Animation Can Show Only the Presence of Errors, Never Their Absence

Tim Miller and Paul Strooper

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

Abstract

A formal specification animator executes and interprets traces on a specification. Similar to software testing, animation can only show the presence of errors, never their absence. However, animation is a powerful means of finding errors, and it is important that we adequately exercise a specification when we animate it.

This paper outlines a systematic approach to the animation of formal specifications. We demonstrate the method on a small example, and then discuss its application to a non-trivial, system-level specification. Our aim is to provide a method for planned, documented and maintainable animation of specifications, so that we can achieve a high level of coverage, evaluate the adequacy of the animation, and repeat the process at a later time.

1. Introduction

Animation, in the context of a formal specification, allows users to pose questions about a specification that can be answered quickly and automatically. This can be used for two different purposes:

- To give end users and field experts a chance to interact with the specification and observe its operational behaviour. Animation is better for this than other methods of formal specification analysis because end users and field experts often have little-to-no experience with formal specifications and notations.

- To provide the specifier with concrete examples of how the specification behaves, so that they can check that the specification reflects the intended properties of their design.

While animation is a cheap, quick alternative to formal proofs, it does not give the same level of assurance. The chance of finding errors in a specification using animation is only as good as the specific cases animated. Specification animation suffers from the same problem as software testing: in all but the most trivial cases, it can show only the presence of errors, never their absence [11]. Also, certain aspects of specifications may be hard or impossible to animate and different methods for checking specifications may have to be used in that case [17].

Even though animation has drawbacks, it can be beneficial if performed well. Performing animation in an ad-hoc fashion gives us no guarantee that the animation is adequate, and such an animation cannot be reviewed or repeated. We therefore have to systematically animate our specification to increase our level of assurance in its correctness.

Most of the current literature on animation describes tools for execution or interpretation of specifications, or mentions that animation has been used with little or no description as to how the animation inputs were selected or how extensive the animation was. In this paper, we present a method for systematically animating specifications so that our confidence in the correctness of the specification is high. The method includes documentation in the form of an animation plan, and guidelines for selecting inputs for the animation.

After reviewing related work in Section 2, we present Possum, an animator for the Sum specification language, which is a modular extension of Z, in Section 3. In Section 4 we describe our method for systematic animation and demonstrate it on a small example. Section 5 presents the results of our method trialed on a non-trivial, system-level specification. Section 6 concludes the paper.
2. Related work

There are several ways to check a formal specification. These include model checking, theorem proving, prototyping, animation and testing. In this section, we discuss only animation and specification-based testing, because these are most relevant to our work.

2.1. Animation

Several tools exist for animating formal specifications. PiZA [23] is animator for the Z specification language [34]. PiZA translates specifications into Prolog to instantiate output variables. PiZA allows Prolog statements to be embedded within specifications, and Prolog calls to be made from within the specifications. The B-Model animator [39] is used to animate the B Method’s [13, 37] model-oriented specification language. Waeselynck and Behnia [39] argue that animation should be performed at each stage of the B development process, not just at the specification stage. This requires the user to “flatten” the concrete representations of the system into an abstract notation suitable for animation. The Software Cost Reduction (SCR) toolset [21, 22] contains an animator that is used to test specifications. The IFAD VDM++ Toolbox [26] contains an interpreter that is also used to test specifications. It has a coverage tool that measures the percentage of the specification that is exercised during a trace.

Pipedream [28] is another animator for Z. 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. [28] describe animation as a light-weight approach to formal methods because although it does not provide the same level of assurance as formal proof, it does provide developers with a quick and cheap alternative to formal proof. Their work is the only that presents a method for selecting animation inputs. The results indicated that animation should be performed at each stage of the development process, not just at the specification stage.

We feel this method does not take full advantage of animation, because it only checks a few basic properties of specifications.

2.2. Specification-based testing

Formal specifications can aid in testing software by providing a starting point for selecting test inputs, executing test cases and determining expected outputs. Research in this area includes work on generating test cases for individual operations [15, 35, 36], generating finite state machines for the purpose of testing [8, 10, 24, 38], generating test oracles [20, 29, 33], and frameworks for specification-based testing that support test case generation, execution, and evaluation [6, 7, 12].

