Extending the Test Template Framework

Ian MacColl    David Carrington
ianm          davec
@csee.uq.edu.au
Software Verification Research Centre
Department of Computer Science & Electrical Engineering
The University of Queensland

Abstract

The Test Template Framework (TTF) is a formal, abstract model of testing, used to derive a hierarchy of test information, including test inputs and outputs, from a model-based formal specification. In this paper we propose two modifications to the framework: first, that testing information derived from state components be factored out from operation-specific information, and second, that focusing on the input space of an operation under test be deferred for as long as possible. The first modification facilitates reuse of derived information in the context of abstract data types (ADTs) and classes to minimise duplication. The second modification supports testing strategies based on operation outputs, and provides a basis for further extensions for testing derived from behaviour-based notations.

1 Introduction

The Test Template Framework (TTF) is a formal, abstract model of testing, used to derive a hierarchy of test information, including test inputs and outputs, from a model-based formal specification. The TTF has been presented by way of case studies deriving testing information from formal specifications, including: the text formatter; the UNIX file Read operation; a symbol table Update operation; the dependency management system CanAdd operation; the triangle problem specification written in a transformational style, i.e., without persistent state; and a topological sort specification, also written in a transformational style [2, 14, 23, 22, 24].

The TTF provides a systematic way to derive and record test cases from a model-based specification for individual operations. The original work focused on specifications written in Z [20, 26] and recent work has extended the framework to specifications written in Object-Z [17, 18].

The existing TTF is applied to operations rather than the entire state space of the specification, and defines testing information in terms of the input spaces of the operations. In this paper we propose two modifications to the framework: factoring out testing information derived from state components and deferring focusing on the input space of an operation. The first modification facilitates reuse of derived testing information. The
second modification supports use of testing strategies considering operation inputs and outputs.

By convention, in Z [20, 26] a state schema describes the state space of a system, and operation schemas describe changes to the state space. The variables of an operation are either incorporated by inclusion of a state schema or directly declared in the operation. We refer to these categories of variables as state and operation variables respectively. By convention, state variables are further classified as before-state (unprimed) and after-state (primed) variables, and operation variables are further classified into inputs (distinguished by trailing '?' and outputs (distinguished by trailing '!').

In the TTF, testing information is derived by constraining the variables of an operation schema. Previous case studies have not classified the variables of an operation either because there was no state schema (transformational specifications) or because testing information was only derived for a single operation. The potential for factoring testing information derived from state components as a means to minimise duplication has therefore not been apparent. In this paper we propose separate derivation of testing information for the state variables to facilitate testing multiple operations with state variables in common. We use the Z schema calculus to achieve this for Z specifications.

The starting point in using the TTF for a particular operation is to determine the valid inputs to the operation which, in Z, is the precondition of the operation. Previous studies have used output and after-state variables for simplifications and for testing strategies such as cause-effect analysis, even though these variables are not accessible. Also, to extend the TTF to encompass behaviour-based notations, we must consider sequences of operation invocations with the output of one invocation establishing the input of another. In this paper we propose deferring the focus on input and before-state variables for as long as possible.

We present our modifications by revisiting the dependency management system (DMS) case study [22]. The next section contains a Z specification for relevant sections of the DMS. Section 3 outlines the derivation of testing information using the unmodified TTF while Section 4 presents the corresponding derivation using our modifications. Section 5 discusses related work and Section 6 concludes.

2 DMS specification

The dependency management system (DMS) case study was a testbed for a project exploring software development methodologies [13]. The DMS is a critical component of a theorem proving tool, tracking dependencies between theorems and assertions in a proof, thus preventing circular reasoning. However, the concept of dependency management generalises to other problems (such as revision control systems), so the DMS is a reasonably generic component.

The DMS tracks dependencies between nodes. The basic DMS maintains a set of nodes, direct dependencies between nodes, and inferred transitive dependencies between nodes. The DMS provides operations for manipulating a dependency graph, from adding nodes and dependencies to calculating all the nodes dependent on some other node. In this paper we consider only features associated with the set of nodes to simplify our examples.

We take as given the set of nodes \( X \)\(^1\) and define the state as a finite set of nodes, where

\( X \)\(^1\)

In the original specification, \( X \) was a parameter of the state and operation schemas. We define the
initially empty.

\[ X \]

\[
\begin{array}{c|c}
\text{Nodes} & \text{Nodes}_{\text{INIT}} \\
\hline
xs : F X & nodes \\
\hline
\end{array}
\]

\[ xs = \emptyset \]

We introduce an enumerated type for boolean output variables.

\[ B ::= \text{True} \mid \text{False} \]

Operations \textit{NoNodes} and \textit{IsNode} can query, but not change, the state (denoted by inclusion of \( \Xi \text{Nodes} \)). \textit{NoNodes} returns true if and only if \( xs \) is empty. \textit{IsNode} returns true if and only if the input node \( x \) is a member of \( xs \).

