A Framework and Tool Support for the Systematic Testing of Model-Based Specifications

TIM MILLER and PAUL STROOPER
University of Queensland

Formal specifications can precisely and unambiguously define the required behaviour of a software system or component. However, formal specifications are complex artifacts that need to be verified to ensure that they are consistent and complete, and validated against the requirements. Current specification testing or animation tools exist to assist with this by allowing the specifier to interpret or execute the specification. However, currently little is known about how to do this effectively.

This paper presents a framework and tool support for the systematic testing of formal, model-based specifications. Several important generic properties that should be satisfied by model-based specifications are first identified. Following the idea of mutation analysis, we then use variants or mutants of the specification to check that these properties are satisfied. The framework also allows the specifier to test application-specific properties. All properties are tested for a range of states that are defined by the tester in the form of a testgraph, which is a directed graph that partially models the states and transitions of the specification being tested. Tool support is provided for the generation of the mutants, for automatically traversing the testgraph and executing the test cases, and for reporting any errors. The framework is demonstrated on a small specification and its application to three larger specifications is discussed. Experience indicates that the framework can be used effectively to test small to medium-sized specifications and that it can reveal a significant number of problems in these specifications.

Categories and Subject Descriptors: D.2.1 [Software Engineering]: Requirements/Specifications; D.2.4 [Software Engineering]: Software/Program Verification—Formal Methods

General Terms: Verification

Additional Key Words and Phrases: Formal Verification, Specification Animation, Testgraphs, Testing

1. SYSTEMATIC SPECIFICATION TESTING

Formal specifications can precisely and unambiguously define the required behaviour of a software system or component. However, formal specifications are complex artifacts that need to be verified to ensure that they are consistent and complete, and validated to ensure that they accurately capture the requirements. Several techniques for specification verification and validation exist, such as theorem proving, model checking, and specification testing. In this paper we focus on...
specification testing, which involves interpreting or executing a specification with the intent of finding errors. Like software testing, specification testing can only show the presence of errors in a specification and not their absence. However, as a more lightweight approach to specification verification and validation compared to model checking and theorem proving, it can be very effective at revealing such errors.

Current specification testing tools, such as Possum [Hazel et al. 1997; 1998], PiZA [Hewitt et al. 1997], Jaza [Utting 2000] and the B-Model animator [Waese-lynck and Behnia 1998], exist to assist with the testing of specifications. Although these tools are often called specification animation tools, we prefer to use the term specification testing, to bring out the similarity with more traditional software testing. Despite the existence of a number of specification testing tools for a variety of specification languages, little is known about how to use these tools most effectively to verify and validate a formal specification. Most of the literature on specification testing presents a tool or simply mentions that a specification was tested, with no or little discussion of how this was done or how effective it was. Moreover, our experience with these tools has shown that using these tools in an ad-hoc fashion can be tedious and time-consuming.

Kazmierczak et al. [2000] and Liu [1999] present approaches for performing specification testing. We build on these approaches by outlining and testing new properties, as well as providing extensive tool support to make the approach more practical.

This paper presents a framework and tool support for the systematic testing of formal specifications. Although the framework and tool support have been developed for the Sum specification language [Kazmierczak et al. 1995], they can easily be adapted to other model-based specification languages such as Z, B, and VDM.

In earlier work [Miller and Strooper 2001], we present an approach for systematically testing specifications that is completely manual, and takes significant time and effort. In this paper, we expand on that work by identifying important generic properties that model-based specifications should satisfy. Following the idea of mutation analysis [DeMillo et al. 1978], we then create variants or mutants of the specification to check that these properties are satisfied. As well as the generic properties, the framework also supports the checking of application-specific properties. To drive the testing, we use the concept of a testgraph [Hoffman and Strooper 1995; 1998; Miller and Strooper 2002b], which is a directed graph that partially models the states and transitions of the specification being tested. Sequences are derived from the testgraph by traversing the testgraph repeatedly from the start node. The framework provides tool support for the generation of the mutants, for defining testgraphs and automatically traversing the testgraph, for executing the test cases during the traversal, and for reporting any errors. The tool support is built on top of the existing Possum specification testing tool [Hazel et al. 1997; 1998]. The framework and tool support thus provide a systematic approach to specification testing that is partially automated and repeatable.

The framework and tool support are demonstrated on a small integer set example. The effectiveness of the approach is also evaluated on a set of student assignments, in which we compare the approach to the manual evaluation of the student assign-
A Framework for the Systematic Testing of Model-Based Specifications

The application of the framework to two more substantial specifications is also discussed. The experience shows which checks are most effective at revealing errors in specifications. It also shows that the framework can be used effectively to test small to medium-sized specifications, in that the human effort involved in setting up the testing is small compared to the effort required for the specification itself. The case studies also show that the framework can reveal a significant number of problems in these specifications.

While our approach is similar to and based on existing approaches to more traditional software (implementation) testing, fundamental differences exist. For example, an important aspect of our framework is the testing of generic properties of formal model-based specifications that can not be done with implementation testing (which typically focuses on testing application-specific properties). Moreover, the tool support for specification testing and implementation testing are fundamentally different. However, we have started investigating the relationship between the two [Miller and Strooper 2003] and plan to expand this work in the future.

This paper is organised as follows. We briefly introduce the Sum specification language and the Possum specification testing tool in Section 2. We then discuss the generic specification properties that we check in Section 3, and describe how we check these properties in Section 4. Section 5 discusses the use of testgraphs to define the states and transitions that will be tested. We discuss tool support for the method in Section 6. In Section 7, we discuss experience with this framework. We then review related work in Section 8 and Section 9 concludes the paper.

2. BACKGROUND

In this section, we present the example used throughout this paper and introduce the Possum specification testing tool [Hazel et al. 1997; 1998], which is used as the basis for the tool support in the framework.

2.1 Example - IntSet

The example used to demonstrate our method of systematic specification testing is an integer set, called IntSet. The specification is shown in Figure 1.

The IntSet specification is written in Sum [Kazmierczak et al. 1995], a modular extension to Z. Sum specifications contain a state schema, an initialisation schema, and zero or more operation schemas. In the IntSet example, the state schema consists of a state variable intset (a power set of integers), and a state invariant, which restricts the intset to a maximum size of 10, defined by the constant maxsize. The initialisation schema is used to specify the initial state of the specification; in this example, intset is initially empty.

Sum uses explicit preconditions in operation schemas using the pre keyword. Like Z, input and output variables are decorated using ? and ! respectively, and post-state variables are primed ('). All operation schemas automatically include the primed and unprimed versions of the state schema in their declarations. changes_only A states that for an operation, only the state variables in the set A are allowed to change when that operation is invoked. Hence, the statement changes_only{} specifies that no state variables are allowed to change. No changes_only statement in an operation schema implies that all state variables can change.
The four operation schemas in the `IntSet` specification are: `add`, which adds an integer to the set if that integer is not already in the set and the set is not full; `remove`, which removes an integer from the set provided it is in the set; `isMember`, which returns a boolean indicating whether an integer is in the set; and `size`, which returns the size of the set.

### 2.2 Possum

Possum is a testing tool for Z and Z-like specification languages, including `Sum`. Possum interprets queries made in `Sum` and responds with simplifications of those queries. A specification can be tested by stepping through operations, and Possum will update the state after each operation. The example below shows a query sent to Possum for the `add` operation in `IntSet` with the value 3 substituted for the input `x`:

`add{3/x?!}`

If we assume that the value of the state variable `intset` before the query is `{1}`, then Possum replies to this query with:

```
⟨ | intset := {1}, intset' := {1, 3} | ⟩
```

This indicates that the value of the state variable `intset` has been updated to `{1,3}`. Possum also displays bindings for any variables that it instantiates, such as output variables, but the `add` operation has none other than `intset` and `intset'`. 