The work on test case generation is directly relevant, because some approaches used to select test cases from formal specifications can be used to select cases for animation. Two approaches are particularly relevant here. Zweben et al. [40] use specifications to perform control and data-flow testing. A flowgraph is developed by using each operation as a node, and placing an arc between two nodes A and B, if and only if the precondition of B can be satisfied following a call to A. They then define three criteria for control-flow testing: all statements, all branches and all paths; and three criteria for data-flow testing: all definitions, all uses and all-DU-paths. Ammann et al. [2, 3] discuss the application of mutation analysis [9] 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.

In a preliminary study [30], Miller used the specification-based testing approaches by Zweben et al. and Ammann et al. to generate animation inputs. The results indicated that the only method that exercised the specification adequately was the all-paths method for control-flow animation, but without tool support this was far too time-consuming to apply to non-trivial specifications.

2.3. Combining testing and animation

Waeselynck et al. [39] discuss using the B-Model animator as an oracle during testing by supplying the animator with the test input and actual output. The animator returns whether this pairing is correct with respect to the specification.

Aichernig et al. [1] discuss validating the expected outputs of test cases by supplying them to the IFAD VDM interpreter along with their inputs. They revealed problems in both the specification and implementation of an air traffic control system using this approach.
3. Sum and Possum

This section presents Sum, the specification language used in the Cogito formal development environment [4, 5, 16], and the Possum animation tool [18, 19], used to animate specifications written in Z and Z-like languages, including Sum.

3.1. Sum

The Sum specification language [27] is a modular extension of Z [34]. Sum supports modules by having three different types of schemas: a state schema, which specifies the state of the module and contains a state invariant that specifies the values a state can have, an init schema, which initialises the module state, and an op schemas, which are schemas that change and query the state. These three different types of schemas provide the ability to specify distinguished state machines within a module. An import statement is available to allow access to schemas from other modules within specifications. Sum also uses explicit preconditions in schemas.

Figure 1 shows an example of a symbol table specification written in Sum. The symbol table stores a set of strings, and a corresponding value (a natural number) for each of these strings. The state of the module consists of one state variable, symtab, which is a partial function from strings to natural numbers (N). The state invariant states that the number of symbol/value pairs must be less than the constant maxsize. The init schema initialises symtab to be empty.

There are five op schemas in the symtbl module. The add operation adds a symbol/value pair if the table is not full and the symbol is not already in the table, the update operation updates the value for an existing symbol, lookup returns the value for an existing symbol, esym returns true if and only if the specified symbol is in the table, and size returns the number of pairs in the table. The maximum number of pairs in the table is restricted to four in this paper to simplify the animation of the specification in following sections.

In Sum, operation schemas implicitly include a primed and unprimed copy of the state schema. All input variables are decorated with a ? and output variables with a !. To help readers not familiar with Sum or Z understand the specification better, we discuss one operation in detail. The lookup operation takes a string s as input. The precondition of this schema, labelled pre, states that s must be in the set of symbols in the table. If the precondition holds, the value of s!, the output variable, will be the corresponding value for s!. In Sum, changes_only A states that for an operation, only the state variables in the set A are allowed to change when that operation is invoked. Hence, in the lookup schema, the statement changes_only {} specifies that no state variables are allowed to change.

3.2. Possum

Possum [18, 19] is an animator for 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, and the simplified response from Possum.

\[ \{ x, y : 0..60 \mid x \times 13 = y \times 11 \} \]

is simplified to

\[ \{(0,0), (11,13), (22,26), (33,39), (44,52)\} \]

This example defines a set of pairs between 0 and 60, where for each pair, the first element times 13 is equal to the second element times 11.

Possum also supports plug-in GUI interfaces written in Tcl/Tk, which allows people not familiar with the specification language to interact with the specification through a user interface.

4. Systematic animation

The method for systematic animation we present in this section uses ideas from various animation and testing techniques. By systematic, we mean a process that is planned, documented and maintained. That is, the animation cases are selected to exercise the entire specification, the process is documented so that it can be understood and evaluated, and the work products are maintained so that the entire set of animation cases can be re-executed after each change made to a specification.