\[
\begin{array}{l}
\text{NoNodes} \\
\hline
\Xi \text{Nodes} \\
result! : B \\
result! = \text{True} \iff xs = \emptyset \\
\end{array}
\]

\[
\begin{array}{l}
\text{IsNode} \\
\hline
\Xi \text{Nodes} \\
x? : X \\
result! : B \\
result! = \text{True} \iff x? \in xs \\
\end{array}
\]

Operations \textit{AddNode} and \textit{RemoveNode} can change the state (denoted by inclusion of \( \Delta \text{Nodes} \)). Node \( x \) is added to \( xs \) by \textit{AddNode}. \textit{AddNode} is undefined if \( x \) is already in \( xs \), since, in Z, an operation gives an undefined result (interpreted as a client responsibility) if invoked outside its precondition. Operation \textit{RemoveNode} removes node \( x \). The precondition is that the input value must be an existing node and the remaining conjunct specifies the removal of the designated node from the set of nodes.

\[
\begin{array}{l}
\text{AddNode} \\
\hline
\Delta \text{Nodes} \\
x? : X \\
x? \notin xs \\
x's = xs \cup \{x?\} \\
\end{array}
\]

\[
\begin{array}{l}
\text{RemoveNode} \\
\hline
\Delta \text{Nodes} \\
x? : X \\
x? \in xs \\
x's = xs \setminus \{x?\} \\
\end{array}
\]

In the next section we derive testing information for \textit{Nodes} using the unmodified TTF and in Section 4 we present the corresponding derivation using our modified TTF. The material in the next section is not new but provides necessary background for our modifications.

3 TTF testing

The Test Template Framework (TTF) \cite{14, 22, 24} is used to derive a hierarchy of test information from a formal, model-based specification. Abstract test data can be instantiated as test inputs and test oracles can be generated.

---

generic parameter as a given set for typographic clarity.

3rd Northern Formal Methods Workshop, 1998
The TTF formally defines test templates as the nodes of the hierarchy. The relationships of the test templates to each other and to the system specification are also defined. A variety of functional testing strategies can be used to derive the templates; using multiple strategies is advocated.

Test templates are used to derive instance templates and oracle templates. Instance templates specify a unique choice of test data defining a specific test input. An oracle is a means of determining whether or not a test is successful. Oracle templates are derived for test templates using the input-output relationship of the formal specification.

In Section 3.1 we define the TTF structures required for our case study, in Section 3.2 we derive test templates, in Sections 3.3 and 3.4 we define instance and oracle templates, and in Section 3.5 we identify problems with the existing approach.

### 3.1 Definitions

The input space (IS) of an operation is the space from which input can be drawn. The input spaces for the Nodes operations are given by Z schemas with signatures restricted to the before-state variables \( (xs) \) and the input variables \( (x?) \), with no predicate part\(^2\).

\[
\begin{align*}
IS_{NN} & \equiv [xs : F X] \\
IS_{IN} & \equiv [xs : F X; x? : X] \\
IS_{AN} & \equiv [xs : F X; x? : X] \\
IS_{RN} & \equiv [xs : F X; x? : X]
\end{align*}
\]

Input to an operation may be type-compatible, i.e., fall within the input space, and still be invalid. The valid input space (VIS) is the subset of the input space for which an operation is defined, and is equivalent to the precondition\(^4\) of the operation. The valid input spaces of the operations are

\[
\begin{align*}
VIS_{NN} & \equiv \text{pre } \text{NoNodes} \\
& \equiv [IS_{NN} \mid \exists \text{result! } \bullet \text{result!} = \text{True} \Rightarrow xs = \emptyset] \\
& \equiv [xs : F X] \\
\end{align*}
\]

\[
\begin{align*}
VIS_{IN} & \equiv \text{pre } \text{IsNode} \\
& \equiv [IS_{IN} \mid \exists \text{result! } \bullet \text{result!} = \text{True} \Rightarrow x? \in xs] \\
& \equiv [xs : F X; x? : X] \\
\end{align*}
\]

\[
\begin{align*}
VIS_{AN} & \equiv \text{pre } \text{AddNode} \\
& \equiv [IS_{AN} \mid \exists xs': F X \bullet x? /\notin xs \land xs' = xs \cup \{x?\}] \\
& \equiv [xs : F X; x? : X \mid x? /\notin xs]
\end{align*}
\]

\(^2\)Here, and in the remainder of the paper, we use two-letter abbreviations of operation names as subscripts.

\(^3\)Note that we do not normalise Z schemas, instead preserving implicit type predicates on the assumption that these will be mapped concrete types in the implementation.

\(^4\)The Z precondition operator (pre) is defined as removal of output and after-state variables from the schema signature and existential quantification of those variables over the schema predicate. We are not aware of a formal definition of this operator in Z.
\[
\begin{align*}
VIS_{RN} & \equiv \text{pre } RemoveNode \\
& \equiv [IS_{RN} \mid \exists x \in xs : F X \cdot x? \in xs \land xs' = xs \setminus \{x?\}] \\
& \equiv [xs : F X ; x? : X \mid x? \in xs]
\end{align*}
\]

The output space (OS) and valid output space (VOS) of an operation are defined over the output signature of the operation similarly to the definitions of the IS and VIS. The output space is used in oracle derivation.

