A Framework for Specification-Based Testing

Phil Stocks and David Carrington, Member, IEEE Computer Society

Abstract—Test templates and a test template framework are introduced as useful concepts in specification-based testing. The framework can be defined using any model-based specification notation and used to derive tests from model-based specifications—in this paper, it is demonstrated using the Z notation. The framework formally defines test data sets and their relation to the operations in a specification and to other test data sets, providing structure to the testing process. Flexibility is preserved, so that many testing strategies can be used. Important application areas of the framework are discussed, including refinement of test data, regression testing, and test oracles.

Index Terms—Specification-based testing, testing strategies, test data, test oracles, Z notation.

1 INTRODUCTION

Testing plays a vital role in software development. Testing is a practical means of detecting program errors that 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, it will always be a necessary verification technique [2].

Formal methods are useful 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 [3]. 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 Goedenough and Gerhart note, testing based only on program implementation is fundamentally flawed [4]. Despite this, only a small portion of the testing literature deals with specification-based testing issues.

This paper examines applications of formal methods to software testing. The formal specification of a software product can be used as a guide for designing functional tests of 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 pinpointing errors and can simplify regression testing. An important application of specifications in testing is providing test oracles [5]. The specification is an authoritative description of system behavior 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, auditing the testing process, and developing tests concurrently with design and implementation. The latter is also useful for breaking “code now/test later” practices in software engineering, and helping develop parallel testing activities for all software life-cycle phases as advocated in [6].

Research on testing from model-based specifications (using VDM [7] or the Z notation [8], [9]) has followed increased interest in this style of specification. Hall [10], [11], an early proponent of specification-based testing, shows how to derive tests from a Z specification. Dick and Faivre [12] use disjunctive normal form (DNF) as the basis of partitioning testing from a VDM specification. Hörcher and Peleska [13], [14] apply the DNF method to Z and extend it to build sequences of test cases. Another approach is demonstrated by Stepney [15] for an object-oriented extension to Z.

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.

1.1 Defining Test Suites

Most research in testing focuses on specific testing subtasks 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 such as functional units, tests and oracles, and establish dependencies.

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

• P. Stocks is with the Department of Computer Science, Rutgers University, Hill Center, Busch Campus Piscataway NJ 08855. E-mail: pstocks@ics.rutgers.edu.
• D. Carrington is with the Department of Computer Science, The University of Queensland, St. Lucia QLD 4072, Australia. E-mail: davec@cs.uq.edu.au.

Manuscript received Mar. 20, 1996; revised Nov. 4, 1996. Recommended for acceptance by J. Gannon.

For information on obtaining reprints of this article, please send e-mail to: trans@computer.org, and reference IEEECS Log Number S9638.

0098-5589/96$05.00 ©1996 IEEE
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.

1.1.1 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 some method is needed to define tests so that all the relevant information is captured.

TSL, Test Specification Language, is a notation used for defining tests derived using category partitioning [18], [19]. Test scripts indicating relevant inputs to operations and global values are derived from specifications. Test scripts also indicate appropriate value selections for the inputs and globals called choices. A test frame is one combination of selections of choices for inputs and globals. Test frames are automatically generated by essentially calculating the cross product of the choices over all the inputs and globals. The intuition of category partitioning is that choices make interesting partitions of values of inputs and globals, and that considering different combinations of choices tests interaction of features in the system. Additional text can be added around test frames to put the frames in a format suitable for execution. More tailored combinations of inputs can be generated, but the specifications required to generate them become unwieldy and counter-intuitive.

A contrasting approach to test specification is taken by Pachl [20], 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.

1.1.2 Structuring Tests
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 this test and other tests, the function being tested, the oracle, and possibly other considerations.

This undervalued area of research is addressed by Ostrand et al. [21]. Their tool for specification-based testing, SPECMAN, assists in structuring tests by acting as record-keeping support for constructing two tables: 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 informal 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.

1.2 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.

Ensuring that the specification is correct (i.e., specification validation) is perhaps the most crucial task in any software engineering project. Unfortunately, the specification can only be compared to 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, beyond detailed inspection.

Kemmerer [22] discusses an approach whereby specifications are made executable and run on some test cases. Kneuper [23] 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.

1.3 Contribution of this Paper
As can be seen, the elements of specification-based testing are varied. Most have received 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 work.

The major contribution of this paper 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. Pachl's method focuses only on test selection criteria and does not introduce sufficiently general criteria to express any arbitrary test requirement formally. TSL also loses the ability to provide a concise, abstract definition of tests when the desired test criteria depart from the category-partition model. The SPECMAN tool does concentrate on the collection of tests, but does not consider oracle information or a formal uniform representation of information. Most specification-based test derivation methods have some implicit notion of test definition, but are 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.

The framework uses the Z notation as a test description language and to define the framework. Appendix A provides an introduction for readers who are unfamiliar with the Z notation. The uniform use of Z improves the clarity and structure of the test information. It would also be possible to define the framework using other model-based specification notations. We feel that not having to introduce and force users to learn a new notation is an important feature of the framework. Another advantage is being able
to define specification-based test suites in the same notation as the specification.

The next section introduces the framework, defining the concepts of test template and test template hierarchy. Section 3 presents a symbol table specification as an example and demonstrates how the test template framework is used to derive abstract tests. In Section 4 additional roles for the framework are discussed. These include constructing actual test data, analyzing test suites and assisting specification validation. Section 5 considers future work that is possible with the framework.

2 THE TEST TEMPLATE FRAMEWORK

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

2.1 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.

The input space (IS) of an operation is the space from which input can be drawn, and is defined as the restriction of the operation’s signature to input components (inputs and prestate components). Essentially, the input space represents type-compatible inputs to the operation. The output space (OS) is similarly defined over the output signature.