4.1. Work product definitions

4.1.1. Animation plan

The first step in our approach is to derive an animation plan. Without adequate planning and documentation, animating a specification is ad-hoc and has limited benefit. There are three sections defined in an animation plan:

- **File:** the file(s) containing the specification to be animated.
- **Assumptions:** describes any assumptions on which the animation may depend.
- **Animation Strategy:** explains how the animation cases are derived.
The animation plan is important because it is the document we use for planning and maintaining our animation. It gives us an insight into why the animation script is formed as it is.

### 4.1.2. Animation script

The animation script of a specification is the implementation of the animation plan for that specification. During animation, Possum records the state of the specification being animated, and displays any variables that were instantiated when a query is run on the state. This means that the Possum is responsible for checking that the state is updated appropriately and that any unbound output variables are instantiated. Possum animation scripts can then be saved and re-used for subsequent animations.

#### 4.2. Deriving animation cases

##### 4.2.1. Performing an initialisation check

As in the Pipedream approach [28], we first perform an initialisation check to ensure that a valid initial state exists. Hence, we want to show:

$$\exists\;\text{state} \cdot \text{init}$$

where \(\text{state}\) is the state of the module and \(\text{init}\) the initialisation schema. As an example, we apply the initialisation check to the symbol table specification. We send:

\[\text{init} \rightarrow \text{Possum}\]

and Possum returns with:

\[\text{symtab}' := \{\}\]

indicating that an initial state exists.

If there is no input for the initialisation schema, which is commonly the case, then this check is a proof witness that an initial state exists. If the initialisation schema has an input, then we select a suitable subset of values for the input

---

**Figure 1. Sum specification of symtbl**

<table>
<thead>
<tr>
<th>sympb</th>
<th>maxsize == 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>symtab : string (\to) (\mathbb{N})</td>
</tr>
<tr>
<td></td>
<td>#symtab (\leq) maxsize</td>
</tr>
<tr>
<td>init</td>
<td>symtab' = {}</td>
</tr>
</tbody>
</table>

**op add**

\(s? : \text{string}\)

\(v? : \mathbb{N}\)

\(\text{pre}(\#\text{symtab} < \text{maxsize} \land s? \notin \text{dom symtab})\)

\(\text{symtab}' = \text{symtab} \oplus \{s? \mapsto v?\}\)

\(\text{changes}_{\text{only}}\{\text{symtab}\}\)

**op update**

\(s? : \text{string}\)

\(v? : \mathbb{N}\)

\(\text{pre}(s? \in \text{dom symtab})\)

\(\text{symtab}' = \text{symtab} \oplus \{s? \mapsto v?\}\)

\(\text{changes}_{\text{only}}\{\text{symtab}\}\)

**op lookup**

\(s? : \text{string}\)

\(v! : \mathbb{N}\)

\(\text{pre}(s? \in \text{dom symtab})\)

\(v! = \text{symtab}(s?)\)

\(\text{changes}_{\text{only}}\{\}\)

**op exsym**

\(s? : \text{string}\)

\(b! : \mathbb{B}\)

\(b! \iff (s? \in \text{dom symtab})\)

\(\text{changes}_{\text{only}}\{\}\)

**op size**

\(\text{size}! : \mathbb{N}\)

\(\text{size}! = \#\text{symtab}\)

\(\text{changes}_{\text{only}}\{\}\)
parameters (see Section 4.2.3) and perform the initialisation check for all of those values.

4.2.2. Module states

As with software testing, the state of the specification provides important information for selecting animation inputs. For example, the lookup operation in the symbol table will behave differently when the table is empty, to when it is non-empty.

Special state values are chosen using heuristics such as the interval rule. The interval rule describes restrictions on the values of the state of a module over an interval \([L, U]\), where \(L\) is the lower bound and \(U\) the upper. The interval rule suggests we select the values \(L, U,\) and at least one value between \(L\) and \(U\). For example, the special values for the \textit{symtbl} module are chosen based on the size of the table, which is restricted to the interval \([0, \text{maxsize}]\). So we may select tables of sizes 0, 1, \text{maxsize}/2 and \text{maxsize}. We choose 0 and \text{maxsize} because they are the boundary of the interval, and \text{maxsize}/2 because this is in between. A table with 1 element in it is considered interesting enough to be included in this set.