The basic unit for defining data in the framework is a test template (TT) which is expressed as a Z schema. A test template is a formally constrained data space. Test templates are generic, abstract, instantiable, and derivable from a formal specification. A test template for an operation is a subset of the valid input space for the operation. We define Z types for the test templates of the operations.

\[
\begin{align*}
TT_{NN} &= P \ VIS_{NN} \\
TT_{IN} &= P \ VIS_{IN} \\
TT_{AN} &= P \ VIS_{AN} \\
TT_{RN} &= P \ VIS_{RN}
\end{align*}
\]

Test templates are derived from some other test template using a testing strategy. We introduce the set of all testing strategies without further specification.

[Strategy]

Test templates are organised into a test template hierarchy (TTH). The root of the hierarchy for an operation is the valid input space of that operation. The test template hierarchy for an operation is a digraph with test templates as nodes and testing strategies as arcs. The hierarchy is usually a tree and each derived test template is typically a subset of its parent in the hierarchy.

\[
\begin{align*}
TTH_{NN} : TT_{NN} \times \text{Strategy} \rightarrow P \ TT_{NN} \\
TTH_{IN} : TT_{IN} \times \text{Strategy} \rightarrow P \ TT_{IN} \\
TTH_{AN} : TT_{AN} \times \text{Strategy} \rightarrow P \ TT_{AN} \\
TTH_{RN} : TT_{RN} \times \text{Strategy} \rightarrow P \ TT_{RN}
\end{align*}
\]

3.2 Test templates

We derive testing information for the operations using two testing strategies: type-based selection and cause-effect partitioning.

A common, intuitive testing strategy is 0-1-many, based on the cardinality of a container type. We refer to this as (one example of) type-based selection. In contrast, cause-effect analysis involves partitioning the input space based on equivalence classes of the output space. We identify type-based selection (TB) and cause-effect analysis (CE) as particular testing strategies.

\[
\begin{align*}
TB, CE : \text{Strategy}
\end{align*}
\]

3rd Northern Formal Methods Workshop, 1998
For operation \textit{NoNodes}, we apply cause-effect analysis to derive two test templates and we establish their places in the hierarchy. The equivalence classes are based on the two possible values of the output variable \textit{result}! reporting whether \(xs\) is empty or not. We distinguish the new test templates by sequentially numbering the subscript. The parent of the test template being derived from (here \(VIS_{NN}\)) is incorporated by schema inclusion and the new constraint is included as the predicate part of the test template schema.

\[
TT_{NN.1} \triangleq [VIS_{NN} \mid xs = \emptyset] \\
TT_{NN.2} \triangleq [VIS_{NN} \mid xs \neq \emptyset] \\
TTH_{NN}(VIS_{NN}, CE) = \{ TT_{NN.1}, TT_{NN.2} \}
\]

Next we apply type-based selection to the newly derived templates, partitioning the valid input space into cases where \(xs\) is empty, a singleton, and contains more than one element. No further templates are derived from \(TT_{NN.1}\) since it only matches one case, but \(TT_{NN.2}\) yields two additional templates\(^5\).

\[
TT_{NN.2.2} \triangleq [TT_{NN.2} \mid \#xs = 1] \\
TT_{NN.2.3} \triangleq [TT_{NN.2} \mid \#xs > 1] \\
TTH_{NN}(TT_{NN.1}, TB) = \{ TT_{NN.2.2}, TT_{NN.2.3} \}
\]

We derive two test templates from operation \textit{IsNode} using cause-effect analysis, with the equivalence classes based on whether \(x?\) is or is not in the set of nodes.

\[
TT_{IN.1} \triangleq [VIS_{IN} \mid x? \in xs] \\
TT_{IN.2} \triangleq [VIS_{IN} \mid x? \notin xs] \\
TTH_{IN}(VIS_{IN}, CE) = \{ TT_{IN.1}, TT_{IN.2} \}
\]

We apply type-based selection to derive five further templates\(^6\).

\[
TT_{IN.1.2} \triangleq [TT_{IN.1} \mid \#xs = 1] \\
TT_{IN.1.3} \triangleq [TT_{IN.1} \mid \#xs > 1] \\
TTH_{IN}(TT_{IN.1}, TB) = \{ TT_{IN.1.2}, TT_{IN.1.3} \}
\]

\[
TT_{IN.2.1} \triangleq [TT_{IN.2} \mid \#xs = 0] \\
TT_{IN.2.2} \triangleq [TT_{IN.2} \mid \#xs = 1] \\
TT_{IN.2.3} \triangleq [TT_{IN.2} \mid \#xs > 1] \\
TTH_{IN}(TT_{IN.2}, TB) = \{ TT_{IN.2.1}, TT_{IN.2.2}, TT_{IN.2.3} \}
\]

We derive three test templates from \textit{AddNode} using type-based selection over \(xs\).

\[
TT_{AN.1} \triangleq [VIS_{AN} \mid \#xs = 0] \\
TT_{AN.2} \triangleq [VIS_{AN} \mid \#xs = 1] \\
TT_{AN.3} \triangleq [VIS_{AN} \mid \#xs > 1] \\
TTH_{AN}(VIS_{AN}, TB) = \{ TT_{AN.1}, TT_{AN.2}, TT_{AN.3} \}
\]

\(^5\) \(TT_{NN.2.1}\) contains the contradiction \(xs \neq \emptyset \land \#xs = 0\).