For example, an operation for removing an element from the front of a sequence

\[ \text{Remove}[\text{ELEMENT}] \]

\[
\begin{align*}
  &s? : \text{seq ELEMENT} \\
  &\text{el} : \text{ELEMENT} \\
  &s? = (\text{el}) \setminus s!
\end{align*}
\]

has the input space
\[ [s? : \text{seq ELEMENT}] \]

and the output space
\[ [\text{el} : \text{ELEMENT}; s! : \text{seq ELEMENT}] \]

We choose to define the input space in terms of the types used in the specification. An alternative approach is to use only the primitive type constructors in the specification notation. For the example, this would give an input space of \([s? : \text{IN}(\mathbb{Z} \times \text{ELEMENT})]\). Fortunately the choice does not affect the framework which is based on the valid input space (defined in the next section).

2.1.1 Valid Input Space

If data is type-compatible with an operation, it may still not be 'sensible' input for the operation: 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.

Consider the \text{Remove} operation above; it is not defined when the initial sequence is empty.

The valid input space (VIS) is 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 precondition 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 integer value one closer to zero

\[ \text{ToZero} \]

\[
\begin{align*}
  &x?, x! : \mathbb{Z} \\
  &\left( x? < 0 \land x! = x? + 1 \right) \lor \\
  &\left( x? > 0 \land x! = x? - 1 \right)
\end{align*}
\]

The input space is
\[ [x? : \mathbb{Z}] \]

but the operation is not defined when \( x? = 0 \) (i.e., \( \neg (x? < 0 \lor x? > 0) \)) so the valid input space is
\[ [x? : \mathbb{Z} \mid x? < 0 \lor x? > 0] \]

2.1.2 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 because the specification defines only what happens for input in the valid input space. No statement can be made about what the operation does when given input outside the valid input space. In the \text{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 cannot 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 behavior. So, nothing can be determined by exercising the operation on input outside the valid input space. However, such inputs are of interest for specification validation and are discussed in Section 4.3.

2.1.3 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 this discussion is deferred to Section 4.3.

2.2 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. Most important is defining the classes of re-
requirements that test data must satisfy. Once these classes of requirements are defined, any actual input satisfying them is appropriate test data.

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. We use Z schemas to model test templates. For example,

$$A_{\text{template}} \models [x?, y? : N | x? < y?]$$

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 a set 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.

### 2.2.1 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 does 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 in Section 2.1.2, 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, $Op$:

$$TT_{Op} = P \text{VIS}_{Op}$$

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

### 2.3 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 [24], [25].

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. In theory, software testing attempts to infer the correctness of a program on all inputs based on the results of a small number of inputs; the result of each test needs to generalize to a whole subset of the input for this to work [4], [11]. To meet this goal, one must derive domains which are equivalence classes of error-detecting ability for the function under test, and which cover the valid input space. That is, domains must be chosen so that each element of a domain has the same error-detecting ability, and so the result of testing one element of the domain applies to all elements of the domain. 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 subdomains, until the tester is satisfied that the domains represent desired equivalence classes.

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 Fig. 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 subtemplates derived represented finer divisions of the parent template. The terminal nodes in a hierarchy represent the final test classes as determined by the human tester.

Some strategies (e.g., random testing [26]) do not advocate domain partitioning and hence 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 does not enforce particular derivation approaches on the tester.
2.3.1 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 define subsets 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:

\[
\text{STRATEGY}
\]

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:

\[
\text{TTH}_o : TT_{op} \times \text{STRATEGY} \rightarrow \mathbb{P} \text{TT}_{op}
\]

Templates are defined in terms of their parents and additional constraints. For example, a template, \( T_1 \), derived from \( VIS_{op} \) with the additional constraint \( cst \) is defined by

\[
T_1 \equiv [VIS_{op}, cst]
\]

If the strategy used in this derivation was \( strat \), then \( T_1 \)'s position in the hierarchy can be described by

\[
T_1 \in \text{TTH}_{op}(VIS_{op}, strat)
\]

If \( T_1 \) is the only template derived from the valid input space using \( strat \), then this section of the hierarchy can be completely defined by

\[
\{T_1\} = \text{TTH}_{op}(VIS_{op}, strat)
\]

Useful relationships among templates, based on the structure of the hierarchy, can be defined. We define two standard functions over templates in a hierarchy: \( \text{children}_{op} \) and \( \text{descendants}_{op} \).

\[
\text{children}_{op} : \text{TT}_{op} \rightarrow \mathbb{P} \text{TT}_{op}
\]

\[
\text{children}_{op} = (\lambda T : \text{TT}_{op} \cdot \bigcup \{S : \text{STRATEGY} \cdot \text{TTH}_{op}(T, S)\})
\]

\[
\text{descendants}_{op} : \text{TT}_{op} \rightarrow \mathbb{P} \text{TT}_{op}
\]

\[
\text{descendants}_{op} = (\lambda T : \text{TT}_{op} \cdot \bigcup \{T_2 : \text{children}_{op}(T) \cdot \text{descendants}_{op}(T_2)\})
\]

The function \( \text{children}_{op} \) determines the set of templates directly derived from some template using any strategy. For example, given the hierarchy in Fig. 1,

\[
\text{children}_{\text{fig1}}(VIS) = \{T_{a_1}, \ldots, T_{a_m}, \ldots, T_{b_1}, \ldots, T_b\}
\]

The function \( \text{descendants}_{op} \) 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 subgraph extending from that template. For example, given the hierarchy in Fig. 1,

\[
\text{descendants}_{\text{fig1}}(VIS) = \{T_{a_1}, \ldots, T_{a_m}, \ldots, T_{b_1}, \ldots, T_b, T_{a_1}, \ldots, T_{a_m}, \ldots, T_{b_1}, \ldots, T_b\}
\]

2.4 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.