4.2.3. Preconditions and postconditions

The next step in our method is to use the preconditions and postconditions of each operation to derive animation inputs.

To exercise the valid input space of the operations, we have to come up with inputs to be used for each special module state. Formal methods for partitioning the input space ensure that as many input domains as possible are animated. Two methods that we have found useful for animation are cause-effect analysis and partition analysis. Stocks and Carrington [36] discuss these and other methods for partitioning the input of operations.

With cause-effect analysis we choose the inputs such that we have an output for each equivalence class in the output space. For example, for the operation \textit{exsym} in the symbol table specification, we would choose at least one input that should produce the output \textit{true}, and one that should produce \textit{false}.

Partition analysis involves splitting the precondition of an operation into disjunctive normal form (DNF). For example, if the precondition is \(P \lor Q\), we would transform this into three disjoint predicates: \(P \land Q\), \(P \land \lnot Q\) and \(\lnot P \land Q\). This gives us disjoint partitions that correspond to input domains of the operation that should be animated.

We also want to animate inputs outside of the valid input space. The process is to negate the precondition, and partition the input using strategies such as partition analysis.

**animation strategy**

- **special values**
  - **state schema**
  - **number of symbols in table:**
    - \{0, 1, \text{maxsize}/2, \text{maxsize}\}
  - ... for each special state schema value add
    - if table is empty
      - \(s?\) not already in the table
    - else if size of table < \text{maxsize}
      - \(s?\) not already in the table
      - \(s?\) already in table
    - else if size of table = \text{maxsize}
      - \(s?\) not already in table
      - \(s?\) already in table

**lookup**

- if table is not empty
  - \(s?\) already in table
  - \(s?\) not already in table

**Figure 2. Animation plan for symtbl**

Where possible, cases just inside and outside the boundary of the valid input space should be selected to check that the precondition is correct. For example, for the \textit{add} operation, we attempt to add a symbol to a table of \text{maxsize}, and the value of \text{maxsize} – 1 is checked when we fill the table with \text{maxsize} pairs.

The above techniques select inputs based on the syntax and semantics of the specification. While useful, this is clearly dangerous since we are trying to determine if the specification is correct in the first place. We must therefore also animate any other special cases that we think may expose errors. This is similar to the “technique” described as error guessing in software testing [32].

4.2.4. Example

For each input we have derived, we animate that input for every module state value. As an example, we use the \textit{add} operation in the symbol table specification. In this operation, the precondition is that no symbol can be added if it is already in the table or the table is full.

To violate the precondition, we negate the precondition and use partition analysis to transform it into DNF. As a result, we come up with three cases:

- The table is already full and a new symbol is added.
- The table is not full and an existing symbol is added.
- The table is full and an existing symbol is added.
For the normal case, partition analysis and cause-effect analysis do not apply to the operation, so the only input is the input that satisfies the precondition. Therefore, we have a total of four cases.

We record each case we derive in our animation plan. The symbol table animation plan is shown in Figure 2. The special values of the state are selected based on the size of the table. The two operations shown in the plan are add and lookup. The add operation is broken up into three if statements. This is because we cannot animate certain cases under certain conditions. For example, we cannot animate the case where the input is already in the table if the table is in the empty state.

Once we have completed our animation plan, we have to come up with actual inputs to animate each condition. We derived seven cases for the add operation because we have to animate our cases for every module state possible. These cases are shown in Figure 3. We expect the first, third and fourth of these cases to be normal cases, and the rest to be exceptional cases.

4.3. Implementation

4.3.1. Performing a type check

Before we animate the specification, we run it through a type checker. We use the Sum type checker, which is part of the Cogito toolset [4, 5, 16]. Type checking is performed first because Possum does not support full type checking.

