SOFTWARE VERIFICATION RESEARCH CENTRE

SCHOOL OF INFORMATION TECHNOLOGY

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 98-22

From Object-Z Specifications to ClassBench Test Suites

David Carrington   Ian MacColl
Jason McDonald   Leesa Murray
Paul Strooper

October 1998

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533
http://svrc.it.uq.edu.au

From Object-Z Specifications to ClassBench Test Suites

David Carrington Ian MacColl Jason McDonald Leesa Murray
Paul Strooper

Abstract

This paper describes a method for specification-based class testing that incorporates test case generation, execution, and evaluation based on formal specifications. This work builds on previous achievements in the areas of specification-based testing and class testing by integrating the two within a single framework.

The initial step of the method is to generate test cases for individual operations from a specification written in the Object-Z specification language. These test cases are combined to produce a finite state machine that is used as the basis for test case execution with the ClassBench test execution framework. An oracle derived from the Object-Z specification is used to evaluate the outputs generated by the test cases. We illustrate the method using a simple example and describe our experience applying it to a more substantial case study.

1 Introduction

Despite its limitations, testing is still the most popular form of verification and validation used in industry. In traditional software development, the software or system under test is executed to produce *failures*, so that the *faults* that cause those failures can be identified and removed. Formal specification and development attempt to shift the emphasis from fault removal to fault prevention. Although fully formal development is possible, formal refinement from a specification into an implementation rarely happens in practice. However, even with fully formal development, there is a need for testing [4]. Whether the software is developed formally or specified formally and then developed informally, the specification can play a major role in the testing process, by allowing the generation of test inputs and the expected outputs for these inputs.

Although a considerable amount of work has been done in the area of class testing [2], most of this work is not formally based. In this paper, we present a method for specification-based class testing, applying the Test Template Framework [33] to an Object-Z specification [7, 8] of the class under test to generate a state machine that allows the class's implementation to be tested with the ClassBench testing framework [12, 13].

Our approach to specification-based class testing is shown in Figure 1. The ovals represent processes and the rectangles represent testing and development artifacts. In [26], we introduce object-orientation to the Test Template Framework [33], a framework originally designed for non-object-oriented specification-based testing. We use the Test Template Framework to formally select inputs and expected outputs for each operation from an Object-Z specification of the class under test. This testing information is analysed to develop a finite state machine representing the class under test [25]. The finite state machine is then transformed into a ClassBench *testgraph* [27]. The specification is also used to construct a ClassBench oracle class [22] for evaluating test results, and is developed separately into an implementation. The implementation is tested under ClassBench, using the testgraph and the oracle class.

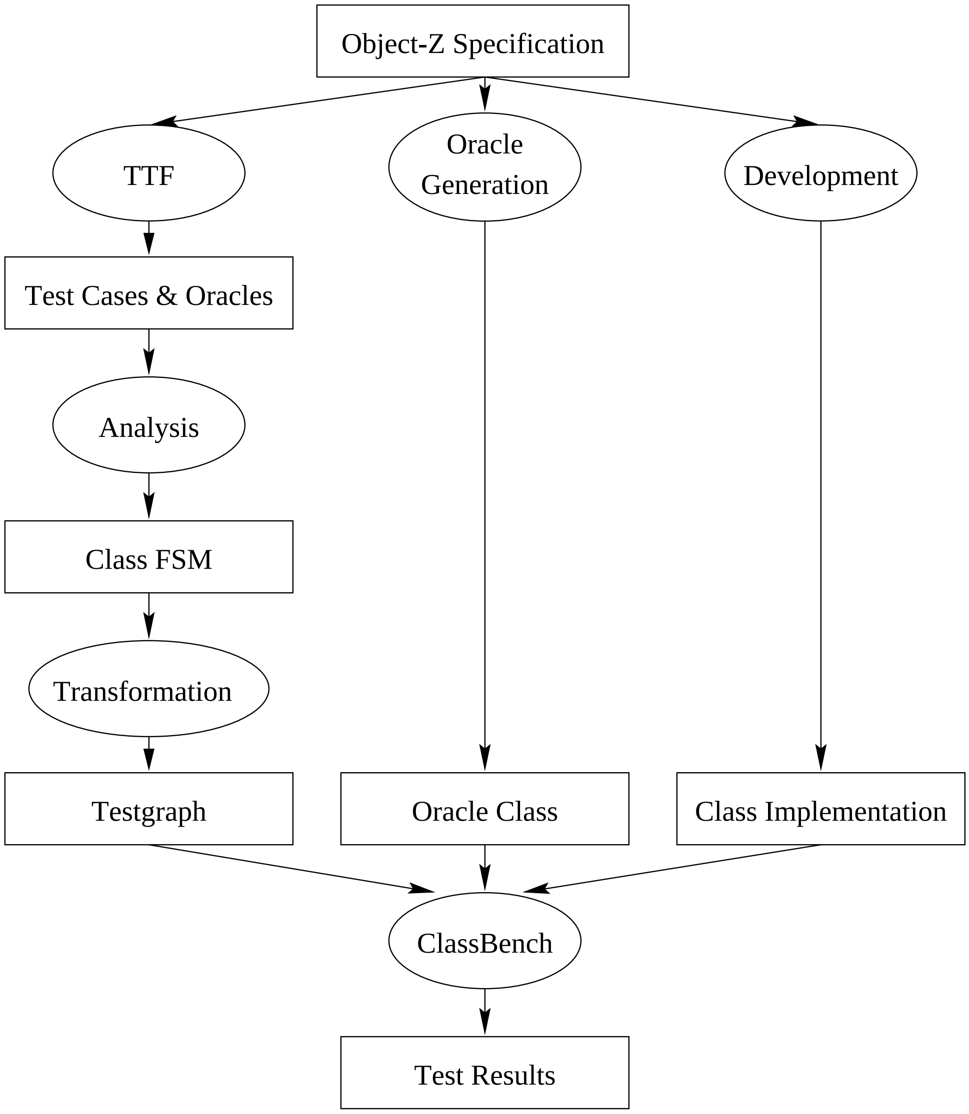

Figure 1: Our approach to specification-based class testing

The Test Template Framework and the ClassBench methodology both support object-oriented features such as genericity and inheritance. However, in this paper we do not discuss these features in detail and instead focus on testing of base classes.

As indicated above, most of the individual steps in our approach have been described elsewhere. In [19], we illustrate all the steps in this approach on a simple class that forms part of a Dependency Management System, but without discussing any of the steps in detail. We use the same example in this paper but we describe each step in our method fully. For each step, we describe the process involved in that step, illustrate how it is applied to the example, and briefly discuss issues related to that step. In addition, we discuss how the method was applied to the full Dependency Management System, a significant case study that raises several interesting issues.

Our paper is organised as follows. In Section 2 we introduce the example that we use through the rest of the paper. Section 3 introduces our object-oriented extension of the Test Template Framework, and in Section 4 we describe how to construct a finite state machine from test inputs and expected outputs for the class under test. Section 5 presents a brief introduction to ClassBench, and in Sections 6 and 7 we describe how to generate a ClassBench testgraph from a finite state machine and an oracle from an Object-Z specification. In Section 8 we discuss our experience in applying our method to the full Dependency Management System. Section 9 presents related work and Section 10 concludes the paper.

Throughout the paper we use *italic* text when referring to specification items, and `monospaced` text when referring to implementation items.

2 Dependency Management System Specification

In this section we present a specification of part of a Dependency Management System, which we use as an example throughout the remainder of the paper. In our approach, an implementation, tests and an oracle are all developed from such a specification. Development of an implementation requires decisions that have an impact on testing. We describe an independently developed implementation of the Dependency Management System and discuss issues that arise in testing an implementation of a formal specification.

The Dependency Management System case study was a testbed for a project exploring software development methodologies [18]. Such a system is a critical component of a theorem proving tool, tracking dependencies between theorems and assertions in a proof to prevent circular reasoning. However, the concept of dependency management generalises to other problems (such as revision control systems), so the Dependency Management System is a reasonably generic component.

Our experience with testing the full Dependency Management System is discussed in Section 8 and the complete specification is presented in Appendix A.

2.1 Specification

A Dependency Management System tracks dependencies between nodes, with each node representing an item under management. Such a system maintains a set of nodes, direct dependencies between nodes, and inferred transitive dependencies between nodes. Operations are provided for manipulating a dependency graph, from adding nodes and dependencies to calculating all the nodes dependent on some other node. The Object-Z specification in Figure 2 describes only those features of the Dependency Management System associated with the set of nodes.

An Object-Z class is represented as a named box with optional generic parameters. The original Dependency Management System specification includes a generic parameter, representing the type of the nodes stored in the Dependency Management System. We have removed this generic parameter for clarity and represent the nodes by integers. The full Dependency Management System case study does use generic parameters as discussed in Section 8.

An Object-Z class contains an unnamed state schema, an initialisation schema (INIT) and zero or more operation schemas (four in this case). As in Z [30], each schema is a named box with an upper declaration part and a lower predicate part constraining the values of the declared variables.

The state variable s represents the finite set of nodes managed by the system. There is no class invariant (i.e., the unnamed state schema has no predicate part) and initially the set is empty.