---

```
<table>
<thead>
<tr>
<th>IntSet</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxsize := 10</td>
</tr>
<tr>
<td>state</td>
</tr>
<tr>
<td>intset : ℤ</td>
</tr>
<tr>
<td>#intset ≤ maxsize</td>
</tr>
<tr>
<td>init' = ∅</td>
</tr>
<tr>
<td>op add</td>
</tr>
<tr>
<td>x? : ℤ</td>
</tr>
<tr>
<td>pre(x? ∉ intset ∧ #intset &lt; maxsize)</td>
</tr>
<tr>
<td>intset' = intset ∪ {x?}</td>
</tr>
<tr>
<td>op remove</td>
</tr>
<tr>
<td>x? : ℤ</td>
</tr>
<tr>
<td>pre(x? ∈ intset)</td>
</tr>
<tr>
<td>intset' = intset \ {x?}</td>
</tr>
<tr>
<td>op isMember</td>
</tr>
<tr>
<td>x? : ℤ</td>
</tr>
<tr>
<td>mem! : ℂ</td>
</tr>
<tr>
<td>changes_only{}</td>
</tr>
<tr>
<td>changes_only{}</td>
</tr>
<tr>
<td>op size</td>
</tr>
<tr>
<td>size! : ℤ</td>
</tr>
<tr>
<td>size! = #intset</td>
</tr>
</tbody>
</table>
```

---

Fig. 1. Sum Specification of `IntSet`
Using queries such as this allows users to execute specifications by stepping through operations of the specification. The behaviour of the specification can be observed and compared to the expected behaviour. For example, after we have added 3 to the set in the last example, we can check whether it is a member of the set:

\[
isMember\{3/x?\} >>
\]

The “\(\Rightarrow\)” symbol means that output variables are displayed separately from other variables. Therefore, Possum replies with:

\[
\{\text{intset }:= \{1\}, \text{intset‘} := \{1, 3\} \}; \\
\text{mem!} := \text{true}
\]

However, Possum is much more powerful and can even determine the set of all elements \(x?\) for which \(isMember\) returns true. For example, assuming that \(\text{intset}\) before the query is \(\{1, 3\}\), Possum will respond to the query

\[
\{x? : \mathbb{Z} \mid isMember\{\text{true/mem!}\}\}
\]

with

\[
\{1, 3\}
\]

It can even enumerate all elements that are not currently in the set in response to the query

\[
\{x? : \mathbb{Z} \mid isMember\{\text{false/mem!}\}\}
\]

We would expect this set to be infinite, because the set of integers is infinite. However, to deal with this problem, Possum has a maximum integer size, which can be set by the user. For example, if the maximum size was set to 32, then Possum would respond to the above query with all integers in the range \(-32..32\), except 1 and 3.

To simplify queries such as this, Possum uses a mix of generate-and-test and other constraint-solving techniques (see [Hazel et al. 1997] for more details).

Possum also supports Tcl/Tk plug-in user interfaces, which allows people not familiar with Sum to interact with the specification through a user interface. Possum defines a simple default user-interface for every specification tested, which displays the current binding for each state variable in the specification being tested.

3. WHAT PROPERTIES DO WE CHECK?

The purpose of specification testing is to validate that the specification behaves as intended. We identify several generic properties for the internal consistency of specifications, which we can check using specification testing. These properties have been selected by evaluating properties that are checked on formal specifications in development environments such as Cogito [Hamilton et al. 1998] and B [Schneider 2001], and specification testing methods such as Pipedream [Kazmierczak et al. 2000]. The properties we have selected are:

— an initialisation check - performed in B and the Pipedream approach;
— a schema satisfiability check - performed in Cogito;
— a precondition check - performed in B, VDM, Cogito and the Pipedream approach; and
— a state invariant check - performed in B.

In this section, we concentrate on the generic properties of Sum specifications, and at the end of the section we discuss how these properties would differ for other specification languages. In the next section, we discuss how we check these properties and also how the framework supports the checking of application-specific properties.

3.1 Definitions
Before explaining the properties in detail, we define some functions and sets.

The state schema. The state schema, including the state invariant, for a specification is denoted by state.

The set of all operations. The set of all operation schemas for a specification is denoted by OP.

The set of all possible states. The set of all possible states that satisfy the signature of state (this does not include the state invariant) is denoted by $S$.

The set of all possible inputs. The set of all inputs for the operation $op : OP$ is denoted by $I_{op}$. For simplicity, we assume a single input for operations.

The set of all possible outputs. The set of all outputs for the operation $op : OP$ is denoted by $O_{op}$. As with input, we assume a single output for operations.

The initialisation function. The function $init : S \rightarrow \mathbb{B}$ returns true if and only if the state $s : S$ satisfies the initialisation schema for the specification.

The pre function. The function $pre : OP \times I \times S \rightarrow \mathbb{B}$ returns true if and only if the precondition for the operation $op : OP$ holds for the state $s : S$ and the input $i : I$.

The post function. The function $post : OP \times I \times S \times S \times O \rightarrow \mathbb{B}$ returns true if and only if the postcondition for the operation $op : OP$, not including the state invariant, is satisfied by the pre-state $s : S$, post-state $s' : S$, the input $i : I$, and the output $o : O$.

The $SI$ function. The function $SI : S \rightarrow \mathbb{B}$ returns true if and only if the state invariant of state holds for the state $s : S$, i.e., the function checks $s \in state$.

3.2 Initialisation Check
The first property we want to check is that there exists a state that satisfies the initialisation schema and state invariant.
Formally, we want the following predicate to hold:

\[ \exists s : S \bullet \text{init}(s) \land \text{SI}(s) \]

In an existential or universal quantification, the \( \bullet \) symbol stands for “such that”. Thus, the predicate states that there exists a state \( s \) in the set \( S \), such that \( s \) satisfies the initialisation schema and state invariant of the specification.

### 3.3 Schema Satisfiability

The next property we want to check is that for all operation schemas, there exists pre- and post states, and values for all input and outputs variables such that these values satisfy the schema.

Formally, for each operation \( op \) in the specification, we want to show:

\[ \exists i : I_{op}; s, s' : S; o : O_{op} \bullet \]

\[ \text{SI}(s) \land \text{pre}(op, i, s) \land \text{post}(op, i, s, s', o) \land \text{SI}(s') \]

Note that the above property is a weak property that just checks that the operation schemas are consistent (i.e., that values exist that satisfy the operation schemas). Moreover, it does not require that the states \( s \) and \( s' \) are reachable from the initial state. As explained in Section 4, in our framework we check a more specific property than this, namely that each operation is satisfiable with reachable state values.

### 3.4 Precondition Checks

**3.4.1 Weak Precondition.** One necessary property of Sum specifications is that the explicit precondition of each operation is not too weak. By this, we mean that there exist no input/state pairs that satisfy the precondition but for which no post-states or outputs exist.

Formally, for each operation \( op \) in the specification \( M \), we want the following predicate to hold:

\[ \forall s : S; i : I_{op} \bullet \text{SI}(s) \land \text{pre}(op, i, s) \Rightarrow \exists s' : S; o : O_{op} \bullet \text{SI}(s') \land \text{post}(op, i, s, s', o) \]

An example of a precondition being too weak is if the precondition of the \textit{add} operation is replaced by \text{pre}(x \notin \intset), in which \#\intset < \text{maxsize} has been removed. Here, the precondition allows elements to be added even when the size is \text{maxsize} or greater. However, the state invariant prevents this from occurring; a pre-state of a full \intset and an input that is a new element satisfies the precondition, but there is no post-state that satisfies the postcondition and state invariant.

**3.4.2 Strong Precondition.** It is also possible for a precondition of an operation to restrict values from being used as input/pre-state even though the postcondition still holds for these values. This can also restrict the possible number of outputs or post-state values. However, this is not necessarily an error, because it may be a deliberate choice by the specifier. For example, if we look at the \textit{add} operation in \textit{IntSet}, one conjunct of the precondition is that the element is not already in the set; however, the postcondition permits an element in a set to be added more than once, although the set will not change. Accepting this, it is clear that specifiers
can unknowingly over-strengthen the precondition, restricting possible input values from being used in the operation.