It should 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

\[
\text{QT1} \equiv [q : \text{seq} \mid \#q = 2 \times \#(\text{ran} q) = 1]
\]

Any instances of this template expressed in \( Z \) describe specific \( Z \) sequences (e.g., \( (1,1) \)), 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.
Our preferred approach to describing instances is to define instance templates since this increases the uniformity within the framework. These are merely templates (schemas) with only one possible instantiation. The instance template corresponding to the instance of QT1 described above is simply defined as:

Q ∋ [QT1 | q = (1, 1)]

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:

| instantiation : STRATEGY |

2.5 Oracles

The formal specification does more than describe conditions on the input. The relationship between input states and output states is specified precisely. 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 precalculated 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 it 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.

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 nondeterministic.

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

\[
(Op \land T) \mid OS_{Op}
\]

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. We use the expression \( oracle_{Op} \) to represent this. For example, with the test template \( T1 \)

\[
oracle_{Op}(T1) = (Op \land T1) \mid OS_{Op}
\]

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:

| A_schema |
|-----------------|-----------------|
| state, state' : STATE |
| in? : IN |
| out! : OUT |

\[
StateComp(A\_schema) = (state, state') \]
\[
ParamComp(A\_schema) = (in?, out!)
\]

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.

\[
State(Op) = Op \setminus ParamComp(Op) \]
\[
Params(Op) = Op \setminus StateComp(Op)
\]

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:

\[
\forall T : descendants_{NoChange}(VIS_{NoChange}) \bullet
State(oracle_{NoChange}(T)) = State(T)
\]

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.

3 Example: A Symbol Table

3.1 Z Specification

A block-structured symbol table is specified in [8] in two parts. The first part introduces a basic table with associated operations. The second part defines a block-structured symbol table as a collection of these tables, defined to handle scoping and visibility. The entire case study is too large to present here. We discuss testing the simple table specification, as it is representative of the issues involved in using our framework.

The symbol and value types are given sets.

\([SYM, VAL]\)

The table is represented as a partial function from symbols to values.

\[
ST
\]

\[
st : SYM \rightarrow VAL
\]

The table is initially empty.

3. This approach relies on the well-established Z conventions that name schema variables to distinguish inputs, outcomes, initial states and final states.

4. This example appeared in [27].
The table operations are defined in three parts. Their basic functionality is defined first, then any error conditions are defined along with success or failure messages. Finally, the complete operation is defined by combining the basic schema with the success and error schemas. By convention, $\Delta$ in an operation schema represents a change of state, and $\Xi$ represents access with no change of state.

$\text{Update}$ adds a new mapping to the table. If the symbol is already in the table, its corresponding value is replaced by the new value.

$$\text{Update} \triangleq \text{success (st$'$ = st$' \uplus \{s? \mapsto v?\})}$$

$\text{LookUp}$ indexes the table mapping on the given symbol, returning the corresponding value. The symbol must be in the table. There is no change of state.

$$\text{LookUp} \triangleq \Xi \text{st}$$

$\text{Delete}$, given a symbol, removes the mapping for that symbol from the table. The symbol must be in the table.

$$\text{Delete} \triangleq \Delta \text{st}$$

The above specifications describe the correct behavior of the system, but are incomplete. The following definitions describe system behavior in error cases. Some sort of report is returned, based on whether a symbol is found in the table or not.

$$\text{REPORT} ::= \text{ok} \mid \text{symbol\_not\_present}$$

$\text{Success}$ issues a report for the correct behavior cases defined above.

$\text{NotPresent}$ issues a report indicating that the symbol is not in the table and allows no change of state.

$$\text{NotPresent} \triangleq \Xi \text{st}$$

$$\text{Update} \triangleq \text{success (st$'$ = st$' \uplus \{s? \mapsto v?\})}$$

$$\text{LookUp} \triangleq (\text{LookUp} \triangleq \text{success}) \lor \text{NotPresent}$$

$$\text{Delete} \triangleq (\text{Delete} \triangleq \text{success}) \lor \text{NotPresent}$$

### 3.2 TTF Testing

We apply the framework to derive tests and oracles from the table specification. For simplicity, we focus on testing the $\text{Update}$ operation.

#### 3.2.1 Input Spaces

The fully expanded definition of $\text{Update}$ is

$$\text{Update}$$

$$\text{st, st'} : \text{SYM} \rightarrow \text{VAL}$$

$$\text{st} \in \text{dom st}$$

$$\text{st}' = \text{st} \uplus \{s? \mapsto v?\}$$

$$\text{rep}' = \text{REPORT}$$

So, the IS is

$$\text{IS}_{\text{up}} \triangleq \{\text{st} : \text{SYM} \rightarrow \text{VAL}; \text{s?} : \text{SYM}; \text{v?} : \text{VAL}\}$$

The precondition of $\text{Update}$ is used to determine the VIS. In this case, the precondition simplifies to $\text{true}$, so the VIS defines the same space as the IS.

$$\text{VIS}_{\text{up}} \triangleq \text{IS}_{\text{up}}$$

#### 3.2.2 Functional Testing

We apply a range of common testing strategies to derive functional tests for $\text{Update}$. Any strategy could be applied. A good starting point is a cause-effect method that partitions the valid input space based on equivalence classes of the output space. However, there are no obvious partitions.
Domain propagation is a specification-based testing strategy we developed for use with model-based specifications. Both it and another strategy we developed, specification mutation, are adaptations of testing strategies and ideas for implementation-based testing and are described more fully in [17], [27]. Domain propagation attempts to capture the notion of branch coverage in a useful way for specification-based testing. A standard approach in specification-based testing is to reduce the specification to disjunctive normal form and choose inputs satisfying the preconditions of each disjunct. This tends to be too simplistic because model-based specifications are generally quite flat, and because specification languages have powerful operators built into the notation which hide the complexity of the input domain from a disjunctive normal form transformation. Domain propagation, then, recognizes that operators and user-defined functions have input domain divisions associated with them, and propagates these divisions to the level in which the operator or function is used by expressing the divisions in terms of the operands. For example, absolute value on integer input, say \( x \), has domains \( x < 0 \) and \( x \geq 0 \). Anywhere absolute value is used, these domains can be expressed in terms of the higher level component arguments. Thus, the input space at the higher level can be partitioned. Essentially, domain propagation is intended to be an "honest" disjunctive normal form partitioning for model-based specifications. Domain propagation is used as a heuristic strategy within our framework.

