SOFTWARE VERIFICATION RESEARCH CENTRE

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 94-4

A tale of two paradigms: Formal methods and software testing

David Carrington and Phil Stocks

Feb 1994

Phone: +61 7 365 1003
Fax: +61 7 365 1533

A tale of two paradigms: Formal methods and software testing

D. Carrington and P. Stocks
Software Verification Research Centre
The University of Queensland
Queensland 4072, Australia
email: davec | phil @cs.uq.oz.au

Abstract

There are two camps of software developers: formal methods advocates battling against traditionalist supporters of software testing and assessment metrics. Surely, as Turing observed, we will (must) never do away with testing in some form. But clearly, formal methods cannot be ignored, and must be the basis of quality assurance in some form. Important impacts of specifications on testing are in test selection, test oracles, and analysis of test suites and theoretical results of testing. We examine uses of formal specifications in software testing, particularly, roles of Z specifications in software testing. We also present our unifying framework for specification-based testing, which is founded on Z.

1 Prelude

It is not the best of times, but neither is it the worst. Software development is in transition, attempting to adopt formal methods without losing positive aspects of older methods. In traditional software development methods, testing occupies a central position of ensuring software quality. Because formal methods are often portrayed as a replacement for such methods, the new role of testing is often misunderstood.

In this paper, we investigate how Z specifications and testing can interact[1]. We review in some detail the relatively sparse literature on combining Z specifications and testing, including our previous work in this area. The intent is to show that many of the concepts developed originally for code-level testing can be promoted to operate effectively with formal specifications, and that formal methods practices of modelling and analysis are useful in testing.

The advantages of combining testing techniques with formal specifications are two-fold:

1. A formal specification is a convenient starting point for the systematic derivation of test suites for testing implementations. This is valuable while fully formal development methods are not widely used and even then, test suites will allow independent checks of correct implementation.

[1] We restrict ourselves to Z here because it is of most interest in this forum. Other specification-based testing work is similarly discussed in [16].

2. The process of generating test-cases from a formal specification is a form of analysis that can assist validation of the specification. Validation is important and difficult since formal techniques cannot ensure the required consistency between the user's informal requirements and the formal specification. Generating test-cases isolates small components of functionality which may be easier for the user to assess.

The paper also presents a framework for conducting specification-based testing and a simple case study to demonstrate its use. The framework is flexible, which enables different techniques focusing on different aspects of testing to be used with a uniform basis.

We identify the following key issues in testing, which are addressed by our discussion of related work and our framework.

Selecting test data

Structured strategies and criteria for choosing test data are required. Testing should not be haphazard.

Providing test oracles

Test data is useless without a means to assess the result of the test. Such a means is called an oracle.

Sequencing tests

Model-based specification notations, such as Z, construct explicit models of the system state as bindings of variables to values of certain types. Operations are represented by state transformations. State variables having complex data types cannot simply be assigned values in the same way as variables of basic data types. Without special state constructor operations, system operations may be required to build certain states. This requires the tests, and the operation testing, to be properly sequenced.

Defining tests (including oracles and sequencing)

Most research in testing focuses on one particular issue, such as a selection strategy, and is usually not concerned with the full picture. 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. It is very important to be able to collect the test information being produced in a uniform and useful way.

Evaluating tests

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. Tests can be evaluated against other properties and criteria to give more meaning to the results of testing.

2 Dramatis personae

The main players and their contributions are introduced.

2.1 Hall

An early proponent of specification-based testing, particularly using Z, is Hall, whose efforts are directed at exploring the relationship between specifications and testing.

Selecting test data

In [7], a general approach is used to derive tests from Z specifications. Simple partitions of the input space are constructed by examining the obvious divisions of input defined in the predicates of operations. This approach is highly structured, but not rigorous. Hall further discusses problems that may arise when checking the state of the system and indicates a preference for algebraic thinking in this regard.

