SOFTWARE VERIFICATION RESEARCH CENTRE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 02-35

A Framework for Systematic Specification Animation

Tim Miller Paul Strooper

September 2002

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

Note: Most SVRC technical reports are available via anonymous ftp, from svrc.it.uq.edu.au in the directory /pub/techreports. Abstracts and compressed postscript files are available via http://svrc.it.uq.edu.au

A Framework for Systematic Specification Animation

Tim Miller Paul Strooper

Abstract

Specification animation allows users to pose questions about specifications that can be answered quickly and automatically. This paper presents a framework for systematically animating specifications. Several generic properties are identified to check on specifications. A method is presented that uses variants of the specification to check these properties using an animation tool, and also uses testgraphs (directed graphs that partially model the specification being animated) to check the properties for a large number of interesting states. Tool support for all of the above is also discussed. The framework is demonstrated on a small specification and its application to two larger specifications is discussed. Experience with the framework indicates that it can be used to effectively animate small to medium-sized specifications and that it can reveal a significant number of problems in these specifications.

1 Systematic Specification Animation

Specification animation allows users to pose questions about formal specifications that can be answered quickly and automatically. While results obtained via animation are less general than results gained from techniques such as theorem proving and model checking, animation requires less expertise and can detect many types of errors in specifications. This gives specification designers a way to test that their specifications behave as intended, but is also useful for demonstrating the behaviour of the specification to end users, who typically have little-to-no knowledge of formal notations and specifications.

Much like testing, performing ad-hoc animation does not give a high-level of assurance. If we try to find errors in a specification using only a small number of cases, we have to ensure that the cases selected adequately cover the specification. Most current literature on animation describes only tools and methods for execution or interpretation of specifications, or simply mentions that animation has been used, with little or no description on how and why specific cases were selected. We aim to provide a simple, systematic framework for specification animation.

In earlier work [23], we present a method for systematically animating specifications. The process is documented using an animation plan and the spec-

ification is used to generate animation cases. This approach was completely manual, and took significant time and effort. In this paper, we define generic properties that we wish to check on specifications, and we create variants of the specification, which we call *mutants* [7], to extract specific information from the specification being animated in order to check these properties. As well as generic properties, we check *application-specific* properties. To drive the animation, we use the concept of a *testgraph* [15, 16, 24], which is a directed graph that partially models the states and transitions of a specification. Animation sequences are derived from the testgraph by traversing the testgraph repeatedly from the start node. This provides a systematic approach to animation that is partially automated and repeatable. Testing using graphs and finite state machines is not a new concept, but, to our knowledge, it has never been applied to animation. Similarly, mutants have been used to generate test cases by creating mutants of both the code being tested and the specification of the code, but not to animate specifications.

Experience so far with this framework indicates that it can be used to effectively verify small to medium-sized specifications and can reveal a significant number of problems in these specifications.

After presenting background work on the specification language and animation tool used in this work in Section 2, we discuss the generic specification properties that we check in Section 3, and describe how to check these properties in Section 4. Section 5 discusses the use of testgraphs to help simplify the process. We discuss tool support for this 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 animation tool [11, 12] used in this work.

2.1 Example - *IntSet*

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

The *IntSet* specification is written in Sum [18], 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 set the initial state of the specification; in this example, it sets *intset* to be 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 ($'$). The *init* schema and all operation schemas

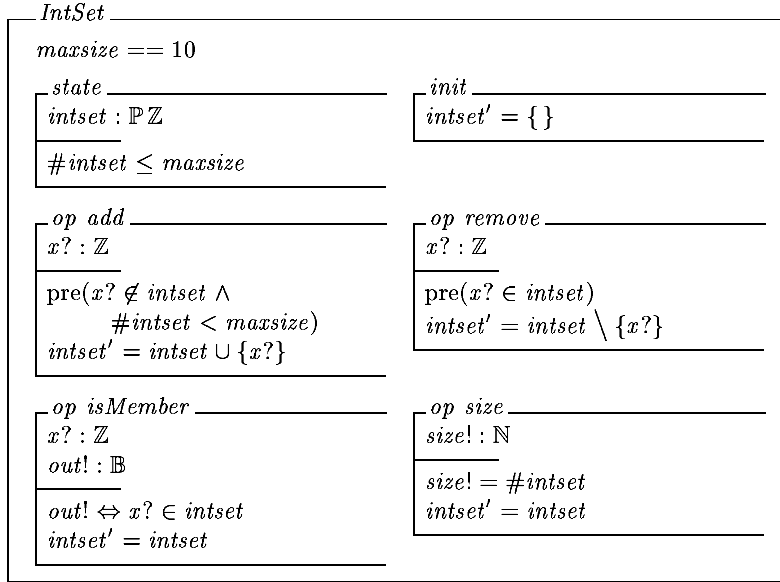

Figure 1: Sum Specification of *IntSet*

automatically include the primed and unprimed versions of the state schema in their declarations.

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 an animator 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 animated 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'*.

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

3 What Properties Do We Check?

The purpose of specification animation is to check that the specification behaves as the designer and user intend it to. We also identify several generic properties that we can check using animation. They are:

- an initialisation check;
- a schema satisfiability check;
- a precondition check; and
- a state invariant check

These properties have been selected by evaluating properties that are checked on formal specifications in development environments such as Cogito [10] and B [26], and animation methods such as the Pipedream approach [19]. In this section, we concentrate on the properties of Sum specifications, and at the end of the section we discuss how these properties would differ for other specification languages.

3.1 Definitions

Before explaining the properties in detail, we define some functions and sets used to define the properties.

The state schema. The state schema, including the state invariant, for the specification is denoted by *state*. It is assumed that *state* is accessible to all axioms defined within the specification.

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 $\mathbb{S}$.

The set of all possible inputs. The set of all inputs for the operation $op : OP$ is denoted by $\mathbb{I}_{op}$. 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 $\mathbb{O}_{op}$. As with input, we assume a single output for operations.

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

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

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

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

Now we have defined these functions, we discuss the properties we wish to check for our specification.

