Aslantest: A Symbolic Execution Tool for Testing Aslan Formal Specifications *

Jeffrey Douglas
Richard A. Kemmerer

Reliable Software Group
Department of Computer Science
University of California
Santa Barbara, CA 93106

Abstract

This paper introduces Aslantest, a symbolic execution tool for the formal specification language Aslan. Aslan is a state-based specification language built on first-order predicate calculus with equality. Aslantest animates Aslan specifications and enables users to interactively run specific test cases or symbolically execute the specification. Testing the formal specifications early in the software life cycle allows one to assure a reliable system that also provides the desired functionality.

1. Introduction

As the size and complexity of software systems have increased, it has become increasingly more difficult for software designers to produce a quality product that meets the customer's requirements in a timely manner. The problem is often a symptom of the informal nature of the design process. Requirements informally agreed upon at the beginning of the process are often misunderstood or misinterpreted, leading to deficiencies in the final software product. In addition, these deficiencies are often not discovered until very late in the design process, causing costly delays in delivery.

As a result, researchers have begun to restructure the design process, introducing approaches and tools, such as formal specification and verification, which enable a designer to rigorously demonstrate that an implementation is consistent with its requirements. Demonstrating that code is consistent with its critical requirements is a difficult process. However, the process can be made tractable by verifying the design at every step.

As shown in [Kem 85] there is a problem with the approach outlined above; although the specification may satisfy its critical requirements there may be no implementation that is consistent with the specification and that at the same time provides the desired functionality. To make things worse, this is usually not discovered until the design has gone through several levels of refinement, with each level being formally verified, and the implementation is in progress or completed. In addition, the cost of formally verifying software is high in both dollars and time. When a design needs to be redone due to the discovery of an error late in the development life cycle, the cost skyrockets.

In order to reduce the cost of developing reliable systems that also provide the desired functionality, it is necessary to provide a means to test the formal specifications early in the software life cycle. The Aslantest tool presented in this paper is a symbolic executor for testing Aslan formal specifications to assure that all of the functional requirements are satisfied. This provides the user with a rapid prototype system early in the software life cycle, and it allows the user to exercise the prototype to see if it meets all of the anticipated needs. In addition to helping the user find errors in the design of the system being developed, the prototype also allows the user to find other useful applications for the completed system before it is delivered.

A number of systems have been developed for executing state-based formal specifications. Some of these translate the specification into a high order language implementation, and others execute the specification directly. The EZ system [VN 91] is an example of the translation approach that can be used to test a subset of the Z language. It compiles Z specifications into C-Prolog prototypes, and the user can run various test cases on the prototype. With this system the user provides a start state and a result state, and the Prolog prototype attempts to use the operations defined in the specification to determine if the result state is accessible from the defined start state.

A system that is similar to Aslantest is the EPROS system, which is part of the VDM development environment [HI 88]. This system, compiles VDM META-IV formal specifications into an executable language called EPROL, which is an extension of META-IV. The system provides an interactive interpreter environment for EPROL where the user can initialize values, execute operations, and check results. The user can examine both the visible results of a sequence of operations, as well as the internal values of state variables. Like Aslantest, the EPROS system actually animates the specification rather than implementing a prototype.

Another system that takes a more graphical approach is the ExSpect system [VSV 91], which animates Z specifications using Petri nets. With this approach tokens move about the net indicating the collection of states the system is currently in. Because tokens provide a graphical representation of how the system is performing, this type of system is useful for animating specifications of concurrent systems.

* This research was partially supported by the National Computer Security Center under grant MDA904-92-C-5149

The remainder of this paper consists of four sections. Section two provides more insight into the role of formal specification and verification. It also reviews classifications for relating the validity of a functional requirement with respect to a formal specification that were originally presented in [Kem 85]. Section three gives a brief overview of the Aslan specification language. Section four describes the features and use of the Aslantest symbolic execution tool and presents an example session. Finally, in the last section some concluding remarks and future considerations are presented.

2. Formal Specification and Verification

Formal specification and verification enables a designer to show that an implementation is consistent with its requirements. The process begins by developing a formal model of the critical requirements of the system, which are informally agreed upon by the customer and the developer. After these have been stated, a high level specification of the system is developed, which precisely describes the behavior of the system. This may be followed by a series of less abstract specifications, eventually resulting in an implementation of the system in a high order programming language.

A step-wise, inductive approach is taken when showing that the resulting implementation is consistent with the original requirements. The first step is to show that the formal model correctly models the critical requirements determined at the requirements stage. Because of the imprecise and ambiguous nature of the original requirements, this is an informal step and is the only informal step in the process. The next step is to prove that the highest level specification satisfies the critical requirements as stated in the formal model. Because the specification and the formal model are stated in a common, mathematically based notation, proof obligations can be generated to show that the relationship holds. Then each successively more detailed specification is shown to be consistent with the specification one level above it. Finally, the implementation must be shown to be consistent with the lowest level specification. By using this step-wise approach to refine down to the high order language implementation, the implementation can be shown to be consistent with the formal model. In addition, because the formal model correctly represents the critical requirements for the system, the implementation is consistent with the critical requirements.

