Model-Based Specification Animation Using Testgraphs

Tim Miller and Paul Strooper

School of Information Technology and Electrical Engineering,
Software Verification Research Centre,
The University of Queensland,
Brisbane, Qld 4072, Australia
timothym@svrc.uq.edu.au
pstroop@itee.uq.edu.au

Abstract. This paper presents a framework for systematically animating specifications using *testgraphs*: directed graphs that partially model the specification being animated. Sequences for the animation are derived by traversing the testgraph. The framework provides a testgraph editor that allows users to edit testgraphs and supports automated testgraph traversal. Experience with the framework so far 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 Introduction

Specification animation allows users to pose questions about the specification that can be answered quickly and automatically. While results obtained via animation are specific to certain cases, as opposed to 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, and, as a result, cannot determine the behaviour of a specification using manual analysis.

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.

In earlier work [13] we present a method for systematically animating specifications. We document the process using an animation plan, and use the specification to generate animation inputs. This approach was completely manual, and took significant time and effort. In this paper, we use the idea of animation using a *testgraph* [9]: a directed graph that partially models the possible states and transitions of a specification. Sequences are derived from the testgraph by traversing the testgraph repeatedly from the start node and cases for animation are generated from these sequences. This provides a systematic

© Springer-Verlag Berlin Heidelberg 2002

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

After reviewing related work in Section 2, we discuss background on the specification language and animation tool used in this work. We then present a method for animation using testgraphs in Section 4, and a framework with tool support for this method in Section 5. We then conclude the paper.

2 Related Work

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

2.1 Animation

There are several animation tools that automatically execute or interpret specifications. PiZA [8] is an 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. The B-Model animator [18] is the animator used in the B formal development process [17]. It is used to animate specifications written in B's model-oriented specification language. The Software Cost Reduction (SCR) toolset [7] contains an animator that is used to test specifications. The IFAD VDM++ Toolbox [10], 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.

Pipedream [12] is another animator for the Z specification language. 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. [12] 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.

2.2 Testing Using Graphs and Finite State Machines

Hoffman and Strooper [9] 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*. Later work by Murray et al. [15] and Carrington et al. [3] describe generating Classbench testgraphs from FSMs. States for these FSMs are derived by using the *Test Template Framework* [16] 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 testgraph. We build on this work by using testgraphs to sequence animation, but rather than derive testgraphs from FSMs, the user can simply generate the testgraph manually. Relying on the specification do 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.

Dick and Faivre [4] also generate test sequences 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 FSA is then traversed, with every branch executed at least once.

Bosman and Schmidt [1] use FSMs to test object-oriented programs. Two state machines are developed. One is the state machine for the specification, called the *design FSM*, and the other is the state machine for the implementation, called the *representation FSM*. If two state machines behave identical for all possible input sequences, then they are considered identical.

Callahan et al. [2] have also used model checking to 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.

3 Background

In this section, we present the example used throughout this paper, and introduce the Possum animation tool [5,6] used in this work.

3.1 Example – IntSet

The example is an integer set called *IntSet*. The specification is shown in Figure 1.

The *IntSet* specification is written in Sum [11], a modular extension to Z, and like all Sum specifications, contains a state schema, an initialisation schema, and zero or more operation schemas. 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, and in this example, it sets *intset* to be empty.

Sum uses explicit preconditions in operation schemas, denoted using the *pre* keyword, and also explicitly defines which state variables can be changed by using the *changes_only* function, which takes, as its sole argument, a set of state variables that are allowed to change for the operation. Like Z, input and output variables are decorated using ? and ! respectively, and post-state variables are primed ($'$). The *init* schema and all operation schemas automatically include the state schema in their declarations.

The four operation schemas in the *IntSet* specification are: *add*, which adds a particular integer to the set if that integer is not already in the set and the set is not full; *remove*, which removes a particular integer from the set provided it is in the set; *isMember*, which returns a boolean indicating whether a particular integer is in the set; and *size*, which returns the size of the set.