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:

$$\exists\, s : \mathbb{S} \bullet init(s) \land SI(s)$$

3.3 Schema Satisfiability

The next property we want to show is that for all operation schemas, there exist values for all the variables that satisfy the schema.

Formally, for each operation op in the specification M, we want to show:

$$\exists\, i : \mathbb{I}_{op};\ s, s' : \mathbb{S};\ o : \mathbb{O}_{op} \bullet \\ SI(s) \land pre(op, s, i) \land post(op, i, s, s', o) \land SI(s')$$

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 are no input/state

pairs that satisfy the precondition but for which there is no post-state or outputs.

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

$$\forall s : \mathbb{S};\ i : \mathbb{I}_{op} \bullet SI(s) \land pre(op, s, i) \Rightarrow$$
$$\quad \exists s' : \mathbb{S};\ o : \mathbb{O}_{op} \bullet SI(s') \land post(op, i, s, s', o)$$

An example of a precondition being too weak is if the precondition of the *add* operation read $pre(x? \notin intset)$, where $\#intset < maxsize$ is removed. Here, the precondition allows elements to be added even when the size is *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 can be a deliberate choice by the designer. For example, if we look at the *add* operation in *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:

$$\forall s, s' : \mathbb{S};\ i : \mathbb{I}_{op};\ o : \mathbb{O}_{op} \bullet$$
$$\quad SI(s) \land SI(s') \land post(op, i, s, s', o) \Rightarrow pre(op, i, s)$$

3.5 State Invariant Checks

3.5.1 Strong State Invariant

A Sum specification is defined as a state, with 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*, $\mathcal{R}$, for a specification as being 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, $\mathcal{R}$, for a specification as:

$$\begin{aligned}\mathcal{R} == \{ss : \operatorname{seq} \mathbb{S} \mid\ & init(head(ss)) \land \\ & (\forall j : 1..\#ss - 1 \bullet \exists op : OP;\ i : \mathbb{I}_{op};\ o : \mathbb{O}_{op} \bullet \\ & \quad pre(op, ss(j), i) \land \\ & \quad post(op, i, ss(j), ss(j+1), o)) \bullet ss(\#ss)\}\end{aligned}$$

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:

$$\forall s : \mathcal{R} \bullet SI(s)$$

While some specification designers might use a state invariant that is stronger than precondition/postcondition pairs for operations to restrict behaviour, this can be considered bad style. This is discussed in more detail in the discussion section at the end of this section.

An example of a strong state invariant would be if the state invariant for *IntSet* is $\#intset < maxsize$, where the $\leq$ is replaced by a $<$. This new state invariant prevents the size of the set from reaching *maxsize*, but there are an infinite number of sets of size *maxsize* in $\mathcal{R}$.

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.

For instance, the requirement that the size of the state variable *intset* is less than the constant *maxsize* can be expressed by having the state invariant $\#intset \leq maxsize$. 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*.

Formally, we want to check:

$$\forall s : \mathbb{S} \bullet SI(s) \Rightarrow s \in \mathcal{R}$$

3.6 Discussion

Most of the properties outlined in this section are taken from formal development environments and other animation methods.

The initialisation check is a proof obligation in the B method [26], and the Pipedream approach [19] advocates checking that an initial state exists. Schema satisfiability is a proof obligation in Cogito [10]. The weak precondition and strong state invariant are common checks made on formal specifications. A check on the preconditions is a proof obligation in both Cogito 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 will 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 [9], 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 the Z specification language [27], 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 are intended by the specifier.

3.6.1 Weak Precondition and Strong State Invariant

Although we check for both weak preconditions and strong state invariants, it is interesting to note that a state invariant that is too strong may in fact be the result of a weak precondition, and vice-versa. The example of a weak precondition in Section 3.4 illustrates this. In this case, it could also be that the state invariant is preventing a set containing more than *maxsize* elements, and the precondition is correct.

However, while there is overlap between these two, there are cases where they are exclusive. To show they are not equal in all cases, we show examples where a strong state invariant does not imply that there exists an operation with a weak precondition, and vice-versa. We then show that if all operations in a specification are deterministic, then a strong state invariant does imply that there exists an operation with a weak precondition.

3.6.2 Strong State Invariant Does Not Imply Weak Precondition

In Section 3.6.4, we will show that if all operations in the specification are deterministic operations, then a strong invariant implies there exists an operation with a weak precondition. However, if the operation that is allowing the invalid reachable state is non-deterministic, it could be that there exists an s' and o that fail to satisfy the state invariant, but there also exists an s' and o that satisfy the state invariant:

$$\begin{aligned}&\exists\, s : \mathbb{S};\ i : \mathbb{I}_{op} \bullet SI(s) \land pre(op, s, i) \land \\ &\quad (\exists\, s' : \mathbb{S};\ o : \mathbb{O}_{op} \bullet \neg\, SI(s') \land post(i, s, s', o)) \land \\ &\quad (\exists\, s' : \mathbb{S};\ o : \mathbb{O}_{op} \bullet SI(s') \land post(i, s, s', o))\end{aligned}$$

An example of such an operation is a non-deterministic *add* operation for *IntSet* called add_{nondet}. This new operation takes an integer $x?$ as input, and non-deterministically adds $x?$ to the set or leaves the set unchanged. The precondition states that if $x?$ is not already in the set, then the size of the set must be less than *maxsize*.

$$\begin{array}{|l}
\text{op } add_{nondet} \\
x? : \mathbb{Z} \\
\hline
pre(x? \notin intset \Rightarrow \# intset < maxsize)) \\
intset' = intset \cup \{x?\} \lor intset' = intset
\end{array}$$

We now replace the state invariant with one that is too strong, e.g. $\# intset < maxsize$, where the $\leq$ is replaced by $<$. Our state invariant is now too strong because there exists a reachable state such that the state invariant is violated, i.e. any integer set of size *maxsize*.

Even though the state invariant is too strong, for all inputs there exists a post-state that satisfies the invariant, because if the invariant is satisfied for the pre-state, then the post-state where the set is not changed ($intset' = intset$) must also satisfy the state invariant. Therefore, we have a strong state invariant, but not a weak precondition.