Set

$s : \mathbb{F}\,\mathbb{Z}$

INIT

$s = \varnothing$

IsEmpty

$result! : \mathbb{B}$

$result! \Leftrightarrow s = \varnothing$

IsIn

$x? : \mathbb{Z}$

$result! : \mathbb{B}$

$result! \Leftrightarrow x? \in s$

Add

$\Delta(s)$

$x? : \mathbb{Z}$

$x? \notin s$

$s' = s \cup \{x?\}$

Remove

$\Delta(s)$

$x? : \mathbb{Z}$

$x? \in s$

$s' = s \setminus \{x?\}$

Figure 2: Object-Z specification of a set of integers

In Z and Object-Z, an operation schema is a transition from one subset of the state space to another, with associated input and output variables. State variables prior to the transition are termed *pre-state* variables and are shown unprimed. The corresponding *post-state* variables have the same identifiers with a prime ($'$) appended. The input and output variables are indicated by query (?) and exclamation (!) decorations respectively. In an Object-Z operation schema, state variables are implicitly equated to their primed post-state equivalents unless included in a Δ-list.

Operation *IsEmpty* returns true if and only if s is empty. Operation *IsIn* returns true if and only if the input node $x?$ is a member of s. The output variable *result*! corresponds to the returned value in each case. *IsEmpty* and *IsIn* do not change the state of the class, indicated by the lack of Δ-lists.

Node $x?$ is added to s by *Add* (note that s is in the Δ-list and so can be changed). An Object-Z operation is disabled outside its precondition, in contrast to Z in which precondition violation gives an undefined result (interpreted as a client responsibility). *Add* is disabled if $x?$ is already in s. Operation *Remove* 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.

2.2 Implementation

The Dependency Management System specification was independently implemented as a generic C++ class with a complex inheritance structure. An excerpt of the actual C++ header file of the implementation is given in Figure 3. As for the specification, we have instantiated generic parameters to integers (`int`) for clarity.

The C++ implementation provides the same interface as the Object-Z specification but it is constructed as a C++ class that inherits from a `DAG` class, and instantiates a `Set` class as a protected data member. The fragment of the class in Figure 3 shows two constructors and the four specified operations.

```
class DepManSys : public DAG {
protected:
        Set *s;
public:
        // constructor
        DepManSys() : DAG() {s = new Set();}
        // constructor
        DepManSys( Set *n) : DAG(),s(n) {}
        // any nodes
        bool IsEmpty() {return s->IsEmpty();}
        // is x a node
        bool IsIn( int *x ) {return s ->IsIn(x);}
        // add a node
        void Add( int *x );
        // remove a node
        void Remove( int *x );
}
```

Figure 3: Excerpt of Dependency Management System implementation in C++

2.3 Discussion

Developing an implementation involves making decisions that affect specification-based testing. The implementation is actually tested in ClassBench so these decisions affect the inputs to ClassBench provided by the tester. We will not attempt to provide an exhaustive list of issues to be considered in implementing a specification, but discuss those issues that have affected our testing.

An implementation is a concrete expression of an abstract specification, which typically involves constraining types and values to permit execution. For example, types that are specified as given types are defined to be infinite and must be mapped to finite types in most implementation languages, with associated overflow or cardinality issues arising as a result. Here, for example, an infinite type $\mathbb{Z}$ is mapped to a finite type `int`.

The issue of implementation constraints is particularly important for the test oracle that is used to evaluate the results of testing. McDonald [20] identifies rules to which the interface of an implementation must conform in order for a C++ oracle to be developed from an Object-Z specification, covering issues such as:

- ordering and naming of generic parameters;
- definition of given types, free types, and specification and class constant values;
- mapping of specification initialisation to implementation constructor(s); and
- operation visibility and signature compatibility.

Also, in some cases, including our example, the development of a test oracle requires access to the state of the implementation. We construct an oracle by inheriting from the class under test, but the addition of extra 'get' methods to access state, or (in C++) `friend` declarations are alternative mechanisms. Without access to the state, it is difficult to evaluate the results of tests.

Genericity is another issue that has an impact on testing. Object-Z permits genericity. If the specification is to be implemented in a language without a mechanism for genericity then generic testing information must be mapped to concrete testing information in the same way the specification is mapped to the implementation. If, as in C++, the implementation language permits genericity, then the genericity in the specification can be preserved in the implementation. However, instantiation of implementation classes for execution and testing requires instantiation of generic parameters. Generic testing information must be mapped to conform with such instantiation.

Other issues arise from the particular choices of specification and implementation notations. For example, Object-Z permits output variables but does not include a notion of return value as found in C++. Operations with single outputs, particularly Boolean checking operations, can be implemented by methods returning the appropriate value. In our example, the specification output variable *result*! in operations *IsEmpty* and *IsIn* is replaced by the return values of the corresponding implementation methods. This is a particular example of signature compatibility, mentioned above.

Decisions concerning preconditions may also have an impact on testing information. In Object-Z an operation is disabled outside its precondition. A decision must be made as to how precondition violation is to be handled, for example, by gracefully ignoring it or by throwing an exception.

In our example, an implementation variable `success` was used to return the success or failure of an operation, including precondition violation. Operations *Add* and *Remove* each have a precondition, namely, that the input element $x?$ is respectively not in and in the set. The implementation of *Add* is actually looser than specified. It accepts an existing element (without adding it to the set) and returns with `success` true. Our approach does not test this unspecified behaviour since our tests are all drawn from the valid input space of the operation, that is, within the precondition. Our approach could be extended to test how operation disabling is implemented. For example, both ClassBench and our method for generating C++ oracles from Object-Z specifications support exceptions.

3 Object-oriented Test Template Framework

The Test Template Framework provides our initial step in generating tests from a specification. This step is identified as the *TTF* process in Figure 1. We select test cases for each operation of the class to partition the state space of the class. In Section 4, we combine the partitions from all operations to form a finite state machine representing the class under test.

The framework, which was originally developed by Stocks [32], provides a systematic way to select and record test cases for individual operations from a model-based formal specification. Test case selection remains under the control of the human tester and multiple testing strategies are supported. The original work focused on specifications written in the Z notation [30]. Recent research has extended the framework to specifications written in Object-Z, with a focus on class-level testing that takes account of object-oriented features like inheritance [25, 26].

3.1 Process

The framework provides a systematic way of finding partitions for each operation. The precondition of an operation is used as the starting point for partitioning since Object-Z operations are disabled for inputs outside the precondition. Hence any testing outside the precondition is testing for robustness. The precondition is called the *valid input space* (VIS). The valid input space is a subset of the *input space* (IS) of an operation, which is defined as the restriction of the operation's signature to input components (inputs and pre-state components). The *output space* (OS) is similarly defined over the output components.

The basic unit for defining test data in the framework is a *test template* (TT). A test template is a generic description of test data, describing only the necessary restrictions on the data and leaving everything else abstract. Templates are defined at the same level of abstraction as the specification. Each template is a constrained subset of the valid input space and is expressed as a Z schema.

A *strategy* in the framework identifies a particular technique for selecting test cases. Both traditional techniques such as input partitioning and boundary analysis, and specialised techniques that exploit the specification notation are used. The framework encourages the use of multiple strategies to build the *test template hierarchy* (TTH). The hierarchy captures the relationships between test templates and strategies, and serves as a record of the test development process. The root of the hierarchy is the valid input space.

The hierarchy is created by applying testing strategies to existing test templates to derive additional ones. A test template hierarchy is usually a directed acyclic graph with the leaf nodes partitioning the valid input space. Strategies are applied to the hierarchy until the tester is satisfied that the leaf templates of the hierarchy represent adequate sources of tests, that is, every instantiation of a leaf template is equally likely to reveal an error in an implementation.

The framework also defines oracle templates corresponding to test templates. These define the expected final state and outputs corresponding to a given test template. An oracle template is calculated by restricting the operation to the input described in the corresponding test template and then projecting onto the operation's output space. Although we do not use these oracles during test execution, they are useful for identifying additional partitions of the state space when the finite state machine is constructed.

3.2 Example

We demonstrate how the framework can be used by selecting test templates for the *Add* operation defined in Section 2.

Add

$\Delta(s)$
$x? : \mathbb{Z}$

$x? \notin s$
$s' = s \cup \{x?\}$

The valid input space for *Add* is:

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

which expands and simplifies to

$$VIS_{Add} \mathrel{\widehat=} [s : \mathbb{F}\,\mathbb{Z};\ x? : \mathbb{Z} \mid x? \notin s]$$

Here we have represented the definition for VIS_{Add} as a horizontal Z schema, which is an alternative to the vertical schema forms used in an Object-Z class definition for state and operations. We define a type for all *Add* test templates (representing the set of all possible subsets of its valid input space):

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

We also define the *Add* test template hierarchy (taking the set of all possible testing strategies as given):

$$[\mathit{Strategy}]$$

$$TTH_{Add} : TT_{Add} \times \mathit{Strategy} \rightarrow\!\!\!\!\mapsto \mathbb{P}\, TT_{Add}$$

The function TTH_{Add} captures the combination of test templates with testing strategies to derive additional test templates. For most strategies, the children partition the parent.