4.3.2. Creating and using the animation script

To run the animation the first time, we run the inputs specified in the animation plan manually through Possum. For example, to “execute” the call to add("00", 0), we would type:
add{"00"/s?, 0/v?}

Possum has an option that will automatically update the state of the module after each call. While we do this, we must check that the response after each command is correct. For each input, we run the case through Possum. If Possum returns “no solution”, either because the precondition is violated or because the operation’s predicate cannot be satisfied, then we check that the input that was used is meant to be an exceptional case. If it is not, then we have found an error in the specification’s precondition. If Possum returns a solution, we have to check that the post-state and any outputs from the operation are correct. For non-deterministic operations, which have more than one possible post-state and/or output for the same input, we check that the post-state and output are one of the possible ones. An alternative to checking the post-state and output, is to bind the expected post-state and output to the post-state and output variables being tested before the operation is executed, and Possum will return whether this binding is correct with respect to the inputs. For example, if we were to lookup the value for the string "00" and we expected the answer 0, then we could send the following call to Possum:
lookup{"00"/s?, 0/v!}

At the end of the session, the user can save the animation script for reuse at a later time. This is useful for re-running the animation case after a change. It can also be used to perform “regression animation”. The user can save an animation script, and later re-run the script, save it under a new name, and use a tool such as Unix’s diff to check that the scripts (including the output) are the same.

Part of the symbol table animation script is shown in Figure 4. This excerpt contains the trace used to animate the last if statement for add from Figure 2, which maps to the final two traces from Figure 3. The table is first initialised, then filled up with four pairs. The second last call attempts to add a new symbol to a full table, and the last call attempts to add an existing symbol to a full table. We expect both of these to fail. We take advantage of the fact that the state of the module is the same for both of these calls, so we do not have to execute the entire final trace of Figure 3, only the final call.

After each call, Possum displays the current value of the state and any instantiated output variables. For the final two calls, this does not happen, because we have violated a precondition in each.

5. Case study - a mass transit railway network

In this section, we briefly discuss a case study that was performed to evaluate the scalability of our method by animating a non-trivial, system-level specification. The specification is of a train network and is taken from [14]. The original specification was written in Object-Z and was translated manually into Sum. The original specification contained eight classes, which were reduced to four modules in Sum. The full specification can be found in an expanded version of this paper [31].

5.1. Informal description of the specification

The Mass Transit Railway Network 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. The value of the ticket is decremented by the fare amount when the passenger leaves the network.

Passengers are able to purchase three different types of tickets. A single-trip ticket is valid for only one trip, and must be worth at least the amount of the fare in order for the
passenger to leave the network. A multi-trip ticket is valid for any number of trips provided the ticket value is greater than 0. Multi-trip tickets can only be purchased for either $50 or $100, and are valid for 2 years. Season tickets are also available for any number of trips and have no monetary value. They can be purchased for a week, month, or year.

All tickets can be reissued, but only as the same type as they were originally issued. Reissuing a ticket updates both the value of the ticket and the expiry date.

5.2. Description of the specification

The specification of the Mass Transit Railway Network consists of four modules. Three of these modules are self-contained, while the fourth is a system-level specification that uses the three self-contained modules to perform its services.

The Clock module is used to record the current date and contains 3 operations. The Tickets module is used to record all tickets that have been issued or reissued, and which tickets are currently active in the network, and has 6 operations. The FareDataBase module is used to record the fare between stations in the network, and contains 4 operations. The fourth module, MassTransitRailway, is the system-level specification. It uses the Clock, Tickets and FareDataBase modules to issue and reissue tickets to passengers, to add and update fares between stations, to allow passengers to enter and exit the network, and to increment the day. This module contains 10 operations.

5.3. Bottom-up animation

To animate a specification containing more than one module, we took a bottom-up approach. This is a commonly used approach in testing, where the low-level modules are tested first, and the level of testing increases until the top-level module is tested, with any errors found in the low-level modules corrected before proceeding to the next
level. In the Mass Transit Railway case study, the Clock, Tickets and FareDataBase modules were animated in isolation, and then the MassTransitRailway module was animated using the three supporting modules instead of stubs.

5.3.1. Errors

Our systematic animation method combined with the bottom-up approach uncovered a total of five errors in the specification. There was one error in the Clock module, three in the Tickets module, and one in the MassTransitRailway module. We did not detect any errors in the FareDataBase module. These errors were introduced during the hand-translation from Object-Z to Sum.

The errors included: allowing state variables to change in operations where they should not, restricting state variables from changing in operations where they should, and swapping the two operands of a domain restriction. All errors found in the low-level modules were discovered and corrected when animating that module in isolation. No extra errors were found in the low-level modules when animating the top-level module.