3.6.3 Weak Precondition Does Not Imply Strong State Invariant

Similarly, a weak precondition does not imply a strong state invariant. To show this, we introduce an operation into the *IntSet* specification called *remove_min*. This operation returns the minimum integer $x!$ in *intset*, and removes it from the set. The size of *intset* must be greater than 0, otherwise there does not exist an $x!$ in *intset*.

$$\begin{array}{|l}
\text{op } remove_min \\
x! : \mathbb{Z} \\
\hline
pre(\# intset > 0) \\
x! = min(intset) \\
intset' = intset \setminus \{x!\}
\end{array}$$

If we change our precondition to make it too weak, by removing the constraint $\# intset > 0$, this leaves the precondition as true. The *IntSet* specification's set of reachable states, $\mathcal{R}$, contains all sets of integers with a size less than or equal to *maxsize*. None of these violate the state invariant. However, when *intset* is empty, there is no minimum value, so the operation has a precondition that is too weak. Therefore, we have a weak precondition, but not a strong state invariant.

3.6.4 For Deterministic Specifications, A Strong State Invariant Implies A Weak Precondition

We now show that if all operations in the specification are deterministic then a strong state invariant implies that there will be at least one operation in the specification with a weak precondition.

A strong state invariant is defined by:

$$SSI \Leftrightarrow (\exists\, \hat{s} : \mathcal{R} \bullet \neg\, SI(\hat{s}))$$

If all operations are deterministic, we know that there exists at most one postcondition for each precondition. Formally:

$$\begin{aligned} DET \Leftrightarrow & \forall\, op : OP;\ s : \mathbb{S};\ i : \mathbb{I}_{op} \bullet SI(s) \land pre(op, s, i) \Rightarrow \\ & (\exists_1\, s' : \mathbb{S};\ o : \mathbb{O}_{op} \bullet post(op, i, s, s', o)) \lor \\ & \neg\, (\exists\, s' : \mathbb{S};\ o : \mathbb{O}_{op} \bullet post(op, i, s, s', o)) \end{aligned}$$

Our weak precondition is defined:

$$\begin{aligned} WPRE \Leftrightarrow & (\exists\, op : OP;\ s : \mathbb{S};\ i : \mathbb{I}_{op} \bullet SI(s) \land pre(op, s, i) \land \\ & \forall\, s' : \mathbb{S};\ o : \mathbb{O}_{op} \bullet \neg\, (SI(s') \land post(op, i, s, s', o))) \end{aligned}$$

We also assume that for all possible initial states, the state invariant holds:

$$INIT \Leftrightarrow (\forall\, s : \mathbb{S} \bullet init(s) \Rightarrow SI(s))$$

Proof 1 *Our proof obligation is:*

$$DET \land SSI \land INIT \Rightarrow WPRE$$

Let ss be the shortest possible sequence of operation calls that end up in a state that violates the state invariant. There may be more than one such ss, but simply let ss be any of these.

Because ss is the shortest possible path, we know that $\forall\, j : 0 \,..\, \#ss - 1 \bullet SI(ss(j))$.

In ss, we know that up until $\#ss - 1$, the state invariant is maintained, because ss is the shortest possible sequence of states that hold. Since we assume that all initial states satisfy the state invariant, we know that $\#ss > 1$. Therefore, before the transition from $ss(\#ss - 1)$ to $ss(\#ss)$, we know that the state invariant holds, i.e. $SI(ss(\#ss - 1))$. We also know that the precondition holds for the transition, but that either:

1. there is no post-state and output that satisfy the postcondition; or
2. there is such a post-state and output, but the post-state does not satisfy the state invariant.

In case 1, clearly the last operation has a weak precondition. In case 2, we know there is only one post-state and output, but since this does not satisfy the state invariant we again have a weak precondition. Therefore, for a deterministic specification, a strong state invariant implies that there exists an operation with a weak precondition. □

4 Checking of Properties

In this section, we discuss how to check the properties from Section 3 using an animation 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. As discussed in Section 6, we can automatically generate the mutants and most of the schemas used for checking the specification using tool support.

4.1 Creating Mutants

Mutation analysis [7], in the traditional sense, is 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 only when all non-equivalent mutants are killed does the testing stop. 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 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 animated. These mutants will 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 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 im-

$$
\begin{array}{l}
\textbf{init} \\
\hline
tgf_report! : \mathbb{F}\{SSI_INIT, INIT_FAIL\} \\
\hline
if\ (\exists\, M.state' \bullet M.init \land changes_only\{M.state\})\ then \\
\quad M.init \land changes_only\{M.state\} \land tgf_report! = \{\} \\
else \\
\quad if\ (\exists\, PRPO.state' \bullet PRPO.init)\ then \\
\quad\quad tgf_report! = \{SSI_INIT\} \land changes_only\{\} \\
\quad else \\
\quad\quad tgf_report! = \{INIT_FAIL\} \land changes_only\{\} \\
\quad fi \\
fi
\end{array}
$$

Figure 2: Schema *top.init* for *IntSet*

port renaming method, as follows: *PR* for the specification containing only preconditions, *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 Sum's module prefixing notation to access the items in each mutant, e.g. $M.add$ for the *add* operation in M.

The state schema of the *top* specification includes only the state schema of M.

4.2 Checking the Properties

Once the specification has proceeded to a new state, we want to check the properties outlined in Section 3. We also want to check application-specific properties, e.g., that after adding the integer 1 to an empty set, the *size* operation returns $size = 1$, and *intset* is equal to $\{1\}$. Except for the initialisation check, all properties are checked for all state values visited during the animation.

4.2.1 Initialisation Check

The first property to check is that an initial state exists. To do this, we define an *init* schema in the *top* specification that checks for an initial state, reporting an error otherwise. Figure 2 shows the schema *top.init* for the *IntSet* specification, which should be the same for any other Sum specification.