Formally, we want to check for each operation \( op \) in the specification \( M \):

\[
\forall s, s': S; \ i: I_{op}; \ o: O_{op} \cdot \\
SI(s) \land SI(s') \land post(op, i, s, s', o) \Rightarrow prev(op, i, s)
\]

### 3.5 State Invariant Checks

A Sum specification defines a state, an invariant (possibly true), and operations that define relations on that state. These operations, along with the state invariant, restrict the state from being assigned certain values. For example, the state invariant and preconditions in the \( IntSet \) specification prevent the \( intset \) state variable from holding more than \( maxsize \) elements. We define the set of reachable states, \( R \), for a specification as the set of all states in a specification that the operations’ preconditions and postconditions (NOT including the state invariant) allow.

Formally, we define the set of reachable states, \( R \), for a specification as:

\[
R = \{ s : S \mid \exists ss : \text{seq}\ S \cdot \text{init}(head(ss)) \land \\
(\forall j : 1..\#ss - 1 \cdot \exists op : OP; \ i: I_{op}; \ o: O_{op} \cdot \\
prev(op, i, ss(j)) \land post(op, i, ss(j), ss(j+1), o)) \land s = last(ss) \}
\]

Thus the set \( R \) consists of all the last elements of all the sequences \( ss \) of states of which the first element satisfies the initialisation schema, and all subsequent pairs of states can be reached by an operation (ignoring the state invariant).

Now that we have defined the set of reachable states for a specification, we define two properties using this set.

#### 3.5.1 Strong State Invariant

A state invariant that is too strong restricts the possible inputs, pre-states, post-states and outputs of an operation, and, as a result, prevents a specification from being in a certain reachable state.

To show that this is not the case, we need to prove the property:

\[
\forall s : R \cdot SI(s)
\]

While a state invariant that is stronger than precondition/postcondition pairs is sometimes used to restrict behaviour, this can be considered bad style, and in some notations, for example B, this is an inconsistency. This is discussed in more detail at the end of this section.

An example of a strong state invariant would be if the state invariant for \( IntSet \) is replaced by \( \#\ intset < maxsize \), in which \( \leq \) is replaced by \(<\). This new state invariant prevents the size of the set from reaching \( maxsize \), even though an infinite number of sets of size \( maxsize \) exist in \( R \). 

#### 3.5.2 Weak State Invariant

Another property that we would like to check for specifications is that the state invariant is not too weak. A weak state invariant is not an error, but the tighter the state invariant, the easier a specification becomes to reason about and understand. Therefore, we also want to check that the state invariant restricts as many input and post-state values as it can, without affecting the behaviour.
Formally, we want to check:

$$\forall s : S \bullet SI(s) \Rightarrow s \in R$$

For instance, the requirement that the size of the state variable intset is less than the constant maxsize is expressed by the state invariant $\#\text{intset} \leq \text{maxsize}$. If this is left out, the state invariant will allow the size of intset to be greater than maxsize. Of course, the precondition of the add operation will prevent any states with the size of intset greater than maxsize.

3.6 Discussion

Most of the properties outlined in this section are taken from formal development environments and other specification testing methods. The initialisation check is a proof obligation in the B method [Schneider 2001], and the Pipedream approach [Kazmierczak et al. 2000] advocates checking that an initial state exists. Schema satisfiability is a proof obligation in Cogito [Hamilton et al. 1998]. The weak precondition and strong state invariant are common checks made on formal specifications. A check on the preconditions is a proof obligation in Cogito, VDM, and the B method. However, the B method requires that all post-states satisfy the state invariant if the precondition holds, not just one.

Different specification languages require different properties to hold on the state invariant. In the B method, the state invariant is not conjoined to the pre- and postconditions, and a proof obligation is required to show that the state invariant is maintained by each operation. In the Object-Z specification language [Duke and Rose 2000], some state variables are known as secondary variables, which means that their values are calculated using other state variables (the primary variables). Therefore, for Object-Z specifications, we may choose to only check the primary variables. But like Sum, Z, and VDM, the state invariant of an Object-Z specification is an implicit part of each precondition and postcondition, so all schemas satisfy the state invariant by definition. Therefore, it is a matter of preference as to whether we check for a strong state invariant.

The weak state invariant and strong precondition were included for completeness. These two conditions may reveal inconsistencies or errors in the specification, but in some cases they are intentionally violated by the specifier.

It is interesting to note that the weak precondition and strong state invariant overlap. Some cases in which an operation has a weak precondition may also mean the state invariant is too strong. In an expanded version of this paper [Miller and Strooper 2002a], we prove that:

—An operation with a weak precondition does not necessarily imply that the state invariant is too strong.

—A strong state invariant does not necessarily imply that there is an operation with a weak precondition in the specification.

—If all operations in a specification are deterministic and if the state invariant is too strong, there exists an operation with a weak precondition in the specification.
4. CHECKING THE PROPERTIES

In this section, we discuss how to check the properties from Section 3 using a specification testing tool combined with mutants of specifications. The mutants are created to extract the necessary information about preconditions, postconditions, and the state invariant, and we attempt to detect differences between these mutants for a subset of the possible states of the specification. The selection of these states is discussed later in Section 5 and the automated generation of the mutants and most of the schemas presented in this section is discussed in Section 6.

4.1 Creating Mutants

Mutation analysis [DeMillo et al. 1978], in the traditional sense, involves applying a slight syntactical change to a program, and then checking that a test data set detects this change. If so, then the mutant is killed. If not, then the tester has to decide whether the mutant program is behaviourally equivalent to the original, or whether the test set is inadequate because it failed to uncover the error in the mutant. Multiple mutants are created for a program, and the number of non-equivalent mutants killed is used as a measure of the adequacy of the test data set. However, we use mutants in a different way. We create mutants of the specification, not the implementation. The mutants we generate are non-trivial and we always generate the same number of mutants. There is also no concept of killing a mutant and we do not need to detect equivalent mutants.

To perform checking of the properties outlined in Section 3, we have to determine, for each operation, the precondition, the postcondition, and the state invariant. With Sum specifications, this is simplified by the use of explicit preconditions, indicated by the pre part of operation schemas (see Figure 1). The postcondition is the rest of the predicate section, and the state invariant is the predicate section of the state schema.

We create three mutants of the specification, $M$, being tested. These mutants give us access to the preconditions, postconditions, and state invariant:

— The first has the state invariant removed from the state schema, and the postconditions removed from all operation schemas. This leaves only the signature and the precondition of the operation, unless the operation has output variables, in which case we remove the output variables from the signature because they are no longer referenced in the schema. We also restrict the operation from changing the state of the specification. This gives us access to the precondition of each operation.

— The second has the preconditions removed from all operations. This gives us access to the postcondition and state invariant of each operation to compare it with the precondition.

— The third has the state invariant removed from the state schema. This allows us to compare the behaviour of the specification with and without the state invariant.

These three mutant specifications, along with the original specification, are imported into a new specification, called $top$, and are renamed, using Sum’s import renaming mechanism, as follows: $PR$ for the specification containing only precondi-
A Framework for the Systematic Testing of Model-Based Specifications

\[ \text{init} \]
\[ \text{tgf_report!} : \forall \{ \text{SSI_INIT, INIT_FAIL} \} \]
\[ \text{if} \ (\exists \ M. \text{state'} \cdot M.\text{init} \land \text{changes_only}\{M.\text{state}\}) \text{ then} \]
\[ M.\text{init} \land \text{changes_only}\{M.\text{state}\} \land \text{tgf_report!} = \{\} \]
\[ \text{else} \]
\[ \text{if} \ (\exists \ \text{PRPO.state'} \cdot \text{PRPO.init}) \text{ then} \]
\[ \text{tgf_report!} = \{\text{SSI_INIT}\} \land \text{changes_only}\{\} \]
\[ \text{else} \]
\[ \text{tgf_report!} = \{\text{INIT_FAIL}\} \land \text{changes_only}\{\} \]
\[ \text{fi} \]
\[ \text{fi} \]

Fig. 2. Schema top.init for IntSet

tions, POSI for the specification containing the postconditions and state invariant, and PRPO for the specification containing only preconditions and postconditions, with no state invariant. The original specification is renamed to \( M \). We can use Sun’s module prefixing notation to access the items in each mutant, e.g., \( M.\text{add} \) for the \( \text{add} \) operation in \( M \).