Although formal verification can be very helpful in ensuring the production of reliable software, it is very costly in terms of time and money. As a result of this high cost, usually only the requirements considered critical are modeled and verified (e.g., security or safety requirements). Therefore, the satisfaction of many non-critical functional requirements are left to the testing phase. This often results in costly delays as functional deficiencies are not discovered until very late in the development process. One way to circumvent this problem is to animate the specifications developed early in the process. This provides the designer and the user with a prototype on which to test various non-critical functional requirements.

There are two major approaches to animating specifications. One is to develop a high order language implementation and use this as the prototype. The second approach is to symbolically execute the specification itself, modeling the system behavior as described in the specification without imposing any implementation or procedural constraints. The advantages and disadvantages of each approach are discussed in detail in [Kem 85]. That paper also defines three relationships that provide a framework for describing whether a functional specification is implicitly implied by a system specification. For these definitions let S be a formal specification, and let F be a functional requirement for the system formally specified by S. The three possible relationships between the functional requirement F and the formal specification S are:

Satisfiability: a functional requirement F is satisfiable with respect to a specification S if there exists an implementation of S that gives the functionality described by F.

Unsatisfiability: a functional requirement F is unsatisfiable with respect to a specification S if none of the possible implementations of S give the functionality described by F.

Validity: a functional requirement F is valid with respect to a specification S if every possible implementation of S gives the functionality described by F.

Ideally, at the specification level, every functional requirement should be shown to be valid with respect to the specification. The major disadvantage of using the implementation approach is that because a particular implementation is chosen when producing the prototype, the best that one can do is to show the satisfiability of a requirement. The validity of a requirement cannot be determined with this approach, since all possible implementations are not tested. In contrast, by symbolically executing the specification itself, no implementation constraints are placed on the animated specification. Therefore, the validity of the requirement can be shown.

After choosing an approach to animation, there are two means of testing functional requirements. The first is through the application of individual test cases using actual values and the second is by using symbolic values. While the first can be effective, it is virtually impossible, in terms of time and energy, to exhaustively test a system. By using symbolic values as the initial values for variables and propagating these values through the operations, a single test case can represent a range of values. Because the results are expressed in terms of symbolic values and constants defined in the specification, these expressions can be formally analyzed. In addition, by making assertions about the initial symbolic values, properties can be proved about the execution. Both of these testing approaches can be used with the Aslantest system.

3. Aslan

The following is a brief overview of the Aslan formal specification language. For a more detailed description, the reader should consult the Aslan user's manual[AK 92].

Aslan is a state-based specification language built on first order predicate calculus with equality. Systems specified in Aslan are viewed as being in one of a collection of *states*, depending on the values of *state variables*. Movement between states is achieved using *state transitions*, which alter the values of state variables. For example, consider an automated teller machine. A specification for an ATM would likely have a state variable to indicate whether the machine is currently active and a state transition to toggle that variable:

```
Transition end_session
 Entry
   active
 Exit
   ~active
```

This transition has an entry condition, which expresses that in the state from which this transition is executed, the boolean variable active must be true. This transition also specifies that in the resulting state, the boolean variable active is false, indicating the end of the ATM session.

In Aslan there are two kinds of critical requirements: invariants and constraints. Requirements that must hold in every reachable state are state *invariants*. For the ATM specification one could have the following invariant:

```
active -> ~restricted(currentcard)
```

where "->" is logical implication. This invariant states that if the ATM is currently in use(i.e., active is true), then the current user (as represented by currentcard) is not restricted from accessing the ATM. *Constraints* express relationships between the values of state variables as they change across state transitions. For example, consider a soda machine that dispenses soda at 50 cents a can. The specification for this machine would likely have the requirement that, if a soda was dispensed, then there was at least fifty cents deposited before hand. This is expressed using the following constraint:

```
(No_of_sodas = No_of_sodas' - 1)
    (Amount_deposited' >= 50)
```

where No_of_sodas and Amount_deposited are state variables, and the prime(′) refers to the value of the variable immediately prior to the state transition.

To assure that a specification is consistent with its requirements, Aslan takes an inductive approach, attempting to show that the specified system always satisfies the critical requirements. The base step of the induction is to show that the system begins only in a state that satisfies the state invariant. In the specification, any assertions regarding possible starting states are expressed in the *initial conditions*. So, the base step is to show:

initial conditions $\rightarrow$ invariant

The inductive step is to show that, for every transition T, if the state invariant is satisfied before the application of T, then the resulting state (i.e., after the application of transition T) will satisfy the invariant and the constraint:

invariant′ & Entry_T' & $\text{Exit}_T \rightarrow$ invariant & constraint

where invariant′ means priming all variable references, "&" is logical conjunction, and Entry_T and Exit_T represent the entry and exit assertions, respectively, for transition T.

Appendix A contains a pedagogical specification for a simple automated teller machine that distributes money in multiples of $20.00. Each customer has an ATM card, which is represented by the unspecified type card. Each card has a pin number and corresponds to a customer's bank account, represented by the constants pin_no and account. Note that the type of account is accountnum, which is also unspecified. This leaves the implementation details of the account number for a lower level specification. The constant cassettesize specifies how many $20 bills can go in a cassette; the ATM can hold one cassette of money at a time.