In this figure, we first check whether an initial state exists for the specification M. If so, we move to that state and report no error. If not, we then check whether the *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 *init* failed.

$$
\begin{array}{|l}
\hline
\textit{check_add_pre} \\
\textit{state} \\
\textit{tgf_report}! : \mathbb{F}\{\textit{WPRE_add}, \textit{SPRE_add}\} \\
\hline
\exists\, \textit{pr_add}, \textit{posi_add} : \mathbb{F}\,\mathbb{Z} \mid \\
\quad \textit{pr_add} = \{PR.\textit{add}\{M.\textit{intset}/\textit{intset}\} \bullet PR.x?\} \land \\
\quad \textit{posi_add} = \{POSI.\textit{add}\{M.\textit{intset}/\textit{intset}\} \bullet POSI.x?\} \bullet \\
\quad (\exists\, x : \textit{pr_add} \bullet x \notin \textit{posi_add}) \Leftrightarrow \textit{WPRE_add} \in \textit{tgf_report}! \land \\
\quad (\exists\, x : \textit{posi_add} \bullet x \notin \textit{pr_add}) \Leftrightarrow \textit{SPRE_add} \in \textit{tgf_report}! \\
\hline
\end{array}
$$

Figure 3: Schema *check_add_pre* for *IntSet*

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 generate all possible inputs that satisfy *PR.add* and *POSI.add*, and use *pr_add* and *posi_add* to "hold" the values for them. Note that 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 that the size of *intset* must be less than *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 ($pr_add = \mathbb{Z} \setminus intset$), while the *POSI* version will allow us to add none ($posi_add = \varnothing$). Therefore, the weak precondition error message is added to the set of messages.

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* ($pr_add = \mathbb{Z} \setminus intset$), and *posi_add* will contain all integers ($posi_add = \mathbb{Z}$), because if an

$$
\begin{array}{l}
\textit{check_add_si} \\
\hline
\textit{state} \\
\textit{tgf_report}! : \mathbb{F}\{\textit{SSI_add}\} \\
\hline
\exists \textit{prpo_add}, \textit{m_add} : \mathbb{F}(\mathbb{Z} \times \mathbb{P}\,\mathbb{Z}) \mid \\
\quad \textit{prpo_add} = \{\textit{PRPO.add}\{\textit{M.intset}/\textit{intset}\} \bullet (\textit{PRPO.x}?, \textit{PRPO.intset}')\} \land \\
\quad \textit{m_add} = \{\textit{M.add}\{\textit{M.intset}/\textit{intset}\} \bullet (\textit{M.x}?, \textit{M.intset}')\} \bullet \\
\quad (\exists x : \textit{prpo_add} \bullet x \notin \textit{m_add}) \Leftrightarrow \textit{SSI_add} \in \textit{tgf_report}!
\end{array}
$$

Figure 4: Schema *check_add_si* for *IntSet*

element already in a set gets added again, the set does not change. Therefore, the strong precondition warning is added to the set of messages.

The schema in Figure 3 examines every possible input trying to find one that reveals a weak precondition. The input to the operation is an integer, therefore, the input space is infinite. However, Possum has a maximum integer size, which can be changed by the user. We set the maximum integer size to 32, allowing only the values in the set $-32..32$ to be added to the set.

In a specification where an input to an operation is of a given set, we assign the given set a value, e.g. $\mathbb{N}$. If the input space is infinite, e.g of type *string*, we select a subset of the input space explicitly by redefining our operations. Although Sum restricts any identifier to be defined more than once, Possum allows, for easy interaction, an operation to be redefined using its previous definition. Therefore, we can redefine the input space of the operation *op*, with the input $s?$ of type string:

$$op \mathrel{\widehat{=}} [\, op \mid s? \in STR \,]$$

where STR is a predefined set of strings. For all operations in the specification that use strings, we use only STR for the input space.

4.2.3 State Invariant Check

Strong State Invariant To check that the state invariant of the specification is not too strong, we try to find an input or 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 generate all possible input/post-state pairs for $PR.add$ and $POSI.add$, and use *prpo_add* and *m_add* to "hold" the values for them. 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 or 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, we know it must be the state invariant restricting M's progress, so we add the *SSI_add* error message to the $tgf_report!$.

To demonstrate this approach, we use the example from Section 3, where 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 except those already in the set, along with the postcondition for *add* with respect to the integer, because the operation without the state invariant will allow more elements to be added. *m_add* will be empty, because the state invariant is preventing elements from being added. Therefore, we know the state invariant is too strong, and add the *SSI_add* error message to the set of messages.

As with the precondition check, output variables can be used to give feedback on which input/post-state pairs are violating the state invariant.

Weak State Invariant Detecting a weak state invariant using an animator 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 infeasible 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 out 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, our search space for the *IntSet* example might be:

$$search_space == \{s : M.state \bullet s.intset\}$$

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:

$$search_space == \{s : \mathbb{P}\,\mathbb{Z};\ d : 0 \mathrel{..} M.maxsize \mid s = 1 \mathrel{..} d \bullet s\}$$

which restricts our search space to only $maxsize + 1$ sets containing $1 \mathrel{..} d$ 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 in Section 7) and the method did not scale to even that small example due to resource limitations.

$$
\begin{array}{|l}
\hline
\textit{CHECK_ONE} \\
\textit{state} \\
\textit{tgf_report}! : \mathbb{F}\, \textit{SPEC_MSG} \\
\hline
M.\textit{isMember}\{1/x?, \text{true}/\textit{out}!\} \Leftrightarrow \textit{ISMEM_TRUE_ERR} \notin \textit{tgf_report}! \\
M.\textit{isMember}\{2/x?, \text{false}/\textit{out}!\} \Leftrightarrow \textit{ISMEM_FALSE_ERR} \notin \textit{tgf_report}! \\
M.\textit{size}\{1/\textit{size}!, \textit{intset}/\textit{intset}'\} \Leftrightarrow \textit{SIZE_ERR} \notin \textit{tgf_report}! \\
M.\textit{intset} = \{1\} \Leftrightarrow \textit{STATE_ERR} \notin \textit{tgf_report}! \\
\hline
\end{array}
$$

Figure 5: Schema *CHECK_ONE* for *IntSet*

Legeard et al. [20] perform a similar search on B specifications using a constraint solver. 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.