The state schema of the \( \text{top} \) specification includes only the state schema of \( M \).

4.2 Checking the Properties

For each special state we have selected, we want to check the properties outlined in Section 3. Except for the initialisation check, all properties are checked for all state values visited during testing. We also discuss the testing of application-specific properties, e.g., that after adding the integer 1 to an empty set, the \text{size} operation returns \( \text{size} = 1 \), and \( \text{intset} \) is equal to \( \{1\} \).

4.2.1 Initialisation Check. The first property to check is that an initial state exists. To do this, we define an \text{init} schema in the \text{top} specification that checks for an initial state, reporting an error if none exist. Figure 2 shows the schema \text{top.init} for the \text{IntSet} specification.

In this figure, we first check whether an initial state exists for the specification \( M \). If so, we change the state and report no error. If not, we check whether the \text{PRPO} specification (the specification with no state invariant) can reach an initial state. If so, then \( M \)’s state invariant is too strong, so we report this. If not, we report an error that \text{init} failed. Reporting errors is performed using the \text{tgf_report!} variable by including the error message in this set. In Figure 2, \text{SSI_INIT} and \text{INIT_FAIL} are abbreviations of error messages. For example, \text{SSI_INIT} is an abbreviation for the string: “Error: state invariant is too strong for ‘init’ schema”. When an error occurs, the state of \( M \) is not changed.

\(^1\)We add the prefix \text{tgf}, which stands for testgraph (see Section 5), to all visible variables in the \text{top} specification to prevent name clashes.
4.2.2 Precondition Check. To check that the precondition of an operation is not weak, we try to find an input that satisfies the PR specification, but that fails for the POSI specification. If this occurs, we know that the precondition of the operation is too weak for the postcondition and state invariant. We can also check that the precondition is not too strong by reversing the above check. That is, if an input satisfies the POSI specification but not the PR specification, the precondition may be stronger than necessary.

For the add operation in IntSet, we define a schema called check_add_pre, shown in Figure 3.

In this figure, WPRE_add is an abbreviation for the string: “Error: precondition for operation ‘add’ is too weak!”, and SPRE_add is defined similarly for a strong precondition.

In this schema, we first assign the set of all possible inputs that satisfy PR.add and POSI.add to pr.add and posi.add respectively. Here we use the power of Possum to search for all values of the input variable x? that satisfy these schemas. This is only feasible because Possum restricts the range of integers (see Section 2.2), and this range can be controlled by the tester. In this case, we restrict the range to the interval \(-32..32\), because we never insert or remove elements from the set outside this range during the testing. Although we cannot guarantee that we do not miss any errors because of this, this seems very unlikely.

Note also that in check_add_pre we substitute each occurrence of intset with M.intset, which means that the operation is checked using the current value of M.intset as the value of intset for the POSI and PR evaluations of add. Then, we check whether there exists a value in pr.add that is not in posi.add. If and only if this is true, we add WPRE_add to tgf_report!. We then do the reverse for the strong precondition check.

As an example, we use the add operation from the IntSet specification, and weaken its precondition as in Section 3 by removing the constraint \#intset < maxsize. If we invoke the check_add_pre schema when the set is full, then the PR version of the specification will allow us to add any integer not already in the set \(\{x : \# -32..32 \mid x \notin \text{intset}\}\), while the POSI version will allow us to add none \(\emptyset\). Therefore, the weak precondition error message is added to the set of messages.
A Framework for the Systematic Testing of Model-Based Specifications

13

check_add_si
state
tgf_report! : P{SSI_add}

\exists \text{prpo_add}, \text{m_add} : P(Z \times P Z) | 
\text{prpo_add} = \{ \text{PRPO_add}(\text{M.intset/intset}) \bullet (\text{PRPO}\cdot x?, \text{PRPO}\cdot \text{intset}') \} \land 
\text{m_add} = \{ \text{M.add}(\text{M.intset/intset}) \bullet (\text{M}\cdot x?, \text{M}\cdot \text{intset}') \} \bullet 
(\exists x : \text{prpo_add} \cdot x \not\in \text{m_add}) \Leftrightarrow \text{SSI_add} \in \text{tgf_report}!

Fig. 4. Schema check_add_si for IntSet

Similarly, if we use the original definition of add, which has a strong precondition, pr_add will contain all integers except those already in intset (pr_add = \{ x : -32..32 | x \not\in \text{intset} \}), and pos_add will be the set -32..32, because if an element already in a set is added again, the set does not change. Therefore, the strong precondition warning is added to the set of messages.

4.2.3 State Invariant Check.

4.2.3.1 Strong State Invariant. To check that the state invariant of the specification is not too strong, we try to find an input and post-state that satisfies the PRPO specification, but not the original specification, M. If this is the case, it means that the state invariant is restricting the input space, or preventing a transition, because the state invariant is the only difference between the two specifications.

As an example, we again use the add operation in the IntSet specification. The schema for checking the state invariant, check_add_si, is shown in Figure 4.

In this schema, we first assign the set of all possible input/post-state pairs for PR.add and POSI.add to prpo_add and m_add respectively. As with the precondition check, we substitute each occurrence of intset with M.intset. Then, we check whether there exists a value in prpo_add that is not in m_add. If and only if this is true, then we have an input and post-state that can occur in the PRPO specification using the operation add, but not for M. The only difference between the two specifications is the missing state invariant, therefore it must be the state invariant restricting M’s progress and the SSI_add error message is added to tgf_report!.

To demonstrate this approach, we use the example from Section 3, in which the state invariant is too strong because the \leq is replaced by <. If we invoke the check_add_si schema in a state with maxsize - 1 elements, prpo_add will contain all integers not already in the set and their corresponding post-value for intset (prpo_add = \{ x : -32..32; s : P -32..32 | x \not\in \text{intset} \land s = \text{intset} \cup \{x\} \}). m_add will be empty (m_add = \emptyset), because the state invariant is preventing elements from being added. Therefore, the state invariant is too strong and the SSI_add error message is added to tgf_report!.

4.2.3.2 Weak State Invariant. Detecting a weak state invariant using a specification testing tool is more challenging because we want to know if all states that satisfy the state schema are reachable via the operations in the specification, so we have to search for these states using the operations in the specification. It is infe-
sible to check every state even for a small example such as the IntSet specification.

To check for a weak state invariant, we perform a search on the state space of the specification using the operations in the specification. We first derive a set of states we wish to reach, and then search the state space to find them. However, we reduce the size of the search space so that we do not have to search all possible states. For example, the search space for the IntSet example is:

\[
\text{search\_space} = \{ s : \mathbb{P} \mathbb{Z} \mid (\exists s.t : M.\text{state} \bullet s = s.t\text{.intset}) \}
\]

which is all possible integer sets of size less than or equal to maxsize (recall that maxsize is defined in the IntSet specification). We can reduce the maximum integer size in Possum to reduce the size of the search space. Alternatively, we could make the search space much smaller:

\[
\text{search\_space} = \{ s : \mathbb{P} \mathbb{Z} \mid (\exists d : 0..M.\text{maxsize} \bullet s = 1..d) \}
\]

which restricts our search space to only maxsize + 1 sets containing 1..d for each different value of d. Reducing the search space clearly reduces the chance of finding a state that is not reachable, but if we can reach them using this approach, then we are guaranteed that the states we have chosen are reachable. Note that the states we are searching for must be a subset of the search space.

This method worked surprisingly well for the IntSet example. We tested the method on a slightly larger example (the Graph example discussed in Section 7) and the method did not scale to that example due to time limitations.

Legard et al. [2002] perform a similar search for partial reachability of states of B specifications using a constraint solver to generate test sequences. The constraint solver is used to simulate the execution of the system, but because it symbolically executes the specification, the state space can be searched much more efficiently. Moreover, their motivation is different from the motivation of this thesis because they focus on generating test sequences that reach specially selected states for specification-based testing. They search for the reachability of only those states that will be used for testing.

4.2.4 Checking Application-Specific Properties. We may also want to check other properties that are specific to the specification being tested. We implement such checks by defining schemas for specific states that we want to test. For example, the CHECK_ONE schema, shown in Figure 5, checks the state \{0\}. It checks the isMember operation for a true and a false case, the size operation, and the value of intset.