The boolean state variable active is true if the machine is currently in use, and currentcard tracks which card is currently in the machine. The number of $20 bills in the machine is recorded by numtwenties. The ATM gives each card holder three tries to correctly enter his/her pin number. The variable retries keeps track of a card holder's remaining attempts. The boolean variable restricted is true if a card holder has exceeded his/her maximum number of attempts; once restricted, the card holder loses access to the ATM. Balance is a variable representing each account's current balance. Finally, the ATM sets a maximum on the amount of money that can be withdrawn each day, and the variable limit represents how much of each user's daily maximum is still available to be withdrawn.

There are two critical requirements for the system. The first states that if the machine is active, the card currently in the machine is not restricted. The second states that if a card is restricted, then no more attempts to use the machine can be made. The initial state of the system specifies that the ATM is not active, and it contains a full cassette of $20 bills. It also specifies that each user's card is not restricted, has a maximum of three retries, and a limit of $300.00.

In Aslan *axioms* are used to express assumptions about types and constants. These assumptions can be used to prove the proof obligations that are generated. As more detail is included in lower level specifications, such as refining an undefined type to an actual type, the axioms of the higher levels can be proved to be true. An axiom for the ATM system is that no two pin numbers are the same.

There are seven transitions in the specification: insert_card, deposit, withdraw, end_session, dayrollover, resetcard, and refill_machine. Insert_card and withdraw contain except-exit pairs, which define possible alternate outcomes should the normal entry condition for the transition not be satisfied.

4. Aslantest

Aslantest is a symbolic execution tool for the Aslan formal specification language. Its features and functionality were strongly influenced by two existing symbolic execution tools, Unisex[KE 85a], which is a symbolic executor for Pascal, and Inatest[KE 85b], which is a symbolic executor for the Ina Jo specification language. Aslantest allows users to test a specification to see if it satisfies various functional requirements, where a functional requirement can be thought of as a test case for the system. An example of a functional requirement for the ATM specification might be:

when the ATM performs its end of the day activity (dayrollover), the remaining number of retries for a user should not be changed.

The following sections explain the Aslantest approach to animation, present some of the features of Aslantest, and show how this functional requirement could be tested in the Aslantest environment.

4.1 Animation of State Machine Specifications

Because of the informal nature of most requirements, the most difficult part of the testing process is to accurately state the requirement in terms of the formal model being used. In a state-based model, a functional requirement consists of a start predicate, a sequence of transitions, and a result predicate. The *start predicate* expresses conditions that must hold in the state from which the sequence of transitions commence, and the *result predicate* expresses the conditions that are to be satisfied upon completion of the sequence of transitions. The start predicate, sequence of transitions, and result predicate would be defined as follows for the example functional requirement introduced above.

```
Start Predicate:
          ~active
        & pin_no(card1) ~= pin1
        & ~restricted(card1)
        & retries(card1) = numtries
        & numtries > 0
        & numtries <= 3
Sequence:
        insert_card(card1, pin1)
        dayrollover
Result Predicate:
        retries(card1) = numtries - 1
```

The start predicate states that the ATM machine is not active, that card1's pin number is not equal to pin1, that card1 is not restricted, and that card1 has numtries retries remaining. The start predicate also makes assertions restricting the range of values for the constant numtries to be between one and three inclusive, which is the valid range of values for a non-restricted card. By using the unspecified constant numtries instead of an exact value for retries(card1), the validity of this requirement can be shown with a single execution. If exact numeric values were used for retries(card1), then separate test cases would have to be run for each possible value of retries(card1). This illustrates one of the advantages of using symbolic values.

The sequence consists of two transitions. First, card1 is inserted using pin1 as the pin number, which will fail, and then dayrollover is applied. The result predicate states that card1 should have "numtries - 1" remaining retries, reflecting the failed attempt, and that dayrollover does not affect the remaining number of retries.

Verifying that the requirement is satisfied involves initializing the system to the state specified by the start predicate, executing the sequence of transitions, and examining the resulting state to see if it satisfies the conditions in the result predicate. In examining a state, there are three components to consider: the current values of the state variables, the state predicate, and the execution path. The state predicate lists all assertions regarding the values of state variables and constants. The execution path provides an ordered listing of transitions that have been executed, showing the outcome of each transition and listing any assumptions that were made while executing the transitions.

4.2 Aslantest Features

In using Aslantest, the user provides an Aslan formal specification to the symbolic executor. The executor uses the initial conditions of the system to determine its initial state, giving the specified values to all variables listed in the initial conditions and symbolic values to all other variables. All initial assertions are placed in the state predicate and the execution path is set to null. The user may then begin to execute transitions, enter start or result states, examine components of the current state, or save a state for later restoration.

In the process of executing a transition, the executor often encounters conditional expressions that need to be evaluated. For instance, the exit assertion of the dayrollover transition for the ATM has the following conditional:

```
FORALL c:card (IF ~restricted'(c)
                  THEN retries(c) = 3
               FI).
```

Because these conditional expressions may contain symbolic values, they can not always be reduced to true or false. Aslantest has a simplifier which uses the current state predicate and state variable values to attempt to automatically reduce these expressions. If the simplifier is successful, then it will move on, following the appropriate branch of execution. If it cannot reduce the expression to either true or false, then it will ask the user to play the role of simplifier. The user may answer true or false if there is enough information in the current state to merit such a response. Otherwise, the user responds neither and then assumes true or false, so that execution can continue. This assumption is then noted in the state predicate and in the execution path.*

The following subsections outline some of the features that are available to the Aslantest user. For a detailed discussion of the Aslantest environment the reader should consult the Aslantest user's manual [Dou 93a].

