# Test Templates: A Specification-based Testing Framework

P. A. Stocks

Department of Computer Science
The University of Queensland
Queensland 4072
Australia

D. A. Carrington

Department of Computer Science
The University of Queensland
Queensland 4072
Australia

## 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 other test data sets, providing structure to the testing process. Flexibility is also 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.*

## 1 Introduction

### 1.1 Specification-based testing

Specification-based Testing (SBT) offers many advantages in software testing. The (formal) specification of a software product can be used as a guide for designing functional tests for the product. The specification precisely defines fundamental aspects of the software, while more detailed and structural information is omitted. Thus, the tester has the important information about the product's functionality without having to extract it from unnecessary detail.

SBT from formal specifications offers a simpler, structured, and more formal approach to the development of functional tests than standard testing techniques. The strong relationship between specification and tests facilitates error pin-pointing and can simplify regression testing. An important application of specifications in testing is providing test oracles. The specification is an authoritative description of system behaviour and can be used to derive expected results for test data. Other benefits of SBT include developing tests concurrently with design and implementation, using the derived tests to validate the original specification, and simplified auditing of the testing process. Specifications can also be analysed with respect to their testability (e.g., [6]).

With so many advantages to be gained, it is surprising to note the relatively limited research into SBT. The idea is not new, but has only recently received renewed attention. One explanation is that usage and stability of formal methods has increased considerably over the past decade, making SBT more effective.

This discussion uses the following terminology. A functional unit is a distinct piece of functionality the system is meant to provide. Usually this is expressed by a single operation, but may involve (parts of) multiple operations. Testing involves more than just the test data, e.g., oracle information to determine the validity of the test, functional units to be tested, constraints on test data, purpose of test data, test plans, dependency information, etc. The general term test information is used to denote any information used in the testing process. A test case is the combination of test data and oracle.

### 1.2 A framework

This paper describes the Test Template Framework (TTF) which is a structured strategy and a formal framework for SBT. The TTF deals with (functional) unit testing, particularly deriving and defining tests for operations defined in the specification. Integration testing is an area for future work. The strategy is fundamentally partition testing, but, as discussed in [10], partition testing can include strategies such as path testing, branch testing, and even mutation testing, if the partitioning is selective enough.

The TTF is not yet another testing strategy, rather it is a general framework for SBT. No particular testing heuristic is better than all the others. They all have strengths and weaknesses, and should be combined to increase the effectiveness of testing. The objective of testing is to identify errors in any way possible. The TTF is very flexible, allowing many different testing strategies to be incorporated. It provides a common foundation for applying testing strategies, and for classifying and comparing these strategies.

The Z specification notation [7, 15] is used as a Test Description Language (TDL), and to define the basic components of the TTF. This enables a formal description of test data, and other test information such as test oracles. The uniform use of the Z notation improves the clarity and structure of the test information. The advantages of using an existing specification language as a TDL are a formal syntax and semantics, reasoning power in the expression of properties and constraints, and familiarity with the notation.

The framework's application in SBT is demonstrated on specifications expressed in model-based notations, which facilitate SBT due to their explicit state transition model. Note that the TTF can be used on specifications expressed in notations other than Z (and the framework can be defined in other languages) that use a state-transition model, particularly where pre- and post-conditions for operations can be determined, e.g., VDM [11] and RSL [13].

In this discussion, the framework is applied to Z specifications. This similarity between test definition and formal specification for a system is another strength of the TTF. The test information is largely derivable from the specification and is represented in the same notation.

### 1.3 Related work

Various methods have been employed to derive test information from specifications, concentrating on generating test data and, in some cases, test oracles. Most SBT work is based on process-based specification languages, e.g., [1, 3]. Some work has been done on test information derivation from model-based specification languages like Z and VDM, e.g., [8, 14], but to the authors' knowledge a framework for SBT using model-based notations is novel.

In some aspects, the framework is quite similar to category-partitioning [12] which uses a description language to define tests derived from the possible combinations of values of variables in the input space. The framework, however, has more structure and doesn't restrict itself to one testing strategy. It allows different heuristics and strategies to be used to determine interesting subdomains of the input space and to select tests for these subdomains.

### 1.4 This document

Familiarity with Z is assumed (tutorial references include [21] and the opening chapters of [7, 15]). Familiarity with testing strategies is also assumed [2].