Domain propagation can be usefully applied to the simple predicate \( st' = st \oplus (s? \Rightarrow v?) \) in Update. The other predicate (\( repl = ok \)) has no interesting subdomains. The definition of functional overriding (\( \oplus \)) is

\[
 f \oplus g = (dom \ g \not= f) \cup g
\]

First, we consider propagation of the domains of set union. For the union of two sets, say \( S \) and \( T \), we consider the following combinations as interesting domain divisions:

1. \( S = \{} \land T = \{} \)
2. \( S = \{} \land T \neq \{} \)
3. \( S \neq \{} \land T = \{} \)
4. \( S \neq \{} \land T \neq \{} \land S \cap T = \{} \)
5. \( S \neq \{} \land T \neq \{} \land S \cap T = \}
6. \( S \neq \{} \land T \neq \{} \land T \subset S \)
7. \( S \neq \{} \land T \neq \{} \land S = T \)
8. \( S \neq \{} \land T \neq \{} \land S \cap T \neq \{} \land (S \subset T) \land (T \subset S) \land S \neq T \)

Substituting \( (dom \ g \not= f) \) for \( S \) and \( g \) for \( T \) contradicts cases 5, 6, 7, and 8 because \( S \) and \( T \) are disjoint. Thus, we have the first four possibilities.

Going further, we examine the domain subtraction (\( - \)) for domain propagations. For some set \( X \) and some relation \( R \), we consider the following domain divisions of \( X \) \( \preceq \) \( R \):

1. \( R = \{} \)
2. \( R \neq \{} \land X = \{} \)
3. \( R \neq \{} \land X \cap dom \ R \)

Unlike LookUp which clearly divides the VIS into two subsets: \( s? \in dom \ st \) and \( s? \not\in dom \ st \).

4. \( R \neq \{} \land X \neq \{} \land X \subset dom \ R \)
5. \( R \neq \{} \land X \neq \{} \land X \cap dom \ R = \{} \)
6. \( R \neq \{} \land X \cap dom \ R \neq \{} \land X \subset dom \ R \)

The domains propagated by the functional overriding are obtained by substituting \( X \preceq R \) for \( S \) in the surviving cases of the set union domain propagation, using all the above possibilities for domain subtraction. This yields the following combinations, after disregarding contradictions:

1. \( f = \{} \land g = \{} \)
2. \( f = \} \land g \neq \{} \)
3. \( f \neq \{} \land g \neq \{} \land dom \ g = dom \ f \)
4. \( \neq \} \land g \neq \{} \land dom \ g \subset dom \ f \)
5. \( \neq \} \land g \neq \{} \land dom \ g \cap dom \ f = \{} \)
6. \( \neq \} \land g \neq \} \land dom \ g \cap dom \ f \neq \} \land (dom \ g \subseteq dom \ f) \)

Finally, by substituting the values for \( f \) and \( g \) from Update, and eliminating any contradictions, we have the input partitions derived from domain propagation. We know \( f = st \) and \( g = (s? \Rightarrow v?) \), so cases 1.1 and 3.2 are eliminated because \( g \) cannot be empty, and case 4.6 is eliminated because \( dom \ g \) has only one element. Thus, given domain propagation as a strategy

\[
 \text{dom\_prop: STRATEGY}
\]

we derive the following test templates from the VIS

\[
\begin{align*}
DP_{2,1} &\overset{\Delta}{=} [VIS_{up}\{st = \{\}\}] \\
DP_{2,3} &\overset{\Delta}{=} [VIS_{up}\{s? \Rightarrow v?\} = dom \ st] \\
DP_{4,4} &\overset{\Delta}{=} [VIS_{up}\{s? \Rightarrow v?\} \subset dom \ st] \\
DP_{4,5} &\overset{\Delta}{=} [VIS_{up}\{st \neq \{} \land (s? \Rightarrow v?) \cap dom \ st = \{\}] \\
(DP_{2,1}, DP_{2,3}, DP_{4,4}, DP_{4,5}) &= \text{TTH}_{up}(VIS_{up}, \text{dom\_prop})
\end{align*}
\]

### 3.2.4 Type-Based Selection

We develop further templates by using type-based selection. Interesting values for each component are chosen based on its type. For example, interesting cases for a set are the empty set, a singleton set, and a nonempty, nonsingleton set. The types SYM and VAL are, as yet, unspecified so we cannot use type-based selection on \( s? \) and \( v? \). The symbol table map is essentially a set, so we have the following possibilities for values of \( st \):

1. \( \#st = 0 \)
2. \( \#st = 1 \)
3. \( \#st \geq 2 \)

Of the domain propagation templates, only \( DP_{4,5} \) allows any freedom of selection of \( st \) with regard to these choices of size, so we derive the following templates:

\[
\begin{align*}
\text{type\_based: STRATEGY} \\
TB_{4,5,2} &\overset{\Delta}{=} [DP_{4,5}\{\#st = 1\}] \\
TB_{4,5,3} &\overset{\Delta}{=} [DP_{4,5}\{\#st \geq 2\}] \\
(TB_{4,5,2}, TB_{4,5,3}) &= \text{TTH}_{up}(DP_{4,5}, \text{type\_based})
\end{align*}
\]
3.3 Instantiation and Instance Templates

After applying all the desired strategies, the template hierarchy is considered complete. If no further subdivision of templates is to be undertaken, it can be assumed that each instance of the terminal templates in the hierarchy graph is equivalent to all others for testing purposes. For a complete description of the test data, the only remaining task is to instantiate the terminal templates in the hierarchy.