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. We identify type-based selection (TB) as a particular testing strategy and we use it to partition the valid input space into cases where s is empty, a singleton, and contains more than one element, generating three test templates (distinguished with numeric subscripts) as a result.

$$TB : \mathit{Strategy}$$

$$\begin{array}{l} TT_{Add.1} \mathrel{\widehat=} [\, VIS_{Add} \mid \#s = 0\,] \\ TT_{Add.2} \mathrel{\widehat=} [\, VIS_{Add} \mid \#s = 1\,] \\ TT_{Add.3} \mathrel{\widehat=} [\, VIS_{Add} \mid \#s > 1\,] \end{array}$$

$$TTH_{Add}(VIS_{Add}, TB) = \{TT_{Add.1}, TT_{Add.2}, TT_{Add.3}\}$$

Although we can continue by applying other testing strategies to these resulting test templates, for this simple example we decide to stop with the above test templates as leaf templates.

The oracles for the above leaf templates are

$$\begin{array}{l} OT_{Add.1} \mathrel{\widehat=} (Add \land TT_{Add.1}) \upharpoonright OS_{Add} \equiv [\, s' : \mathbb{F}\,\mathbb{Z} \mid \#s' = 1\,] \\ OT_{Add.2} \mathrel{\widehat=} (Add \land TT_{Add.2}) \upharpoonright OS_{Add} \equiv [\, s' : \mathbb{F}\,\mathbb{Z} \mid \#s' = 2\,] \\ OT_{Add.3} \mathrel{\widehat=} (Add \land TT_{Add.3}) \upharpoonright OS_{Add} \equiv [\, s' : \mathbb{F}\,\mathbb{Z} \mid \#s' > 2\,] \end{array}$$

3.3 Discussion

For brevity, we do not show the selection of test templates for the other operations. For the *Remove* operation, we apply a similar type-based strategy. For the *IsEmpty* and *IsIn* operations, we apply a cause-effect strategy to the value of the *result*! output. A total of ten leaf test templates are generated from the four operations of the *Set* class. A diagrammatic representation of the four test template hierarchies is shown in Figure 4.

4 Derivation of a Class's Finite State Machine

This section details the development of a class's finite state machine from testing information derived from its Object-Z specification. This is identified as the *Analysis* process in Figure 1.

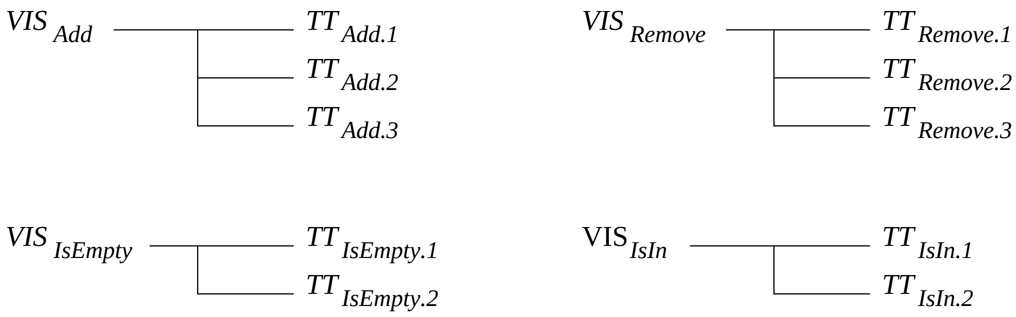

Figure 4: Test template hierarchies for the operations of *Set*

4.1 Process

We identify the states of the finite state machine from the *INIT* schema and the leaf test and oracle templates for all operations. When selecting test templates from an operation's specification, we partition the operation's input space. However, while some operation invocations will transition to a point where further invocations can occur (i.e., the input and outputs spaces overlap), some may transition to a state outside the input space where no further invocations can occur. Further, the equivalence classes in the output and input spaces may also overlap, providing an opportunity to increase the number of states. Therefore, it is also necessary to use the oracles of corresponding test templates to identify states for the finite state machine. We consider oracle templates only for operations that change the state of the class, as operations that do not change the state cannot contribute any new states. The collection of states gained from the *INIT* schema and the test and oracle templates gives us the set of states for the class's finite state machine, referred to as *state templates* (ST).

To derive state templates from test templates, we use schema hiding to restrict the signatures of the templates to just the class's state variables. Hiding involves removing the input variables from the template's declaration and existentially quantifying them over the template's predicate.

To derive state templates from oracle templates, we rename the primed state variables to their unprimed equivalents. Any output variables in the oracle templates are hidden and the templates simplified, so only the class's state variables appear in the declarations.

Once the set of state templates is identified, we consider whether the templates are disjoint. If they are not, it is necessary to resolve any overlap of states so that we have a maximal partition[1] of the class's state space. To achieve this, we use the canonical disjunctive normal form (DNF) to construct a partition, based on the technique described by Dick and Faivre [5], using equivalences of the form

$$A \lor B \equiv (A \land B) \lor (\neg\, A \land B) \lor (A \land \neg\, B)$$

If the *INIT* schema is partitioned in the process of formulating disjoint state templates, the subscript *Init* is added to the names of the resulting templates. This identifies those templates that are initial states of the class's finite state machine. We represent each initial state by an incoming arc that does not originate in a state and has no label.

We then identify the transitions of the finite state machine by considering each pair of state templates and checking whether the pair is related by each operation (see [25] for a

[1]By maximal partition we mean that we make the 'granularity' of the partition as small as possible, i.e., we minimise the amount of the state space covered by each state template in the partition, or, conversely, we maximise the number of state templates.

formal model of this process). A valid transition occurs when a pair of states, (ST_i, ST_j), is related by an operation. For each valid transition identified, an arc labelled with the operation's name, beginning in state ST_i and terminating in state ST_j, is added to the class's finite state machine.

Once all the valid transitions are identified, we determine if all states in the finite state machine are reachable. By reachable, we mean that there is a sequence of transitions from an initial state to the state in question. Any unreachable states are removed from the finite state machine, along with any transitions that originate or terminate in them.

4.2 Example

The state template identified from the *INIT* schema is

$$ST_{Init} \mathrel{\widehat=} [s : \mathbb{F}\,\mathbb{Z} \mid s = \varnothing]$$

The leaf test templates for the *Add* operation (selected in Section 3 and shown with VIS_{Add} fully expanded here) are:

$$\begin{aligned}
TT_{Add.1} &\mathrel{\widehat=} [s : \mathbb{F}\,\mathbb{Z};\ x? : \mathbb{Z} \mid \#s = 0]\\
TT_{Add.2} &\mathrel{\widehat=} [s : \mathbb{F}\,\mathbb{Z};\ x? : \mathbb{Z} \mid x? \notin s \land \#s = 1]\\
TT_{Add.3} &\mathrel{\widehat=} [s : \mathbb{F}\,\mathbb{Z};\ x? : \mathbb{Z} \mid x? \notin s \land \#s > 1]
\end{aligned}$$

To formulate state templates, we take each test template, hide the input variables and simplify.

$$\begin{aligned}
ST_1 &\mathrel{\widehat=} TT_{Add.1} \setminus (x?) \equiv [s : \mathbb{F}\,\mathbb{Z} \mid \#s = 0]\\
ST_2 &\mathrel{\widehat=} TT_{Add.2} \setminus (x?) \equiv [s : \mathbb{F}\,\mathbb{Z} \mid \#s = 1]\\
ST_3 &\mathrel{\widehat=} TT_{Add.3} \setminus (x?) \equiv [s : \mathbb{F}\,\mathbb{Z} \mid \#s > 1]
\end{aligned}$$

For operations that change the state of *Set* (*Add* and *Remove*) we calculate oracle templates by conjoining each template with the operation and projecting onto the operation's output space (OS). State templates are formed from oracle templates by renaming the primed state variables to their unprimed equivalents, and hiding any output variables. The oracle templates for *Add* and their corresponding state templates are

$$OT_{Add.1} \mathrel{\widehat=} (Add \land TT_{Add.1}) \upharpoonright OS_{Add} \equiv [s' : \mathbb{F}\,\mathbb{Z} \mid \#s' = 1]$$
$$ST_4 \mathrel{\widehat=} [s : \mathbb{F}\,\mathbb{Z} \mid \#s = 1]$$

$$OT_{Add.2} \mathrel{\widehat=} (Add \land TT_{Add.2}) \upharpoonright OS_{Add} \equiv [s' : \mathbb{F}\,\mathbb{Z} \mid \#s' = 2]$$
$$ST_5 \mathrel{\widehat=} [s : \mathbb{F}\,\mathbb{Z} \mid \#s = 2]$$

$$OT_{Add.3} \mathrel{\widehat=} (Add \land TT_{Add.3}) \upharpoonright OS_{Add} \equiv [s' : \mathbb{F}\,\mathbb{Z} \mid \#s' > 2]$$
$$ST_6 \mathrel{\widehat=} [s : \mathbb{F}\,\mathbb{Z} \mid \#s > 2]$$

State templates are derived from the leaf test templates of *Remove*, *IsEmpty* and *IsIn* and from the oracle templates of *Remove* in a similar way.

Removing duplicates, our complete collection of state templates is

$$\begin{aligned}
ST_{Init} &\mathrel{\widehat=} [s : \mathbb{F}\,\mathbb{Z} \mid s = \varnothing]\\
ST_2 &\mathrel{\widehat=} [s : \mathbb{F}\,\mathbb{Z} \mid \#s = 1]\\
ST_3 &\mathrel{\widehat=} [s : \mathbb{F}\,\mathbb{Z} \mid \#s > 1]\\
ST_5 &\mathrel{\widehat=} [s : \mathbb{F}\,\mathbb{Z} \mid \#s = 2]\\
ST_6 &\mathrel{\widehat=} [s : \mathbb{F}\,\mathbb{Z} \mid \#s > 2]\\
ST_8 &\mathrel{\widehat=} [s : \mathbb{F}\,\mathbb{Z} \mid s \neq \varnothing]\\
ST_9 &\mathrel{\widehat=} [s : \mathbb{F}\,\mathbb{Z}]
\end{aligned}$$

Note that ST_8 is derived from *IsEmpty* and ST_9 is derived from *IsIn*.