Section 2 defines the framework. Section 3 discusses the framework and its many applications. Space limitations require this discussion to be somewhat abstract.

## 2 Test template framework

This section describes the Test Template Framework (TTF) beginning with a general discussion of test templates and a test hierarchy, then discussing each aspect in more detail.

### 2.1 Test template overview

The central concept of the framework is the Test Template (TT). A TT is a description of data that is

- generic,
- instantiable,
- derivable from a formal specification, and
- refinable.

TTs describe sets of data, most commonly test inputs. Test data usually have restrictions that they must satisfy in order to test a particular feature. There are usually many possible data that can satisfy the restrictions. These classes of test data are defined with TTs, that are instantiated to produce actual test data when required. TTs constrain the important features of data without placing unnecessary restrictions.

### 2.2 Constructing a test hierarchy

Test data derivation is simplified by a structured approach involving systematic application of various testing heuristics; templates are derived in stages. TTs are organised into a hierarchy representing the tests for a functional unit. The algorithm for constructing the test hierarchy is based on the familiar partition testing approach, where the input space of an operation is partitioned into subspaces whose elements display some common characteristics.

The Input Space (IS) of an operation is the set of all possible input to the operation based on the signature of the input alone. For example, if the input is of type integer, then the IS is the set of integers. The Valid Input Space (VIS) of an operation is the subset of the IS for which the operation is defined, i.e., the subset which satisfies the operation's pre-condition. For example, the VIS of the factorial function defined on integers is the non-negative integers. The VIS may equal the IS.

The first step is to determine the VIS of the functional unit under test. The VIS is derived directly from the specification of the functional unit, and is largely an automatic process. There is one VIS for each functional unit.

The next step is to subdivide the VIS into the desired subsets, or partitions (called *domains*). Choice of domains is not determined by the TTF. Rather, testing strategies and heuristics should be used to subdivide the input space. Standard practice is to choose tests for a domain whose success is intended to demonstrate the correctness of the operation on 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. To preserve the flexibility to choose tests for domains selectively, the domain derivation step is used repeatedly, dividing domains into further sub-domains, until the tester is satisfied that the domains are equivalence classes of the valid input space.

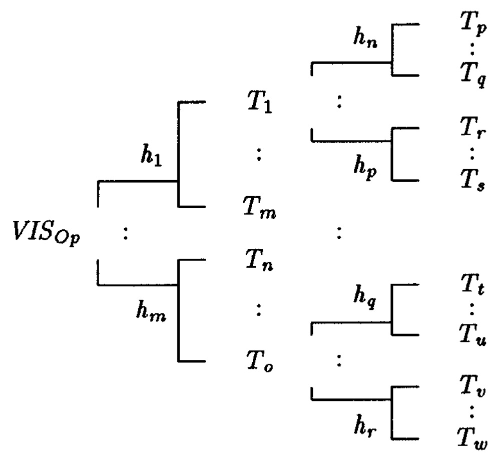

Figure 1: Typical TTF Hierarchy

This derivation results in a collection of test templates, related to each other by their derivation and the heuristics used in their derivation. The collection can be considered a directed graph of templates, where the heuristics represent the edges of the graph. Typically a TT hierarchy looks something like Figure 1. Some strategies do not advocate domain partitioning (e.g., random testing), in which case the valid input space forms the hierarchy. 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 heuristics may advocate further subdividing of already derived templates. Figure 1 shows a common structure. The terminal nodes of this graph are the templates representing the final test data for the functional unit.

## 2.3 Model of test templates

Z schemas define test templates. Schemas, rather than sets, are used because they provide a more general representation of tuples, and the schema calculus of Z simplifies the definition of test templates.

For each operation, a VIS template is derived. All templates in the hierarchy for this operation are subsets of the VIS. The hierarchy is defined in terms of heuristics and subsets of the VIS. The generic set of heuristics is introduced and deliberately left abstract:

$$[HEURISTIC]$$

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

$$| \; TTH_{Op} : \mathbb{P}\ VIS_{Op} \times HEURISTIC \twoheadrightarrow \mathbb{P}(\mathbb{P}\ VIS_{Op})$$