Selecting the application-specific properties to test is similar to the test case selection problem for traditional software testing (of implementations). While specification-based testing approaches exist for selecting test cases from formal specifications for testing implementations of those specifications, this makes much less sense when those test cases are used to test the specification itself. Instead we rely on more traditional heuristics for software testing to define application-specific tests that we perform. For example, for the isMember operation, we test one value in the set (if the set is not empty) and one value not in the set. Note that although the selection of these tests is not automated, the execution and checking is.
4.2.5 Schema Satisfiability. The satisfiability of schemas does not need to be explicitly checked, because if the application-specific properties are checked at reachable pre- and post-states then this implies that the schemas are satisfiable. However, we always check that the schemas are satisfiable before performing the precondition and state invariant checks, otherwise they may not check anything. For example, if an operation is not satisfiable, then no input exists for that operation, and the set of all possible inputs and post-states would be empty for all mutants of the specification. Therefore, we will never find an input that satisfies one mutant and not another, and the schemas that check the generic properties will report no errors. The application-specific checks are different because they check the value of the state and the return values of operations. If these tests pass, then the schema must be satisfiable. Therefore, it makes sense to run the application-specific checks before the precondition and state invariant checks.

Note that this is a more specific check than defined in Section 3, because we actually check that the schema is not only satisfiable, but that it is satisfiable with reachable state values.

4.3 Evaluation of Error Messages

The error messages generated by our method are straightforward in telling the user what the problem is, e.g., a weak precondition, but not very useful for discovering why the error occurred. If Sum supported concatenation of strings, more meaningful error messages could be generated by supplying information such as the current state and, for example, the specific value that satisfies the PR version of add, but not the POSI version.

To improve the feedback given to the user, we use output variables in our check schemas that give us more detailed feedback on errors found. For example, as in the check_add_pre schema in Figure 3, we can declare an output variable weak_add! : PZ and set its value to be the set of inputs that satisfy the PR version of the schema, but not the POSI version (weak_add! = pr_add \ posi_add). This gives us feedback as to which values satisfy the precondition but not the postcondition.

It is possible for apparent contradictions to occur in the error messages. For instance, it is possible for both weak precondition and strong precondition error messages to be reported. As an example, if we were to remove the #intset <
maxsize from the precondition of `add`, we would have a weak precondition. However, because the precondition does not allow elements already in the set to be added, we also have an instance of a strong precondition. Our framework will detect both problems and display both error messages.

4.4 Discussion

The properties outlined in Section 3 are checked for a subset of the possible states, and at each of these states, we thoroughly check the specification to detect problems. However, the weak state invariant check is different from the other checks. This check requires us to search the state space to find a selected set of states. This is similar to the approach taken by model checkers.

Model checking [Clarke et al. 1999] is an automated technique for verification of formal, state-based specifications. Model checking involves supplying certain properties to the model checker that must hold for all states of a specification. The model checker will then check every possible execution path of the specification, returning true if the specified properties hold, or providing counter-examples that show why the properties do not hold. Due to its exhaustive nature, state explosion is a major problem with model checking.

Our method for checking the preconditions and for a strong state invariant combines ideas from testing and model checking. As with other forms of testing we avoid the problem of state explosion by selecting only a small subset of the states to be tested, but for those states we exhaustively check a large input space as in model-checking. However, for the weak state invariant check, the problem of state explosion arises again, because we may have to search the entire state space to determine if certain states are reachable. For the `IntSet` example, narrowing the search space was straightforward, but for other examples, this has not been the case. We believe an approach such as model checking may be better suited for detecting a weak state invariant.

5. DEFINING A MODEL OF THE SPECIFICATION

In this section, we discuss using testgraphs to model a subset of states and transitions of the specification. We use testgraphs because they are straightforward to derive, and deriving tests from testgraphs can be done quickly and automatically. Testgraphs give us a systematic approach to specification testing that allows us to analyse the specification as a whole instead of testing each of the operations in isolation.

We generate our testgraphs manually from the informal requirements. Approaches exist for generating testgraphs from formal specifications [Carrington et al. 2000; Murray et al. 1998], but when the testgraphs are used to validate the specification this makes much less sense. Moreover, our practical experience has shown that defining testgraphs manually is straightforward.

5.1 Deriving a Testgraph

A testgraph is a directed graph that partially models the states and transitions of the specification being tested. Each node in the testgraph represents a possible state in the specification, and each arc a possible transition (a sequences of calls
to operations) between states. One state in the testgraph is selected as the start
node, and this node represents one of the initial states.

In Section 3, we discuss checking properties for all states of a specification. How-
ever, using a specification testing tool, it is infeasible to check the entire state space
of specifications, except for specifications with very small state spaces. Even the
IntSet example has such a large state space that we cannot afford to check all states.
Therefore, we select a subset of the state space for testing.

The state and operations of a specification provide important information about
the selection of states to test. For example, the add operation in the IntSet speci-
fication will behave differently when the set is full (has maxsize elements in it) to
when it is not full.

Standard testing practice advocates several heuristics for selecting special states,
such as the interval rule. For the IntSet specification, we select our states based
on the size of the set. We test an empty set, a set containing one element, a set
that is half-full, a set with maxsize – 1 elements, and a set that is full. For our
testgraph states, we split the set with multiple elements into two different sets:
one containing only odd elements and one containing only even elements. We split
these states to give good coverage of the specification, and we choose the odd and
even states because they are easy to generate. For each one of these special states,
we select one possible instantiation of the state. For example, the state containing
one element becomes the set \{0\}. Each of these instantiations becomes a node in
our testgraph.

Once we have our testgraph nodes, we derive arcs for the testgraph to be used
as transitions during testing. A requirement is imposed that each node in the
testgraph is reachable from the start node. In addition to making sure each node is
reachable, we also add arcs if there exists any specific transitions we want to test.

Figure 6 shows the testgraph for the IntSet specification. Here, we have six
nodes representing the states defined above: EMPTY, ONE, ODD, EVEN, FULL-
1, and FULL. EMPTY is the start node and this is indicated by the node being
shaded. The seven arcs on the testgraph change the state of IntSet from one state
to another. For example, the ADDODD arc represents a transition that adds 1, 3,
5, 7 and 9 to the set. This takes us from EMPTY to ODD.

5.2 Traversing a Testgraph

To traverse a testgraph, the user must define a transition schema for each arc in
the testgraph. This schema is called when that arc is traversed. For simplicity, we
use the labels of the testgraph as the schema names.

When a transition fails, we would like to receive feedback as to why. We therefore
define a wrapper schema similar to the ones shown in Section 4 that uses the
mutants to determine the cause of the problem when a transition fails. For example,
if the transition fails, we check whether the PRPO version of the specification
can make that transition. If so, we know that the state invariant is preventing
the transition. As discussed in Section 6, these wrapper schemas are generated
automatically.

We generate our test sequences by traversing the testgraph to achieve arc coverage.
The other two types of coverage considered were node and path coverage.

Node coverage of a testgraph is achieved if every node of the testgraph is reached.
Fig. 6. Testgraph for IntSet

This criteria is inadequate because we can achieve node coverage without traversing all arcs. For example, node coverage can be achieved on the IntSet testgraph without traversing the arc REMOVEODD.

Node coverage is subsumed by arc coverage. Arc coverage is achieved if every arc of the testgraph is traversed. If each arc is traversed, then each node must also be reached.

Arc coverage is subsumed by path coverage. Path coverage is achieved if every path in the testgraph is traversed. This is impossible to achieve for any testgraph with a cycle, such as the IntSet graph.

Arc coverage traverses every arc and visits every node (provided all testgraph nodes and arcs are reachable from the start node). As discussed in Section 6.1, we have implemented a straightforward traversal algorithm that achieves arc coverage. The algorithm executes in a reasonable amount of time, but can generate a large number of paths. Therefore we use the traversal algorithm for simpler testgraphs, but for more complicated testgraphs, or when execution time becomes an issue, we manually define a set of paths that achieve arc coverage instead (this has been straightforward in practice).