* Because the user would normally want to test the results when assuming true as well as when assuming false, the normal procedure would be to save the current state after responding neither, assume true (false), restore the saved state after the execution of interest completes, and then assume false (true) to test the alternate path.

Execution Commands

The execution commands are the commands used to actually animate the specification. Start or result predicates may be entered using the Init command. When a new start predicate is entered it is treated as a new set of initial conditions, and the system is initialized according to that start predicate. After the start state is initialized, the "check current" command can be used to assure that the start state satisfies the invariants (see Generating Proof Obligations section below). This does not, however, assure that the start state is a reachable state.

Transitions may be executed one at a time or as a sequence, and the transitions to be executed may be entered interactively or read from a file. In fact, all Aslantest commands can be read from a textfile, which is very helpful for regression testing.

Miscellaneous Displays

Whenever the Aslantest executor is waiting for input from the user, the user may use the display commands to get information about the current state of the execution. The user may display each of the transitions of the specification, the current state predicate, current variable values, current execution path, start or result predicates, axioms of the specification, and even the specification itself.

Saving and Restoring States

At any time during the symbolic execution session, the user can save the current state, restore a previously saved state or remove a state from the set of saved states. Saving/restoring a state consists of saving/restoring that state's variable values, state predicate and execution path. States can be restored in any order, and restoring a saved state does not automatically remove the state from the set of saved states.

Predicate Commands

A user can add assertions to the current state predicate by using the Addpred command. By making assertions about constants or state variables, the user is improving the knowledge base of the simplifier. This enables the user to add equality and first order predicates. Aslantest also has a special "reduces_to" operator, which allows a user to state that a particular predicate reduces to another. This operator is useful in presenting axiomatic relationships to the simplifier.

Generating Proof Obligations

Using the "Check" command, two proof obligations can be generated during an Aslantest session. The first, check current, verifies that the current state satisfies the invariant for the system:

current state $\rightarrow$ invariant

The second, check result, verifies that the result predicate holds in the current state:

current state $\rightarrow$ result predicate.

As is the case with conditional expressions, the Aslantest simplifier will attempt to automatically reduce these proof obligations to true or false. If unsuccessful, the reduced expression will be presented to the user, who must determine if it can be reduced to true or false.

General Commands

There are three environment flags within Aslantest that may be set to tailor the execution session to the user's needs. The simplify flag controls the amount of simplification performed by the system. The verbose flag controls the amount of information displayed as execution progresses. The autosave flag is set to save the state of the system after the execution of each transition. When the autosave flag is set to false, states are only saved at the explicit request of the user. Aslantest also provides a number of general commands that provide help, allow the user to enter descriptive comments, reinitialize the executor, load in a new specification, and exit the executor.

4.3 An Example

In this section, an example session is presented using the ATM specification and the functional requirement introduced earlier. Appendix B contains a numbered listing of the example symbolic execution session. The Aslantest user prompt is "#", and comments entered by the user are enclosed between "/*" and "*/".

For this session the start predicate is entered manually (lines 11-30), and the resulting current state is listed using the vars command (lines 32-63), the display command (lines 64-68), and the path command (lines 69-70). The current state consists of the following:

```
State Variables:
        retries(card1) = numtries
        restricted(card1) = false
        active = false
 All other variable instances are undefined and given
  symbolic values
State Predicate:
          pin_no(card1) ~= pin1
        & numtries > 0
        & numtries < 4
Execution Path: NULL
```

Note that the variable instances active, retries(card1), and restricted(card1) take on the values specified in the start predicate. All other variable instances are given symbolic values, and the assertions involving card1's pin number and the restrictions on numtries are placed in the state predicate.* The execution path is null, because at this point no transitions have been executed.

After entering the result predicate (lines 71-80), the sequence of transitions is entered(lines 86-94) and the first transition, insert_card(card1, pin1), is evaluated. Insert_card has an entry condition, so it is evaluated. When evaluating an entry condition, all variable references are replaced by their current values and the resulting expression is evaluated. This yields the expression:

```
~false & ~false & pin1 = pin_no(card1),
```

which evaluates to false, because the state predicate specifies that pin1 is not equal to pin_no(card1). Since there are except-exit pairs, they are evaluated in the order listed. In evaluating the except-exit pairs, the executor encounters an expression that it cannot reduce to true or false, and the user is asked to evaluate the expression(lines 97 - 101). In this case, there is not enough information to reduce the expression to true or false, so the user responds neither and then assumes the expression is false.** After having the user evaluate another expression (lines 105-112), the corresponding exit predicate is applied, and the resulting state is:

```
State Variables:
        active = false
        retries(card1) = numtries - 1
        restricted(card1) = false
 All other variable instances are undefined and given
  symbolic values
State Predicate:
          numtries ~= 1
        & pin_no(card1) ~= pin1
        & numtries >= 2
        & numtries < 4
Execution Path
        Added conditions:
        numtries >= 2
        Start: insert_card(card1, pin1)
        Executed Except #2
        No assumptions
        Finished
```

Note that the state predicate now contains the assumptions that the user made while evaluating the except-exit pairs. Also, the execution path indicates that the assumptions were made immediately before the execution of transition insert_card (i.e., when the entry assertions were being evaluated). The execution path also indicates which except-exit pair was executed and whether or not there were any assumptions made during the application of the exit predicate(there were none).