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

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

If the heuristic used in this derivation was $h$, then the TTH is updated by introducing the heuristic and adding the mapping to the TTH:

$$| \; h : HEURISTIC$$

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

The set of all child templates derived from a template is determined by the function

$$\begin{array}{|l} children_{Op} : \mathbb{P}\, VIS_{Op} \to \mathbb{P}(\mathbb{P}\, VIS_{Op}) \\ \hline children_{Op} = (\lambda\, T : \mathbb{P}\, VIS_{Op} \bullet \\ \quad \bigcup\{h : HEURISTIC \bullet TTH_{Op}(T, h)\}) \end{array}$$

and the set of all templates descended from a template is determined by the function

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

## 2.4 Valid input space

The VIS can be derived for an operation by using its pre-condition as the constraint on the IS. The pre-condition is the minimum constraint which must hold on the input for the operation not to fail (note this does not require a deterministic post-state to exist for every input satisfying the pre-condition).

For example, the operation

$$\begin{array}{|l} \textit{Converge} \\ \hline x, x' : \mathbb{N} \\ \hline (x < 5 \land x' = x + 1) \lor \\ (x > 5 \land x' = x - 1) \\ \hline \end{array}$$

has VIS

$$\begin{aligned} VIS_{Converge} &\mathrel{\widehat{=}} \text{pre } Converge \\ &= [x : \mathbb{N} \mid x < 5 \lor x > 5] \end{aligned}$$

All inputs for which the operation is defined conform to this structure. This is significant because at this level it is meaningless to derive test data outside the valid input space since there is no description of intended behaviour for such input, i.e., any behaviour satisfies the specification.

Consequently, as mentioned in [17], the expression $\neg\ VIS_{Op}$ describes input not conforming to the requirements of the operation. It is a concise description of what is not acceptable input for the operation and may alert the designer to some error in the specification. Depending on design decisions and testing strategies, this may also be the source of robustness tests for the operation. This need not always be the case, though. It may be convenient in the implementation for each operation to assume its pre-conditions are checked elsewhere, and so robustness testing operations in isolation is not meaningful. If this is the case, then it is necessary to show that the pre-condition for an operation holds when the operation is invoked.

## 2.5 Domain partitioning

Heuristics determine interesting features of the operation to be used as determining factors in domain definition. Using heuristics introduces informality, but test derivation can still be rigorous.

Including a heuristic in the framework involves stating guidelines and perhaps defining some properties of the heuristic. For example, consider path testing: dividing the VIS into "paths" through the specification (in other words, distinctions of input treated the same way by the operation). Domains are determined according to classes of input for which the operation does the same thing. The whole input space could be thus partitioned.

In the previous example of operation *Converge*, it is clear that the input is divided by the classes $x < 5$ and $x > 5$.

These are expressed as TTs as follows

$$\begin{aligned} P1 &\mathrel{\widehat{=}} [VIS_{Converge} \mid x < 5] \\ P2 &\mathrel{\widehat{=}} [VIS_{Converge} \mid x > 5] \end{aligned}$$

and included in the hierarchy.

$$\begin{array}{|l} path : HEURISTIC \end{array}$$

$$TTH_{Converge}(VIS_{Converge}, path) = \{P1, P2\}$$

In path testing, the domains are required to partition the input space. That is, they must cover it and have no elements in common. Corresponding relationships between templates in general can be expressed by

$$\begin{array}{|l} \_\underline{covers}\_ : \mathbb{P}(\mathbb{P}\, VIS_{Converge}) \leftrightarrow \mathbb{P}\, VIS_{Converge} \\ \hline \forall\, TS : \mathbb{P}(\mathbb{P}\, VIS_{Converge});\ T : \mathbb{P}\, VIS_{Converge} \bullet \\ \quad TS\ \underline{covers}\ T \Leftrightarrow \bigcup TS = T \end{array}$$

$$\begin{array}{|l} \_\underline{mutex}\_ : \mathbb{P}\, VIS_{Converge} \leftrightarrow \mathbb{P}\, VIS_{Converge} \\ \hline \forall\, T1, T2 : \mathbb{P}\, VIS_{Converge} \bullet \\ \quad T1\ \underline{mutex}\ T2 \Leftrightarrow T1 \cap T2 = \{\} \end{array}$$

The following predicates express the partitioning property for the path testing heuristic.

$$\begin{aligned} &\forall\, anc : \mathbb{P}\, VIS_{Converge} \bullet \\ &\quad TTH_{Converge}(anc, path)\ \underline{covers}\ anc \end{aligned}$$

$$\begin{aligned} &\forall\, anc : \mathbb{P}\, VIS_{Converge} \bullet \\ &\quad (\forall\, T1, T2 : TTH_{Converge}(anc, path) \mid T1 \neq T2 \bullet \\ &\qquad T1\ \underline{mutex}\ T2) \end{aligned}$$

The heuristic is thus incorporated in the TTF as a systematic application of guidelines for selecting interesting classes of the input space and as a collection of properties, expressed using Z, of the domain templates derived.