4.2.4 Checking Application-Specific Properties

There are also other properties that we may want to check that are specific to the specification being animated. For example, the *CHECK_ONE* schema, shown in Figure 5, checks the *isMember* operation for a true and a false case, the *size* operation, and the current state. Like the checks for generic properties, this schema is checked for every special state value.

4.2.5 Schema Satisfiability

The satisfiability of schemas does not need to be explicitly checked, because if the application-specific properties are checked 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 our generic-check schemas 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.

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,

```
tgf_report! =
  {"Error: precondition for operation 'add' is too weak!",
   "Warning: precondition for operation 'add' is stronger than necessary"}
```

Figure 6: Extract from Report Generated for *IntSet*

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 can also have output variables in our check schemas that give us more detailed feedback on errors found. For example, *check_add_pre* schema in Figure 3, we can declare an output variable $weak_add! : \mathbb{F}\,\mathbb{Z}$ and set its value to be the set of inputs that satisfy the *PR* version of the schema, but not the *POSI* version: $weak_add! = pr_add \setminus posi_add$. This gives us feedback as to which values are satisfying 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, we consider the *add* operation in *IntSet*. If we were to remove the part of the precondition that stated the size of the set had to be less than *maxsize*, 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 method will detect both problems and display both error messages.

Figure 6 shows a sample output from *add_check_pre* where the part of the precondition enforces a maximum size of a set has been removed from the *add* operation.

4.4 Discussion

In this section, we discuss our approach to checking the properties outlined in Section 3. These properties 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 to the other checks. This check requires us to search out the state space to find a select set of states. This is similar to *model checking*.

Model checking [3] 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. This is where there are a large or infinite number of states, and checking every state becomes infeasible.

Our method combines testing, animation, and model checking, to avoid the

problem of state explosion by selecting only a small subset of the states to be checked. However, for the weak state invariant check, this problem arises again, because we are searching out a large set of states. 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 and Traversing Animation States

In this section, we discuss using testgraphs to define and traverse a number of interesting states during animation. We use testgraphs because they are straightforward to derive, and deriving the animation from testgraphs can be done quickly and automatically. Testgraphs give us a systematic approach to animation that allows us to analyse the specification as a whole instead of animating each of the operations in isolation.

5.1 Deriving a Testgraph

A *testgraph* is a directed graph that partially models the states and transitions of the specification being animated. Each node in the testgraph represents a possible state that the specification can reach, and each arc represents a transition (a sequences of calls to operations) that moves the specification from one state to another. 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. However, using an animator, it is infeasible to check the entire state space of specifications, except for specifications with very small state spaces. If we look at the *IntSet* example, the size of the state space is infinite, and this is a small specification by industry standards. Therefore, we select a subset of the state space for animation.

The state and operations of a specification provide important information about the selection of states to animate. For example, the *add* operation in the *IntSet* specification 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 animate 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 $\{1\}$. Each of these instantiations becomes a node in our testgraph.

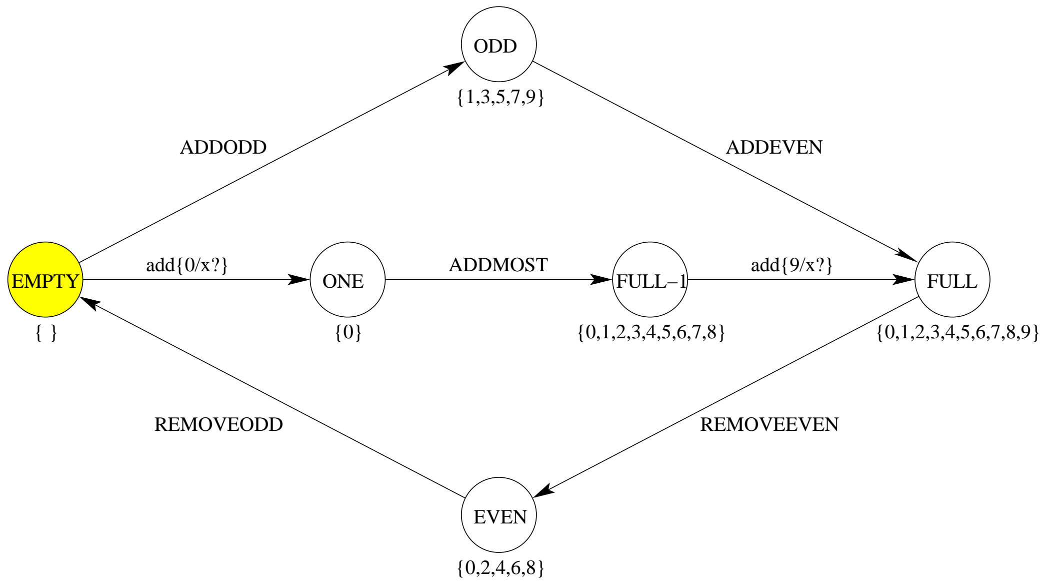

Figure 7: Testgraph for *IntSet*

Once we have our testgraph nodes, we derive arcs for the testgraph to be used as transitions during animation. We require each node, except the start node, to have at least one arc leading in to it, otherwise the node will be unreachable. In addition to making sure each node is reachable, we also add arcs if there are any for any specific transitions we want to animate.