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 TTH:

\[
\text{instantiation} : \text{STRATEGY}
\]

For this example, we use characters for the data type SYM and natural numbers for the data type VAL, and derive the following concrete instance templates.

\[
\begin{align*}
T1 & \doteq [DP_{2,1} \mid s ? = 'a' \land v ? = 1] \\
T2 & \doteq [DP_{2,3} \mid st = \{ 'a' \mapsto 1 \} \land s ? = 'a' \land v ? = 2] \\
T3 & \doteq [DP_{4,4} \mid st = \{ 'a' \mapsto 1, 'b' \mapsto 1 \} \land s ? = 'a' \land v ? = 2] \\
T4 & \doteq [TB_{4,5,2} \mid st = \{ 'a' \mapsto 1 \} \land s ? = 'b' \land v ? = 2] \\
T5 & \doteq [TB_{4,5,3} \mid st = \{ 'a' \mapsto 1, 'b' \mapsto 1 \} \land s ? = 'c' \land v ? = 2] \\
\end{align*}
\]

\[
\{ (DP_{2,1}, \text{instantiation}) \mapsto \{ T1 \}, \ (DP_{2,3}, \text{instantiation}) \mapsto \{ T2 \}, \ (DP_{4,4}, \text{instantiation}) \mapsto \{ T3 \}, \ (TB_{4,5,2}, \text{instantiation}) \mapsto \{ T4 \}, \ (TB_{4,5,3}, \text{instantiation}) \mapsto \{ T5 \}, \} \subset \text{TTH}_{\text{up}}
\]

The relationships between templates in the test template hierarchy for the Update operation are shown in Fig. 2.

\[
\begin{align*}
\text{DP}_{2,3} & \longrightarrow T1 \\
\text{DP}_{2,3} & \longrightarrow T2 \\
\text{DP}_{4,4} & \longrightarrow T3 \\
\text{DP}_{4,5} & \longrightarrow \text{TB}_{4,5,2} \longrightarrow T4 \\
\text{TB}_{4,5,2} & \longrightarrow T5 \\
\text{VIS}_{\text{Update}} & \\
\end{align*}
\]

Fig. 2. Graphical relationship between test templates for Update.

3.4 Oracles

The output space (OS) of Update is

\[
\text{OS}_{\text{up}} \doteq [st' : \text{SYM} \rightarrow \text{VAL}; \text{rep}! : \text{REPORT}]
\]

We use the expression \(\text{oracle}_{\text{up}}\) to represent the restriction of an operation's input to that defined in the test template projected onto the output space. For example

\[
\text{oracle}_{\text{up}}(T1) = (\text{Update} \land T1) \big| \text{OS}_{\text{up}}
\]

That is,

\[
\text{oracle}_{\text{up}}(T1) = [\text{OS}_{\text{up}} \mid st' = \{ 'a' \mapsto 1 \} \land \text{rep}! = \text{ok}]
\]

This section has demonstrated how the test template framework can be applied to the update operation of a simple symbol table. Further examples can be found in [16], [27], [17], [3], [28].

4 DISCUSSION

With the framework, we derive abstract descriptions of test suites as hierarchies of test cases derived using various strategies. This section 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 section also looks at how the formal basis of the framework assists in analyzing the testing, and uses of the framework in other phases of the software life-cycle.

4.1 Constructing Actual Test Suites

The framework generates abstract test hierarchies which are descriptions of test suites. At some stage in testing, actual tests that can be executed and evaluated need to be produced. 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.

4.1.1 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 refined 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.

First, 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.

Second, 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 structural test cases, which is easier than selecting
structural tests based on the whole implementation, and ensures that the structural tests are developed in an appropriate context. Indeed, reification illustrates the limitations of a functional-structural dichotomy.

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 [9]: 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 conjointly 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 symbol table example of Section 3.

Consider implementing the symbol table by two arrays (each modeled as a Z sequence).

\[
\begin{align*}
&\text{\textit{st} }^R \\
&\text{symb} : \text{seq SYM} \\
&\text{vals} : \text{seq VAL} \\
&\\text{#symb} = \#\text{vals} \\
&\\text{#symb} = \#\text{ran symb}
\end{align*}
\]

The relation between abstract and concrete symbol tables in this case is

\[
\begin{align*}
&\text{abs} \\
&\text{st} \\
&\text{st}^R \\
&\text{st} = \text{sym}^z \text{; vals}
\end{align*}
\]

where the abstract table is defined in terms of the concrete variables using relational inversion (\(\sim\)) and relational composition (\(\cdot\)).

To see how this reification can be applied to the test templates, consider \(DP_{4,5}\) derived in Section 3:

\[
D_{4,5} = \left[ \text{VIS}_{up} \mid s \neq 1 \wedge \{s\} \cap \text{dom } st = \{1\} \right]
\]

Using Abs, the template reified from the abstract template \(DP_{4,5}\) (also represented by the superscript R on the name) is

\[
DP^R_{4,5} = (DP_{4,5} \wedge Abs) \ \backslash \ (st)
\]

That is,

If test derivation is conducted separately on the reified specification (\(Update^z\)), one of the templates produced using domain propagation analysis is equivalent to \(DP^R_{4,5}\). The same is true of oracle templates. Graphically, these relationships are shown in Fig. 3.