### 2.5.1 Notes on the choice of domains

Obviously, the heuristic used for determining domains is very important, and perhaps hard to formalise. However, most partition methods are very structured and quite straightforward. For example, determining equivalence classes of input in the VIS can be a very simple process given a Z specification of the operation; consider operation *Converge* discussed in section 2.4. But it isn't always easy. Consider the similar operation below.

$$\begin{array}{|l} \textit{Converge2} \\ \hline x, x' : \mathbb{N} \\ \hline x < 5 \Rightarrow x' = x + 1 \\ x > 5 \Rightarrow x' = x - 1 \\ \hline \end{array}$$

Here the pre-condition is *true* since the operation does not fail when $x = 5$, it is merely non-deterministic. However, it is reasonable to use the same domains for *Converge*2 as were used for *Converge*, with the additional domain $\{x : \mathbb{N} \mid x = 5\}$, because to guarantee behaviour $x' = x+1$ or $x' = x-1$ the input must satisfy $x < 5$ or $x > 5$ respectively. This example shows that a cause-effect method is really being used: selecting input paths based on what output will be achieved. Again, though, the domains are obvious given the specification. Now consider the equivalent operation formed by expressing the implications as disjunctions

$$\begin{array}{|l} \textit{Converge3} \\ \hline x, x' : \mathbb{N} \\ \hline (x \geq 5 \lor x' = x + 1) \land \\ (x \leq 5 \lor x' = x - 1) \\ \hline \end{array}$$

The pre-condition is *true*, but what is not obvious is that the domains are $x < 5$, $x > 5$, and $x = 5$, as for *Converge*2. This is so because these predicates guarantee each *different* behaviour of the operation.

The point is that heuristics should be clear about what factors really determine the domains of interest. These domains will not always be simple to derive. In most cases, though, it is a simple process and it is reasonable to assume that partitioning can be done. Detailed examination of partitioning the VIS into domains based on path selection and other criteria is an area for future work.

### 2.5.2 Continuing subdivision

A curious aspect of most partition testing methods is the assumption that each element of a partition is as good as every other element at detecting errors. On the surface it is a valid assumption, and simplifies the testing process by vastly reducing the required number of test cases. Studies have shown that, under certain conditions, partition testing is no more effective than random testing, which is often considered the benchmark for a testing heuristic [10]. However, due to the assumption, these studies do not consider further subdividing the partitions based on heuristics. Further subdividing the partitions can increase the accuracy of tests.

The TTF permits further heuristics to be applied to derive the most desirable test data for particular domains. For example, choosing points close to the domain boundaries:

$$B1 \mathrel{\widehat{=}} [P1 \mid x = 4]$$
$$B2 \mathrel{\widehat{=}} [P2 \mid x = 6]$$

$$\mid \; \textit{boundary} : \textit{HEURISTIC}$$

$$TTH_{\textit{Converge}}(P1, \textit{boundary}) = \{B1\}$$
$$TTH_{\textit{Converge}}(P2, \textit{boundary}) = \{B2\}$$

Note that these templates have only one possible instantiation, but this need not always be so.

## 2.6 Instance templates

After applying all the desired heuristics to derive the template hierarchy, instance templates can be derived. 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. Instance templates make the specific choice of test data. They are simply templates describing only one element of their parent templates. This step is taken to provide a uniform representation of test data in the hierarchy. The final translation of instance templates to concrete test data is implementation dependent. The heuristic to derive instance templates is assumed in the TTH:

$$\mid \; \textit{instantiation} : \textit{HEURISTIC}$$

## 2.7 Common properties of templates

It is possible to build a library of common properties of templates, such as the relations covers and mutex defined above. These properties can be used in defining properties of testing heuristics.