We now transform this collection into a set of disjoint state templates. ST_9 is partitioned by ST_{Init} and ST_8 ($ST_9 \equiv ST_{Init} \vee ST_8$) and hence is discarded. Similarly, ST_3 ($ST_3 \equiv ST_5 \vee ST_6$) and ST_8 ($ST_8 \equiv ST_2 \vee ST_5 \vee ST_6$) are discarded. We rename the four remaining disjoint state templates (ST_{Init}, ST_2, ST_5, ST_6) as the states of the finite state machine.

$$\begin{array}{lcl} ST_0 & \widehat{=} & [s : \mathbb{F}\mathbb{Z} \mid \#s = 0] \\ ST_1 & \widehat{=} & [s : \mathbb{F}\mathbb{Z} \mid \#s = 1] \\ ST_2 & \widehat{=} & [s : \mathbb{F}\mathbb{Z} \mid \#s = 2] \\ ST_M & \widehat{=} & [s : \mathbb{F}\mathbb{Z} \mid \#s > 2] \end{array}$$

ST_0 is the only initial state of *Set* since it has not been partitioned.

We choose two possible *Set* transitions to illustrate the identification of valid and invalid transitions of a class. The intuition is to determine whether an operation can be executed in one state and terminate in another (including the same) state. To show that (ST_0, ST_0') are not related by *Add* and therefore that *Add* does not form a valid transition between these states in the finite state machine, we prove

$$\begin{array}{l} \exists\, IS_{Add};\ OS_{Add} \bullet ST_0 \wedge Add \wedge ST_0' \\ \equiv \exists\, s : \mathbb{F}\mathbb{Z};\ x? : \mathbb{Z};\ s' : \mathbb{F}\mathbb{Z} \bullet \\ \qquad \#s = 0 \wedge x? \notin s \wedge s' = s \cup \{x?\} \wedge \#s' = 0 \\ \equiv \mathit{false} \end{array}$$

ST_0' refers to the state template ST_0 with the state variables primed.

On the other hand, to show that (ST_0, ST_1') are related by *Add*, we prove

$$\begin{array}{l} \exists\, IS_{Add};\ OS_{Add} \bullet ST_0 \wedge Add \wedge ST_1' \\ \equiv \exists\, s : \mathbb{F}\mathbb{Z};\ x? : \mathbb{Z};\ s' : \mathbb{F}\mathbb{Z} \bullet \\ \qquad \#s = 0 \wedge x? \notin s \wedge s' = s \cup \{x?\} \wedge \#s' = 1 \\ \equiv \mathit{true} \end{array}$$

The complete finite state machine of class *Set* is shown in Figure 5. Since all states are reachable, there is no need to remove any states from this finite state machine.

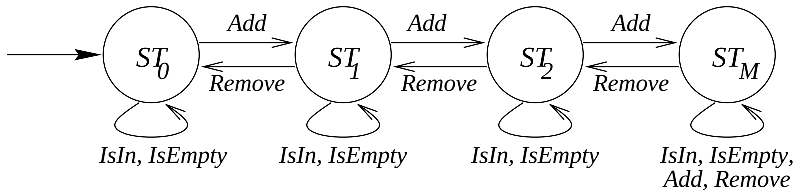

Figure 5: The finite state machine representing *Set*

4.3 Discussion

It is possible to construct finite state machines that have unreachable states from specifications (an example is shown in [25]). The goal of generating a finite state machine for a class is so we can construct sequences of tests to exercise the class. We are not only interested in testing each operation in isolation but also testing interactions between operations. While a class's specification may generate unreachable states, and it may be possible to execute operations in these states and perform legal transitions, if the interaction of the class's operations does not allow us to reach these states they cannot be tested. Therefore, when unreachable states occur we eliminate them from the finite state machine along with any transitions that originate or terminate in these states.

The identification of valid transitions requires n^2 calculations for each of the class's operations (where n is the number of states in the finite state machine). This work is mechanical, but detailed, and tool support would be of great assistance. However, automation of this process is beyond the scope of this paper.

5 ClassBench

This section describes the development of a traditional ClassBench test suite. Parts of the original ClassBench approach described in this section have been replaced by processes identified as *TTF*, *Analysis*, *Transformation* and *Oracle Generation* in Figure 1. The ClassBench framework [12, 13] is a collection of tools and techniques supporting the automated testing of C++ container classes.

5.1 Process

To develop a ClassBench test suite in the traditional way, a tester performs three tasks:

1. *Develop a testgraph.* The testgraph is a directed graph that models a subset of the possible states and transitions of the class under test. The testgraph describes the states and transitions that will be exercised during testing. Each node of the testgraph represents a possible state of the class under test and each arc represents a sequence of member function calls required to transform the class under test from one state to another. The testgraph is typically vastly smaller than the state transition graph of the class under test. The choice of nodes and arcs is made by the tester.

2. *Develop an* `Oracle` *class.* The `Oracle` is used to check the correctness of the behaviour of the class under test. It provides essentially the same operations, but supports only the testgraph states and transitions. As a result, the `Oracle` is usually significantly cheaper to implement than the class under test, and is more reliable.

3. *Develop a* `Driver` *class.* The `Driver` class contains the objects `cut` and `orc`, instances of the class under test and `Oracle` respectively. `Driver` also provides three public member functions: `reset` places both `cut` and `orc` in the initial state, `arc` generates the transitions in `cut` and `orc` associated with each testgraph arc traversed, and `node` checks that, at each node, the `cut` behaviour is consistent with the `orc` behaviour.

To add flexibility to the framework, each test suite may have a *test suite parameter*. This parameter is typically used to determine the size of the container to be tested, so that containers of different sizes may be tested without recompiling the test suite.

To support the tester, the ClassBench framework provides a graph editor, a graph traversal algorithm, and support code in the form of skeletons and demonstration test suites. The testgraph editor provides the facilities commonly available in graph editors, allowing testgraphs to be created, saved and modified. The testgraph traversal routines automatically traverse a stored testgraph, calling the appropriate `Driver` member functions: `reset` is called at the start of each path, `arc` is called each time a testgraph arc is traversed, and `node` is called each time a testgraph node is visited. The traversal routines ensure the arcs of the testgraph are covered.

5.2 Example

A possible testgraph for the *Set* class is shown in Figure 6. The testgraph has four nodes, `EMPTY`, `ALL`, `EVEN`, and `ODD`. The test suite parameter, *parm* indicates the maximum element value (10 in Figure 6). Thus the testgraph nodes represent the empty set, the set of all integers in the range $0..parm - 1$, the set of all even integers in the range $0..parm - 1$, and the set of all odd integers in the range $0..parm - 1$, respectively. We distinguish the node representing the initial state of the class under test, in this case `EMPTY`, by an unlabeled arc with no source node.

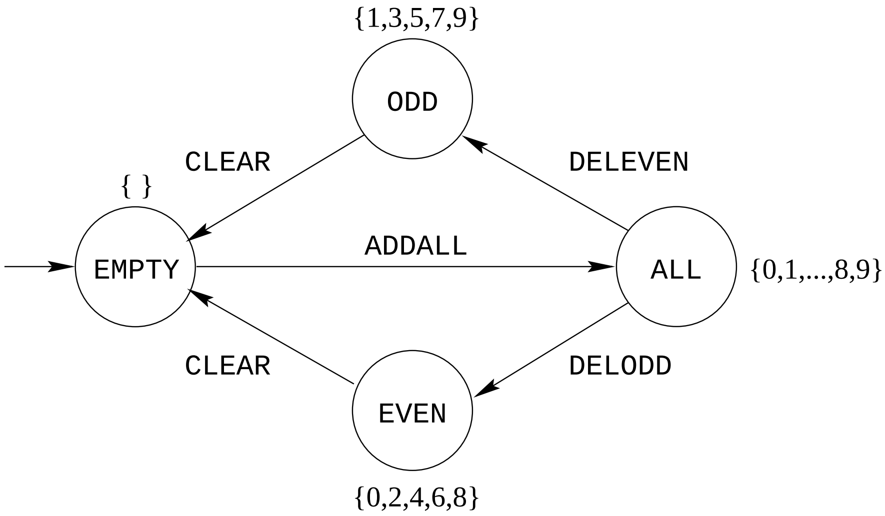

Figure 6: Testgraph for *Set* test suite with $parm = 10$

Figure 7 shows part of the `Oracle` class declaration. The oracle maintains an array, `s`, of Boolean values such that the value of `s[`i`]` indicates whether or not i is in the set. This implementation does not work in general, because it can only store elements in the range $0..$`MAXSIZE`-1. However, it is sufficient to represent the states in the *Set* testgraph, provided that the value of `MAXSIZE` is at least as big as the largest value chosen for the test suite parameter.

```
class Oracle {
public:
        Oracle(int Node, int MaxSize);
        int IsEmpty() const;
        int IsIn(int i) const;
        void Add(int i);
        void Remove(int i) const;
private:
        bool s[MAXSIZE];
};
```

Figure 7: Oracle declaration for *Set* test suite

```
void Driver::arc(int arcTag)
{
        switch (arcTag) {
        case ADDALL:
                for (int i = 0; i < parm; i++) {
                        cut.Add(i);
                        orc.Add(i);
                }
                break;
        ...
}

void Driver::node() const
{
        CHECKVALBOOLEAN(orc.IsEmpty(), cut.IsEmpty());
        for (int i = 0; i < parm; i++)
                CHECKVALBOOLEAN(orc.IsIn(i), cut.IsIn(i));
}
```

Figure 8: Partial driver for *Set* test suite

Figure 8 shows part of the `Driver` class implementation. `Driver::arc()` applies the transitions associated with each testgraph arc to both the class under test and oracle. For `ADDALL`, it adds all elements in the range $0..\texttt{parm} - 1$ to both. `Driver::node()` checks that the class under test and oracle are in equivalent states at each testgraph node by comparing the output of each observer function with the output of the corresponding oracle function. The function `CHECKVALBOOLEAN` is provided by the ClassBench framework and is used to compare the return values of Boolean functions and to report any discrepancies. In this case, we compare the return values for the class under test and the oracle for `IsEmpty` and for `IsIn`(i), for all values of i in the range $0..\texttt{parm} - 1$.