\(^6\) \(TT_{IN.1.1}\) contains the contradiction \(x? \in xs \land \#xs = 0\).
We derive three test templates from RemoveNode using type-based selection over $xs'$. Note that we are exploring values of $xs$ in the output space (0-1-many) and translating these to the input space (1-2-many).

\[
TT_{RN,1} \equiv [\text{VIS}_{RN} \mid \#xs = 1]
\]
\[
TT_{RN,2} \equiv [\text{VIS}_{RN} \mid \#xs = 2]
\]
\[
TT_{RN,3} \equiv [\text{VIS}_{RN} \mid \#xs > 2]
\]
\[
TTH_{RN}(\text{VIS}_{RN}, TB) = \{ TT_{RN,1}, TT_{RN,2}, TT_{RN,3} \}
\]

We choose to stop the derivations at this point, a decision that is left to the individual human tester under the TTF. Figure 1 is a graphical representation of the test template hierarchies derived in this section.

![Diagram of test template hierarchies](image)

Figure 1: Graphical representation of TTHs

### 3.3 Instances

We define instance templates for the leaf test templates of the test template hierarchy. The instance templates specify a choice of test data on the assumption that each element of a given test template is equivalent for testing purposes.

We define an individual member of $X$ to use in instance templates.

\[
| \quad x : X
\]

For example, the fully expanded form of $TT_{IN,1.2}$ is

\[
TT_{IN,1.2} \equiv [xs : F X ; x? : X \mid x? \in xs \land \#xs = 1]
\]

A possible instance template derived from $TT_{IN,1.2}$ is

\[
IT_{IN,1.2} \equiv [xs : F X ; x? : X \mid xs = \{x\} \land x? = x]
\]
3.4 Oracles

An oracle is some means of determining the result of a test. An oracle template is calculated by conjoining a test or instance template with the operation under test and then projecting the result onto the output space (OS) of the operation\(^7\). The general expression for an oracle template for a test template \(TT\) of operation \(Op\) is

\[
OT_{TT} \equiv (Op \land TT) \upharpoonright OS_{Op}
\]

The output and valid output spaces of \(IsNode\) are

\[
OS_{IN} \equiv [xs' : F \ X; \ result! : \mathbb{B}]
\]

\[
VOS_{IN} \equiv [xs' : F \ X; \ result! : \mathbb{B}]
\]

and the oracle template of instance template \(IT_{IN,1,2}\) is

\[
OT_{IN,1,2} \equiv [xs' : F \ X; \ result! : \mathbb{B} \mid xs' = \{x\} \land result! = \text{True}]
\]

3.5 Discussion

In this paper we propose two modifications to the framework: factoring out testing information derived from state components and deferring focusing on the input space of an operation. The first modification facilities reuse of derived testing information. The second modification supports use of testing strategies considering operation inputs and outputs.

We applied type-based selection over the value of \(xs\) to each of the operations of \(Nodes\); four times in the input space of an operation (twice for \(IsNode\) and once, for \(RemoveNode\), in the output space. Deriving these templates once and reusing them will reduce the derivation effort, particularly if simplifications or contradictions arise due to the state invariant (not the case here).

The existing test template approach has an early focus on the input space of the operation under test, resulting in problems for strategy application and simplification. One of the first steps in using the test template framework is to calculate the input space and the valid input space. The valid input space is then used as the root of the test template hierarchy. As a result, testing strategies that require the output space, such as cause-effect analysis, cannot be easily formalised since the information required for their application is not immediately available. For example, the application of cause-effect analysis for \(NoNodes\) and \(IsNode\) uses the value of \(result!\) which is not available in \(VIS_{NN}\) and \(VIS_{IN}\).

Similarly, testing strategies that could make use of the output space, such as the application of type-based selection for \(RemoveNodes\), do not have the relevant variables available (in this case \(xs'\)). Also, simplification of testing information typically uses operation results in addition to precondition constraints. Also, the initial step of calculating the valid input space involves existentially quantifying output and after-state variables over the predicate of the operation under test. Simplification of the quantified predicate may not be straightforward, possibly resulting in a complex valid input space.

The existing approach also results in problems for oracles. Oracle templates, which are constrained subsets of the output space, are calculated by reversing the initial focus on

\(^7\) Schema projection \(S \upharpoonright T\) hides all components of \(S\) except those that are shared by \(T\).
the input space (by conjoining the operation with the test or instance template). It has
proved difficult to formalise oracle templates due to limitations on Z type rules, principally
because schemas are not well integrated in the Z notation, as noted in [25]. In ‘ZRM’ Z
[20], schema expressions are distinguished from other expressions and our problems arise
in mixing the two. We have not investigated using the Z Standard [26] to deal with these
problems.