5.3 Checking States and Operations

When we reach a new node in the testgraph, we want to check properties of the current state of the specification, e.g., the precondition and state invariant checks, as well as the application-specific checks. For that purpose, we use the schemas defined in Section 4, such as \texttt{check\_add\_pre} in Figure 3 and \texttt{CHECK\_ONE} in Figure 5.

If a node has been visited previously in the traversal, we need not perform a check like the one above, but instead check that the current value of the state is the same as the previous visit. Possum makes this possible because it displays bindings for variables associated with a specification. If the states are the same, our checks will...
not find anything different. If not, we have uncovered a problem in our specification or our testgraph.

5.4 Problems Found in $\text{IntSet}$

The only error found in the $\text{IntSet}$ specification was a weak precondition for the $\text{add}$ operation. There was no property in the precondition forcing the size of the set to be less than $\text{maxsize}$ when adding an element. This error was introduced by accident when it was decided to add the maximum size constraint to the specification (previously there was no bound on size), and the precondition of $\text{add}$ was overlooked, with only the state invariant being updated.

In addition, two warning messages are generated. Both the $\text{add}$ and $\text{remove}$ operations have strong preconditions. The $\text{add}$ operation does not allow an element to be added if already in the set, and the $\text{remove}$ operation does not allow an element to be removed if it is not in the set. This was a deliberate choice by the specifier to make the user aware that the element will not be added/removed.

6. TOOL SUPPORT

Applying the method outlined in Sections 4 and 5 is tedious and time-consuming. In this section, we describe tool support for this method.

The tool support for our framework is broken up into two components. The first component allows us to edit, save, and restore testgraphs, and performs partial automation of testgraph traversal and report compilation. The second automatically generates the mutant and top specifications.

With the above tool support, the tester typically performs the following steps when testing a specification:

1. Define a testgraph using the testgraph editor (Section 6.1).
2. Define the state transition schemas associated with the arcs and the application-specific checking schemas. This must be done manually.
3. Generate the mutants and the top-level specification. This is fully automated (Section 6.2).
4. Traverse the testgraph and execute the test cases. As described in Section 6.1, three modes exist for doing this. Initially, the tester will typically perform the transitions manually to detect and fix any obvious problems in both the specification being tested, but also any problems in the manually written test schemas. Eventually, after most problems have been fixed and also during any regression testing after the specification has changed, the test cases are executed automatically using either paths generated by the traversal algorithm or using manually constructed paths. In either case, the paths are generated or chosen so that they achieve arc coverage of the testgraph.

6.1 The Testgraph Framework

The Testgraph Framework allows the user to build testgraphs using a testgraph editor and provides assistance for testgraph traversal.

The testgraph editor, shown in Figure 7, allows the user to add/remove nodes and arcs to/from the testgraph and to identify the start node. A label is associated with each arc to represent the name of the transition and the corresponding schema,
which must be defined by the user and will be invoked during the traversal of the arc. The user can associate a schema or sequence of schemas with a node, which are invoked during the traversal of the testgraph to perform a check on a state. Testgraphs can be saved to disk and opened again at a later time.

The testgraph framework also generates a sequence of paths that achieve arc coverage using a depth-first search. It starts at the start node and adds each node to the current path until a node that is already in the path is reached, or no more nodes leading from the current node exist. If there exists unreachable nodes or arcs, the path generation algorithm ignores these. The execution time, and the space and time complexity of the algorithm are discussed in Section 6.3.

The framework allows the user to save paths that have been generated, and open them again at a later time. This allows users to view and edit paths, or manually generate their own paths.

The user can traverse the testgraph in three ways: using manual traversal, for which the user selects the next arc to traverse; stepping through the testgraph, for which the user tells the framework to traverse the next arc in the generated or specified paths; or using automatic traversal, for which no interaction is required and the framework traverses the paths automatically.

For the latter two traversal options, we traverse the testgraph, calling the appropriate check schema(s) and transitions for each node and arc respectively. The pseudocode for the testgraph traversal is:
for each path $P$ to be traversed
  call the init command
    for each arc $A(S, D)$ in $P$
      where $S$ is the source and $D$ the destination
        call the transition for arc $A$
          sequentially call all check schemas for node $D$

The init command is a user-defined schema for initialising the specification. This is generally the top specification’s init schema.

Report compilation can also be automated using the framework. The check operations discussed in Section 4, such as CHECK.ONE, report problems using a variable called tgf_report!, which is a set containing error and warning messages. During testing, the testgraph framework reads the value of this variable every time it changes, and records its contents, along with the current transition, source and destination nodes. The result is a report containing all messages generated and at which nodes and after which transitions they occurred.

The testgraph tool also supports regression testing. When a node is reached for the first time during a traversal, our tool associates that node with the current state value, and this value is saved when the testgraph is saved. On subsequent traversals, the state at that node can be compared to its saved value. This option can be turned on/off.

6.2 Generating the top and Mutant Specifications

From experience, we found that generating the schemas that perform checking of the state invariant and preconditions is tedious and error-prone, even with the smallest of specifications. This prompted us to write tool support to generate these specifications.

To generate the checking schemas, the user simply creates a script containing: the name of the file in which the testgraph is stored, the state variables in the specification, and for each operation, the name of the operation and the variables the user wishes to check. From this script, the tool will generate most of the top specification. All error messages are defined, the top state and init schemas, and all schemas for checking the preconditions and state invariant are generated. In addition, it will read the arc labels from the testgraph and define the wrappers for the transition schemas (see Section 5), and create skeletons for application-specific schemas provided they are specified in the testgraph.

We have also fully automated the generation of mutants. The generation of mutants is simplified in Sum, because the precondition and postcondition are explicitly separated. A tool was developed that reads in a Sum specification, and creates three new files containing the PR, POSI, and PRPO versions of the specification.

6.3 Execution Time

All execution times reported below and in Section 7 were obtained on a SUNW Ultra-80 machine with a 400MHz processor running Solaris 5.8.

The current testgraph traversal algorithm was easy to implement, but is clearly not very efficient. For example, a testgraph with $n$ nodes can have at most $O(n^2)$ arcs, and the traversal algorithm can generate $O(n!)$ paths for such a graph. However, the run-time of the traversal algorithm has not been a problem in practice,
because testgraphs typically remain fairly small (our largest testgraph contains 19 nodes and 28 arcs) and the most substantial overhead in execution time is the time that Possum takes to check the generic properties. However, as we discuss below, the number of paths generated by this naive algorithm can be prohibitively large.

Checking large state spaces and large input spaces consumes much CPU time on machines. To traverse a testgraph with simple checks is not a problem, but the time that Possum takes to check the generic properties is the limiting factor during execution. We have controlled certain parameters to help reduce the execution time, such as selecting only special states to test. However, even reducing the input space by reducing the maximum integer in Possum can lead to a large amount of inputs, therefore we have added a feature, called memoisation, to our tool support to help control this.

Memoisation uses the feature of regression testing of storing state values so that once a node is visited, the checks associated with that node are not performed on subsequent visits. The only check is that the value of the states are the same. If the value of the states are the same, then we need not perform the other checks. If not, an error is added to the report. At first glance, this may appear the same as regression testing, however, we can use the regression testing while still forcing the checks at every node. Memoisation can be turned on/off.

As an example of how useful memoisation can be, we ran our Graph specification (discussed in Section 7) without memoisation, and then with memoisation. The time it takes to execute the Graph without memoisation is 23 minutes. If we use the memoisation feature, we can reduce the time to 5 minutes and 15 seconds. Even for this small example, the time it takes to execute the tests is significantly reduced to under 23% of the original time.

In addition, we can also reduce the amount of transitions and checks made in the testgraph by generating paths manually. Rather than use the paths generated by the traversal algorithm, we can define our own paths in a file. For the Graph specification, if we define our own paths (that still achieve arc coverage) the execution time with memoisation and our own paths is 2 minutes and 25 seconds, which is approximately 10.5% of the original execution time. Although these execution times are non-trivial, it should be noted that the execution is fully automated, so that the human effort involved is minimal.