Evaluating tests

Some preliminary ideas on test evaluation found in [7] form some of the basis for a later paper [8] examining the relationship between specifications and testing in the context of what is learned about software by its performance on a test suite. The contrast lies between deducing software correctness and between inducing software correctness based on test results. Dijkstra's classic statements on software testing identify the limits of inducing software correctness from performance on a test suite and imply a deductive approach proving software correctness is required. Hall argues that, apart from the possibility of introducing errors into a proof, an inductive approach still has value because it is recognised that software may be faulty when shipped, and may indeed be satisfactory despite any residual faults. An inductive approach can give insight into a measure of software quality. Hall suggests an approximation measure of software quality by measuring the difference between specification and implementation in the output domain weighted by the probability of the inputs occurring.

2.2 Dick and Faivre

The work of Dick and Faivre is a major contribution to the use of formal methods in software testing [4, 5]. Though their work uses VDM, their ideas are applicable to all model-based specification notations.

Selecting test data

A standard strategy 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.

Sequencing tests

A similar application of reduction to DNF is useful in test sequencing. 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.

Tool support

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. Construction of finite-state automaton and test selection is not yet automated.

2.3 Richardson and O'Malley

Also early to recognise the potential of formal methods in software testing were Richardson and O'Malley, whose early work considered various methods of choosing tests but whose main contribution lies in deriving test oracles from specifications.

Selecting test data

Richardson et al. [14] examine strategies for selecting tests by extending implementation-based testing techniques to be applicable to formal specifications. Testing strategies are classified into error-based and 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.

Providing test oracles

A quite detailed approach to the important problem of providing test oracles is described in [13]. 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.

2.4 Extras

There are few other results in Z-based testing.

Category partitioning

Category partitioning is a testing strategy developed by Ostrand and Balcer [12], and is designed to extract a set of functional tests from a specification (usually informal). It is a structured strategy for deriving tests. Obviously, it is much simpler to apply category partitioning to formal specifications (especially model-based specifications) because all the considerations that have to be tediously extracted from a natural language specification are spelled out formally. This is the thrust of [1] and [10]. This area is somewhat disappointing because no special connection to Z specifications is made, especially how knowledge of Z could lead to superior test selection and more complete test suites.

Conformance testing

Cusack and Wezeman [3] derive labelled transition systems from Object-Z specifications[2]. The labelled transition systems are concerned with the external behaviour of the specification and the CO-OP method [19] is used to derive canonical testers from the transition systems which test conformance of the external behaviours of the implementations to the specifications.

Oracles

Hayes [9] 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.

2.5 Stocks and Carrington

We have developed a framework for conducting specification-based testing, in which Z plays an integral role [17, 18]. Details of the framework are discussed in the next section. The framework is mainly concerned with structuring test information, but we have achieved some results in other specification-based testing areas.

Selecting test data

We advocate using as many testing strategies as desired or possible, since no one strategy is better than all the others. Our framework is a vehicle for using strategies. However, we have developed two strategies which we commonly use in testing.

The first, domain propagation, is based on reduction to DNF (also used by Dick and Faivre), but also considers effects of sub-operations on the domain partitions. Input domains for each Z operator are constructed[3] and propagated to the higher level operations to see what effect they have on the inputs of the higher level operations. Domains are propagated to the level of the operation under test.

[2] Object-Z is an object-oriented extension to Z developed at the University of Queensland [6].

[3] Not always using reduction to DNF. Most are a combination of experience, good sense, and fine-tuning.

The second resulted from adapting mutation testing to the specification level. This mutation testing has a different focus to mutation testing at the implementation level, where the goal is to provide an assessment of a test suite based on the number of program mutants it can kill. Here, specification mutants are considered and tests are chosen to distinguish these mutants from the original specification. This sort of testing is aimed at showing that certain errors do not exist in the implementation. Specification mutation testing is aimed at detecting misunderstandings of the specification in the implementation, and has some informal impact on specification validation.