IntSet

$maxsize == 10$

state

$intset : \mathbb{P}\,\mathbb{Z}$

$\#intset \leq maxsize$

init

$intset' = \{\,\}$

op add

$x? : \mathbb{Z}$

$\mathrm{pre}(x? \notin intset \land$
$\quad \#intset < maxsize)$
$intset' = intset \cup \{x?\}$
$changes\_only\{intset\}$

op remove

$x? : \mathbb{Z}$

$\mathrm{pre}(x? \in intset)$
$intset' = intset \setminus \{x?\}$
$changes\_only\{intset\}$

op isMember

$x? : \mathbb{Z}$
$out! : \mathbb{B}$

$out! \Leftrightarrow x? \in intset$
$changes\_only\{\}$

op size

$size! : \mathbb{N}$

$size! = \#intset$
$changes\_only\{\}$

Fig. 1. Sum Specification of *IntSet*

3.2 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 $x?$. Let us assume that the value of the state variable *intset* before the query is $\{1\}$:

```
add {3/x?}
```

Possum returns with:

```
[intset := {1}, intset' := {1, 3}]
```

This means that the value of the state variable *intset* has been updated to $\{1, 3\}$. Possum also displays any bindings for any variables that it instantiates, 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.

4 Animation Using Testgraphs

In this section, we discuss using testgraphs to perform animation. We use testgraphs because they are straightforward to derive, and deriving cases from testgraphs can be done quickly and automatically. Testgraphs give us a planned, documented, and repeatable approach to animation that allows us to analyse the specification as a whole instead of animating each of the operations in isolation.

4.1 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 the initial state.

Using animation, 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, which is a small specification by industry standards, the size of the state space is infinite. Therefore, we select a subset of the state space as nodes for our testgraph. In the context of animation, the state space contains all states that *can* be reached; the testgraph nodes are the set of states that *will* be reached during animation.

The state of a specification provides important information about the selection of states for animation. For example, the *add* operation in *IntSet* will behave differently when the set is full (has *maxsize* elements in it) to when it is not full.

Standard testing practice advocates many methods for selecting special state values using rules such as *the interval rule*. For the *IntSet* example, we select our states based on the size of the set, and include four states: an empty set, a set that is half-full containing only odd numbers, a set that is half-full containing only even numbers, and a set that is full containing both even and odd numbers.

Once we have our testgraph nodes, we derive arcs for the testgraph to be used as transitions during animation. We require each node to have at least one arc leading in to it, except the start node, otherwise the node will be unreachable.

Figure 2 shows the testgraph for the *IntSet* specification. Here, we have four nodes representing the states derived above: *EMPTY*, *ODD*, *EVEN*, and *ALL*. *EMPTY* is the start node and this is indicated by the node being shaded. The five 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*.

4.2 Traversing the 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. For example, the *ADDODD* transition in the *IntSet* testgraph is shown in Figure 3, where $\fatsemi$ represents the sequential composition of operations by identifying the post-state of the first call as the pre-state of the next.

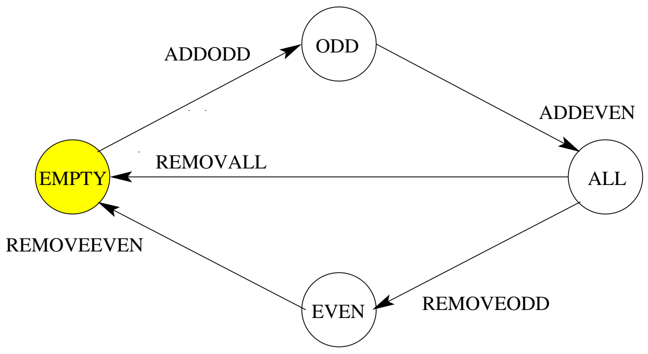

Fig. 2. Testgraph for *IntSet*

$$ADDODD == add\{1/x?\} \mathbin{;} add\{3/x?\} \mathbin{;} add\{5/x?\} \mathbin{;} add\{7/x?\} \mathbin{;} add\{9/x?\}$$

Fig. 3. *ADDODD* operation for *IntSet*

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 traverse 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.

4.3 Checking States and Operations

Once the testgraph has proceeded to a new node, we want to check properties of the current state of the specification, e.g., that after the transition *ADDODD* from *EMPTY* to *ODD*, *intset* $= \{1, 3, 5, 7, 9\}$, and the *size* operation returns *size*! $= 5$.

There are two ways to do this: manually using the standard Possum interface, or partially automated using *CHECK* schemas.

The manual approach involves checking the current value of the state is correct for each node, and manually invoking the operations that we wish to check. For example, after the *ADDODD* transition, we would check that the new value of *intset* is $\{1, 3, 5, 7, 9\}$. We would then invoke the *size* operation, expecting *size*! $= 5$ to be returned, and invoke the *isMember* operation for at least two values, one that returns true and one that returns false.