7. EXPERIENCE

In this section we discuss three case studies that have been performed using our framework. The first is a collection of Graph specifications submitted by students as assessment for an honours-level course at the University of Queensland. The other two case studies are more substantial specifications. We performed these case studies to measure both the cost of using our framework and its ability to find errors.

7.1 Graph

The Graph case study was performed to look at the types of errors that are made during specification, and to see whether our framework picks up such errors. To obtain a large number of specifications, we evaluated a group of assignments from 26 fourth-year students studying a subject called High Integrity Software Engineering.
at the University of Queensland. The assignment required them to write a Sum specification of a graph.

7.1.1 Description. The graph specification takes, as a parameter, the set \( DATA \).

The state schema maintains a set of nodes of type \( DATA \) that represent the nodes in the graph, and a set of edges, which are pairs of nodes representing directed edges in the graph. The state invariant states that for all edges, the start and end node must be in the set nodes, and that no edges exist from a node to itself. The initialisation schema initialises the graph to have no nodes or edges. The students were given the state schema, the signature for each operation, and asked to complete the five operations for the specification. They were also given a specification of a Dependency Management System with similar behaviour. The students were then asked to test their specifications using Possum until they thought it was correct.

We hand-analysed the submitted specifications to find any problems and categorise them. We then applied our framework to all of the student assignments. A script was used to run the mutants generation tool over all the assignments and to copy all files needed for the framework, e.g., the files containing the testgraph and automatically generated top specification. We ran the testgraph framework over the students’ submissions using automatic testgraph traversal and saved the resulting report.

7.1.2 Results. As stated in Section 6 all execution times reported in this section were obtained on a SUNW Ultra-80 machine with a 400MHz processor running Solaris 5.8.

The cost of setting up the testgraph for the Graph specification was minimal, taking between 2 and 3 hours. The testgraph contains 9 nodes and 18 arcs. By using our own paths and memoisation (see Section 6.3), the execution time for one specification is under 3 minutes per assignment (and fully automated).

The results of applying the framework are very positive. Table I shows a summary of the problems found. The first column gives the number of errors found using manual analysis. The second gives the errors found using the framework. All of the non-satisfiable operation, weak precondition, strong precondition, and strong state invariant problems found by hand were found by the framework. The weak state invariant is not checked for, so we detected none of these. The problems classed as “other” were also all detected.

From this table, we can see that the student specifications contained a large num-

<table>
<thead>
<tr>
<th>Message</th>
<th>Manual Analysis</th>
<th>Automated Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialisation</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Non-satisfiable schema</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Weak Precondition</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>Strong Precondition</td>
<td>120</td>
<td>120</td>
</tr>
<tr>
<td>Strong State Invariant</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Weak State Invariant</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>Loop in Path</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>“Out of Bounds” Error</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Other</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Total</td>
<td>192</td>
<td>165</td>
</tr>
</tbody>
</table>

Table I. Summary of Reports for Graph Specifications

ber of errors that violate generic properties. This shows clearly that these checks are worthwhile. The most common occurrences are strong preconditions, which is not surprising considering these are considered quite normal in specifications and are not necessarily classed as errors.

The “Out of Bounds” errors occur when students attempt to index a sequence outside of its domain. However, in the four occurrences, they were all indexed incorrectly inside a universal quantifier:

\[ \forall i : 1 \ldots (\#s!) \bullet (s!(i), s!(i + 1)) \in \text{edges} \]

instead of

\[ \forall i : 1 \ldots (\#s! - 1) \bullet (s!(i), s!(i + 1)) \in \text{edges} \]

Here, for the case when \( i = \#s! \), the expression \( \#s!(i + 1) \) is undefined. However, Possum only evaluates the indexes that are in the domain and ignores the rest. As a result, these errors cannot be detected using Possum.

The “Loop in Path” errors are when the \textit{FindPath} operation would find a path of length one when given the same source and destination nodes. The requirement that no edges exist from a node to itself meant that this should not happen.

While we were not able to uncover all problems in the specifications, we uncovered most at a fraction of the cost compared to manual detection (on average, it took the marker of the assignments 15 minutes per specification once he was up to speed and the initial specifications took much longer). The human effort for applying our framework to all 26 specifications in this case was greatly reduced because we defined only 1 testgraph and 1 set of application-specific schemas to test them.

7.2 Mass Transit Railway

The MTR specification is taken from Miller and Strooper [2001] and this version of the specification was manually translated from Duke and Rose [2000]. The specification is a simplified model of the Hong Kong Mass Transit Railway ticketing system. This specification is of particular interest to us because it contains more than one module: three low-level modules and one top-level module that uses the three low-level modules to perform its services.

7.2.1 Description. The MTR consists of a set of passengers and a set of stations that the passengers travel between. To enter the network, a passenger must obtain a ticket, which is supplied to the system upon entering and exiting the network. Tickets can be single-trip, multi-trip, or season tickets. Each ticket has an expiry date and a value. The value of the ticket is decremented by the fare amount when the passenger leaves the network. Fare amounts are stored in a database that supports the addition of new fares and the updating of existing fares. All tickets can be reissued, but only as the same type as they were originally issued. The current date can be incremented.

The MTR consists of four modules: one to record and increment the date, one to store and retrieve fares, one to store and retrieve tickets and their values, and a top-level specification that uses these three to perform the behaviour described above.

The MTR specification contains 23 operations, and about 400 lines of Sum.
7.2.2 Results. Again, the results of this case study were positive. The cost of setting up the testgraphs was minimal, taking under one person-day of effort. The testgraph for the module to record the date contains 4 nodes and 3 arcs, and the tests take only 2 seconds to execute. The testgraph for the module to store fare information contains 5 nodes and 5 arcs, and the tests take 4 seconds to execute. The module to store ticket information contains 12 nodes and 15 arcs, and takes 27 minutes to execute. The testgraph for the top level specification contains 11 nodes and 15 arcs, and takes 1 hour and 43 minutes to execute. Therefore, the overall execution time is 2 hours and 10 minutes. However, since the testing is fully automated, no interaction from the tester is required during this time, so we do not perceive this as a major issue.

Previously, we performed testing on the Mass Transit Railway specification, uncovering five errors in total [Miller and Strooper 2001]. With the expanded framework and tool support, we found these five and one additional error: a weak precondition. Other errors included: allowing state variables to change when they should not, swapping operands of domain restrictions and subtractions, and a non-satisfiable schema. We also found 14 strong precondition warnings. As discussed in Section 3, this is not necessarily a problem with the specification, however, in the Tickets module, we did uncover an error in the postcondition from a strong precondition warning message. Table II shows the number of errors and warnings found in each category.

7.3 TrackCAD

TrackCAD [Robinson et al. 2001] is part of a joint project between Queensland Rail and the School of Information Technology and Electrical Engineering at the University of Queensland. The aim of this project is to develop a prototype toolset to aid in construction of functional specifications of railway signalling layouts. TrackCAD’s purpose is to model the connectivity information of railway track topologies.

This specification is of interest to us because the corresponding specification is being developed toward use in a commercial tool. Until we applied the framework to this specification, our case studies had either been small examples or a more substantial specification written by ourselves (the MTR specification).

7.3.1 Description. The TrackCAD tool is used to model the topology of railway track layouts. A segment is the most basic part of a track and is represented by a directed arc between two nodes. Each track segment has an identifier associated with it and multiple segments may be grouped together to form a track, which also has an identifier. Two tracks can be joined together using a joint. A point...
is used to connect three segments together, with each segment belonging to the same track. Signals are associated with nodes and track segments as additional information about the layout.

The user of TrackCAD can add and remove segments, points, and signals, and perform checks on the well-formedness of the track layout, e.g., that no connection is connecting more than four track segments.

The TrackCAD specification contains two modules. The first module contains 13 state variables, 14 operations, 16 axioms, and 378 lines of Sum, and the second module contains 9 state variables, 8 operations, 2 axioms and 170 lines of Sum. While neither of these contain many operations, the schemas are considerably more complex than any other specifications we tested.

7.3.2 Results. Our framework was easy to apply to the TrackCAD specifications, and we discovered a significant number of errors in both modules. Table II gives a breakdown of the errors and warnings generated using our framework.