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 [29] 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.

4.1.2 Notes on Abstract Test Suites

These examples show two strengths of the framework's abstract test specifications. First, 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. Second, 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.

4.2 Analysis

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.

4.2.1 Properties of Templates from Strategies

First, 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 actually do partition the input space from 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:

\[-covers - : P \tau TTo \leftrightarrow \tau TTo\]
\[\forall \text{SetofTT}: P \tau TTo; T: \tau TTo \bullet\]
\[\text{SetofTT covers } T \Rightarrow \bigcup \text{SetofTT} = T\]

\[-disjoint - : \tau TTo \leftrightarrow \tau TTo\]
\[\forall T1, T2: \tau TTo \bullet\]
\[T1 \text{ disjoint } T2 \Rightarrow T1 \cap T2 = \{\}\]

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

\[\text{partitioning : STRATEGY}\]

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

\[\forall \text{Space}: \tau TTo \bullet\]
\[\text{TH}_{\text{Op}}(\text{Space, partitioning}) \text{ covers } \text{Space}\]

\[\forall \text{Space}: \tau TTo \bullet\]
\[\left(\forall T1, T2: \text{TH}_{\text{Op}}(\text{Space, partitioning})\right)\]
\[T1 \neq T2 \bullet T1 \text{ disjoint } T2\]

Consider the symbol table case study from Section 3 where four templates were derived from the VIS.

\[DP_{2.1} \triangleq [VIS_{\text{Op}} \mid st = \{\}]\]
\[DP_{2.3} \triangleq [VIS_{\text{Op}} \mid \{s?\} = \text{dom } st]\]
\[DP_{4.4} \triangleq [VIS_{\text{Op}} \mid \{s?\} \subset \text{dom } st]\]
\[DP_{4.5} \triangleq [VIS_{\text{Op}} \mid st \neq \{\} \wedge \{s?\} \cap \text{dom } st = \{\}]\]

To check that the domain propagation templates do indeed partition the VIS, we need to check the following properties

\[\text{TH}_{\text{Update}}(\text{VIS}_{\text{Op}}, \text{dom}_\text{prop}) \text{ covers } \text{VIS}_{\text{Op}}\]
\[\forall T1, T2: \text{TH}_{\text{Update}}(\text{VIS}_{\text{Op}}, \text{dom}_\text{prop}) \mid T1 \neq T2 \bullet\]
\[T1 \text{ disjoint } T2\]

That is, we need to show that

\[\bigcup \{DP_{2.1}, DP_{2.3}, DP_{4.4}, DP_{4.5}\} = \text{VIS}_{\text{Op}} \wedge\]
\[DP_{2.1} \cap DP_{2.3} = \{\} \wedge DP_{2.1} \cap DP_{4.4} = \{\} \wedge\]
\[DP_{2.3} \cap DP_{4.5} = \{\} \wedge DP_{4.4} \cap DP_{4.5} = \{\} \wedge\]

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.

These properties can be used to analyze other strategies. Consider cause-effect testing. Certainly, cause-effect templates should be disjoint, unless the operation is intended to be nondeterministic. 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.

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

\[\text{notcovered} : \tau TTo \rightarrow \tau TTo\]
\[\text{notcovered} = (\lambda T: \tau TTo \bullet\]
\[T \setminus \bigcup (\text{children}_{\text{Op}}(T))\]

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 enforce the entire input domain to be covered. For example, with an ideal, revealing domain-partitioning [24], 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 help detect incorrect use of strategies when defining templates. Expression of such properties also increases our understanding of strategies. It is possible to build a library of common template proper-
ties, such as the relations defined above, which can be used to show properties of templates derived using certain strategies.

### 4.2.2 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 Goode-nough and Gerhart [4] depend on analyzing 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 cannot 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. Nevertheless, some criteria are expressible. For example, Weyuker and Jeng [30] list 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

\[ \forall T \in \mathbb{T}_{Op} \cdot \\
(\forall c_1, c_2 : \text{children}_{Op}(T) \cdot \\
\#c_1 = \#c_2 \land \\
\left( \mathbb{T}_{Op}(c_1, \text{instantiation}) = \mathbb{T}_{Op}(c_2, \text{instantiation}) \right) \]

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

Parrish and Zweben [31] study test data adequacy criteria in detail. They refine the adequacy criteria proposed in the literature to seven independent criteria. All but one of these properties would be widely considered simple common sense properties, such as that an empty set of tests is inappropriate and that adding more tests does not reduce the effectiveness of the tests. These fundamental properties are very important, of course! The other property they mention is that the test set should achieve def-use coverage of the program. This is not a property that can be extended to specification-based testing, as there is no way to determine its satisfaction by examining only the specification. However, perhaps a suitable replacement can be found. Many works on specification-based testing [12], [14], [32] look for an equivalence partitioning based on the disjunctive normal form of the operation's specification. This is a kind of branch coverage of the specification, and seems like a sensible criterion. The domain propagation strategy discussed in Section 3.2 satisfies this criterion. In fact, it satisfies it better than does a top-level reduction to disjunctive normal form since it also considers "branches" of suboperations.

### 4.3 Specification Validation

As mentioned in Section 1.1, 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 expected 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 for 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 precondition 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 summarizes all the legal output from the operation and may help the specifier spot 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. Again, the hope is that this summary of unspecified behavior helps 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 are not well tested, and still be detected this way [33]. 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 focused form of review.