Providing test oracles

The key element of our framework is the construction of abstract descriptions of test data, called test templates. This facilitates a fairly simple approach to providing abstract oracles. Similarly to Hayes, we use the input-output relation defined in the specification to construct abstract descriptions of output from given input templates, giving a description of the expected output for certain input. One feature of such descriptions in Z is that the state components of the input and output are clearly distinguished from the parameter components, which is helpful if the states need to be constructed and checked using existing operations.

Defining tests

To define test suites, we construct abstract definitions of test classes, which also show how the tests were derived. Other requirements of testing (such as the actual (concrete) data, oracles, and sequencing) can be synthesised from this information and the specification. These definitions are formal, which enables analysing and evaluating test suites.

Evaluating tests

The unifying framework facilitates various analyses of test suites and test strategies. Adequacy criteria of test suites can be shown to hold, for example. Though specification mutation focuses on selecting tests, test suites can be evaluated using mutation analysis and given a rating based on the number of specification mutants distinguished by the suite. Also, test suites derived using different strategies can be compared for overlap and performance, to give insight into which strategies are better in which circumstances.

3 Framework overview[4]

The goal is to provide a generic and flexible framework for specification-based testing. The formal specification notation Z [15, 2] is used as a test description language in the framework. There is no dependence between the notation used in the framework and the notation used in the specification from which tests will be derived. The framework consists of two parts: a formal model of tests and a method for using the model in testing. These are described below and illustrated with a small case study in the next section.

Z schemas are used in the framework to represent test data. Schemas used this way are called test templates (TTs). A test template is a generic description of test data, describing only the necessary restrictions on the data, and leaving everything else abstract. Templates are

[4]This overview of our framework is drawn from [17].

defined at the same level of abstraction as the specification; the final form of the test data depends on other aspects related to the final implementation.

The Input Space (IS) is the set from which values of operation components may be drawn. The Valid Input Space (VIS) is the subset of the IS for which the operation is defined. In Z specifications, the IS of an operation is its signature restricted to its input variables, and the VIS of an operation is its pre-condition.

Templates are organised into a Test Template Hierarchy (TTH) for each functional unit. The root of the hierarchy for each functional unit is the VIS. All the other templates are directly or indirectly derived from the VIS. Every template represents a subset of the VIS, and usually a subset of its parent in the hierarchy. The template hierarchy is nearly always a tree, but it is possible to derive equivalent templates from different parents, so the hierarchy is actually a digraph.

Further templates are derived from the VIS according to testing strategies. Strategies are incorporated into the framework as a name, a set of guidelines for selecting tests, and (optionally) some general properties of the strategy.

Nodes in the hierarchy represent templates, while arcs are derived from strategies. There are many testing strategies, each with its own strengths and weaknesses. In the framework, we advocate using multiple strategies and aim to give the tester freedom to use whatever strategies are desired. Thus templates are derived from other templates (initially from the VIS template) until the tester is satisfied with the tests produced. The leaves of the resulting tree represent the final test data. Instantiation templates for these templates may be derived by restricting the templates so that there is only one possible instance.

We introduce the generic set of testing strategies

$$[STRATEGY]$$

and for each operation, *Op*, we define the valid input space.

$$VIS_{Op} \mathrel{\widehat{=}} \mathrm{pre}\ Op$$

Since all the templates are subsets of the VIS, we can define a type synonym for templates to aid readability.

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

The hierarchy defines the set of child templates derived from some parent template using some strategy. Thus, it is a mapping from template/strategy pairs to sets of templates.

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

Also, the following functions are defined for each operation. *Children* applied to some template returns the set of templates in the hierarchy derived from it using any strategy. *Descendents* returns the set of all templates in the hierarchy directly or indirectly derived from some template using any strategy.

$$children_{Op} : TT_{Op} \rightarrow \mathbb{P}\ TT_{Op}$$

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

$$\begin{array}{l} descendants_{Op} : TT_{Op} \rightarrow \mathbb{P}\ TT_{Op} \\ \hline descendants_{Op} = (\lambda\ T : TT_{Op} \bullet \\ \quad children_{Op}(T) \cup \bigcup\{T2 : children_{Op}(T) \bullet descendants_{Op}(T2)\}) \end{array}$$

4 Case study

To demonstrate how the test template framework can be used and how test cases can be derived from Z specifications, we use a simple case study. This particular case study is well-known in the testing community [11].