Next, dayrollover is executed (line 165). Its entry condition requires only that the machine is not active, which is satisfied in the current state. Thus, the exit predicate is applied:

```
  FORALL a:accountnum (limit(a) = 30000)
& FORALL c:card (IF ~restricted(c)
                 THEN retries(c) = 3
              FI)
```

which specifies that in the resulting state, all accounts are given a new $300.00 daily withdrawal limit and for all cards, if the card is not restricted, the number of retries is reset to three. So the resulting state is:

```
State Variables:
        active = false
        retries(card1) = 3
        restricted(card1) = false
        FORALL _p1:accountnum(
           limit(_p1) = (IF true
                           THEN 30000
                        FI) )
 All other variable instances are undefined and given
  symbolic values
State Predicate:
          numtries ~= 1
        & pin_no(card1) ~= pin1
        & numtries >= 2
        & numtries < 4
Execution Path:
        Added conditions:
        numtries >= 2
        Start: insert_card(card1, pin1)
        Executed Except #2
        No assumptions
        Finished
        Start: dayrollover
        No assumptions
        Finished
```

Having executed the entire sequence of transitions, the user directs the executor to generate the proof obligation to see if the resulting state satisfies the result predicate(lines 219-222). That is, does:

* Note how the simplifier alters some expressions by putting them in a canonical form to aid simplification.

** At this point the user could have saved the state so that he/she could restore it later and test the results when the expression was assumed to be true.

```
            numtries ~= 1
          & pin_no(card1) ~= pin1
          & numtries >= 2
          & numtries < 4
    ->
          retries(card1) = numtries - 1?
```

After substituting the current value for retries(card1) the question becomes does:

```
          3 = numtries - 1?
```

From the state predicate, the user knows that numtries is less than 4, so (numtries - 1) must be less than 3. Since the maximum value for numtries - 1 is 2, the expression reduces to false, and the requirement is not satisfied by the specification. It is worth noting that at this point the user must decide if the requirement needs to be satisfied. If so, then the specification must be changed. In this case, changing the dayrollover exit predicate to only reset each user's limit would make this requirement satisfiable.

4.4 XAslantest

In addition to the teletype interface to Aslantest, which was introduced in the previous section, Aslantest has an X-Windows graphical user interface component, called XAslantest. XAslantest allows a user to symbolically execute Aslan specifications with the option of accessing all commands using the point and click technology of graphical user interfaces. The screen layout is shown in Figure 1. It consists of two scrollable windows which are adjustable in size, a message window, and a command panel. A numbered listing of the currently loaded specification is shown in the upper window; this display automatically scrolls to the part of the specification currently being executed. The lower window is an emulation of the teletype interface of Aslantest. Users can choose to type commands and responses directly into this window or enter them using the command panel. The message window provides informational messages and prompts. Figure 1 also shows a popup window that appears when the user selects a particular transition to be executed. For more details on XAslantest see [Dou 93b].

5. Conclusions and Future Work

This paper has presented the features and functionality of the Aslantest symbolic execution tool. The Aslantest environment provides designers and users with a flexible means of animating system specifications to test for various functional requirements. Aslantest has the potential for future integration into the design process for a number of reasons. First, the language that it animates, Aslan, is a highly readable and easy to follow formal specification language. Also, Aslantest executes the specification itself, modeling its behavior as described in the specification. Because no implementation constraints are placed on the resulting model, the validity of requirements can be shown. Its ability to symbolically execute a specification enables users to perform a wide variety of tests on the specifications. A user can initialize variables with actual values, symbolic values, or a combination of both. Finally, Aslantest's graphical interface and various environmental flags make it a system that can be customized to a particular user's preferences.

Because the functional requirements used for testing an Aslan specification are analogous to test cases, the test cases that were developed to test the specification during the design phases can be used for system testing of the implementation. This is particularly useful if the implementation is only proved to be consistent with the critical requirements of the system and not with all of the desirable functional requirements.

The Aslantest environment has been used for testing a number of formal specifications of real systems as well as pedagogical examples. In particular, aslantest was used to demonstrate a flaw in a key distribution protocol for a digital mobile communications system. Other system specifications that were tested include a hospital database, a secure release terminal, and a university library database.

Although Aslantest is a sound and useful tool, there are a number of enhancements that could be made. Included in these are improved type checking, improved evaluation and simplification modules, the ability to execute mappings to lower level specifications, and the addition of an interactive theorem prover. The improvements in the simplifier would not only include improving its simplification capabilities, but also improving it's ability to reproduce more readable expressions after performing its simplification operations. The interactive theorem prover is a module that would work interactively with the user to step-wise reduce an expression to true or false. Work on each of these areas is currently in progress.

In summary, it is clear that the use of formal specification techniques can greatly improve the design process. Aslantest is an example of an easy-to-use tool that demonstrates how the use of symbolic execution in conjunction with traditional testing techniques can enhance the ability of formal verification techniques to produce useful and reliable software.

