N91-20023 p.28

1990 NASA/ASEE SUMMER FACULTY FELLOWSHIP PROGRAM

JOHN F. KENNEDY SPACE CENTER
UNIVERSITY OF CENTRAL FLORIDA

FORMAL SPECIFICATION OF HUMAN-COMPUTER INTERFACES

PREPARED BY:	Dr. Brent Auernheimer
ACADEMIC RANK:	Associate Professor
UNIVERSITY AND DEPARTMENT:	California State University - Fresno Department of Computer Science
NASA/KSC	
DIVISION:	Data Systems
BRANCH:	Real Time Systems
NASA COLLEAGUE:	Mr. Les Rostosky
DATE:	August 10, 1990
CONTRACT NUMBER:	University of Central Florida NASA-NGT-60002 Supplement: 4

Acknowledgements

The support of my NASA colleagues Oscar Brooks, Steve Bryan, Bill Drozdick, Linda Koch, Les Rostosky, Bill Sloan, Lynn Svedin, and Larry Wilhelm is gratefully acknowledged. Thanks also to Steve Eckmann and Richard Kemmerer of the University of California, Santa Barbara for providing helpful comments and ideas.

Abstract

This report describes a high-level formal specification of a human-computer interface. A typical window manager is modeled. Previous work is reviewed and the ASLAN specification language is described. Top-level specifications written in ASLAN for a library and a multiwindow interface are discussed.

Table of Contents

I **Introduction**
I.1 Organization of the Paper
I.2 Specification and Verification Terminology
I.3 Previous Work
I.4 Formal Techniques and the Development of User Interfaces
I.5 Two Views of Specifications
I.6 Testing vs. Proving Specifications

II **Formal Specification**
II.1 The Aslan Specification Language
II.2 The Aslan Approach
II.3 A Specifier-friendly Feature

III. **Specification of A Multiwindow User Interface**
III.1 Overview
III.2 Types
III.3 Constants
III.4 State Variables
III.5 Definitions (Macros)
III.6 Initial Conditions
III.7 Critical Correctness Requirements
III.8 State Transitions
III.8.1 Window Closing (Iconifying)
III.8.2 Window Opening
III.8.3 Window Destruction
III.8.4 Window Creation
III.8.5 Shifting Input Focus
III.8.6 Moving Windows
III.8.7 Window Resizing
III.8.8 Window Restacking

IV **Concluding Remarks**

I. Introduction

I.1 Organization of the Paper

This paper documents an attempt to formally specify a multiwindow user interface. The paper is organized as follows:

Section I briefly reviews the foundation laid in last summer's work and discusses motivations and expected results of using formal specification techniques on user interfaces. Section II[1] introduces the ASLAN formal specification language through the example in Appendix A. Section III is a detailed look at an abstract specification of a typical multiwindow interface. The formal specification discussed in Section III is in Appendix B. Finally, Section IV contains concluding remarks and recommendations.

I.2 Specification and Verification Terminology

Specifications are statements about the functionality of a system. Specifications express what is a system is to accomplish, not how it is to do it. In this paper, *formal specifications* are assertions about the behavior of a system. *Critical correctness criteria* are assertions that the specification and all refinements and implementations are to satisfy. *Formal verification* techniques demonstrate that implementations satisfy their specifications. In addition, it is useful to show that specifications meet their critical correctness criteria. This is sometimes called *design verification*. Neumann explains [12]:

> Formal verification has often been talked about as a technique for demonstrating consistency between code and assertions about that code, in some cases between code and specifications. Somewhat less popular has been the easier notion of using formal verification to demonstrate that a set of formal specifications is consistent with its formally axiomatized requirements, i.e., carrying out *design verification*.

I.3 Previous Work

This report describes a continuation of work on formalisms for user interface specification and design described in [1]. That work examined several recent research results in human-computer interaction (HCI) that may be applicable to NASA applications. One of the results examined was the formal specification of direct manipulation user interfaces for a secure military message system [7, 8].

[1] also contains an introduction to formal specification and verification including objections to the approach and reasonable expectations. It was recommended that a pilot study using formal techniques on a small, well-defined piece of a user interface be done. The subsystem to be specified should have clear, high-level correctness properties that must be met. The specification given in Appendix B of this paper is the portion of a user interface that manipulates windows. The correctness property that must be maintained is that users are not allowed to

[1] The information in II.2 and Appendix A is based on a portion of a paper by the author and Daniel Stearns, "Using the ASLAN specification language in undergraduate software engineering courses," submitted to *Computer Science Education*, July 1990

move, close, or resize certain windows. This correctness property was derived from current interface prototypes developed at NASA KSC.