> *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).*

Our first task is to create a Z specification of the required program. We define a free type *TRIANGLE* representing the possible classification outcomes.

$$\begin{array}{l} TRIANGLE ::= \\ \quad EQUILATERAL \mid ISOSCELES \mid SCALENE \mid INVALID \end{array}$$

We define the set of all valid triangles:

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

The classification operation is defined using schemas:

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

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

Schema disjunction is used to build the final schema definition from smaller components.

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

For *Classify*, the IS and VIS are identical as *Classify* is total, that is, it is defined over all input values.

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

Since all the test templates are subsets of the VIS, we define a type synonym for templates.

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

The template hierarchy for *Classify* is defined by

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

We derive functional test cases for *Classify* by applying several testing strategies. We begin with the cause-effect method that partitions the valid input space based on equivalence classes of the output space.

$$|\ cause_effect : STRATEGY$$

This strategy 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*!.

$$CE_{equ} \mathrel{\widehat=} [VIS_{Classify} \mid (x?, y?, z?) \in validTriangle \land \#\{x?, y?, z?\} = 1]$$

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

$$CE_{sca} \mathrel{\widehat=} [VIS_{Classify} \mid (x?, y?, z?) \in validTriangle \land \#\{x?, y?, z?\} = 3]$$

$$CE_{inv} \mathrel{\widehat=} [VIS_{Classify} \mid (x?, y?, z?) \notin validTriangle]$$

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

Our next strategy uses the partition analysis of Dick and Faivre [5]. 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.

$$|\ partition_analysis : STRATEGY$$

Using this strategy requires expanding and simplifying the current test templates:

$$CE_{equ} = [VIS_{Classify} \mid x? = y? \land y? = z? \land x? \neq 0]$$

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

$$\begin{aligned} CE_{sca} = [VIS_{Classify} \mid\ & x? < y? + z? \land y? < x? + z? \land z? < x? + y? \land \\ & x? \neq y? \land y? \neq z? \land z? \neq x?] \end{aligned}$$

$$CE_{inv} = [\, VIS_{Classify} \mid x? \geq y? + z? \vee y? \geq x? + z? \vee z? \geq x? + y?]$$

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

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

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

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

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

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

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

Knowledge of properties of arithmetic allow us to simplify this to

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

Thus generating

$$PA_{inv.1} \mathrel{\widehat=} [CE_{inv} \mid x? = 0 \wedge y? = 0 \wedge z? = 0]$$

$$PA_{inv.2} \mathrel{\widehat=} [CE_{inv} \mid x? \neq 0 \wedge x? = y? \wedge z? = 0]$$

$$PA_{inv.3} \mathrel{\widehat=} [CE_{inv} \mid x? \neq 0 \wedge y? = 0 \wedge x? = z?]$$

$$PA_{inv.4} \mathrel{\widehat=} [CE_{inv} \mid x? = 0 \wedge y? \neq 0 \wedge y? = z?]$$

$$PA_{inv.5} \mathrel{\widehat=} [CE_{inv} \mid x? \geq y? + z? \wedge y? < x? + z? \wedge z? < x? + y?]$$

$$PA_{inv.6} \mathrel{\widehat=} [CE_{inv} \mid x? < y? + z? \wedge y? \geq x? + z? \wedge z? < x? + y?]$$

$$PA_{inv.7} \mathrel{\widehat=} [CE_{inv} \mid x? < y? + z? \wedge y? < x? + z? \wedge z? \geq x? + y?]$$

$$\begin{aligned} &\{PA_{inv.1}, PA_{inv.2}, PA_{inv.3}, PA_{inv.4}, PA_{inv.5}, PA_{inv.6}, PA_{inv.7}\} = \\ &\quad TTH_{Classify}(CE_{inv}, partition_analysis) \end{aligned}$$

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.