References

[AK 92] Auernheimer, B. and R.A. Kemmerer, "ASLAN User's Manual," Department of Computer Science, University of California, Santa Barbara, California, TRCS 84-10, April 1992.

[Dou 93a] Douglas, J., "Aslantest User's Manual," Department of Computer Science, University of California, Santa Barbara, California, TRCS 93-12, July 1993.

[Dou 93b] Douglas, J., "XAslantest User's Manual," Department of Computer Science, University of California, Santa Barbara, California, TRCS 93-26, July 1993.

[HI 88] Hekmatpour, S. and Darrel Ince, *Software Prototyping, Formal Methods, and VDM*, Addison Wesley, New York, New York, 1988.

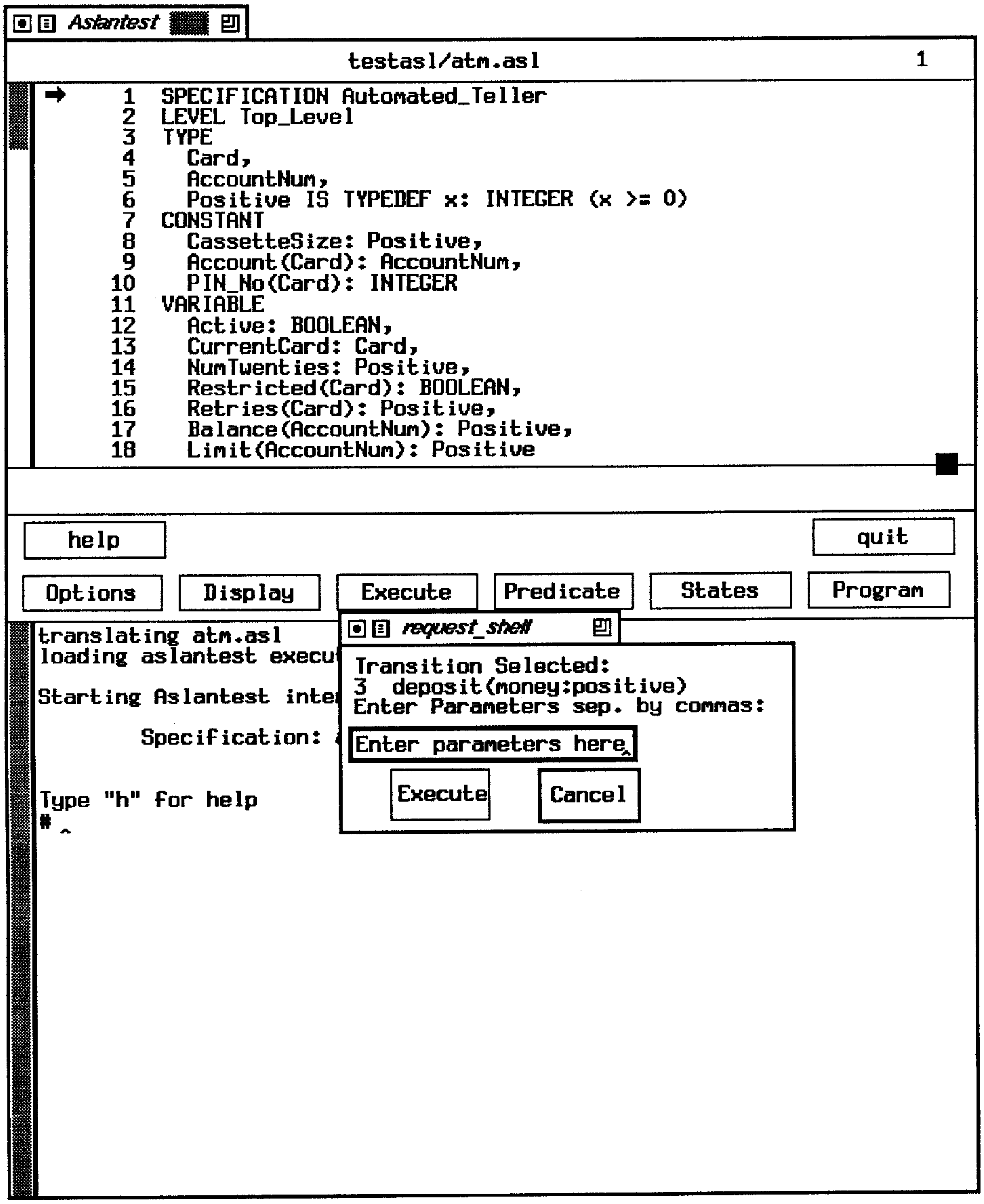
[Kem 85] Kemmerer, R.A., "Testing Formal Specifications to Detect Design Errors," *IEEE Transactions on Software Engineering*, pp. 32-43, vol SE-11, No. 1, January 1985.

[KE 85a] Kemmerer, R.A. and S.T. Eckmann, "UNISEX: a UNIx-based Symbolic EXecutor for Pascal," *Software-Practice and Experience*, pp. 439-458, vol. 15, No. 5, May 1985.

[KE 85b] Kemmerer, R.A. and S.T. Eckmann, "INATEST: an Interactive Environment for Testing Formal Specifications," *Software Engineering Notes*, vol. 10, No. 4, August 1985.