In this section we derived testing information for the Nodes operations using the existing
TTF approach. In the next section we demonstrate our modified approach by deriving
testing information for the same operations.

4 Modified TTF testing

We propose to modify the TTF derivation approach by factoring out state components to
facilitate their reuse in other operations and by delaying the focus on input and before-
state variables for as long as possible.

We achieve the first aim by deriving hierarchies for the state components of operations.
We achieve the second aim by making the operation the root of the test template hi-
erarchy, rather than its precondition. In Section 4.1 we derive testing information for
the state components and in Section 4.2 we derive testing information for the operations
using our modified approach.

4.1 State

In this section we derive hierarchies for the state components of operations. The testing
information contained in such hierarchies is factored out to facilitate sharing between the
operations. We use the Z schema calculus to combine the shared testing information with
operation-specific tests.

In conventional Z, the state space is represented by a schema. Operation schemas specify
relationship between input and output variables and a pair of states, the before- and
after-states. The name of the state schema is used to represent the before-state, and a
prime (') is added to represent the after-state.

A further convention is used to construct pairs of states for operations. A schema name
with \( \Delta \) prepended is implicitly defined as the unprimed and primed schemas with no
additional constraints. This signifies making all components available for change. A
schema name with \( \Xi \) prepended is implicitly defined as the unprimed and primed schemas
with each unprimed component equated to the corresponding primed component. This
signifies make all components available for reference.

We are only interested in deriving templates in either the input space or the output space
in any particular derivation step so we restrict our attention to the unprimed state schema
and signify the output space by primed templates. We derive a test template hierarchy
for Nodes (which as an after-state would be \( \text{Nodes}' \)).

The modified approach considers the entire state space of a component rather than fo-
cusing on the input (output) and valid input (output) spaces. The state space \( SS \) of a
component is the declaration part (signature) of the state schema.

\[
SS_{\text{Nodes}} \triangleq [x : F X]
\]

3rd Northern Formal Methods Workshop, 1998
In the modified TTF approach, we define a type for test templates as a subset of the component or operation under test, in contrast to the original TTF approach of defining a type for test templates as a subset of the valid input space of an operation. We use the name NewTT to distinguish templates defined over the entire state space from TTs which are defined over the input space\(^8\).

\[
\text{NewTT}_{\text{Nodes}} \equiv P \text{ Nodes}
\]

\[
\text{NewTT}_{\text{Nodes}} : \text{NewTT}_{\text{Nodes}} \times \text{Strategy} \rightarrow P \text{ NewTT}_{\text{Nodes}}
\]

We derive three test templates for Nodes using type-based selection over \(xs\). These can be transferred to the output space by referring to the primed template name (e.g., \(\text{NewTT}'_{\text{Nodes},1}\)).

\[
\text{NewTT}_{\text{Nodes},1} \equiv [\text{Nodes} \mid \#xs = 0]
\]

\[
\text{NewTT}_{\text{Nodes},2} \equiv [\text{Nodes} \mid \#xs = 1]
\]

\[
\text{NewTT}_{\text{Nodes},3} \equiv [\text{Nodes} \mid \#xs > 1]
\]

\[
\text{NewTTH}_{\text{Nodes}}(\text{Nodes}, \text{TB}) = \{\text{NewTT}_{\text{Nodes},1}, \text{NewTT}_{\text{Nodes},2}, \text{NewTT}_{\text{Nodes},3}\}
\]

It is technically possible to derive instance and oracle templates from NewTTs derived over the state schema. This is not particularly useful because these NewTTs will be combined with operation-specific information which will impose additional constraints on instance and oracle templates.

### 4.2 Operations

#### 4.2.1 Definitions

In the original TTF approach, a type for test templates is defined as a subset of the valid input space. In the modified approach, we define types for test templates based on the relation denoted by the operation rather than the valid input space, and we define corresponding test template hierarchies.

\[
\text{NewTT}_{\text{NN}} \equiv P \text{ NoNodes}
\]

\[
\text{NewTT}_{\text{IN}} \equiv P \text{ IsNode}
\]

\[
\text{NewTT}_{\text{AN}} \equiv P \text{ AddNode}
\]

\[
\text{NewTT}_{\text{RN}} \equiv P \text{ RemoveNode}
\]

\[
\text{NewTTH}_{\text{NN}} : \text{NewTT}_{\text{NN}} \times \text{Strategy} \rightarrow P \text{ NewTT}_{\text{NN}}
\]

\[
\text{NewTTH}_{\text{IN}} : \text{NewTT}_{\text{IN}} \times \text{Strategy} \rightarrow P \text{ NewTT}_{\text{IN}}
\]

\[
\text{NewTTH}_{\text{AN}} : \text{NewTT}_{\text{AN}} \times \text{Strategy} \rightarrow P \text{ NewTT}_{\text{AN}}
\]

\[
\text{NewTTH}_{\text{RN}} : \text{NewTT}_{\text{RN}} \times \text{Strategy} \rightarrow P \text{ NewTT}_{\text{RN}}
\]