Noting that this 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 the set $\{x?, y?, z?\}$ so "many" is constrained to two. The result is nine further templates:

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

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

$$ZOM_{inv.5.2} \mathrel{\widehat=} [PA_{inv.5} \mid y? = 0 \land z? = 0]$$

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

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

$$ZOM_{inv.6.2} \mathrel{\widehat=} [PA_{inv.6} \mid x? = 0 \land z? = 0]$$

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

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

$$ZOM_{inv.7.2} \mathrel{\widehat=} [PA_{inv.7} \mid x? = 0 \land y? = 0]$$

$$\{ZOM_{inv.5.0}, ZOM_{inv.5.1}, ZOM_{inv.5.2}\} = TTH_{Classify}(PA_{inv.5}, zero_one_many)$$

$$\{ZOM_{inv.6.0}, ZOM_{inv.6.1}, ZOM_{inv.6.2}\} = TTH_{Classify}(PA_{inv.6}, zero_one_many)$$

$$\{ZOM_{inv.7.0}, ZOM_{inv.7.1}, ZOM_{inv.7.2}\} = TTH_{Classify}(PA_{inv.7}, zero_one_many)$$

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.

$$PA_{inv.5.1.1} \mathrel{\widehat=} [ZOM_{inv.5.1} \mid y? = 0 \land z? \neq 0]$$

$$PA_{inv.5.1.2} \mathrel{\widehat=} [ZOM_{inv.5.1} \mid y? \neq 0 \land z? = 0]$$

$$PA_{inv.6.1.1} \mathrel{\widehat=} [ZOM_{inv.6.1} \mid x? = 0 \land z? \neq 0]$$

$$PA_{inv.6.1.2} \mathrel{\widehat=} [ZOM_{inv.6.1} \mid x? \neq 0 \land z? = 0]$$

$$PA_{inv.7.1.1} \mathrel{\widehat=} [ZOM_{inv.7.1} \mid x? = 0 \land y? \neq 0]$$

$$PA_{inv.7.1.2} \mathrel{\widehat=} [ZOM_{inv.7.1} \mid x? \neq 0 \land y? = 0]$$

$$\{PA_{inv.5.1.1}, PA_{inv.5.1.2}\} = TTH_{Classify}(ZOM_{inv.5.1}, partition_analysis)$$

$$\{PA_{inv.6.1.1}, PA_{inv.6.1.2}\} = TTH_{Classify}(ZOM_{inv.6.1}, partition_analysis)$$

$$\{PA_{inv.7.1.1}, PA_{inv.7.1.2}\} = TTH_{Classify}(ZOM_{inv.7.1}, partition_analysis)$$

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

$$| \; permutation : STRATEGY$$

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

$$PERM_{xyz} \mathrel{\widehat=} [CE_{sca} \mid x? < y? < z?]$$

$$PERM_{xzy} \mathrel{\widehat=} [CE_{sca} \mid x? < z? < y?]$$

$$PERM_{yxz} \mathrel{\widehat=} [CE_{sca} \mid y? < x? < z?]$$

$$PERM_{yzx} \mathrel{\widehat=} [CE_{sca} \mid y? < z? < x?]$$

$$PERM_{zxy} \mathrel{\widehat=} [CE_{sca} \mid z? < x? < y?]$$

$$PERM_{zyx} \mathrel{\widehat=} [CE_{sca} \mid z? < y? < x?]$$

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

We have generated twenty-six test templates: one for the equilateral case, three for the isosceles, six for the scalene, and sixteen for invalid triangles (see Figure 1 for 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 depends 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 produces test-cases that check that all four outcomes are able to be generated by the software under test. Myers [11] 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.