For the partially automated approach, we define schemas that automatically check the properties for the current state of the specification. For example, the *CHECK_ODD* schema, shown in Figure 4, is used to check: that for the *isMember* operation, every element that returns true is in the *intset* state variable, and vice versa[1]; that the *size* op-

[1] Although the set $\mathbb{Z}$ is infinite, Possum has a maximum bound for this that can be changed by the user

eration returns *size*! $= 5$ and does not change the state, and that *intset* $= \{1, 3, 5, 7, 9\}$. In this schema, the variable *tgf_report*![2] is a finite set of *MSG*, where *MSG* is a previously declared set containing the four possible error messages: *ISMEM_TRUE_ERR*, *ISMEM_FALSE_ERR*, *SIZE_ERR*, and *STATE_ERR*. These error messages are defined as abbreviations

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{op CHECK\_ODD}} \\
\hline
\textit{tgf\_report}! : \mathbb{F}\,\textit{MSG} \\
\hline
(\textit{intset} \neq \{\textit{isMember}\{\textit{state.intset}/\textit{intset}, \text{true}/\textit{out}!\} \bullet x?\}) \Leftrightarrow \\
\qquad \textit{ISMEM\_TRUE\_ERR} \in \textit{tgf\_report}! \\
(\mathbb{Z} \setminus \textit{intset} \neq \{\textit{isMember}\{\textit{state.intset}/\textit{intset}, \text{false}/\textit{out}!\} \bullet x?\}) \Leftrightarrow \\
\qquad \textit{ISMEM\_FALSE\_ERR} \in \textit{tgf\_report}! \\
(\exists\, s : \mathbb{N};\ t : \mathbb{P}\,\mathbb{Z} \mid \textit{size}\{s/\textit{size}!, t/\textit{intset}'\} \bullet s \neq 5 \lor t \neq \textit{intset}) \Leftrightarrow \\
\qquad \textit{SIZE\_ERR} \in \textit{tgf\_report}! \\
\textit{intset} \neq \{1, 3, 5, 7, 9\} \Leftrightarrow \textit{STATE\_ERR} \in \textit{tgf\_report}! \\
\hline
\end{array}
$$

Fig. 4. Schema *CHECK_ODD* for *IntSet*

To use the *CHECK_ODD* operation to check the *ODD* state, we simply run the operation by typing *CHECK_ODD* at the Possum prompt.

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. The time and effort saved by checking whether the current state has been visited depends on how long the checks take to perform.

5 Tool Support

In this section, we describe tool support for the method outlined in Section 4. Applying this method manually is time-consuming.

The tool described in this section, called the *Possum Testgraph Framework*, is a tool we have plugged into Possum to allow us to edit, save, and restore testgraphs. It also has options for partial automation of testgraph traversal and report compilation.

5.1 Constructing a Testgraph

The first step is to construct a testgraph. When opened, the editor presents the user with a blank canvas on which the user can design their testgraph.

[2] We use the prefix *tgf_*, which stands for *testgraph*, to prevent variable name clashes.

The user can add nodes to the canvas, 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.

The default labels of nodes placed on the graph are determined by the order they are added. However, the user can change the label to any string not containing spaces.

Users can also associate a schema with a node. This schema is invoked during the traversal of the testgraph when the user wishes to perform a check on a state.

By default, arcs do not have a label. They are uniquely identified by the source and destination nodes. However, users can add labels to arcs. An arc label is also the name of the transition schema associated with that arc.

Testgraphs can be saved to disk and opened again at a later time. The user also has the option of clearing the canvas and starting a new testgraph.

5.2 Generating Paths

As discussed in Section 4, we traverse the testgraph to achieve arc coverage. We use the testgraph framework to automatically generate a sequence of paths that achieve arc coverage.

The path generation algorithm performs 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.

There are two paths generated for the *IntSet* example:

$$< EMPTY, ODD, FULL, EVEN, EMPTY >$$
$$< EMPTY, ODD, FULL, EMPTY >$$

The framework allows the user to save paths that have been generated, and open them again at a later time. This is for three reasons:

– The user can see the paths that have been generated by the algorithm.
– The user can remove some of the paths by editing the file the paths are saved in, thus reducing animation time.
– The user can manually generate paths, save them in a file, and open them for use in the framework.