4.4 Maintenance

After maintenance changes to a system, tests must be rerun and new tests derived to ensure the correctness of the changes. Assuming the changes involve updating the specification and proceeding from there as is widely advocated, the TTF can be used in the rederivation of tests. There is no automatic statement of tests which can be rerun and new tests. Rather, the TTF’s grounding in the specification and structured approach simplifies the task. Any templates which are unchanged in the new derivation are tests that can be rerun.

This section describes a small modification to the symbol table to give it an upper size limit, and the corresponding changes to the TTs. This is not a realistic example of maintenance, but is indicative of the issues involved.

4.4.1 Changes to Specification

We introduce a constant Max and define a new state schema:

| Max : N |

ST2 ⊨ [ST | #st ≤ Max]

The revised Update operation is defined by replacing ∆ST by ∆T2 which alters its precondition.

4.4.2 Changes to Tests

A VIS for the revised Update needs to be derived. It would seem that the new VIS would just add the extra predicate, but care is required. The VIS is the precondition of the operation. The precondition of Update is the set of input states for which the operation does not fail. It assumes the existence of a table in the post-state with no more than Max elements. Hence, if a new symbol will change the size of the table, the initial size of the table must be strictly less than Max. The size of the symbol table is increased only when s? is not in the domain of the table. The VIS for the revised Update is

\[
\text{VIS}_\text{up} \triangleq [\text{IS}_\text{up} ]
\]

\[
s? \in \text{dom } st \land #st \leq Max \\
    s? \notin \text{dom } st \land #st < Max
\]

Now, we obviously have two cause-effect partitions of the VIS:

| cause_effect : STRATEGY |

CE1 ⊨ [VIS_up ] s? ∈ dom st ∧ #st ≤ Max

CE2 ⊨ [VIS_up ] s? ∈ dom st ∧ #st < Max

[CE1, CE2] = TTH_up(VIS_up, cause_effect)

The TTs derived previously are still valid, though their derivation should stem from these new templates. Thus, from each of CE1 and CE2, the four domain propagation templates can be derived as before. Here, we must be aware of the implicit constraints that strategies have on their derived templates. Templates derived using cause-effect methods are subsets of their immediate parents. This means that there could be contradictions introduced by the new derivation, and indeed there are.

There are two approaches available for redefining templates with altered derivations. The first is to redefine each template by placing the appropriate parent template in the signature and resolving contradictions. This is sometimes the only approach available, but can involve more work than necessary. The other approach is to extend the definitions of strategies wherever possible to include constraints on templates derived using them. For example, defining the cause-effect strategy by

\[
\text{cause_effect : STRATEGY} \\
\forall T : \text{TT}_0, \bullet \\
(\forall \text{tchild} : \text{TTH}_\text{op}(T, \text{cause_effect}) \bullet \\
\text{tchild} \subseteq T)
\]

enforces the restriction on derived templates and removes any need to redefine templates. Only the TTH needs to be altered accordingly.

In each of the cause-effect cases, two domain propagation templates contradict their new derivation and hence are eliminated, leaving us with the same four templates as before, but with a different derivation:

\[
\text{TTH}_{\text{up}}(\text{CE}_1, \text{dom}_\text{prop}) = \{\text{DP}_{2,3}, \text{DP}_{4,5}\}
\]

\[
\text{TTH}_{\text{up}}(\text{CE}_2, \text{dom}_\text{prop}) = \{\text{DP}_{2,3}, \text{DP}_{4,5}\}
\]

The additional upper bound introduced requires more test cases. We can extend the type-based testing by adding the following cases for when the size of the table nears its maximum

4. #st = Max - 1
5. #st = Max

and deriving the following templates

\[
\text{TB}_{4,4,4} \triangleq [\text{DP}_{4,4} | #st = \text{Max} - 1]
\]

\[
\text{TB}_{4,4,5} \triangleq [\text{DP}_{4,4} | #st = \text{Max}]
\]

\[
\text{TB}_{4,5,5} \triangleq [\text{DP}_{4,5} | #st = \text{Max} - 1]
\]

\{\text{TB}_{4,4,4}, \text{TB}_{4,4,5}\} = \text{TTH}_{\text{op}}(\text{DP}_{4,4}, \text{type}_\text{based})

\{\text{TB}_{4,5,2}, \text{TB}_{4,5,3}, \text{TB}_{4,5,5}\} = \text{TTH}_{\text{op}}(\text{DP}_{4,5}, \text{type}_\text{based})

Note that we key test the case where #st = Max for DP_{5,5}, since this is not a subset of DP_{4,5}. The behavior of Update on this case is unspecified.

4.4.3 More Complete Changes

At some stage of development, some statement of how to treat adding new members to the full symbol table should be made. The following is an example of such a statement.

\[
\text{REPORT} ::= \text{ok} \mid \text{symbol_not_present} \mid \\
\text{symbol_table_full}
\]
5.1 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 are not implemented correctly. There can be a simple mapping from specification structure to implementation structure, but this need not necessarily be true [39]. If the implementation structure does correspond closely to the specification structure, it is easier to find where the fault lies. Even if there is not a close correspondence between the specification and implementation structures, knowing which parts of the specification are not 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.

5.2 Extensions to the Framework

There is room for extension to existing aspects of the framework. Certainly, our libraries of propagations should be completed, and 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 [40], [36], [5].

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 structural tests.

A potential extension to the framework is suggested by our maintenance experiments. We saw in Section 4.4 that when we “split” a template high in the hierarchy, we usually derive a similar set of subtemplates 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.

6 CONCLUSIONS

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 Section 1.1, 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.

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. The framework is not restrictive and allows many strategies to be used.

APPENDIX A — Z NOTATION OVERVIEW

Z is a general purpose specification language developed by J.-R. Abrial and the Programming Research Group at Oxford. It is based on set theory and predicate calculus, uses fairly standard mathematical notation and supports standard set and predicate operations.