4.1 Oracles

Oracles can be derived for test templates using the input-output relationship of the formal specification. The oracle template is defined on the output space of the operation. We define the Output Space (OS) of an operation as the signature of the operation restricted to the output variables (and the final state). For the *Classify* operation, the OS is simply

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

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

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

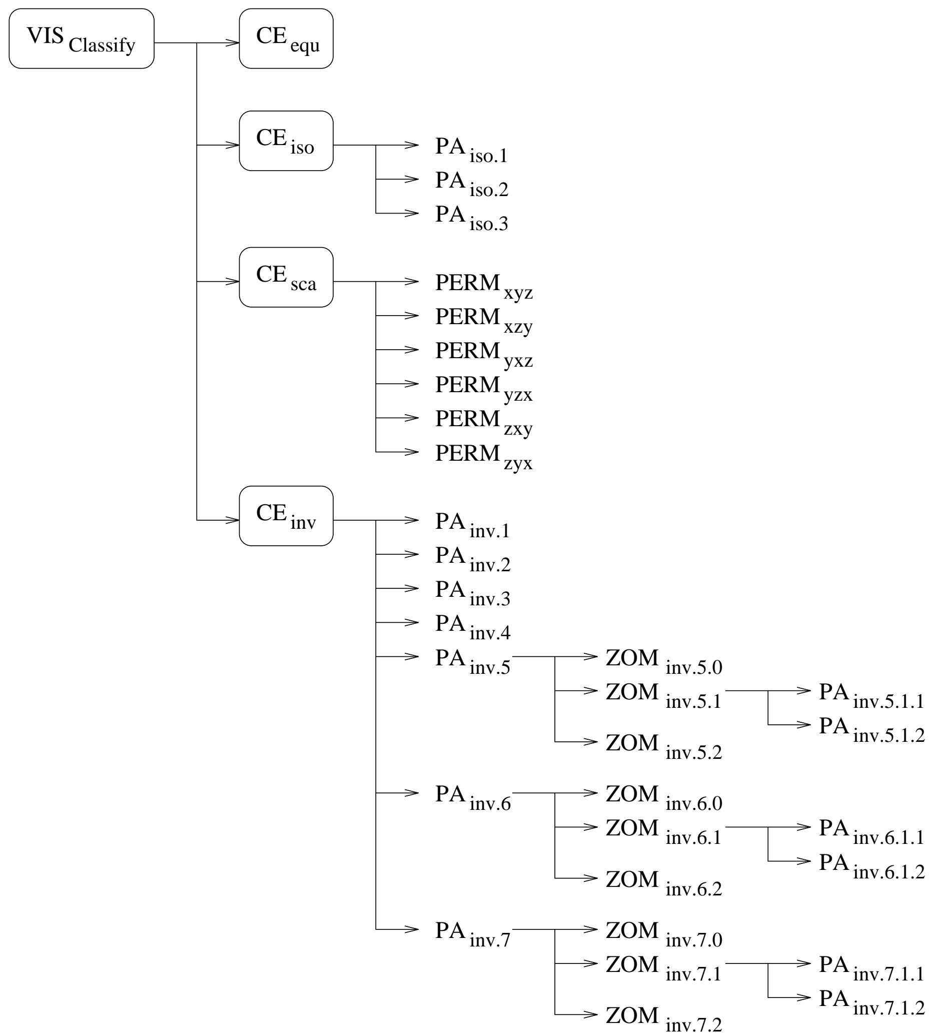

Figure 1: Test templates for the triangle classification problem.

We use the notation $oracle_{Op}(T)$ for this. For the case study, we need

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

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, for example

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

4.2 Maintenance and regression testing

When (inevitably) the specification is changed to accommodate new user requirements, it is useful to observe the benefits of the test template framework. We postulate that the user becomes interested in knowing about right-angle triangles as well. The *TRIANGLE* type is revised to

$$TRIANGLE ::= EQUILATERAL \mid RA_ISOSCELES \mid ISOSCELES \mid \\ RA_SCALENE \mid SCALENE \mid INVALID$$

The effect on the test template derivation is localised. The cause-effect strategy generates two new templates for *RA_ISOSCELES* and *RA_SCALENE*, and modifies those for *ISOSCELES* and *SCALENE*. Similar changes are required for the oracles.