\(^8\)Note that we are assuming the definition of Strategy from Section 3.1.
4.2.2 Test templates

For operation NoNodes, we apply cause-effect analysis to derive two test templates and we establish their places in the hierarchy. The effect, which is the source of the test, is explicit in the template, and the cause can be derived.

\[
\text{NewTT}_{\text{NN},1} \triangleq \text{NoNodes} \mid \text{result!} = \text{True}
\]
\[
\text{NewTT}_{\text{NN},2} \triangleq \text{NoNodes} \mid \text{result!} = \text{False}
\]
\[
\text{NewTT}_{\text{NH}}(\text{NoNodes}, \text{CE}) = \{\text{NewTT}_{\text{NN},1}, \text{NewTT}_{\text{NN},2}\}
\]

Next we apply type-based selection over the state variable \(xs\). The new templates are constructed by conjoining a test template and a state template\(^9\).

\[
\text{NewTT}_{\text{NN},2.1} \triangleq \text{NewTT}_{\text{NN},2} \land \text{NewTT}_{\text{Nodes},2}
\]
\[
\text{NewTT}_{\text{NN},2.2} \triangleq \text{NewTT}_{\text{NN},2} \land \text{NewTT}_{\text{Nodes},3}
\]
\[
\text{NewTT}_{\text{NH}}(\text{NewTT}_{\text{NN},1}, \text{TB}) = \{\text{NewTT}_{\text{NN},2.1}, \text{NewTT}_{\text{NN},2.2}\}
\]

We derive two test templates from operation IsNode using cause-effect analysis, and five further templates using type-based selection.

\[
\text{NewTT}_{\text{IN},1} \triangleq \text{IsNode} \mid \text{result!} = \text{True}
\]
\[
\text{NewTT}_{\text{IN},2} \triangleq \text{IsNode} \mid \text{result!} = \text{False}
\]
\[
\text{NewTT}_{\text{INH}}(\text{IsNode}, \text{CE}) = \{\text{NewTT}_{\text{IN},1}, \text{NewTT}_{\text{IN},2}\}
\]

\[
\text{NewTT}_{\text{IN},1.1} \triangleq \text{NewTT}_{\text{IN},1} \land \text{NewTT}_{\text{Nodes},2}
\]
\[
\text{NewTT}_{\text{IN},1.2} \triangleq \text{NewTT}_{\text{IN},1} \land \text{NewTT}_{\text{Nodes},3}
\]
\[
\text{NewTT}_{\text{INH}}(\text{NewTT}_{\text{IN},1}, \text{TB}) = \{\text{NewTT}_{\text{IN},1.1}, \text{NewTT}_{\text{IN},1.2}\}
\]

\[
\text{NewTT}_{\text{IN},2.1} \triangleq \text{NewTT}_{\text{IN},2} \land \text{NewTT}_{\text{Nodes},1}
\]
\[
\text{NewTT}_{\text{IN},2.2} \triangleq \text{NewTT}_{\text{IN},2} \land \text{NewTT}_{\text{Nodes},2}
\]
\[
\text{NewTT}_{\text{IN},2.3} \triangleq \text{NewTT}_{\text{IN},2} \land \text{NewTT}_{\text{Nodes},3}
\]
\[
\text{NewTT}_{\text{INH}}(\text{NewTT}_{\text{IN},2}, \text{TB}) = \{\text{NewTT}_{\text{IN},2.1}, \text{NewTT}_{\text{IN},2.2}, \text{NewTT}_{\text{IN},2.3}\}
\]

We derive three test templates from AddNode using type-based selection over \(xs\).

\[
\text{NewTT}_{\text{AN},1} \triangleq \text{AddNode} \land \text{NewTT}_{\text{Nodes},1}
\]
\[
\text{NewTT}_{\text{AN},2} \triangleq \text{AddNode} \land \text{NewTT}_{\text{Nodes},2}
\]
\[
\text{NewTT}_{\text{AN},3} \triangleq \text{AddNode} \land \text{NewTT}_{\text{Nodes},3}
\]
\[
\text{NewTT}_{\text{ANH}}(\text{AddNode}, \text{TB}) = \{\text{NewTT}_{\text{AN},1}, \text{NewTT}_{\text{AN},2}, \text{NewTT}_{\text{AN},3}\}
\]

\(^9\)An alternative definition is

\[
\text{NewTT}_{\text{NN},2.2} \triangleq [\text{NewTT}_{\text{NN},2} \mid \text{NewTT}_{\text{Nodes},2}]
\]

3rd Northern Formal Methods Workshop, 1998
We derive three test templates from \textit{RemoveNode} using type-based selection over $xs'$, expressed by conjoining primed state templates.

\[
\begin{align*}
\text{NewTT}_{RN,1} & \triangleq \text{RemoveNode} \land \text{NewTT}^{\prime}_{Nodes,1} \\
\text{NewTT}_{RN,2} & \triangleq \text{RemoveNode} \land \text{NewTT}^{\prime}_{Nodes,2} \\
\text{NewTT}_{RN,3} & \triangleq \text{RemoveNode} \land \text{NewTT}^{\prime}_{Nodes,3} \\
\text{NewTT}_{R,T}(\text{RemoveNode}, TB) & = \{ \text{NewTT}_{RN,1}, \text{NewTT}_{RN,2}, \text{NewTT}_{RN,3} \}
\end{align*}
\]

The graphical representation of the test template hierarchies derived in this section is identical to Figure 1, i.e., we derive the same templates by different means.