A notion of template equivalence is useful when using multiple heuristics in test development, since some of the templates derived may be equivalent, and can thus be discarded. The Z schema calculus has an equivalence operation on schemas: $\Leftrightarrow$.

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

$$\begin{array}{|l} \underline{notcovered} : \mathbb{P}\ VIS_{Op} \rightarrow \mathbb{P}\ VIS_{Op} \\ \hline \underline{notcovered} = (\lambda\ T : \mathbb{P}\ VIS_{Op} \bullet \\ \quad T \setminus \bigcup(children_{Op}(T))) \end{array}$$

Not all domain subdivisions will enforce the entire input domain to be covered, for example, an ideal revealing-domain-partitioning [19], where the input is partitioned into the set of all error-causing inputs and the set of all correct inputs, and only the error-causing domain examined.

Checking these properties is useful in detecting incorrect use of heuristics when defining templates. Expression of such properties also helps increase understanding of heuristics.

## 3 Discussion

The previous sections describe the the basic mechanics of the TTF. The following sections discuss ideas on further applications of the framework beyond specification of test data, and future work areas. Due to space limitations, this discussion is brief. A more complete discussion of these matters, in the context of a substantial example, can be found in [18]. The TTF has also been used to derive tests for a symbol table [16] and for a dependency management system.

### 3.1 Analysis of test set

Expressing the test data using a formal notation facilitates analysis of the test set. For example, [10] lists a number of properties of partition testing (using random selection of tests within partitions) useful for comparing the test set with a collection of random tests. For example, Observation 4 is that if all partitions are of the same size and the same number of tests for each partition is chosen, then the test set is at least as good as random testing. Since templates are Z schemas, and Z schemas are sets, we use the cardinality operator on sets (#) to represent the size of a template. Observation 4 may be expressed

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

### 3.2 Analysis of heuristics

The TTF provides common ground for comparing and contrasting testing heuristics. Domain testing [20, 4], usually derives fewer test points, but it is a more difficult strategy to apply, and is not applicable in many cases.

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

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

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

### 3.3 Application areas

#### 3.3.1 More testing heuristics

The TTF facilitates development of more SBT heuristics, particularly in the partitioning of the input space into domains. What are good partitioning strategies? Is analysis of the output space to determine unique input partitions for all cases worthwhile? Is it ever acceptable not to consider portions of the input space? Similar questions are raised when deriving sub-templates. What information is there in the specification that indicates good subdivisions for a partition?

Also of interest are examples showing how to incorporate existing SBT and other testing strategies into the TTF.

### 3.3.2 Refinement

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. Where the specification is refined to different implementations, so the TTs can be refined to describe (more concretely) the test data and test information for the refined specifications.

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. The treatment of refinement in the TTF is in its early stages, and deals primarily with data refinement.

Between a specification (at the abstract level) and a refinement of the specification (at the concrete level, though not necessarily the final implementation) there exists an abstraction relation, say *Abs* (see Chapter 1 of [15] and [18] for more details and examples).

Refined templates are derived from their abstract counterparts by conjoining the abstract template and the abstraction relation, and then hiding the components of the abstract data representation.

For example, consider the data type *File* defined

$[BYTE]$

$| \; MaxFileSize : \mathbb{N}$

```
┌─ File ──────────────────
│ file : seq BYTE
├─────────
│ #file ≤ MaxFileSize
└─────────────────────────
```

and implemented using a combination of an array of bytes (represented as a mapping from $\mathbb{N}$ to $BYTE$) and a size variable representing the size of the array

```
┌─ File1 ─────────────────
│ elts : ℕ ⇸ BYTE
│ size : ℕ
├─────────
│ size ≤ MaxFileSize
└─────────────────────────
```

The relation between abstract and concrete files is

```
┌─ Abs ───────────────────
│ File
│ File1
├─────────
│ #file = size
│ ∀ i : dom file •
│     file(i) = elts(i)
└─────────────────────────
```

Suppose the following template is derived, using domain testing, from the abstract specification of a read operation which takes the file and the number of bytes to be read as input. It represents one test point (an ON point from domain D1, hence the name), where no characters are read from a large-as-possible file.

$$Read\_D1_{ON1} \mathrel{\widehat=} [VIS_{Read} \mid \#file = MaxFileSize \wedge len? = 0]$$