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.

5 Epilogue

5.1 Formal methods practices and testing

Given the formal definition of a test suite, we can apply standard formal analysis practices to verify properties of the suite.

We can add to our definitions of strategies by expressing formal relationships amongst the derived tests. For example, we have some expectations of input partitions that can be expressed formally. Namely, that the derived domains do partition the input space. This can be expressed in the framework. Using some partitioning strategy

$$partitioning : STRATEGY$$

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:

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

$$\begin{array}{l} _\underline{disjoint}_ : TT_{Op} \leftrightarrow TT_{Op} \\ \hline \forall\, T1, T2 : TT_{Op} \bullet T1 \,\underline{disjoint}\, T2 \Leftrightarrow T1 \cap T2 = \{\} \end{array}$$

So, we can state that any templates derived using partitioning must partition the template (or space) from which they are derived:

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

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

Thus we could determine whether or not the cause-effect templates from the case study partition the valid input space of *Classify*, by proving

$$\begin{array}{l} TTH_{Classify}(VIS_{Classify}, cause_effect)\, \underline{covers}\, VIS_{Classify} \\ \wedge \\ (\forall\, T1, T2 : TTH_{Classify}(VIS_{Classify}, cause_effect) \mid T1 \neq T2 \bullet \\ \quad T1\, \underline{disjoint}\, T2) \end{array}$$

That is, by proving

$$\begin{array}{l} \{CE_{equ}, CE_{iso}, CE_{sca}, CE_{inv}\}\, \underline{covers}\, VIS_{Classify} \wedge \\ CE_{equ}\, \underline{disjoint}\, CE_{iso} \wedge \\ CE_{equ}\, \underline{disjoint}\, CE_{sca} \wedge \\ CE_{equ}\, \underline{disjoint}\, CE_{inv} \wedge \\ CE_{iso}\, \underline{disjoint}\, CE_{sca} \wedge \\ CE_{iso}\, \underline{disjoint}\, CE_{inv} \wedge \\ CE_{sca}\, \underline{disjoint}\, CE_{inv} \end{array}$$

which is straightforward.

Other properties of templates can be defined similarly to those above. Also, criteria for test suites to meet can be defined, which can be used to induce correctness based on results of testing. Another interesting formal method practice that affects testing is specification reification. The TTs 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. The specification is refined to an implementation; corresponding refinements can be made to the templates to describe (more concretely) the test data and test information for the refined specification. This simplifies test derivation in two ways. Firstly, it is usually easier to derive tests at higher levels of abstraction. Secondly, more information about the final implementation is introduced in stages, so that additional tests due to increased knowledge of structure are required in small manageable amounts, which greatly simplifies structural, or white-box, testing. Further discussion of these points can be found in [18, 16].

5.2 Testing practices and formal methods

Validation of a formal specification is important because of the central role of a specification. Several methods have been proposed for validation, including proving properties of

the specification and prototyping. The first method requires some determination of the desired properties to be proved, and the second requires test data. Our proposal for test case generation can be combined with either or both methods to assist the validation process.

Since each testing strategy typically has some general properties associated with it, proof of these properties for a particular test template hierarchy is a useful validation activity. For example, the first strategy applied in the case study is cause-effect mapping. As discussed above, we could reasonably require the cause-effect templates to cover the valid input space and to be disjoint. If the cause-effect templates do not cover the valid input space, it means that there are some causes without effects, i.e., that the operation is under-specified. If the cause-effect templates are not disjoint, it means that the same cause may produce different effects, which can often mean an error. Consider a naive extension of the case study to identify right-angle triangles without considering the impact on other types of triangle. If we simply add a new cause-effect template

$$\begin{aligned} CE_{ra} \mathrel{\widehat{=}} [\, & VIS_{Classify} \mid \\ & (x?, y?, z?) \in validTriangle \land \\ & (x?^2 = y?^2 + z?^2 \lor y?^2 = x?^2 + z?^2 \lor z?^2 = x?^2 + y?^2)] \end{aligned}$$

we can no longer prove that the cause-effect templates are disjoint.