### 4.2.3 Reified templates

The usual step at this point is to define instance templates that specify a unique choice of test data defining a specific test input. We have deferred the focus on input so we define reified templates (RT) that specify a unique choice of test data defining both test input and output.

For example, the fully expanded form of $\text{NewTT}_{IN,1,2}$ is

\[
\text{NewTT}_{IN,1,2} \triangleq [xs : F \ X; x? : X \mid x? \in xs \land \#xs = 1]
\]

A possible reified template derived from $\text{NewTT}_{IN,1,2}$ (using $x : X$ from Section 3.3) is

\[
\begin{align*}
\text{RT}_{IN,1,2} & \triangleq [xs, xs' : F \ X; x? : X; \text{result}! : \mathbb{B} \mid \\
& xs = \{ x \} \land x? = x \land xs' = \{ x \} \land \text{result}! = \text{True}]
\end{align*}
\]

### 4.2.4 Inputs and outputs

The step from reified test data to specific test inputs and outputs is straightforward. We define reified input and output templates (RIT and ROT respectively) using the Z precondition operation (\textit{pre}), and a corresponding postcondition operation (\textit{post})\textsuperscript{10}. The reified input and output templates for $\text{NewTT}_{IN,1,2}$ are defined as

\[
\begin{align*}
\text{RIT}_{IN,1,2} & \triangleq \text{pre } \text{RT}_{IN,1,2} \\
\text{ROT}_{IN,1,2} & \triangleq \text{post } \text{RT}_{IN,1,2}
\end{align*}
\]

which expand to

\[
\begin{align*}
\text{RIT}_{IN,1,2} & \triangleq [xs : F \ X; x? : X \mid xs = \{ x \} \land x? = x] \\
\text{ROT}_{IN,1,2} & \triangleq [xs' : F \ X; \text{result}! : \mathbb{B} \mid xs' = \{ x \} \land \text{result}! = \text{True}]
\end{align*}
\]

\textsuperscript{10}We use a symmetric definition of \textit{post}, i.e., remove input and before-state variables from the schema signature and existentially quantify them over the schema predicate.
4.3 Discussion

We have proposed modifying the TTF by factoring state-specific testing information and deferring focusing on the input space. The first modification is achieved by deriving testing information from the state schema and incorporating the resulting templates by name rather than as predicates included on an operation by operation basis. The second modification is achieved by deriving testing information from the operation schema rather than its precondition, deferring the move to the input space until concrete test data has been derived.

Although these modifications achieve our aims of facilitating reuse and supporting testing strategies that use operation outputs, there is some cost. Previous case studies have observed that tool support is essential to manage bookkeeping tasks associated with any but the most trivial TTF derivations. Reusing state-based templates by name will permit more complex strategy applications but will strengthen the requirement for tool support to enable expansion of included schemas.

Also, in a sense we have exchanged reuse of template derivation for reuse of precondition simplification. In the existing TTF, the precondition is calculated and simplified once for each operation. In the modified TTF, the precondition is calculated for each reified template. Storage and application of simplification steps as tactics (in the theorem prover sense) is an obvious area for tool support.

There is a problem with our second modification that may affect other approaches that support testing based on outputs. The problem can arise when tests are based on outputs in the context of non-determinism. Consider an abstract non-deterministic square root operation with tests defined over the output space.

\[
Asqrt \triangleq [i? : \mathbb{N}; o! : Z \mid i? = o!* o!]
\]

\[
NewTT_{Asqrt} = = P\ Asqrt
\]

\[
|\ NewTT_{Asqrt} : NewTT_{Asqrt} \times Strategy \rightarrow P\ NewTT_{Asqrt}
\]

\[
NewTT_{Asqrt,1} \triangleq [Asqrt \mid o! \geq 0]
\]

\[
NewTT_{Asqrt,2} \triangleq [Asqrt \mid o! < 0]
\]

\[
NewTT_{Asqrt}(Asqrt, TB) = \{NewTT_{Asqrt,1}, NewTT_{Asqrt,2}\}
\]

We could reasonably define symmetrical reified templates from these tests with the obvious input and output templates.

\[
RT_{Asqrt,1} \triangleq [i? : \mathbb{N}; o! : Z \mid i? = 1 \land o! = 1]
\]

\[
RT_{Asqrt,2} \triangleq [i? : \mathbb{N}; o! : Z \mid i? = 1 \land o! = \leftrightarrow]
\]

However, an implementation of Asqrt can reduce the non-determinism in the output and could, for example, perform as follows (note that o! is of type \(\mathbb{N}\) rather than \(\mathbb{Z}\)).

\[
Csqrt \triangleq [i? : \mathbb{N}; o! : \mathbb{N} \mid i? = o!* o!]
\]

Performing the test corresponding to \(RT_{Asqrt,2}\) on such an implementation will return a false failure. We are still considering solutions to this problem.
The modified TTF presented in this paper has much in common with the approaches of Dick et al., Hall and Hierons, and Stepney. Also, Hörcher et al. extend the work of Dick et al. to Z, and Donat derives testing information using an input-output relationship based on implication rather than conjunction.