Using formal specification techniques is costly. Benefits are realized when there are readily identifiable, critical correctness properties that must hold. This is the case for portions of proposed interfaces to shuttle and space station software.

Further, it is not necessary to carry out formal specification and verification to their full extent to realize benefits. Finding an appropriate level of formality and analysis can result in systems that users can have a high degree of confidence in. To summarize [1, section IV.2.2]:

> The general goal is to lower expectations for formal specification – the goal isn't necessarily provably correct software – but to specify important functionality and correctness criteria in a way that is reviewable by software engineers and integrates usefully in other software development efforts.

I.4 Formal Techniques and the Development of User Interfaces

As noted in [1], there is some controversy about the usefulness of formal specification and verification techniques in general. Because modern interfaces are visually complex and becoming more aural, some HCI researchers believe that prototyping and user interface management system (UIMS) are the correct approach to interface specification and development. Fischer is a critic of formal specification of interfaces [6, p. 50,51]:

> Static specification languages have little use in HCI software design. First, detailed specifications do not exist. Second, the interaction between a system and its user is highly dynamic and aesthetic, aspects that are difficult, if not impossible, to describe with a static language. ...A prototype makes it much easier and productive for designers and users to cooperate because users do not have to rely on written specifications, which do not indicate an interface's qualities. ...Validation and verification methods from other software domains have limited use in HCI. Formal correctness is crucial, but it is by no means a sufficient measure of the effectiveness and success of an HCI system.

Fischer's statements are true for most interface development efforts. Formal techniques aimed at low level or aesthetic portions of user interfaces may not be productive.

However, for critical aspects of NASA interfaces, such as alarm areas, critical correctness requirements are apparent and easily expressed. By using techniques employed in the specification of secure systems, formal specification becomes a valuable approach.

Further, it is possible to combine formal specifications of functionality with usability specifications. Carrol and Rosson have studied the design process and recommend an iterative approach of developing and integrating functionality and usability specifications [5, p. 1]:

> Much has been said about this "usability" problem regarding current interface designs. Less has been said about how to solve the problem. ...we develop an approach to the problem based on *usability specifications:* precise, testable statements

of performance goals for typical users carrying out tasks typical of their projected use of the system.

I.5 Two Views of Specifications

Most specification efforts target one of two goals: an executable specification (prototype system), or a proof that a specification meets critical correctness requirements ("design verification"). HCI specification are usually developed with the intent of having a prototype system that can be "checked for certain undesirable properties" [8, p. 211]. Because these specifications are to result in realistic prototypes, it is necessary to specify low-level events such as mouse clicks and beeps. Not surprisingly, it is not feasible to combine a huge quantity of implementation details with design verification of the specification.

The approach explored in this paper is to write abstract specifications and critical correctness requirements for a portion of a user interface without getting bogged down in implementation details. High-level specifications have been very successful in the field of secure systems. The formal specifications for these systems are shown to be consistent with their critical correctness criteria without becoming bogged down in implementation details. For example, in a short, high-level specification, Kemmerer shows fundamental flaws in a cryptosystem [10].

Although Jacob focuses on executable specifications and prototyping, he briefly discusses extensions to his techniques [8, p. 237]:

> In designing a secure message system, it is desirable to prove assertions about the security of the system formally. Such proofs are usually based on a formal specification of the system (with the proviso that the final software and hardware correctly implement the specification). This approach has not generally been used at the user interface level, but, if one had a formal specification of the user interface, it would be possible to provide proofs about the user interface.

I.6 Testing vs. Proving Specifications

A common and persistent criticism of formal specification and verification is that the quantity of proofs that must be done is overwhelming. a formal specification and statement of correctness, it is possible to gain insights into the system and confidence in the specification without performing proofs. The informal analysis of a formal specification can be a valuable technique for communication between software engineers.

More formally, symbolic execution tools have been developed [11]. Specifications can then be tested against correctness requirements. Testing specifications allows software engineers to play what-if games with the specification and may result in the discovery of system states that do not satisfy the correctness requirements.