The derived tests are a simple and different representation of the system, that with some analysis may reveal errors in the specification.

5.3 In summary...

We have defined a framework and testing methodology based on a formal methods approach to testing. The flexible framework does not exclude the techniques of other specification-based testing work. We have demonstrated the framework on a familiar case study.

Dijkstra's famous remark on software testing is often falsely interpreted to mean that testing has little or no value, and that it is irrelevant for formal methods of software development. In this paper, we have shown how the concepts of software testing can be combined with formal specifications to extend the role of the formal specification. We have also shown how formal specification techniques can systematise the application of testing strategies by defining a framework such as ours.

References

[1] Amla N, Ammann P. Using Z specifications in category partition testing. In Proceedings of COMPASS 1992, the Seventh Annual Conference on Computer Assurance, pp 3–10, 1992.

[2] Brien SM, Nicholls JE. Z base standard version 1.0. Technical report, Programming Research Group, Oxford University Computing Laboratory, Oxford University, 1992.

[3] Cusack E, Wezeman C. Deriving tests for objects specified in Z. In Bowen JP, Nicholls JE (eds), Z User Workshop. Springer-Verlag, 1993.

[4] Dick J, Faivre A. 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, 1992.

[5] Dick J, Faivre A. Automating the generation and sequencing of test cases from model-based specifications. In Woodcock JCP, Larsen PG (eds), FME'93 Industrial Strength Formal Methods, Lecture Notes in Computer Science 670, pp 268–284. Springer-Verlag, 1993.

[6] Duke R, King P, Rose G, Smith G. The Object-Z specification language version 1. Technical Report 91-1, Software Verification Research Centre, The University of Queensland, Queensland 4072, Australia, 1991.

[7] Hall PAV. Towards testing with respect to formal specifications. In Second IEE/BCS Conference on Software Engineering 88, pp 159–163. IEE, 1988.

[8] Hall PAV. Relationship between specifications and testing. Information and Software Technology, 33(1):47–52, 1991.

[9] Hayes IJ. Specification directed module testing. IEEE Transactions on Software Engineering, 12(1):124–133, 1986.

[10] Laycock G. Formal specification and testing: A case study. Journal of Software Testing, Verification and Reliability, 2(1):7–23, 1992.

[11] Myers GJ. The Art of Software Testing. Business data processing. Wiley-Interscience, 1979.

[12] Ostrand TJ, Balcer MJ. The category-partition method for specifying and generating functional tests. Communications of the ACM, 31(6):676–686, 1988.

[13] Richardson DJ, Aha SL, O'Malley TO. Specification-based test oracles for reactive systems. In Proceedings of the 14th International Conference on Software Engineering, pp 105–118, 1992.

[14] Richardson DJ, O'Malley O, Tittle C. Approaches to specification-based testing. Software Engineering Notes, 14(8):86–96, 1989. Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3).

[15] Spivey JM. The Z Notation: A Reference Manual. Series in Computer Science. Prentice Hall International, second edition, 1992.

[16] Stocks P. Applying formal methods to software testing. PhD thesis, The University of Queensland, 1993. (Under examination).

[17] Stocks P, Carrington DA. Test template framework: A specification-based testing case study. In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'93), pp 11–18, 1993.

[18] Stocks P, Carrington DA. Test templates: A specification-based testing framework. In Proceedings of the 15th International Conference on Software Engineering, pp 405–414, 1993.

[19] Wezeman CD. The CO-OP method for compositional derivation of canonical testers. In Brinksma E, Scollo G, Vissers CA (eds), Protocol Specification, Testing and Verification IX. North Holland, 1990.