The template refined from the above abstract template using *Abs* (represented by the superscript R on the name) is

$$Read\_D1^{R}_{ON1} \mathrel{\widehat=} (Read\_D1_{ON1} \wedge Abs) \setminus (file)$$

That is,

$$Read\_D1^{R}_{ON1} \mathrel{\widehat=} [elts : \mathbb{N} \pfun BYTE;\ size : \mathbb{N};\ len? : \mathbb{N} \mid size = MaxFileSize \wedge len? = 0]$$

If testing is conducted separately on the refined specification, one of the TTs produced when domain testing is applied to the refined read operation, *Read*1, is [18]

$$Read1\_D1_{ON1} \mathrel{\widehat=} [elts : \mathbb{N} \pfun BYTE;\ size : \mathbb{N};\ len? : \mathbb{N} \mid size = MaxFileSize \wedge len? = 0]$$

As expected

$$Read\_D1^{R}_{ON1} \Leftrightarrow Read1\_D1_{ON1}$$

Graphically, this relationship is

$$\begin{tikzcd}
\text{SPEC} & \text{TT} \\
\mathit{Read} \arrow[r] \arrow[d, "\text{refine (using } \mathit{Abs})"'] & \mathit{Read\_D1}_{ON1} \arrow[d] \\
\mathit{Read1} \arrow[r, "\text{derive template}"'] & \mathit{Read\_D1}^{R}_{ON1} = \mathit{Read1\_D1}_{ON1}
\end{tikzcd}$$

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

The new VIS derived after this refinement is applied to *Read* is

$$\begin{array}{l} VIS_{Read2} \mathrel{\widehat{=}} \\ \quad [file : \mathrm{seq}\, BYTE;\ len? : \mathbb{Z} \mid \\ \qquad \#file \leq MaxFileSize \wedge len? \geq 0] \end{array}$$

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

### 3.3.3 Maintenance

In a similar vein to the above, the TTF is also useful in software maintenance. When the specification is updated, the TTs that require updating can be determined and re-derived.

This reduces the effort required when regression testing software. A tool that performed dependency analysis on the specification would be very useful since effects of specification changes may propagate to unexpected areas of the specification.

### 3.3.4 Oracles

The formal specification does more than describe conditions on the input. The relationship between input states and output states is precisely specified. This means that the specification can serve as a test oracle. Deriving test oracles from specifications has received some attention, e.g., [14]. A simple approach to deriving an oracle in the TTF is to generate output TTs corresponding to input TTs.

An oracle template is derived for a test data template by using the input-output relationship of the operation to derive an expression for the output components given certain input. The oracle template is defined over the output space of the operation. Note that the oracle template defines a set of possible outputs if the operation is non-deterministic, and that an oracle template can be determined for a TT that isn't an instantiation TT, also resulting in a set of possible outputs.

We define the output space (OS) of an operation, similarly to the IS, as the signature of the operation restricted to the output variables. A general expression for the oracle template of any test template $T$ derived from operation $Op$ is

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

We use the expression $oracle_{Op}$ to represent this. For example

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

Thus, a description of the expected output for each TT can be derived. Again, final concrete instantiation of this oracle template depends on the final implementation. The combination of test data and test oracle forms a test case.

### 3.3.5 Robustness tests

As mentioned earlier, the negation of the VIS could be used as the source of robustness tests for an operation. $\neg\ VIS_{Op}$ could form the root of another TT hierarchy and robustness TTs could then be derived using the TTF.

### 3.3.6 Software design: Components

Brad Cox describes a software engineering ideal where software development involves constructing programs from pre-defined components, similar to conventional engineering's use of nuts and bolts [5]. A software engineering revolution similar to the industrial revolution is discussed which will move software development from a "build everything from scratch" approach to a "build from re-usable components where possible" approach. Real, hardware components are defined by a specification of their purpose/parameters and some sort of gauge to determine whether a proposed piece of hardware is satisfactory. The TTF is useful in component definition because the set of test cases derived from the specification can act as the component gauge. The test cases can be used to determine whether implementations of the component conform to the specification.

### 3.4 Future work

The TTF described in this paper is usable, but additional work is required. Many interesting areas of future work have been identified from applying the framework, notably

- examination of methods for selecting input partitions,
- extending the refinement model, and
- refining the test oracle model.