Dick et al. [1, 4] present techniques for using a model-based specification (written in VDM) to generate test cases and sequence tests. They derive test cases, termed sub-operations, by reducing operations to a canonical disjunctive normal form (DNF). Test sequencing is achieved by constructing a finite state automaton, using the sub-operations as transitions and the disjoined sub-operation before- and after-states (input and output spaces) as nodes. The modified TTF similarly provides a symmetrical treatment of input and output spaces, but permits use of a variety of testing strategies, including DNF partitioning.

Hörcher et al. [12, 11, 16] use DNF partitioning to derive test inputs from Z specifications and have developed a Z predicate compiler to automate test result evaluation.

Hall and Hierons [7, 8] also describe the generation of a finite state machine from a Z specification. The input domain is partitioned using the category-partition method of Ostrand and Balcer [19]. The Test Template Framework, in contrast, permits use of a variety of testing strategies, including category-partition. The states for the finite state machine are calculated by rewriting the specification as disjoined input and output predicates (effectively preconditions and postconditions). The transitions are determined by pairwise consideration of states as operation pre- and postconditions. The effect is similar to the machine constructed by Dick et al. Hierons also notes previous work on test control and test sequencing from finite state machines but does not pursue these aspects.

Donat [5] also converts specifications to predicate logic but uses conjoined implications of the form $(S_i \Rightarrow R_i) \wedge \ldots$, where $S$ and $R$ are stimulus and response expressions respectively. A stimulus is a predicate that refers only to input and before-state variables, and a response is a predicate containing at least one reference to an output or after-state variable. The modified TTF input-output relationship is disjoined conjunctions as used by Dick et al. and Hall and Hierons.

Stepney [21] describes a method for formally specifying testing information based on applying abstraction to a specification. The method uses the ZEST object-oriented variant of Z [3], and involves systematically introducing non-determinism into the specification. The method includes three techniques: conjunctive partitioning, simplification by hypothesis introduction, and weakening of after-state constraints. A firing condition interpretation of precondition is used, the method is supported by the Zyla tool, and testing information is structured as an inheritance tree of specifications. The inheritance tree corresponds to the TTF test template hierarchy but the nodes are related by abstraction rather than strategy application. Input and output spaces are not treated symmetrically due to differing abstraction (refinement) relationships. Stepney’s method is more closely aligned with formal development than the TTF, but the TTF is more firmly grounded in software testing.

In related work of our own, we are extending the TTF to accommodate specifications written in Object-Z [6]. In [18], we adapt the TTF to object-oriented formal specifications, deriving testing information for operations (test inputs and expected outputs) from
Object-Z specifications, and inheriting testing information from parent classes. In [17], we focus on testing the class as a whole, achieved by deriving a class’ finite state machine from its specification, which allows us to formulate sequences of the class’ operations to conduct testing. In [15], we present a case study demonstrating a complete process for object-oriented specification-based testing, including integration with the Classbench test execution framework [9, 10]. The latter studies suggested some of the modifications presented in this paper.

6 Conclusion

In this paper we proposed two modifications to the Test Template Framework: first, that testing information derived from state components be distinguished from operation-specific information, and second, that focusing on the input space of an operation under test be deferred for as long as possible. The first modification facilitates reuse of derived testing information and the second supports testing strategies that use operation outputs. As mentioned above, we are extending the TTF to accommodate specifications written in Object-Z. We are also extending the TTF to accommodate multiparadigm specifications of interactive systems, involving model-based and behavioural sub-specifications. The symmetrical treatment of input and output spaces proposed in this paper is an important step in that work, enabling consideration of sequences of operation invocations with the output of one invocation establishing the input of another.

Acknowledgements

The dependency management system (DMS) was originally specified in Object-Z by Gordon Rose in at least two different versions. In this paper we have used a Z specification developed by Phil Stocks [22] that was derived from one of Gordon’s specifications. We also acknowledge the contribution of our colleague, Paul Strooper, for a suggestion that facilitated the state factoring aspect of this paper.

We thank Mark Utting for pointing out the problem associated with deriving tests in the output space in the context of non-determinism and we also thank the anonymous reviewers for their detailed and helpful comments.

Ian MacColl is supported by an Australian Postgraduate Award and a Telstra Research Laboratories Postgraduate Fellowship.

References


3rd Northern Formal Methods Workshop, 1998