5.3 Discussion

The ClassBench framework has been used to test a number of commercial container class libraries, including the ANSI C++ Standard Template Library, the Borland C++ Data Structures library, and the IBM CSet++ library [13]. More recently it has been used to develop a test suite for the container classes of the Standard Template Library (STL) [21].

Traditionally with ClassBench, the tester is responsible for ensuring that the test cases defined by the testgraph are adequate to test the class under test and that the oracle accurately reflects the expected behaviour. In Section 6 we show how to derive a testgraph from the finite state machine derived from the specification of the class under test, and in Section 7 we show how to derive a test oracle from the same specification. These systematic derivations help reduce the potential for human error in testing.

6 Transforming Finite State Machines to Testgraphs

To test a class's implementation with ClassBench, we must provide a testgraph. This section details the derivation of a testgraph from a class's finite state machine, identified as the *Transformation* process in Figure 1. The arcs and nodes of the testgraph cover the transitions and states of the finite state machine.

6.1 Process

A class's testgraph is derived from its formally derived finite state machine in three steps:

1. *States → nodes.* We map each finite state machine state to a testgraph node. To do this, we choose concrete values for the class's state variables that conform with the finite state machine's state, the types chosen for the class's generic parameters, and the values assigned to constants in the class's implementation.

2. *Choose a start node.* A testgraph has one start node. The node that is derived from the finite state machine's initial state in step 1 becomes the testgraph's start node (we discuss the possibility of multiple initial states below).

3. *Transition coverage*

   Our aim is to cover the transitions of the finite state machine. Each transition in the finite state machine that changes the state of the class must map to a testgraph arc. The start and end state of each such transition is mapped to the corresponding testgraph nodes, and an arc between them is formulated by identifying a sequence of operation calls that transforms the starting node to the end node. The sequence of calls must contain at least one call to the operation of the finite state machine's transition.

   If a case arises where it is difficult or impossible to create such a sequence, we identify a sequence of calls that does transform the state of the node derived from the transition's starting state, to a node that complies with the transition's ending state. A new node is then added to the testgraph representing this state.

   Transitions in the class's finite state machine that do not change the class's state are not mapped to arcs in the derived testgraph. Instead we test and use the operations corresponding to these transitions in `Driver::node()`, as is the normal practice for ClassBench.

6.2 Example

We map each state of the example finite state machine (shown in Figure 5) to a testgraph node. We map Z integers ($\mathbb{Z}$) to C++ integers (`int`). The nodes of the testgraph for *Set* are shown in Figure 9. The value chosen for the state variable s is shown above each node. The name of each node relates it directly to the finite state machine state from which it is derived, e.g., `TG0` maps to ST_0. Note that for `TGM` we have chosen a set of four integers, but any set with more than two integers would have been equally acceptable and compliant with ST_M, the corresponding state.

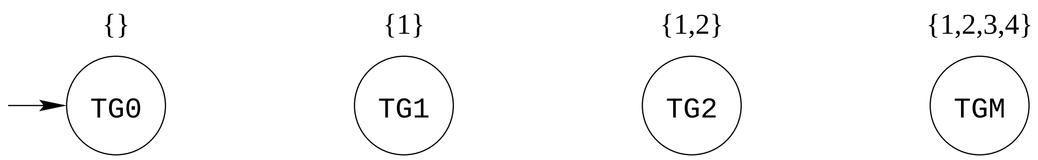

Figure 9: The nodes of the testgraph for *Set*

The finite state machine for *Set* has one initial state, ST_0. The testgraph node `TG0` is derived from ST_0 and becomes the start node.

Transitions of the finite state machine that change the state of *Set* are now transformed. The result is the complete, testgraph for *Set* shown in Figure 10. Most testgraph arcs are obvious; for example, the arc from TG0 to TG1 corresponding to transition *Add* is a single call of method Add with 1 as argument. Constructing the testgraph arcs corresponding to the *Add* and *Remove* transitions from ST_M to itself is not straightforward, however. Consider the *Add* transition: ST_M maps to node TGM which has a state of $\{1, 2, 3, 4\}$. Obviously, if we add a new integer, the state will not remain the same, and it is impossible to create an arc from TGM to itself that adds an element. Instead, we create a new node, TGMA, with state $\{1, 2, 3, 4, 5\}$. TGMA is also derived from ST_M and allows the translation of the *Add* transition to the arc Add(4). Similarly, the *Remove* transition from ST_M to itself in Figure 5 is translated to arc Remove(5).

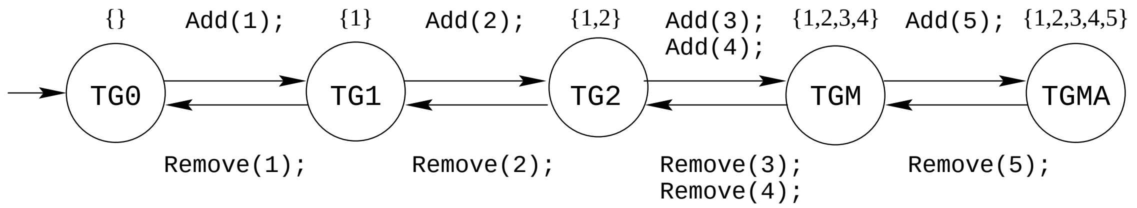

Figure 10: Testgraph for *Set*

The transitions of the finite state machine that do not change the state of *Set*, that is, all *IsEmpty* and *IsIn* transitions, are mapped to code in Driver::node(), which in this instance is the same as in the traditional driver shown in Figure 8.

6.3 Discussion

Several issues arise when deriving a testgraph from a finite state machine. Most of these issues occur because the finite state machine is abstract in that it covers the state space of the specification, and a testgraph is more concrete because it covers the state space (or part of it) of the implementation.

The choice of a start node for a testgraph can be complicated if the finite state machine has more than one initial state. When this situation arises, we look to the implementation for guidance. In the best case, the implementation's constructor method will initialise the class's state variables to values that match one of our nodes, identifying that node as the start node. If there is no match, we change the values given to the state variables for a node to match those of the implementation, ensuring consistency with the state specified by the finite state machine from which the node is derived. This node becomes the start node. In choosing one out of many possible initial states as the start node, it is possible that some of the states not chosen become unreachable.

A testgraph node represents one state in the class's state space. In a finite state machine, a state represents an equivalence class, or set of states. The concrete nature of the testgraph nodes means that it may be necessary for a finite state machine state to be represented by more than one testgraph node to achieve satisfactory testing.

A testgraph arc represents a sequence of calls. The sequence can contain one or many calls to the class's operations. These may be multiple calls to the same operation, or calls to different operations. A finite state machine transition represents one call to one operation. Therefore, in a finite state machine we have the situation where the starting state of a transition satisfies the operation's precondition and the ending state of a transition satisfies the operation's postcondition. This is difficult to translate to a testgraph where an arc could be a sequence of calls to many different operations. The call to the operation corresponding to the finite state machine transition could occur at three places in the sequence: the beginning, the end, or somewhere in the middle. We prefer the second option so that the postcondition of the operation is tested immediately after the operation. In cases where this is difficult, we settle for one of the other options.

It is possible that a node in the testgraph is not reachable, even though the state it is derived from is reachable in the class's finite state machine. One option when this occurs is to choose new values for the class's state variables for the node (making sure the node remains consistent with the finite state machine state from which it is derived), so the node becomes reachable. Another option is to consider changing the values for other nodes in the path from the start node to the node in question (maintaining consistency with the finite state machine). It is also possible to introduce new nodes as representatives of the same finite state machine states; consequently some finite state machine states will map to more than one testgraph node.

7 Deriving Oracles from Specifications

Traditionally, ClassBench oracles are developed informally, based on the tester's understanding of the expected class behaviour. However, where a formal specification exists, we can reduce the potential for human error by systematically deriving the test oracle from the specification. This section describes the derivation of a test oracle from an Object-Z specification, identified as the *Oracle Generation* process in Figure 1.

7.1 Process

There are two stages in mapping an Object-Z specification to a test oracle: optimisation and translation. Optimisation transforms the specification to a form more suited to systematic translation to an oracle. Oracle translation maps the optimised specification to C++ code.