Figure 7 shows the testgraph for the *IntSet* specification. Here, we have six nodes representing the states derived 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

We specify operations for our specification that make the transitions defined in the testgraph using the arc labels as the name for the operations. To step from one state to another, we use the schemas defined in the original specification. We create a new schema in the *top* specification that checks the operations during a transition. This new schema uses the four mutants to either step M to the new state, or to report a failure if the transition does not succeed, and possibly why this failure occurred. The *add* operation for the *top* specification is shown in Figure 8. Note that this new *add* schema is not the same as the *add* schema defined in Figure 1, because it is defined within the *top* module,

$$
\begin{array}{l}
\textit{op add} \\
\hline
x? : \mathbb{Z} \\
tgf_report! : \mathbb{F}\{SSI_add, WPRE_add, SPRE_add, add_FAIL\} \\
\hline
\textit{if } (\exists M.state' \bullet M.add\{x?/x?\} \land changes_only\{M.state\}) \textit{ then} \\
\quad M.add\{x?/x?\} \land changes_only\{M.state\} \land tgf_report! = \{\} \\
\textit{else} \\
\quad \textit{if } (\exists PRPO.state' \bullet PRPO.add\{M.intset/intset, x?/x?\}) \textit{ then} \\
\quad\quad tgf_report! = \{SSI_add\} \land changes_only\{\} \\
\quad \textit{else} \\
\quad\quad \textit{if } (\exists PR.state' \bullet PR.add\{M.intset/intset, x?/x?\})) \textit{ then} \\
\quad\quad\quad tgf_report! = \{WPRE_add\} \land changes_only\{\} \\
\quad\quad \textit{else} \\
\quad\quad\quad \textit{if } (\exists POSI.state' \bullet POSI.add\{M.intset/intset, x?/x?\})) \textit{ then} \\
\quad\quad\quad\quad tgf_report! = \{SPRE_add\} \land changes_only\{\} \\
\quad\quad\quad \textit{else} \\
\quad\quad\quad\quad tgf_report! = \{add_FAIL\} \land changes_only\{\} \\
\quad\quad\quad \textit{fi} \\
\quad\quad \textit{fi} \\
\quad \textit{fi} \\
\textit{fi}
\end{array}
$$

Figure 8: *add* schema for the *top* specification

whereas the first definition is in the *IntSet* module.

In this schema, we first check whether M can proceed to the next state. This is the normal case, so if it can, we advance to that state, and report nothing. However, if this fails, we check whether the *PRPO* version can advance to this state. If it succeeds, then the state invariant must be restricting M's progression, because the only difference between the two specifications is the removed state invariant, so we set *tgf_report!*[1] to a set containing *SSI_add*, which is defined as *"Error: strong state invariant prevented transition add"*. If the transition fails for the *PRPO*, then we try to make the transition for *PR*. If this succeeds, then the postcondition must be stronger than expected, because the only difference between the *PRPO* and *PR* specifications is the removed postcondition, so we raise the error *WPRE_add*, which means the operation has a weak precondition. If the transition fails for *PR*, then we check it for *POSI*. If this succeeds, then the precondition must be stronger than expected, so we raise the warning message *SPRE_add*, which means the operation's precondition is stronger than necessary. If this fails, we report that the transition has failed using *add_FAIL*. Note that in this schema, the calls made to *add* for the mutant specifications have all references to the state variable *intset* substituted with the current value

[1] We add the prefix *tgf_*, which stands for *testgraph*, to all visible variables in the *top* specification to prevent name clashes.

of *M.intset*. This is because the mutant specifications do not have a current state; we are just using their definitions to check certain properties.

We generate our animation sequences by traversing the testgraph to achieve *arc* coverage. The other two types of coverage considered were *node* and *path* coverage. Node coverage does not force the traversal of every transition in a graph, and path coverage is infeasible for graphs with a cycle. Arc coverage traverses every arc, visits every node (provided all testgraph nodes and arcs are reachable from the start node), and is straightforward to achieve.

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. Therefore, we use the schemas defined in Section 4, such as *check_add_pre* in Figure 3 and *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 *IntSet*

The only error found in the *IntSet* specification was a weak precondition for the *add* operation. There was no property in the precondition forcing the size of the set to be less than *maxsize* when adding an element. This 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 *add* was overlooked, with only the state invariant being updated.

In addition, two warning messages are generated using our method. Both the *add* and *remove* operations have strong preconditions. The *add* operation will not allow an element to be added if already in the set, and the *remove* operation will 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 would not be added/removed.

6 Tool Support

Applying the method outlined in Sections 4 and 5 is 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. A more detailed

description of this can be found in [24]. The second automatically generates the mutant and *top* specifications.

6.1 The Testgraph Framework

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

The user can add nodes to the graph, and add a directed arc between any two nodes, provided there is not already an arc with the same source and destination nodes. An arc can be removed from between two nodes, and nodes can be removed. Removing a node also removes any arcs that have that node as the source or destination node. Users can also select one node to be the start node of the testgraph. Users can associate a schema or sequence of schemas with a node. The schemas are invoked during the traversal of the testgraph when the user wishes 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 there are no more nodes leading from the current node. If there are unreachable nodes or arcs, the path generation algorithm ignores these. 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 different ways: using manual traversal, where the user selects the next arc to traverse; stepping through the testgraph, where the user tells the framework to traverse the next arc in the automatically generated paths; or automatic traversal, where no interaction is required and the framework traverses the paths automatically.

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 animation, 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 where they occurred.

Some advanced features of the framework include *memoisation*, which is where once a node is visited the checks are not performed on subsequent visits. The only check is that the value of the states are the same. This reduces execution time. The framework also allows *regression animation*, which allows nodes to be associated with state values, and these values are saved when the testgraph is saved. On subsequent traversals, the states can be compared to their respective values at each node. Both of these options can be turned on/off.

6.2 Generating the Mutant and *top* Specifications

The most error-prone part of our method is generating the *top* specification. From experience, we found that generating the schemas that perform checking of the state invariant and preconditions is tedious and error-prone and that these problems occurred regularly when dealing with even the smallest of specifications.

This prompted us to write tool support to generate these specifications. 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 almost all of the top specification. All error messages, e.g. $WPRE_add$, are defined, the *top* state and init schemas, and all schemas for checking the preconditions and state invariant. On top of this, it will read the arc labels from the testgraph and define the transition schemas (e.g. Figure 8), and create skeletons for all schemas associated with nodes that are not precondition or state invariant checks.

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 takes in a Sum specification, and creates three new files. One file contains the PR version of the specification, one the $POSI$ version, and one the $PRPO$ version.

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 subject as the University of Queensland. The other two case studies are more substantial specifications than the previous specifications we have used our framework with.

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 method 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 asked to write a Sum specification of a graph.