The expression \( P S \) where \( S \) is some set denotes the powerset of \( S \), i.e., the set of all subsets of \( S \). For example, \( P \{1, 2\} = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\} \). The operator \( \# \) yields the size of a set. Z has a range of basic sets like \( \mathbb{N} \) (the natural numbers). Additional sets can be introduced as given sets when we are not interested in their internal structure nor in enumerating their elements.

Systems are described in Z using a state-based model. System state is usually defined using Z schemas. A Z schema consists of a signature part containing declarations and a predicate part constraining the variables defined in the signature. For example:

\[
\text{Coord} \equiv [x, y : \mathbb{R} \mid x \geq 0 \land y \geq 0]
\]

System events are modeled as operations on the system state and are defined in terms of the states before and after the operations. Operations are also defined using schemas. To distinguish variables representing the initial state from those representing the final state, Z uses the lexical convention of appending a prime (') to final state variables. Similarly, inputs are distinguished with a ? and outputs with a !. For example, an operation to transpose the coordinate is defined as:

\[
\begin{align*}
\text{Transpose} \\
\Delta \text{Coord} \\
x' &= y \\
y' &= x
\end{align*}
\]

By convention, \( \Delta \text{Coord} \) is a schema defined as \([\text{Coord}; \text{Coord}'] \) representing the initial and final states, including the constraining predicate for each state. In the schema Transpose, it is an example of schema inclusion.

Another example is an operation to output the distance from the origin:

\[
\begin{align*}
\text{CalcDistance} \\
\exists \text{Coord} \\
dist() : \mathbb{R} \\
dist() &= \sqrt{x^2 + y^2}
\end{align*}
\]

The schema \( \exists \text{Coord} \) is similar to \( \Delta \text{Coord} \) in that it defines two states but additionally it constrains the two states to be equal. It is used for operations like \( \text{CalcDistance} \) that do not modify the system state.

Z also provides a schema calculus, supporting operations on schemas so that schemas can be combined or composed to describe new states or operations. For example, joining two schemas with the \( \land \) operator produces a composition with a merged signature (common variables must agree on type) and the conjunction of the two predicate parts. Axiomatic definitions in Z introduce global variables; for example, a function square defined over the natural numbers:

\[
\begin{align*}
square &: \mathbb{N} \to \mathbb{N} \\
\forall n : \mathbb{N} \\
square n &= n \times n
\end{align*}
\]

This function is total (i.e., defined for all naturals), signified by \( \to \). Partial functions are signified by \( \nrightarrow \). Functions in Z are sets of pairs so operations on sets are also applicable to functions. The notation \( x \to y \) denotes the tuple \((x, y)\).

Z includes some special notations that apply to functions (and more generally, to relations). For example: if \( f : A \nrightarrow B \) is a function and \( S \seq A \), the domain restriction \( S \downarrow f \) and domain subtraction \( S \setminus f \) are defined by
This expression denotes a function that is like $f$ except at the domain element $x$. If $f$ is not defined at $x$, the expression is defined and maps $x$ to $y$. If $f$ is defined at $x$, that mapping is overridden by the right operand. In general, $f @ g = ((\text{dom } g) \cap \text{dom } f) U g$

In $\mathbb{Z}$, a sequence is a function whose domain is the contiguous set of natural numbers from 1 to the length of the sequence.

$\text{seq } X = \{ s : \mathbb{N}_+ \rightarrow X | \exists n : \mathbb{N} \cdot \text{dom } s = 1..n \}$

The sequence $(a_1, a_2, a_3)$ is equivalent to $(1 \rightarrow a_1, 2 \rightarrow a_2, 3 \rightarrow a_3)$. The operator $\sim$ concatenates two sequences.

**ACKNOWLEDGMENTS**

This work was initiated at the University of Massachusetts at Amherst during a six month visit by Phil Stocks. Many thanks are due to Lori Clarke for her efforts and interest in this work. We are grateful for the useful discussions on this work held with Barbara Lerner and other members of the Software Development Laboratory at the University of Massachusetts. We thank Ian Hayes and Ian MacColl, at the University of Queensland, for some very insightful comments on the basic model of test templates and the test template hierarchy.

Phil Stocks was supported by an Australian Postgraduate Research Award and a scholarship with the Centre for Expertise in Distributed Information Systems, funded by Telecom Research Laboratories Australia under Telecom Australia Contract No. 7015. Some of the material in this article was presented in papers at ICSE in May 1993 and ISTA in June 1993. This work was done as part of Phil Stocks’ Ph.D program at The University of Queensland, Australia.

**REFERENCES**


Phil Stocks holds BS (Hons) and PhD degrees from The University of Queensland, Australia, awarded in 1989 and 1994, respectively. Dr. Stocks was a member of technical staff at Siemens Corporate Research in Princeton, New Jersey, in 1994 and 1995. In his spare time over the 1995 spring semester, he was an adjunct professor at Rutgers University. He is currently a postdoctoral research associate in the Department of Computer Science of Rutgers University, New Brunswick, New Jersey, working with the programming languages research group on compile-time analysis of large programs. His research interests are programming languages, compile-time analysis, and software testing. Dr. Stocks is a member of the Association for Computing Machinery.

David Carrington received the BSc (Hons) and PhD degrees from the University of New South Wales, Australia, in 1975 and 1984, respectively. He is a senior lecturer in the Department of Computer Science and an academic member of the Software Verification Research Centre at The University of Queensland, Australia. Dr. Carrington has a broad range of research interests in the areas of software development and user interfaces, including techniques and tools for formal specification, refinement techniques, design methods, programming environments, and specification-based testing methods and tools. He is a member of the IEEE Computer Society and the Association for Computing Machinery.