In the optimisation phase, we rearrange the specification to simplify translation to an implementation language. The most difficult task of the optimisation phase is replacing constructs that are not finitely evaluable with equivalent finitely evaluable constructs. If, at the end of the optimisation phase, there remain any constructs that cannot be transformed into finitely evaluable constructs then the translation fails. The other optimisation tasks are comparatively simple. They include transformations such as flattening inheritance and schema inclusions (and performing any associated renamings), and replacing concurrency with equivalent sequential constructs.

In the oracle translation phase, the optimised specification is translated to a test oracle class implemented in C++. As recommended by Hörcher [14], the oracles evaluate in the abstract state space of the specification rather than the concrete state space of the implementation. In particular, the oracle defines an abstraction function, `abs()`, to relate the concrete state of the implementation to the abstract state of the specification. The implementation of this function must be supplied by the tester.

The oracle class inherits from the class under test and augments it with code to check the behaviour. Thus the oracle acts as a self-checking version of the class under test that can be used in place of the class under test during testing. In addition to the abstraction function, the oracle consists of:

- an invariant check, `inv()`, derived from the invariant in the specification;
- a constructor, which provides a wrapper around the constructor of the class under test, and a corresponding constructor check, which checks the behaviour of the class constructor against the specification;

- an augmented operation for each operation in the class under test, which provides a wrapper, and a corresponding operation check, which again checks the behaviour of the operation's implementation against the specification.

To assist in producing oracles, we have developed a library of classes that implement some of the standard types of the Z Mathematical Toolkit [30]. At present, this library provides implementations of the set, sequence, relation and function types. The implementations are in the form of generic C++ classes. For example, the definition $s : \mathbb{F}\,\mathbb{Z}$ in Object-Z is translated into the declaration `Z_Set<int> s` in the test oracle. The Z Toolkit classes are used to store and evaluate the abstract state of specifications in test oracles.

7.2 Example

We illustrate the translation method using the *Set* class. No optimisations are needed for the *Set* specification. A partial declaration for the *Set* oracle is shown below and a description of each of the oracle member functions follows.

```
class SetOracle : public Set {
        // abstraction function
        void abs( Z_Set<int>& state_s ) const;
        // invariant check
        void inv( const Z_Set<int> state_s ) const;
        // Constructor
        SetOracle();
        // Constructor check
        void check_init( const Z_Set<int> state_s ) const;
        // Augmented operations
        bool IsEmpty() const;
        bool IsIn( const int x ) const;
        void Add( const int x );
        void Remove( const int x );
        // Operation checks
        void check_Add( const Z_Set<int> pre_s,
                const Z_Set<int> post_s, const int x ) const;
        ...
};
```

The abstraction function of `SetOracle` is shown below. It takes a reference parameter for each element of the abstract state and places the abstract representation of the concrete state into these parameters. It iterates over the concrete state of `Set` adding each element to the Z-toolkit set representing the abstract state.

```
void SetOracle::abs( Z_Set<int>& state_s ) const
{
        for (int i = 1; i <= s->Size(); i++)
                state_s.add(*(s->SelectElmt(i)));
}
```

The invariant check, `inv()`, has the abstract state of the class under test as a parameter and evaluates whether that state satisfies the class invariant of the specification. The invariant check for `SetOracle` is empty as *Set* has an empty class invariant.

The implementation of the oracle constructor, shown below, calls the inherited constructor. It then declares a variable, `state_s`, to represent the abstract specification state and calls the abstraction function to retrieve the value of the abstract state. The call to `inv()` checks that the abstract state satisfies the class invariant and the call to `check_init()` checks that the state satisfies the *INIT* schema of the specification.

```
SetOracle::SetOracle() : Set()
{
        Z_Set<int> state_s;
        abs(state_s);
        inv(state_s);
        check_init(state_s);
}
```

The constructor check, `check_init()`, shown below, checks the behaviour of the class under test constructor against the specification of the initial state described in the *INIT* schema. The abstract state of the class under test is a parameter and `check_init()` evaluates whether or not that state satisfies the predicate of the *INIT* schema. For *Set*, it simply checks that the current state represents the empty set using the `==` operator provided by the Z-toolkit set class. Like `CHECKVALBOOLEAN`, the function `CHECKTRUE` is provided by the ClassBench framework for error reporting.

```
void SetOracle::check_init(const Z_Set<int> state_s)
{
        CHECKTRUE(s == Z_Set<int>());
}
```

The oracle replaces each public operation of the class under test by an augmented version that calls the original operation, checks its behaviour, and signals any inconsistencies between the specified behaviour and the actual behaviour of the operation. The augmented version of `Add()` is as follows.

```
void SetOracle::Add( const int x )
{
        Z_Set<int> pre_s;
        abs(pre_s);
        inv(pre_s);
        Set::Add(x);
        Z_Set<int> post_s;
        abs(post_s);
        inv(post_s);
        check_Add(pre_s, post_s, x);
}
```

Before calling the inherited class under test operation, the oracle records the abstract pre-state and calls `inv()` to check that the class invariant is satisfied. After calling the inherited operation, the oracle records the outputs and the abstract post-state, calls `inv()` to check that the post-state satisfies the class invariant, and calls the appropriate operation check to check that the combination of inputs, outputs, pre- and post-states satisfies the specification of the operation. The invariant is checked both before and after calling the inherited operation because we view the invariant as an additional pre- and post-condition on each operation.

The operation checks compare the behaviour of the class under test operations with their expected behaviour as described in the operation schemas of the specification.

Each operation check is passed the abstract pre-state and post-state, and the inputs and outputs of the operation and evaluates whether they satisfy the specification of that operation. For example, the check for `Add()` is as follows.

```
void SetOracle::Add( const int x )
void SetOracle::check_Add( const Z_Set<int> pre_s,
        const Z_Set<int> post_s, const int x ) const
{
        CHECKTRUE(!pre_s.isMember(x));
        CHECKTRUE(post_s == pre_s + Z_Set<int>(x));
}
```

The implementation of each check is a translation of the predicate part of the corresponding operation schema in the specification. In the example, the check ensures that the input `x` is not a member of the pre-state of `s`, and that the post-state of `s` is equal to the pre-state with the input `x` added.

Additionally, for each state item that is not modified by an operation (i.e., those items that do not appear in the Δ list), the operation check assesses whether the pre-state and post-state of the item are equal. For example, the check for `IsEmpty()` is as follows.

```
void SetOracle::check_IsEmpty( const Z_Set<int> pre_s,
        const Z_Set<int> post_s, const bool result ) const
{
        CHECKTRUE(result == (pre_s == Z_Set<int>()));
        CHECKTRUE(pre_s == post_s);
}
```

Again, the first statement of this check is a straightforward translation from the Object-Z specification for *IsEmpty* to the corresponding expression using the Z-toolkit library. The second statement checks that the pre-state and post-state of `s` are equal.

7.3 Discussion

The systematic derivation of test oracles from formal specifications raises an important practical issue. In order to derive testing code that will communicate with the class under test, we must know precisely the interface that the class under test will have. Therefore, it has been necessary to develop a set of rules for the style of public interface to which the C++ implementation of each class must adhere [20]. Some of these issues were discussed above in Section 7.1

The derived oracles are self-checking versions of the class under test. This simplifies the ClassBench driver code in two ways. First, the oracle may be used in place of the class under test, so the driver need only call the oracle member functions, instead of calling both oracle and class under test member functions. Second, as the oracle checks the behaviour of all class under test functions, the driver need not check the output values returned by the oracle functions. Part of the new driver for *Set* is shown below. This is considerably simpler than the original driver shown in Figure 8.

```
void Driver::arc(int arcTag)
{
        switch(arcTag) {
        case A1:
                orc.Add(1);
```

```
                break;
...
}

void Driver::node() const
{
        orc.IsEmpty();
        for (int i = 0; i < parm; i++)
                orc.IsIn(i);
}
```

Recall that in the traditional ClassBench approach, the oracle is required to handle only those states that appear in the testgraph. This causes maintenance problems if the testgraph changes. A further advantage to the use of the derived oracles is that they can handle all valid states of the class under test.

Work is presently in progress on tool support for automated oracle generation. The tool builds on the `wizard` Object-Z type checker [15]. The oracle generator will soon be used in a significant case study testing the ANSI C++ Standard Template Library container classes.

8 Experience

In this section we discuss our experience applying our method to the full Dependency Management System specification, which is presented in Appendix A. The complete specification includes 11 additional operations for manipulating dependencies. Both the specification and the implementation of the Dependency Management System include a generic parameter, representing the type of the nodes stored in the Dependency Management System.

Construction of the test template hierarchies for the complete specification was straightforward. Deriving the finite state machine was considerably more difficult, however, mainly due to the tedious, error-prone, manual checking required for states and transitions. The resulting state machine can be viewed as a composition of two sub-machines, one concerning nodes and the other dependencies. It consists of 8 states and 45 transitions, 7 'get' transitions, 10 'set' transitions that do not change the finite state machine state and 28 that do change the state.

The testgraph derived from the finite state machine is the largest to be used with ClassBench to date, comprising 13 nodes and 38 arcs. The finite state machine has only one initial state, so selecting the start node of the testgraph presented no problem. However, the state machine includes a large number of *self-transitions*, transitions that begin and end in the same state, and *multi-transitions*, transitions joining a single pair of states with multiple labels. ClassBench includes a self-arc capability but it has not been widely used because such arcs cannot be displayed by the editor. They were used in this study. Multi-arc capability was added to ClassBench for this study and has been used for a subsequent study.