7.1.1 Description

The graph specification takes, as a parameter, the set $DATA$. The state of the 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

Message	Manual Analysis	Automated Analysis
Initialisation	0	0
Non-satisfiable schema	2	2
Weak Precondition	28	28
Strong Precondition	120	120
Strong State Invariant	5	5
Weak State Invariant	23	0
Loop in Path	8	8
"Out of Bounds" Error	4	0
Other	2	2
Total	192	165

Table 1: Summary of Reports for *Graph* Specifications

node must be in the set *nodes*, and that there are no edges 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 signatures 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 animate 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 method 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 method, e.g., the files containing the testgraph and automatically generated *top* specification. The testgraph and *top* specification were written before the hand-analysis was performed, so as to not influence the way they were approached, and the same testgraph and *top* specification were used for all assignments. We ran the testgraph framework over the students' submissions using automatic testgraph traversal, and saved the resulting report.

7.1.2 Results

The results of applying the method are promising. 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.

Table 1 gives 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.

From this table, we can see that the students specified a large amount errors that we consider generic properties. This shows clearly that these checks are worth performing. The most common occurrences are strong preconditions,

which is not surprising considering these are considered quite normal in specifications, and are not 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:

$$\forall i : 1 \mathrel{..} (\#s!) \bullet (s!(i), s!(i+1)) \in \mathit{edges}$$

instead of

$$\forall i : 1 \mathrel{..} (\#s! - 1) \bullet (s!(i), s!(i+1)) \in \mathit{edges}$$

where 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 *FindPath* operation would find a path of length one when given the same source and destination nodes. The requirement that there are no edges from a node to itself meant that this should not happen.

7.2 Mass Transit Railway

The MTR specification is taken from [23] and this version of the specification was manually translated from [9]. 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.

There are four modules in the MTR: 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.

Message	**MTR**	**TrackCAD**
Initialisation	0	0
Non-satisfiable schema	1	1
Weak Precondition	1	2
Strong Precondition	14	11
Strong State Invariant	0	0
Other	4	11
Total	20	25

Table 2: Summary of Report for *MTR* and *TrackCAD* Specifications

7.2.2 Results

Previously [23] we performed animation on the Mass Transit Railway specification, uncovering five errors in total. Our method found not only these five errors, but one extra error: a weak precondition. This extra error could not be uncovered using the approach in [23], because the restrictions missing in the precondition were present in the postcondition, therefore, using only the original specification and no mutants, the behaviour is the same. This is an error because the precondition is no longer preserving the state invariant. Other errors included: allowing state variables to change when they were not supposed to, swapping operands of domain restrictions and subtractions, and a non-satisfiable schema. Table 2 shows the number of errors found in each category.

7.3 TrackCAD

TrackCAD [22] is part of a joint project between Queensland Rail and the Software Verification Research Centre. 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 towards 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.

There are two modules for TrackCAD, but these modules are not used together. One module is used to model the input to the program, similar to the graphical user interface, while the other is used to model the output of the program. These two specifications contain similar operations and states, and there is a mapping between the output module's state and most of the input module's state, although there is extra information in the input that is not modelled by the output module. The output module models similar information at a more abstract level than the input specification.

The input specification contains 13 state variables, 14 operations, 16 axioms, and 378 lines of Sum, and the output specification contains 9 state variables, 8 operations, 2 axioms and 170 lines of Sum. While there are not many operations in either of these, the schemas are considerably more complex than any other specifications we animated.

7.3.2 Results

Our method was easy to apply to the TrackCAD specifications, and we discovered a significant number of errors in both. Table 2 gives a breakdown of the errors and warnings generated using our framework.

The errors 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, and similar to the weak precondition in the *Mass Transit Railway* case study, the weak precondition could not be uncovered using the method in [23].

8 Related Work

In this section, we present related work on animation, mutation analysis of specifications, and testing using finite state machines (FSMs) and graphs.

8.1 Animation

There are several animation tools that automatically execute or interpret specifications. The animation tool we use in this work is Possum [11, 12], which animates Z and other Z-like languages. PiZA [14] is another animator 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 [30], which stands for *Just*

Another Z Animator, is a Z animator that appears to be more advanced in its execution than other animators, because it uses different data structures for representing different types of sets. The B-Model animator [31] is the animator used in the B formal development process [29]. It is used to animate specifications written in B's model-oriented specification language. The Software Cost Reduction (SCR) toolset [13] contains an animator that is used to test specifications. The IFAD VDM++ Toolbox [17], 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. [19] are one of only a few groups to discuss how they perform animation. They use a tool called Pipedream. Pipedream transforms the specification into first-order logic to determine predicates and finite sets, which help Pipedream establish which specifications are executable. Kazmierczak et al. outline an approach for specification animation using Pipedream containing three steps: performing an initialisation check; verifying the preconditions of schemas; and performing a simple reachability property.

Lui [21] also talks about how to use animation tools. Lui 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 animators such as Possum, PiZA, and Jaza.

8.2 Mutation Analysis

Ammann et al. [1, 2] discuss the application of mutation analysis [7] 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. [4] 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

There is a lot of literature that discusses testing using testgraphs and finite state machines (*FSMs*). In this section we discuss only the ones that are directly relevant to our use of FSMs for specification animation.

Dick and Faivre [8] were the first to generate FSMs from formal specification. 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.

There are a number of other approaches that extend on Dick and Faivre's work.

Hoffman and Strooper's approach [15, 16] is the most related to ours. 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. [25] and Carrington et al. [5] describe generating Classbench testgraphs from FSMs. States for these FSMs are derived by using the *Test Template Framework* [6, 28] 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 animation, but rather than derive testgraphs from FSMs, the user generates the testgraph manually. Relying on the specification to generate the testgraph does not make as much sense in our application, because we want to use the testgraph to determine the correctness of the specification.

9 Conclusions

Specification animation can be used to check properties and the behaviour of specifications. While not offering the same assurance as proofs, animation can increase our confidence in the correctness of a specification.