The cost of setting up the testgraph and writing schemas for checking the application-specific properties was approximately two person-days of effort. While this may appear significant, it is significantly less than it took to write the specification (it is hard to provide an estimate of this effort, since the specification was developed and reviewed by a number of people over a period of a few months). The testgraph of the first specification contains 19 nodes and 28 arcs, and the testgraph of the second specification contains 19 nodes and 27 arcs.

To execute the tests took significantly longer than the other case studies that we have performed. With both memoisation and manually specified paths, the tests for the first module took 2 hours and 40 minutes, and for the second module, 2 hours and 3 minutes. However, again this is not a significant problem as during this time no interaction from the tester is required.

The errors that were found included schemas stating what values are not allowed in a set, but not stating what values are allowed in the set. As a result, Possum was making a non-deterministic choice of the other values, when the correct behaviour was to remove the elements from the existing set. Other problems included oversights, such as forgetting to remove elements from sets, using 0 instead of 1 as the first index of a sequence, and in one case, allowing all state variables to change when only one should. A weak precondition was also uncovered.

From Table II, it can be seen that again we have a large number of strong precondition warnings, and only one of these warnings uncovered an error. The large number of these warnings indicates that if the designer of the specification is content to use strong preconditions, then this check could be more of a hindrance than a benefit. In this case, such a check could be dropped.

8. RELATED WORK

Several approaches exist for specification verification and validation, including theorem proving, model checking, and specification testing. Currently, theorem proving and model checking require more sophisticated tool support and specialised knowledge than specification testing. We therefore restrict a detailed comparison to the related work on specification testing, mutation analysis of specifications, and testing using finite state machines (FSMs).
8.1 Specification Testing

Several specification testing tools exist that automatically execute or interpret specifications. The specification testing tool we use in this work is Possum [Hazel et al. 1997; 1998], which executes Z and other Z-like languages. PiZA [Hewitt et al. 1997] is another testing tool for Z. PiZA translates specifications into Prolog to generate output variables. PiZA provides a facility to embed Prolog statements within the Z specifications and make calls to Prolog from the specifications. Jaza [Utting 2000], which stands for Just Another Z Animator, is a Z testing tool that appears to be more advanced in its execution than other testing tools, because it uses different data structures for representing different types of sets. However, Jaza is less flexible than Possum, and its user interface is more difficult to use. The B-Model animator [Waeselynck and Behnia 1998] is used in the B formal development process [Treharne et al. 1996]. It is used to test specifications written in B’s model-oriented specification language. The Software Cost Reduction (SCR) toolset [Heitmeyer et al. 1998] contains an interpreter that is used to test specifications. The IFAD VDM++ Toolbox [IFAD], used for development from the object-oriented extension of VDM, contains an interpreter. This interpreter is used to test specifications, and contains a coverage tool that measures what percentage of specification statements are exercised for each operation during a trace.

Kazmierczak et al. [2000] are one of only a few groups to discuss how they perform specification testing. They use a tool called Pipedream and outline an approach for specification testing using Pipedream containing three steps: performing an initialisation check; verifying the preconditions of schemas; and checking a simple reachability property. We build on this approach by defining new properties to check, and by providing a well-defined framework for checking these properties, including tool support.

Liu [1999] also discusses how to use specification testing tools. Liu uses specification testing to test proof obligations. These proof obligations are tested to check that they hold for a subset of possible traces of the specification. However, it is unclear how the proof obligations could be tested using tools such as Possum, PiZA, and Jaza, and tool support for the method is not discussed.

8.2 Mutation Analysis

Mutants have been used to generate test cases by creating mutants of both the implementation being tested and the specification of the implementation, but not to test specifications. Below, we discuss the related work on the application of mutation analysis to formal specifications, but in both cases the mutants are generated to assist with testing the implementation.

Ammann et al. [1998] and Ammann and Black [1999] discuss the application of mutation analysis to specifications to generate test data sets. They apply slight syntactical changes to a specification to generate mutants, and use model checking to detect equivalent mutants and to detect counter-examples. The equivalent mutants are discarded and the counter-examples are used as test cases.

Callahan et al. [1998] have used model checking combined with mutation analysis to help drive testing. They use the counter-example feature found in model checkers to derive sequences for testing. They apply slight syntactical changes to
specifications to create mutants that purposely force the model-checker to find a counter-example of a property, and then use the paths in these counter-examples to derive FSMs for driving the testing process.

8.3 Testing Using Graphs and Finite State Machines

Much work has been done on testing using finite state machines (FSMs). In this section we discuss the work that is directly relevant to our use of testgraphs for specification testing.

Dick and Faivre [1993] generate FSMs from formal specifications. They generate FSMs by retrieving the pre- and post-states from test cases generated by partitioning schemas into *disjunctive normal form* (DNF), and using them as the states of FSMs. A transition is created between two states if the two states can be related via an operation. The FSM is then traversed, with every branch executed at least once. A number of other approaches exist that extend Dick and Faivre’s work [Chang et al. 1998; Hierons 1997; Turner and Robson 1995].

Hoffman and Strooper’s approach [1995; 1998] is the most relevant to our work. They generate test cases for C++ classes by automatically traversing a testgraph, a directed graph that partially models the states and transitions of the class-under-test, using *Classbench*. These testgraphs are usually derived manually, without the aid of a formal specification. Later work by Murray et al. [1998] and Carrington et al. [2000] describe generating Classbench testgraphs from FSMs. States for these FSMs are derived by using the *Test Template Framework* [Carrington and Stocks 1994; Stocks and Carrington 1996] to specify sets of test cases, and extracting pre- and post-states from these test cases. Transitions are then drawn between each node if possible, and the FSM is converted into a testgraph. We build on this work by using testgraphs to sequence specification testing, but rather than derive testgraphs from FSMs, the user generates the testgraph manually. Relying on the specification to generate the testgraph does not make sense in our application, because we want to use the testgraph to determine the correctness of the specification.

9. CONCLUSIONS

Specification testing can be used to verify and validate formal specifications by testing both generic and application-specific properties. While not offering the same assurance as proofs, specification testing can increase our confidence in the correctness of a specification.

In this paper we presented a framework for systematic specification testing. This framework uses mutants of specifications to analyse the precondition and postcondition of operations, as well as the state invariant, to help us check generic properties of specifications. Test execution is driven by the traversal of testgraphs: directed graphs that model a subset of the states and transitions of the specification being tested. We provide tool support to help users construct testgraphs and automate their traversal, as well as tools to generate mutants and the schemas used for checking the mutants. The framework was illustrated using a small integer set example, and we also discussed its application to a collection of student assignments and two non-trivial specifications.

Experience with the framework shows that it can be effective for testing small to medium-sized specifications. The development time of test suites is typically
significantly less than the time it takes to develop the specification that is being tested. Although test execution time can be significant and needs to be controlled, test execution can be fully automated, so that this is not a significant problem. The test suites that we developed revealed a significant number of problems in the specifications we tested, but not all checks were equally useful. In particular, we have found it difficult to test for a weak state invariant with our approach and an approach such as model-checking may be more appropriate for this. While checking for a strong precondition is easy, and this check produced a large number of warnings in the case studies, most of these were not actual errors. Depending on the specification style preferred by the specifier, it may not be worthwhile to include this check. The checks for satisfiability, weak precondition, strong state invariant and application-specific checks all appear to be very worthwhile and revealed problems in all case studies.

Although the framework and tool support were developed for testing Sum specifications, the approach can easily be applied to other model-based specification languages, such as Z, B, and VDM.

Plans for future work include:

—Continue exploring the relationship between testing a specification and testing an implementation of that specification [Miller and Strooper 2003].

—Develop appropriate measures of coverage for formal specifications to help us evaluate the adequacy of our testing.

ACKNOWLEDGMENTS
The authors would like to thank Neil Robinson from the Software Verification Research Centre, Bruno Legeard from the University of Franche-Comté (France), and the anonymous referees for their insightful comments on earlier versions of this paper.

REFERENCES


A Framework for the Systematic Testing of Model-Based Specifications


Received October 2002; revised July 2003; accepted December 2003