The following statistics summarise a typical ClassBench run for the complete specification.

Nodes visited	246
Arcs traversed	228
Paths	18
Comparisons	100818

The nodes and arcs figures indicate the totals required to achieve arc coverage of the testgraph. The number of paths is the number of times the `reset` function was called (to start a new traversal from the start node of the testgraph). The count of comparisons is the total number of checks conducted within the oracle class (invocations of functions like `CHECKTRUE`).

Testing revealed two problems with memory handling in the implementation and a coding error in the oracle class:

- The method that adds a dependency also updates the inferred transitive closure. In two places in the update, the implementation deleted the dynamically allocated memory for the recently added dependency, leading to a catastrophic failure.
- The specification for removal of an element requires that the element being removed not be depended upon, and that its dependencies (if any) also be removed within the removal operation. The comparison to identify dependencies for removal was performed on pointers rather than values with the result that dependencies were not deleted correctly.
- An oracle class wrapper for the method to check whether an element was depended upon lacked a `return` statement. It was used to guard a call to a wrapper for the method to return some direct dependent. Since non-zero values are interpreted as true, the guard was usually (accidentally) true and the guarded call was made in violation of the precondition, resulting in catastrophic failure.

9 Related Work

Considerable work has been done in the areas of class and object-oriented testing [2]. Similarly, a significant amount of work has been done in the area of specification-based testing, including the generation of test cases for single operations [9, 31], the generation of finite state machines for testing purposes [5, 11, 34], and the generation of oracles [1, 14, 23, 28]. In the remainder of this section, we focus on frameworks for class testing that incorporate test case generation, execution, and evaluation. We first review the frameworks that incorporate formal specifications.

Bosman and Schmidt [3] use finite state machines to test classes. Their method uses state machines that result from Statecharts [10] used in object-oriented analysis and design techniques. A *design finite state machine* for a class is generated from the state diagrams and defines the expected outcomes of test cases. A *representation finite state machine* is an abstraction of the class's implementation and is used to drive the testing of the class. It also provides a mapping between the design and implementation of the class. Their approach is similar to ours but starts from a behavioural rather than model-based specification. Their tool support appears less developed than ClassBench.

ASTOOT, developed by Doong and Frankl [6], provides tools for automatic driver generation from a class interface specification and semi-automatic test case generation from algebraic class specifications. Each test case contains an original message sequence and a simplified message sequence. The two sequences should leave objects of the class under test in observationally equivalent states. However, it is left to the tester to provide a means of determining whether the states are actually equivalent. Our framework uses access to the state to achieve this.

There are also a number of class testing frameworks that do not use formal specifications. Kung et al. [17] describe the Object-Oriented Testing and Maintenance (OOTM) environment. OOTM uses a mathematically defined *test model*, derived from the source code. This highlights a limitation of the system as it only supports program-based testing.

Murphy et al. [24] describe testing of classes and clusters using the Automated Class Exerciser (ACE). The ACE tool was developed to support the testing of C++ and Eiffel classes. The tool takes a *test script* and generates a test harness. Each test script describes how to compile the class under test and provides code for the test cases and the expected results of the tests. The test harness runs the test cases and reports the results, any failures being entered into a problem management system for later correction.

Smith and Robson [29] present the Framework for Object-Oriented Testing (FOOT). The framework allows a class to be tested either manually or automatically, using either a test strategy or a previously defined test sequence. FOOT provides several testing strategies, including exhaustive testing of method combinations, testing for memory leaks, and testing that sequences of operations that should be identities do actually leave the state unchanged.

10 Conclusions

Software testing involves three activities: generating, executing and evaluating tests. This paper shows how a formal specification can be used as the starting point for all three of these activities.

An Object-Z specification is a precise, abstract description of the interface and behaviour of the system to be tested. The Test Template Framework provides a systematic approach to generating tests. We have extended the Test Template Framework from its initial focus on individual operations to generate tests for class specifications. From the tests, we can derive a finite state machine that captures possible sequences of operations.

ClassBench is an existing tool for executing and evaluating tests. It uses the concept of a testgraph to model the states and transitions to be tested. Our method demonstrates how an abstract finite state machine derived using the Test Template Framework can be transformed into a testgraph for ClassBench.

ClassBench requires an oracle be supplied to evaluate tests. The oracle checks the behaviour of the class under test at every state and for every transition in the testgraph. Traditionally, ClassBench oracles are constructed by hand. With our method, the oracle is derived from the Object-Z specification. The process is not fully automatic, but the systematic approach makes the task easier and less error-prone.

At present, the method requires considerable human effort to apply, but there is scope for mechanising some of the activities. A current research project is building a tool to support the Test Template Framework. The tool will be interactive with the user making choices about testing strategies while the tool records the results and guides the process. A tool to support the clerical parts of oracle transformation is also under development.

The paper describes a method for using a formal specification for all aspects of software testing. Such a method increases the value of a formal specification and unifies two approaches to software quality that are often considered to be in competition.

Acknowledgements

The Dependency Management System was originally specified in Object-Z by Gordon Rose in at least two different versions. Our Object-Z specification is based on a Z specification by Phil Stocks [32] which was derived from Gordon's.

The implementation was originally developed by Wendy Johnston [16] under Unix from one of the original Object-Z specifications. Minor modifications have been made to handle

a change of operating system to Microsoft Windows 95 and a compiler change to Borland C++.

We thank Dan Hoffman and Graeme Smith for reading drafts of this paper.

This work is supported by Australian Research Council Grant A4-96-00282. Ian MacColl is supported by an Australian Postgraduate Award and a Telstra Research Laboratories Postgraduate Fellowship. Jason McDonald is supported by an Australian Postgraduate Award.

References

[1] S. Antoy and R. Hamlet. Objects that check themselves against formal specifications. Technical Report 91–1, Dept. of Computer Science, Portland State University School of Engineering and Applied Sciences, 1991.

[2] R. V. Binder. Testing object-oriented software: a survey. *Software Testing, Verification and Reliability*, 6:125–252, 1996.

[3] O. Bosman and H. W. Schmidt. Object test coverage using finite state machines. In *Technology of Object-Oriented Languages and Systems (TOOLS 18)*, pages 171–178. Prentice-Hall, 1995.

[4] J. P. Bowen and M. G. Hinchey. Ten commandments of formal methods. *Computer*, 28(4):56–63, 1995.

[5] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Formal Methods Europe (FME93)*, volume 670 of *LNCS*, pages 268–284. Springer-Verlag, 1993.

[6] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, 1994.

[7] R. Duke and G. Rose. *Formal Object-Oriented Specification and Design Using Object-Z.* Academic Press, 1998. To appear.

[8] R. Duke, G. Rose, and G. Smith. Object-Z: A specification language advocated for the description of standards. *Computer Standards & Interfaces*, 17:511–533, 1995.

[9] M.-C. Gaudel. Testing can be formal too. In *Theory and Practice of Software Development (TAPSOFT95)*, volume 915 of *LNCS*, pages 82–96. Springer-Verlag, 1995.

[10] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[11] R.M. Hierons. Testing from a Z specification. *Software Testing, Verification and Reliability*, 7(1):19–33, 1997.

[12] D. Hoffman and P. Strooper. The testgraph methodology: Automated testing of collection classes. *Journal of Object-Oriented Programming*, 8(7):35–41, 1995.

[13] D.M. Hoffman and P.A. Strooper. ClassBench: A methodology and framework for automated class testing. *Software Practice and Experience*, 27(5):573–597, 1997.

[14] H.-M. Hörcher. Improving software tests using Z specifications. In *Z User Meeting (ZUM95)*, volume 967 of *LNCS*, pages 152–166. Springer-Verlag, 1995.

[15] W. Johnston. A type checker for Object-Z. Technical Report 96–24, Software Verification Research Centre, The University of Queensland, Australia, 1996.

[16] W. Johnston and G. Rose. Guidelines for the manual conversion of Object-Z to C++. Technical Report 93–32, Software Verification Research Centre, The University of Queensland, Australia, 1993.

[17] D. Kung, P. Hsia, J. Z. Gao, Y. Toyoshima, C. Chen, Y.-S. Kim, and Y.-K. Song. Developing an object-oriented software testing and maintenance environment. *Communications of the ACM*, 38(10):75–87, 1995.

[18] P. Lindsay. The Dependency Management System case study. Technical Report 94–01, Software Verification Research Centre, The University of Queensland, Australia, 1994.

[19] I. MacColl, L. Murray, P. Strooper, and D. Carrington. Specification-based object-oriented testing: A case study. In *International Conference on Formal Engineering Methods (ICFEM98)*. IEEE, 1998.

[20] J. McDonald. Translating Object-Z specifications to passive test oracles. Technical Report TR98–04, Software Verification Research Centre, The University of Queensland, Australia, 1998.

[21] J. McDonald, D. Hoffman, and P. Strooper. Programmatic testing of the standard template library containers. In *Automated Software Engineering (ASE'98)* (accepted). IEEE Computer Society, 1998.

[22] J. McDonald and P.A. Strooper. Translating Object-Z specifications to passive test oracles. In *International Conference on Formal Engineering Methods (ICFEM98)* (accepted). IEEE, 1998. Also SVRC TR98–04.