In this paper we presented a framework for systematic specification animation. 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. The animation is driven using testgraphs: directed graphs that model a subset of the states and transitions of the specification being animated. Sequences for animation are derived by traversing the testgraph. We presented tool support to help users construct testgraphs and automate their traversal, as well as tools to generate mutants and to generate the specification used for checking the mutants. This framework was explained using a small integer set example, and we also discussed the application of this method on a collection of student assignments and two non-trivial specifications. The results from these case studies are promising, because they took little effort and time, and uncovered several significant problems in all case studies.

Even though we only apply this framework to Sum specifications, we believe that the approach could be applied to other model-based specification languages, such as Z, B, and VDM.

Plans for future work in this area are:

- Explore the relationship between animating a specification and testing an implementation of that specification.

- Develop a way to measure the coverage of a specification by an animation script.

References

[1] P. Ammann and P. Black. A specification-based coverage metric to evaluate test suites. In *4th IEEE Internationla High-Assurnce Systems Engingeering Symposium*, pages 239–248, 1999.

[2] P. Ammann, P. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proceedings of the Second IEEE International Conference on Formal Engineering Methods*, pages 46–54, 1998.

[3] J. M. Atlee and J. D. Gannon. State-based model checking of event-driven systems requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, 1993.

[4] J. Callahan, E. S., and T. Montgomery. Generating test oracles via model checking. TR NASA-IVV-98-015, NASA / West Virginia University Software Research Laboratory, 1998.

[5] D. Carrington, I. MacColl, J. McDonald, L. Murray, and P. Strooper. From Object-Z specifications to Classbench test suites. *Journal on Software Testing, Verification and Reliability*, 10(2):111–137, 2000.

[6] D. Carrington and P. Stocks. A tale of two paradigms: Formal methods and software testing. In J.P.Bowen and J. Hall, editors, *ZUM'94, Z User Workshop*, pages 51–68, 1994.

[7] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, 1978.

[8] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Formal Methods Europe (FME'93)*, pages 268–284, 1993.

[9] R. Duke and G. Rose. *Formal Object-Oriented Specification Using Object-Z*. MacMillan Press Limited, London, 2000.

[10] N. Hamilton, D. Hazel, P. Kearney, O. Traynor, and L. Wildman. A complete formal development using Cogito. In *Computer Science '98: Proceedings of the 21st Australasian Computer Science Conference*, pages 319–330, 1998.

[11] D. Hazel, P. Strooper, and O. Traynor. Possum: An animator for the Sum specification language. In *Proceedings of the Asia-Pacific Software Engineering Conference and International Computer Science Conference*, pages 42–51. IEEE Computer Society, 1997.

[12] D. Hazel, P. Strooper, and O. Traynor. Requirements engineering and verification using specification animation. In *Proceedings of the 13th IEEE International Conference on Automated Software Engineering*, pages 302–305. IEEE Computer Society, 1998.

[13] C. Heitmeyer, J. Kirby, B. Labaw, and R. Bharadwaj. SCR*: A toolset for specifying and analysing software requirements. In *Computer-Aided Verification 10th Annual Conference*, pages 526–531, 1998.

[14] M. Hewitt, C. O'Halloran, and C. Sennett. Experiences with PiZA, an animator for Z. In *ZUM'97: The Z Formal Specification Notation*, volume 1212 of *LNCS*, pages 37–51, 1996.

[15] D. M. Hoffman and P. A. Strooper. The testgraphs methodology — automated testing of collection classes. *Journal of Object-Oriented Programming*, pages 35–41, 1995.

[16] D. M. Hoffman and P. A. Strooper. ClassBench: A methodology and framework for automated class testing. In D. C. Kung, P. Hsia, and J. Gao, editors, *Testing Object-Oriented Software*, pages 152–176. IEEE Computer Society, 1998.

[17] IFAD. Features of VDM tools. http://www.ifad.dk/products/vdmtools/features.htm.

[18] E. Kazmierczak, P. Kearney, O. Traynor, and L. Wang. A modular extension to Z for specification, reasoning and refinement. TR 95-15, SVRC, The University of Queensland, Australia, Feb. 1995.

[19] E. Kazmierczak, M. Winikoff, and P. Dart. Verifying model oriented specifications through animation. In *Proceedings of the Asia-Pacific Software Engineering Conference*, pages 254–261, 1998.

[20] B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from Z and B. In *Formal Methods Europe, 2002*, pages 21–40, Copenhagen, Denmark, 2002.

[21] S. Lui. Verifying consistency and validity of formal specifications by testing. In *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems*, pages 896–914, 1999.

[22] T. McComb and N. Robinson. Assuring graphical computer aided design tools. TR 02-18, SVRC, The University of Queensland, Australia, May 2002.

[23] T. Miller and P. Strooper. Animation can show only the presence of errors, never their absence. In *Proceedings of the Australian Software Engineering Conference (ASWEC 2001)*, pages 76–85. Australian Computer Society, 2001.

[24] T. Miller and P. Strooper. Model-based specification animation using testgraphs. In *Proceedings of the 4th International Conference on Formal Engineering Methods (To appear)*, 2002.

[25] L. Murray, D. Carrington, I. MacColl, J. McDonald, and P. Strooper. Formal derivation of finite state machines for class testing. In J. B. Hinchey, A. Fett, and M.G., editors, *ZUM'98: The Z Formal Specification Notation*, volume 1493 of *Lecture Notes in Computer Science*, pages 42–59. Springer Verlag, 1998.

[26] S. Schneider. *The B-Method: An Introduction.* Cornerstones of Computing. Palgrave, 2001.

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

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

[29] H. Treharne, B. Ormsby, J. Draper, and T. Boyce. Evaluating the B-Method on an avionics example. In *Proceedings of the DASIA Conference*, pages 89–97, 1996.

[30] M. Utting. Data structures for Z testing tools. In *Proceedings of FM-TOOLS 2000*, Germany, 2000.

[31] H. Waeselynck and S. Behnia. B-Model animation for external verification. In *Proceedings of the International Conference for Formal Engineering Methods*, pages 36–45. IEEE Computer Society, 1998.