5.3.2. Discussion

Deriving the animation inputs from the preconditions and postconditions was difficult because a precondition of a top-level operation is the conjunction of the preconditions from all low-level operations used in that operation. This meant that some operations had large preconditions, which made the input difficult to partition. As a result, a lot of effort was required to animate the entire specification.

One minor problem we encountered was with an inconsistency between Possum and the type checker. The type checker would not raise a problem when the max function, which returns the maximum value of a set or pair of values, was called using round brackets, e.g., max(0,1). However, Possum required these brackets to be curly brackets during its own type check, e.g., max\{0,1\}. This is due to the fact that Possum expects one argument to max in the form of a set, and the type checker expects two separate arguments.

5.4. Top-Level animation

Because animating the MassTransitRailway specification was costly, we investigated animating the specification at the top-level only, and compared it to the bottom-up approach. As with testing, animating only the top-level specification would clearly save time overall, however, testing only the top-level module of an implementation often fails to reveal errors in the lower-level modules.

The same animation script used to animate the system-level module was used again, but this time, we animated the original specification, not the corrected one. Using the bottom-up approach, these errors had been corrected before proceeding with top-level animation.

Using the top-level-only approach, only 4 of the 5 errors that had been previously discovered were discovered again. No extra errors were found using the top-level approach. The error that was not found was in the Clock module. The statement changes_only{}, which prevents the state of the Clock module from changing, had been left out of the operation that retrieved the current date. As a result, the date was reset each time this operation was invoked. We found that the system-level module was masking this error, because the system-level operations that used this operation had a changes_only statement that prevented the Clock state from changing.

We also found that locating errors with the top-level approach is significantly more difficult. When Possum fails to instantiate variables, which is often the case when an error occurs, the only feedback from Possum is the message “no solution”. Therefore, when a failure occurred, it was difficult to detect which module contained the error.

These results indicate that performing top-level animation only is not adequate when compared to the bottom-up approach.

6. Conclusions and future work

Specification animation is used to check properties of a specification. While animation does not prove that a specification is correct, we have demonstrated in this paper that there are clear benefits from animating formal specifications.

Randomly executing operations in the specification does not make sufficient use of animation. To uncover as many errors as possible requires us to plan and document the entire process. Running our animation cases on the specification without finding errors strengthens our assurance that the specification is correct.

In this paper, we presented a method for systematic animation that requires us to document the process using an animation plan, and to systematically select animation inputs to achieve high assurance of the specification. This method of animation is intended more for the specifier rather than for the end user or field expert, because we are testing the specification from the specifier’s point of view. Also, animating the low-level modules might not be as useful to end users as animating the top-level, because the end user will typically not care about the details of the low-level modules, only the behaviour of the system.

We demonstrated the method on a small symbol table specification. We also discussed the results of a case study.
that was performed on a non-trivial, system-level specification. The results from this case study were promising because we found several errors in the specification. However, our method was difficult and time-consuming to apply to the larger, system-level specification, which indicates that we will have problems with scaling it up to large systems.

Ideas for future work in this area are:

- Address the issue of scalability of the method, particularly by investigating tool support to eliminate the amount of manual work needed to enter animation scripts, and to perform mechanical tasks such as partitioning the input space of operations.

- Applying the method to larger, real-life, industry specifications to refine and evaluate the scalability of the method.

- Analysing the coverage the method achieves on animation methods such as control and data-flow animation, and mutation analysis, as described in [30].

- Investigate whether the work products produced during animation of a specification could be used to help test the implementation of that specification.

- Investigate whether the method could be applied to specifications with a GUI interface. For example, an interface written in Tcl/Tk for Possum. Using a GUI interface will allow a domain expert to interact with a complex specification, hence we can investigate how well our method works for end users exploring the specification, not just specifiers.

- Compare our systematic animation method with formal proof techniques and model checking on criteria such as effort and number of errors found.

References


[31] T. Miller and P. Strooper. Animation can show only the presence of errors, never their absence. TR 01-09, Software Verification Research Centre, The University of Queensland, Brisbane 4072, Australia, Mar. 2001.