[VSV 91] van Hee, K.M., L.J. Somers, and M. Voorhoeve, "Z and High Level Petri Nets," *Software Development Methods, 4th International Symposium of VDM*, pp. 204-219, Noordwijkerhout, The Netherlands, 1991.

[VN 91] Veronika, D. and R. Nicholl, "EZ: A System for Automatic Prototyping of Z Specifications," *Software Development Methods, 4th International Symposium of VDM*, pp. 189-203, Noordwijkerhout, The Netherlands, 1991.



```
testasl/atm.asl                                                1

SPECIFICATION Automated_Teller
LEVEL Top_Level
TYPE
  Card,
  AccountNum,
  Positive IS TYPEDEF x: INTEGER (x >= 0)
CONSTANT
  CassetteSize: Positive,
  Account(Card): AccountNum,
  PIN_No(Card): INTEGER
VARIABLE
  Active: BOOLEAN,
  CurrentCard: Card,
  NumTwenties: Positive,
  Restricted(Card): BOOLEAN,
  Retries(Card): Positive,
  Balance(AccountNum): Positive,
  Limit(AccountNum): Positive
```

```
translating atm.asl
loading aslantest execu

Starting Aslantest inte

        Specification:


Type "h" for help
#
```

```
Transition Selected:
3  deposit(money:positive)
Enter Parameters sep. by commas:
Enter parameters here
```

Figure 1   XAslantest Screen Layout

Appendix A   Automated Teller Specification

```
SPECIFICATION automated_teller
LEVEL Top_Level

TYPE
  card,
  accountnum,
  positive IS TYPEDEF x: INTEGER (x >= 0)

CONSTANT
  cassettesize: positive,
  account(card): accountnum,
  pin_no(card): INTEGER

VARIABLE
  active: BOOLEAN,
  currentcard: card,
  numtwenties: positive,
  retries(card): positive,
  restricted(card): BOOLEAN,
  balance(accountnum): positive,
  limit(accountnum): positive


INITIAL
    ~active
  & (numtwenties = cassettesize)
  & FORALL c:card (~restricted(c) & retries(c) = 3)
  & FORALL a: accountnum (limit(a) = 30000)


INVARIANT
    (active -> ~restricted(currentcard))
  & FORALL c:card (restricted(c) -> (retries(c) = 0))


AXIOM
  FORALL c, c1:card (pin_no(c) = pin_no(c1) -> c = c1)


TRANSITION insert_card(c: card, pin: INTEGER)
  ENTRY
      ~active
    & ~restricted(c)
    & pin = pin_no(c)
  EXIT
      active
    & currentcard = c
    & retries(c) BECOMES 3
  EXCEPT
      ~active
    & ~restricted(c)
    & pin ~= pin_no(c)
    & retries(c) = 1
  EXIT
      restricted(c) BECOMES TRUE
    & retries(c) BECOMES 0
    & NOCHANGE(active)
  EXCEPT
      ~active
    & ~restricted(c)
    & pin ~= pin_no(c)
    & retries(c) > 1
  EXIT
      (retries(c) BECOMES (retries'(c) - 1))
    & NOCHANGE(active)
  EXCEPT
      ~active
    & restricted(c)
  EXIT
    NOCHANGE
```

```
TRANSITION deposit(money: positive)
  ENTRY
    active
  EXIT
    balance(account(currentcard')) BECOMES
      (balance'(account(currentcard')) + money)


TRANSITION withdraw(nt: positive)
  ENTRY
      active
    & (balance(account(currentcard)) - (2000 * nt)) >= 0
    & (limit(account(currentcard)) - (2000 * nt)) >= 0
    & numtwenties >= nt
  EXIT
      numtwenties = numtwenties' - nt
    & balance(account(currentcard')) BECOMES
        (balance'(account(currentcard')) - (2000 * nt))
    & limit(account(currentcard')) BECOMES
        (limit'(account(currentcard')) - (2000 * nt))
  EXCEPT
      active
    & (balance(account(currentcard)) - (2000 * nt)) >= 0
    & (limit(account(currentcard)) - (2000 * nt)) < 0
    & numtwenties >= nt
  EXIT
    NOCHANGE(numtwenties,balance,limit)
  EXCEPT
      active
    & (balance(account(currentcard)) - (2000 * nt)) < 0
    & numtwenties >= nt
  EXIT
    NOCHANGE(numtwenties,balance,limit)
  EXCEPT
      active
    & numtwenties < nt
  EXIT
    NOCHANGE(numtwenties,balance,limit)


TRANSITION end_session
  ENTRY
    active
  EXIT
    ~active


TRANSITION dayrollover
  ENTRY
    ~active
  EXIT
    FORALL a: accountnum (limit(a) = 30000)
  & FORALL c:card (IF ~restricted'(c)
                   THEN retries(c) = 3 FI)


TRANSITION resetcard(c: card)
  ENTRY
    restricted(c)
  EXIT
      restricted(c) BECOMES FALSE
    & retries(c) BECOMES 3


TRANSITION refill_machine
  ENTRY
    ~active
  EXIT
    numtwenties = cassettesize


END Top_Level
END automated_teller
```