The file format is simple, with each path being a sequence of node labels in the order they are visited, separated by a space. Paths are separated by a line break.

5.3 Traversing the Testgraph

There are three ways that the framework allows users to traverse the testgraph: manual traversal, partially automated traversal, and fully automated traversal. Whichever method is used, the current node and arc are highlighted in the testgraph during the traversal. The user can switch between any of the traversal methods during a session.

Manually Traversing the Testgraph: The user can manually traverse an arc in the testgraph by right-clicking on that arc, and selecting *"Traverse Arc"* from the menu.If the source node of the selected arc is not the current state, an error message is displayed to the user. If the source node is the current state, the arc is traversed, the transition associated with that arc is sent to Possum, and the current node is updated to the destination node. The tool waits for Possum to complete the transition before sending the schema associated with the destination node to Possum.

Stepping Through the Testgraph: The user can also choose to step through the paths generated by the testgraph framework. By this, we mean traverse one arc at a time. They can do this by holding *Shift* and clicking the middle mouse button, or by selecting this option from a menu. The next arc is traversed, sending the transition associated with the arc to Possum and updating the current node to the destination node. The tool waits for Possum to complete the transition before sending the schema associated with the destination node to Possum.

Automatically Traversing the Testgraph: Automatically traversing the graph uses the paths generated by the framework, but unlike stepping through the testgraph, no user interaction is required. The user simply selects, from the *Animate* menu in the menu bar, the option *"Traverse All Paths"*. The framework traverses the first arc, sends the transition to Possum, updates the current node to be the destination node, and waits for Possum to perform the transition before sending the operation associated with the destination node to Possum. It then waits for Possum to finish running the operation, and performs the next transition in the path. This continues until all arcs in all paths have been traversed.

5.4 Report Generation

The check operations discussed in Section 4, such as *CHECK_ODD*, 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, destination and source nodes. The result is a report containing all messages generated and where they occurred.

For example, if after the transition *ADDODD* from the *EMPTY* to *ODD* nodes, the *CHECK_ODD* operation returned an error indicating that the *size* operation returned an incorrect value, the report would include:

```
   Transition: (ADDODD, EMPTY |--> ODD); CHECK_ODD returned:
Error: operation 'size' returning unexpected value for 'size!'.
```

After sending a transition or check to Possum during traversal, the traversal algorithm will wait until the value of *tgf_report*! is read back from Possum. Therefore, if a transition fails, the traversal will not continue. To recover from this, we define the *ADDODD* operation in Figure 3 as an auxiliary operation, *ADDODD_AUX* which we then use to

```
op ADDODD
tgf_report! : 𝔽 ADDODD_FAIL
────────────
if ∃ state′ • ADDODD_AUX then
      ADDODD_AUX ∧ tgf_report! = {}
else
      tgf_report! = {ADDODD_FAIL}
fi
```

Fig. 5. Updated *ADDODD* Operation for *IntSet*

define an updated *ADDODD*, shown in Figure 5. The new *ADDODD* operation first checks to see if the transition can be made. If so, the transition is made and *tgf_report*! is set to empty. If not, the error message *ADDODD_FAIL*, which is a previously declared abbreviation for the string: *"Error: Transition ADDODD failed unexpectedly"*, is included in the report variable, *tgf_report*!.

The user can view the report at any point during or after traversal. The report is displayed in a new window.

Once the user has run the traversal, new additions to the report generated during subsequent traversals are appended to the end. The user can clear the report or save it to a text file.

5.5 Advanced Features

Regression Animation: The testgraph framework gives the user the option to perform *regression animation*: where results from previous runs are used to check the results of new runs. When a node is visited for the first time, the tool records the value of the state at that node. A check is performed on the value of the state against this recorded value for subsequent visits to that node.

For this to happen, the user has to define a operation in the specification called *retrieve*, which retrieves the value of the state for the specification being animated. Figure 6 shows the *retrieve* function for *IntSet*.

When a testgraph is saved, the value of the state at each node will also be saved. When the testgraph is opened, these values will be loaded and associated with their respective nodes, and on subsequent runs, these values are compared to their respective values at each node. If there is a difference, an error is added to the report.