A major area for future work is tool support. A simple tool interface to the template collection would greatly assist using the TTF. Such a tool would keep track of the derivation structure so that only new information would need to be entered. Other highly useful tools that are more complicated are a pre-/post-condition analyser that derives the pre- or post-condition from an operation's specification, a theorem prover that can assist in verifying properties of TTs, and test data generator that automatically instantiates TTs given the TT definition.

Another aspect for consideration is applying the TTF beyond unit testing. The specification defines operation interfaces, which is useful in module and integration testing. As it stands, the TTF is useful for deriving test data, but means of testing modules on the data are required. Integrating the TTF with module testing approaches such as in [8, 9], which focus on developing test scripts to apply (sequences of) tests rather than deriving tests, should be investigated.

### 3.5 Conclusion

A generally applicable testing framework has been described, which allows test information to be structured in a useful and usable manner. Expressing properties of test classes and data is facilitated by the formal nature of the language used to define test information.

The TTF is a specification-based testing method with the advantages of SBT, with applications beyond testing software. Its flexible nature facilitates incorporating new heuristics. The existence of useful specification-based methods and tools may encourage increased use of formal methods.

Finally, one interesting point about a specification-based testing method is that, if there are certain styles of specification where the method is more applicable or easier to use, it will encourage specifiers to structure their specifications in these styles. Design for testability encourages specifiers and designers to consider factors that could otherwise be delayed far into the development process, and usually improves the clarity of the specification or design.

## Acknowledgements

Much of this work was conducted at the University of Massachusetts at Amherst during a six month visit by the first author. 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, at the University of Queensland, for some very insightful comments on the basic model of TTs.

Phil Stocks is supported by an Australian Postgraduate Research Award and a scholarship with the Centre for Expertise in Distributed Information Systems, which is funded by Telecom Research Laboratories Australia under Telecom Australia Contract # 7015.

## References

[1] J. Arkko, V. Hirvisalo, J. Kuusela, and E. Nuutila. Supporting testing of specifications and implementations. *EUROMICRO Journal*, 30(1-5):297–302, August 1990. EUROMICRO'90.

[2] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, second edition, 1990.

[3] G. Bernot, M.-C. Gaudel, and B. Marre. Software testing based on formal specifications: A theory and a tool. *Software Engineering Journal*, 6(6):387–405, November 1991.

[4] L. A. Clarke, J. Hassell, and D. J. Richardson. A close look at domain testing. *IEEE Transactions on Software Engineering*, 8(4):380–390, July 1982.

[5] B. J. Cox. Planning the software industrial revolution. *IEEE Software*, 7(6):25–33, November 1990.

[6] R. S. Freedman. Testability of software components. *IEEE Transactions on Software Engineering*, 17(6):553–564, June 1991.

[7] I. Hayes, editor. *Specification Case Studies*. Series in Computer Science. Prentice Hall International, 1987.

[8] I. J. Hayes. Specification directed module testing. *IEEE Transactions on Software Engineering*, 12(1):124–133, January 1986.

[9] Daniel Hoffman and Christopher Brealey. Module test case generation. *Software Engineering Notes*, 14(8):97–102, December 1989. Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3).

[10] B. Jeng and E. J. Weyuker. Some observations on partition testing. *Software Engineering Notes*, 14(8):38–47, December 1989. Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3).

[11] C. B. Jones. *Systematic Software Development Using VDM*. Series in Computer Science. Prentice Hall International, 1990. Second Edition.

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

[13] The RAISE Language Group. *The RAISE Specification Language*. BCS Practitioner Series. Prentice Hall International, 1992.

[14] D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th International Conference on Software Engineering*, pages 105–118, Melbourne, Australia, May 1992.

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

[16] P. Stocks and D. Carrington. Test template framework: A specification-based testing case study. Submitted to ISSTA'93, Cambridge MA, June 1993.

[17] P. Stocks and D. Carrington. Deriving software test cases from formal specifications. In *6th Australian Software Engineering Conference*, pages 327–340, July 1991.

[18] P. Stocks and D. Carrington. Test templates: A specification-based testing framework. Technical Report 243, Key Centre for Software Technology, Department of Computer Science, The University of Queensland, 1992.

[19] E. J. Weyuker and T. J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, 6(3):236–246, May 1980.

[20] L. J. White and E. I. Cohen. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, 6(3):247–257, May 1980.

[21] J. Woodcock and M. Loomes. *Software Engineering Mathematics*. Pitman, 1988.