Appendix B Session Log

```
valerie: aslantest atm.asl
translating atm.asl
loading aslantest executor...

Starting Aslantest interpreter...

        Specification automated_teller


Type "h" for help
# init start
Select the source of the start state:
    d   default - use the initial state of the spec
    f   file - read from a file
    k   keyboard - read from the terminal

Your choice: k
Type the predicate. When finished type '.' on line by itself:
  ~active
& pin_no(card1) ~= pin1
& ~restricted(card1)
& retries(card1) = numtries
& numtries > 0
& numtries <= 3
.
Processing new Start State...

declaring new constant  card1:card
declaring new constant  pin1:integer
declaring new constant  numtries:nonnegative
# /* display current state resulting from start predicate */
# vars
  limit:
FORALL  _p1:accountnum (
   limit(_p1) = ( IF true THEN limit$0(_p1) FI  )
)

  balance:
FORALL  _p1:accountnum (
   balance(_p1) = ( IF true THEN balance$0(_p1) FI  )
)

  retries:
FORALL  _p1:card (
   retries(_p1) = ( IF _p1 = card1 THEN numtries
    ELSE  IF true THEN retries$1 FI  )
)

  restricted:
FORALL  _p1:card (
   restricted(_p1) = ( IF _p1 = card1 THEN false
    ELSE  IF true THEN restricted$1 FI  )
)

  numtwenties:
numtwenties = numtwenties$0

  currentcard:
currentcard = currentcard$0

  active:
active = false

# display current
current state predicate:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
pin_no(card1) ~= pin1 & numtries >= 1 & numtries < 4
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# path
# /* no execution path because no transitions executed yet */
# init result
Select the source of the result state:
```

```
    d   default - use the final state of the execution
    f   file - read from a file
    k   keyboard - read from the terminal

Your choice: k
Type the predicate. When finished type '.' on line by itself:
retries(card1) = numtries - 1
.
# display result
current result predicate:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
retries(card1) = numtries - 1
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# sequence
Enter the sequence of transitions, one per line.
Finish by typing a single '.' at the beginning of a line.
insert_card(card1,pin1)
dayrollover
.
Show the Sequence?[(y)es or (n)o]: n
Pause between transitions? [(y)es or (n)o]: y
Executing the sequence...
entry condition for insert_card not satisfied
Trying Except Conditions
Evaluate condition:
------------
numtries = 1
------------
(t)rue, (f)alse, or (n)either:
# n
Assume (t)rue or (f)alse:
# f
Evaluate condition:
------------
numtries >= 2
------------
(t)rue, (f)alse, or (n)either:
# n
Assume (t)rue or (f)alse:
# t
executing insert_card(card1,pin1)
Sequence Breakpoint:
# /* display current state after insert_card(card1,pin1) */
# vars
  limit:
FORALL  _p1:accountnum (
   limit(_p1) = ( IF true THEN limit$0(_p1) FI  )
)

  balance:
FORALL  _p1:accountnum (
   balance(_p1) = ( IF true THEN balance$0(_p1) FI  )
)

  retries:
FORALL  _p1:card (
   retries(_p1) = ( IF _p1 = card1 THEN numtries - 1
    ELSE  IF true THEN retries$1 FI  )
)

  restricted:
FORALL  _p1:card (
   restricted(_p1) = ( IF _p1 = card1 THEN false
    ELSE  IF true THEN restricted$1 FI  )
)

  numtwenties:
numtwenties = numtwenties$0

  currentcard:
currentcard = currentcard$0

  active:
active = false

```

```
# display current
current state predicate:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   numtries ~= 1
&  pin_no(card1) ~= pin1
&  numtries >= 2
&  numtries < 4
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# path
Added conditions:
numtries >= 2
Start: insert_card(card1, pin1)
Executed Except #2
No assumptions
Finished
# /* Next: execute dayrollover */
# go
executing dayrollover
Sequence completed
# /* display state after sequence */
# vars
  limit:
FORALL  _p1:accountnum (
   limit(_p1) = ( IF true THEN 30000 FI  ) )

  balance:
FORALL  _p1:accountnum (
   balance(_p1) = ( IF true THEN balance$0(_p1) FI  )
)

  retries:
FORALL  _p1:card (
   retries(_p1) =
      ( IF ~restricted(_p1) & _p1 = card1 THEN 3
       ELSE  IF ~restricted(_p1) THEN 3
       ELSE  IF true THEN retries$2(_p1) FI  )
)

  restricted:
FORALL  _p1:card (
   restricted(_p1) = ( IF _p1 = card1 THEN false
    ELSE  IF true THEN restricted$1 FI  )
)

  numtwenties:
numtwenties = numtwenties$0

  currentcard:
currentcard = currentcard$0

  active:
active = false

# display current
current state predicate:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   numtries ~= 1
&  pin_no(card1) ~= pin1
&  numtries >= 2
&  numtries < 4
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# path
Added conditions:
numtries >= 2
Start: insert_card(card1, pin1)
Executed Except #2
No assumptions
Finished
Start: dayrollover
No assumptions
Finished
# check result
Attempting to simplify result state...

current state -> result state: False
```

```
# /* Therefore, the requirement is not satisfied */
# /* because dayrollover reset all non-restricted cards */
# quit
Return to Unix? (n/y)
# y
Bye.
valerie:
```