The user can turn the regression checking on and off. By default, this option is off.

```
op retrieve
tgf_state! : ℙ ℤ
────────────
tgf_state! = intset
changes_only{}
```

Fig. 6. *retrieve* function for *IntSet*

Memoisation: As discussed in Section 4, once a node has been visited and the state associated with that node checked, it is not necessary to check the state on subsequent visits to that node if the state is the same as on previous visits. Therefore, the framework has an option to not check the state of a node that has been previously visited. Instead, it records the state of the specification at each node the first time the node is visited, and checks the states are equal on subsequent visits. If the states are the same, the check is not performed. This is called *memoisation*: where the results of a previous calculation are transparently saved and reused to reduce calculation time. This technique is also used in functional and logical programming languages to improve efficiency of programs. If the states are different, an error is added to the report informing the user.

Memoisation and regression animation are similar, but neither subsumes the other because regression animation will check if the current value of the state at a node is the same as a previous value at that node, but will still perform the check. Like regression animation, memoisation requires a *retrieve* function to get the current value of the state.

The user can turn the memoisation on and off. By default, this option is on.

5.6 Experience

As well as the *IntSet* example, we have used our framework to check several other specifications, including two larger case studies. These case studies show that the method scales to medium-sized specifications, and can uncover a significant amount of errors in these specifications. For brevity, a discussion of these is not included here, but is available in an expanded version of this paper [14].

6 Conclusions and Future Work

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 animation 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. This framework was explained using a small example of an integer set, and we also discussed the application of this method on two non-trivial specifications. The results from these case studies were promising, because they took little effort and time, and uncovered several significant problems in both case studies.

Plans for future work in this area are to investigate more generic, specification-independent properties to be checked on specifications.

References

1. O. Bosman and H. W. Schmidt. Object test coverage using finite state machines. In *Proceedings of TOOLS Pacific '95*, pages 171–178, Melbourne, Australia, 1995.
2. J. Callahan, Easterbrook. S., and T. Montgomery. Generating test oracles via model checking. TR NASA-IVV-98-015, NASA / West Virginia University Software Research Laboratory, 1998.
3. 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.
4. 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.
5. D. Hazel, P. Strooper, and O. Traynor. Possum: An animator for the Sum specification language. In *Proc. Asia-Pacific Soft. Eng. Conf. and Int. Comp. Sci. Conf.*, pages 42–51. IEEE Computer Society, 1997.
6. D. Hazel, P. Strooper, and O. Traynor. Requirements engineering and verification using specification animation. In *Proc. 13th IEEE Int. Conf. on Automated Soft. Eng.*, pages 302–305. IEEE Computer Society, 1998.
7. C. Heitmeyer, J. Kirby, B. Labaw, and R. Bharadwaj. SCR*: A toolset for specifying and analysing software requirements. In *Comp-Aided Verif. 10th Annual Conf*, 1998.
8. 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.
9. 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.
10. IFAD. Features of VDM tools. http://www.ifad.dk/products/vdmtools/features.htm.
11. E. Kazmierczak, P. Kearney, O. Traynor, and L. Wang. A modular extension to Z for specification, reasoning and refinement. TR 95-15, SVRC, University of Queensland, Australia, February 1995.
12. E. Kazmierczak, M. Winikoff, and P. Dart. Verifying model oriented specifications through animation. In *Proc. Asia-Pacific Soft. Eng. Conf.*, pages 254–261, 1998.
13. T. Miller and P. Strooper. Animation can show only the presence of errors, never their absence. In *Proc. Aust. Soft. Eng. Conf (ASWEC 2001)*. Aust. Comp. Soc., 2001.
14. T. Miller and P. Strooper. Model-based specification animation using testgraphs. TR 02-15, SVRC, The University of Queensland, Australia, 2002. http://www.svrc.uq.edu.au/Publications/2002/svrc2002-015.html.
15. L. Murray, D. Carrington, I. MacColl, J. McDonald, and P. Strooper. Formal derivation of finite state machines for class testing. In *ZUM'98: The Z Formal Specification Notation*, volume 1493 of *LNCS*, pages 42–59. Springer Verlag, 1998.
16. P. Stocks and D. Carrington. A framework for specification-based testing. *IEEE Transactions on Soft. Eng.*, 22(11):777–793, 1996.
17. H. Treharne, B. Ormsby, J. Draper, and T. Boyce. Evaluating the B-Method on an avionics example. In *Proc. DASIA Conference*, pages 89–97, 1996.
18. H. Waeselynck and S. Behnia. B-Model animation for external verification. In *Proc. Conf. for Formal Eng. Methods*. IEEE Computer Society, 1998.