep\_on} \\
\hline
\end{array}
$$

SomeDirectDependent is an interrogation of the state which non-deterministically outputs any one of the possibly several direct dependents of the nominated node $x?$. The precondition requires that $x?$ is an existing node with at least one direct dependent (i.e. that it is in the range of dir_dep_on). The second conjunct requires that the actual node output is directly dependent on the input.