[23] E. Mikk. Compilation of Z specifications in C for automatic test result evaluation. In *Proceedings of the Ninth Annual Z User Meeting*, volume 967 of *LNCS*, pages 167–180. Springer-Verlag, 1995.

[24] G. C. Murphy, P. Townsend, and P. Wong. Experiences with class and cluster testing. *Communications of the ACM*, 37(9):39–47, 1994.

[25] L. Murray, D. Carrington, I. MacColl, J. McDonald, and P. Strooper. Formal derivation of finite state machines for class testing. In *Z User Meeting (ZUM98)* (accepted), volume 1493 of *LNCS*. Springer-Verlag, 1998. Also SVRC TR98–03.

[26] L. Murray, D. Carrington, I. MacColl, and P. Strooper. Extending Test Templates with inheritance. In *Australian Software Engineering Conference (ASWEC97)*, pages 80–87. IEEE Computer Society, 1997.

[27] L. Murray, J. McDonald, and P. Strooper. Specification-based class testing with ClassBench. In *Asia-Pacific Software Engineering Conference* (accepted), 1998. Also SVRC TR98–12.

[28] D.K. Peters and D.L. Parnas. Using test oracles generated from program documentation. *IEEE Transactions on Software Engineering*, 24(3):161–173, 1998.

[29] M. D. Smith and D. J. Robson. A framework for testing object-oriented programs. *Journal of Object-Oriented Programming*, 5(3):45–53, 1992.

[30] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 2nd edition, 1992.

[31] S. Stepney. Testing as abstraction. In *Z User Meeting (ZUM95)*, volume 967 of *LNCS*, pages 137–151. Springer-Verlag, 1995.

[32] P. Stocks. *Applying formal methods to software testing.* PhD thesis, Department of Computer Science, The University of Queensland, 1993.

[33] P. Stocks and D. Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, 1996.

[34] C. D. Turner and D. J. Robson. A state-based approach to the testing of class-based programs. *Software—Concepts and Tools*, 16:106–112, 1995.

A Complete DMS specification

The dependency management system (DMS) case study was a testbed for a project exploring software development methodologies [18]. 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.

State variable *nodes* represents the finite set of axioms or theories stored in the system's database. State variables *ddo* and $\succ$ represent the direct dependency relationship and its transitive closure respectively; both are relations on X. An aid to reading the specification is to think of the domain of *ddo* as dependents and the range as supporters.

The class invariant requires that only nodes are related by *ddo*, that $\succ$ is the transitive closure of *ddo*, and that the transitivity does not induce a circularity.

Initially the DMS has no nodes and, by the state invariant, the direct dependency relation and its transitive closure are empty.

DMS[X]

State:

$$nodes : \mathbb{F}\, X$$
$$ddo : X \leftrightarrow X$$
$$\Delta$$
$$\succ : X \leftrightarrow X$$

$$ddo \subseteq nodes \times nodes$$
$$\succ = ddo^{+}$$
$$\nexists x : X \bullet x \succeq x$$

INIT

$$nodes = \varnothing$$

NoNodes

$$result! : \mathbb{B}$$

$$result! \Leftrightarrow nodes = \varnothing$$

Returns true if and only if *nodes* is empty.

IsNode

$$x? : X$$
$$result! : \mathbb{B}$$

$$result! \Leftrightarrow x? \in nodes$$

Returns true if and only if the input node $x?$ is a member of *nodes*.

$$\begin{array}{|l} \textit{AddNode} \\ \hline \Delta(nodes) \\ x? : X \\ \hline x? \notin nodes \\ nodes' = nodes \cup \{x?\} \end{array}$$

Node $x?$ is added to $nodes$ if it is not already in $nodes$.

$$\begin{array}{|l} \textit{RemoveNode} \\ \hline \Delta(nodes, ddo) \\ x? : X \\ \hline x? \in nodes \setminus \operatorname{ran} ddo \\ nodes' = nodes \setminus \{x?\} \\ ddo' = \{x?\} \mathbin{\lhd\!\!\!-} ddo \end{array}$$

Node $x?$ is removed from $nodes$ and ddo if it is an existing node and no node depends on it (i.e., it is not a supporter).

$$\begin{array}{|l} \textit{NoDependencies} \\ \hline result! : \mathbb{B} \\ \hline result! \Leftrightarrow ddo = \varnothing \end{array}$$

Returns true if and only if ddo is empty.

$$\begin{array}{|l} \textit{DependedUpon} \\ \hline x? : X \\ result! : \mathbb{B} \\ \hline x? \in nodes \\ result! \Leftrightarrow x? \in \operatorname{ran} ddo \end{array}$$

Returns true if and only if some node depends on $x?$.

$$\begin{array}{|l} \textit{IsDependency} \\ \hline x?, y? : X \\ result! : \mathbb{B} \\ \hline \{x?, y?\} \subseteq nodes \\ result! \Leftrightarrow (x?, y?) \in ddo \end{array}$$

Checks if $(x?, y?)$ is in ddo.

$$\begin{array}{|l} \textit{CanAdd} \\ \hline x?, y? : X \\ result! : \mathbb{B} \\ \hline \{x?, y?\} \subseteq nodes \\ result! \Leftrightarrow (\neg\, y? \succeq x? \wedge x? \neq y?) \end{array}$$

Checks if the pair $(x?, y?)$ can be added to the direct dependency relation, that is, returns true if and only if $x?$ and $y?$ are existing nodes, and $x?$ depending on $y?$ does not cause a circularity, i.e., $y?$ does not already depend on $x?$ (directly or indirectly) and $x?$ and $y?$ are distinct.

$$\begin{array}{|l} \textit{AddDependence} \\ \hline \Delta(ddo) \\ x?, y? : X \\ \hline \{x?, y?\} \subseteq nodes \\ \neg\, y? \succeq x? \wedge x? \neq y? \\ ddo' = ddo \cup \{(x?, y?)\} \end{array}$$

Add $(x?, y?)$ to the dependency relation.

$$\begin{array}{|l} \textit{RemoveDependence} \\ \hline \Delta(ddo) \\ x?, y? : X \\ \hline (x?, y?) \in ddo \\ ddo' = ddo \setminus \{(x?, y?)\} \end{array}$$

Removes $(x?, y?)$ from ddo provided that $(x?, y?)$ is in ddo.

$$\begin{array}{|l} \textit{Dependents} \\ \hline x? : X \\ ns! : \mathbb{F}\, X \\ \hline x? \in nodes \\ ns! = \{n : nodes \mid n \succeq x?\} \end{array}$$

Returns the set of all nodes that depend on $x?$

$$\begin{array}{|l} \textit{Supporters} \\ \hline x? : X \\ ns! : \mathbb{F}\, X \\ \hline x? \in nodes \\ ns! = \{n : nodes \mid x? \succeq n\} \end{array}$$

Returns the set of all nodes on which $x?$ depends.

$UltSupporters$

$x? : X$
$ns! : \mathbb{F}\, X$

$x? \in nodes$
$ns! = \{n : nodes \mid x? \succeq n \land n \notin \mathrm{dom}\, ddo\}$

Returns the set of all non-dependent nodes on which $x?$ depends.

$CandidateSupporters$

$x? : X$
$ns! : \mathbb{F}\, X$

$x? \in nodes$
$ns! = \{n : nodes \mid \neg\, n \succeq x? \land n \neq x?\}$

Returns the set of nodes that are candidate supporters of $x?$.

$SomeDirectDependent$

$x? : X$
$n! : X$

$x? \in \mathrm{ran}\, ddo$
$(n!, x?) \in ddo$

Outputs (nondeterministically) any one of the possibly several direct dependents of the nominated node $x?$.

Operation $NoNodes$ returns true if and only if $nodes$ is empty. Operation $IsNode$ returns true if and only if the input node $x?$ is a member of $nodes$.

Node $x?$ is added to $nodes$ by $AddNode$. $AddNode$ is disabled if $x?$ is already in $nodes$. Operation $RemoveNode$ removes node $x?$. The precondition is that the input value must be an existing node and that no node depends on it (i.e., it is not a supporter). The remaining two conjuncts specify the removal of the designated node from the set of nodes and from the direct dependency relation respectively.

Operation $NoDependencies$ returns true if and only if ddo is empty. Given $x? \in nodes$, operation $DependedUpon$ checks whether anything depends on $x?$.

Operation $IsDependency$ checks if $(x?, y?)$ is in ddo. Operation $CanAdd$ checks if the pair $(x?, y?)$ can be added to the direct dependency relation. $x?$ and $y?$ must be existing nodes, and $x?$ depending on $y?$ must not cause a circularity, i.e., $y?$ does not already depend on $x?$ (directly or indirectly) and $x?$ and $y?$ are distinct.

$AddDependence$ specifies the addition of the nominated dependency $(x?, y?)$ to the dependency relation. Operation $RemoveDependence$ removes $(x?, y?)$ from ddo. The operation is disabled if $(x?, y?)$ is not in ddo.

Operation $Dependents$ returns the set of all nodes that depend on $x?$. $Supporters$ returns the set of all nodes on which $x?$ depends.

Operation $UltSupporters$ returns the set of all non-dependent nodes on which $x?$ depends.

Operation $CandidateSupporters$ returns the set of nodes that are candidate supporters of $x?$. Operation $SomeDirectDependent$ outputs (nondeterministically) one of the (possibly several) direct dependents of the nominated node $x?$.