These tools have been successfully used in the development of secure systems. Kemmerer explains the use of the Inatest tool on a cryptosystem specification [10, p. 453]:

> With the Inatest tool, it is possible to introduce assumptions about the system interactively, execute sequences of transforms, and check the results of these execu-

tions. This provides the user with a rapid prototype for testing properties of the cryptographic facilities ...

II. Formal Specification

II.1 The Aslan Specification Language

Software engineers' lack of exposure to formal specification systems is particularly disturbing in light of increasing dependence on critical software systems. Neumann describes examples of problems with specifications in four application areas: human safety, reliability, security, and user interfaces [12]. Neumann concludes (emphasis added):

> There are many contributions that good software engineering practice could have made to the prevention or minimization of these and many other problems. In particular, the sound use of system structuring, *specification languages capable of meaningful abstraction, and rigorous analysis of specifications* could all have had significant effects.

The ASLAN formal specification language is a partial solution to the above problem. Software engineers can use ASLAN to formally specify complex systems and develop their specifications through arbitrary levels of abstraction. When a specification is passed through the ASLAN Language Processor (ALP), software engineers receive a set of correctness conjectures.

The following sections describe features of ASLAN using a specification of a library as an example. The library example has been used in many formal specification workshops. A library specification written in the InaJo language appears in [11].

II.2 The ASLAN Approach

The ASLAN language is built on first order predicate calculus. Systems being specified are thought of as being in *states*, defined by the values of the system variables. Logical assertions are used to define the critical correctness requirements that must hold in every state and those that must hold between two consecutive states. The former are *state invariants*, while the latter are *constraints* on state transitions.

To prove that a specification satisfies its invariant and constraints, the ALP generates *correctness conjectures*. Correctness conjectures are logical statements whose proof ensures the correctness of the specification with respect to the invariant and constraint.

Appendix A contains a high-level specification of a library. Although the library could be further specified through more detailed levels of specification, only the top-level specification will be examined here.

The state variables for this system appear in the `VARIABLE` section of the specification. `Library` is a variable whose type is a collection of `Book`. At this level, `Book` is left as an unspecified type. A state variable `Checked_Out` maps each book into the boolean domain, while `Number_Out` maps library users to the the number of books they have checked out.

The specification contains an initial assertion defining possible starting states of the library. This assertion states that the library is initially empty, that no users have books out, and that, indeed, no books are checked out.

The library specification contains an invariant assertion to specify the essential properties that the system must have. The invariant states that

- when a book is checked out, it cannot be available. Similarly, an available book cannot simultaneously be checked out.
- the limit on the number of books checked out by any user is enforced
- no user has more than one copy of the same book checked out

Clearly, we want the initial state of the system to fulfill the invariant. The ALP will generate a logical implication that

$$\text{initial} \rightarrow \text{invariant}$$

The particular correctness conjecture generated is

```
    Library = EMPTY
  & FORALL u: User (Number_Books(u) = 0)
  & FORALL b: Book (~Checked_Out(B))
->
    FORALL b:Book( b ISIN Library ->
                     Checked_Out(b) & ~Available(b)
                   | ~Checked_Out(b) & Available(b))
  & FORALL u:User (Number_Books(u) <= Book_Limit)
  & FORALL u:User,b1, b2:Book(
                     Checked_Out_To(u, b1)
                   & Checked_Out_To(u, b2)
                   & Copy_Of(b1, b2)
                   -> b1 = b2)
```

It is up to the specifier, possibly with the help of a theorem prover, to prove the above correctness conjecture.

An empty library is not very interesting. The specification must define how the library can expand; that is, how the library can change from a current ('old') state to a new state in which more books are present.

Allowable state changes are specified as `transitions`. An ASLAN transition consists of a precondition (`ENTRY`) and a postcondition (`EXIT`). Transitions in the library example have only postconditions. The ALP assumes that omitted preconditions are true.

The `Add_A_Book` transition allows the library to expand. Because `Add_A_Book` does not have an entry assertion, it can be applied at any time. The exit assertion states the effect the application of the transition has on the state variables. It asserts that

- the user adding the book must be a member of the library staff,
- and assuming the user is a staff person and the book isn't already in the library (the apostrophe is the ASLAN notation for 'old value'),
- the book is added to the library
- the book is not checked out
- and in particular, this book has *never* been checked out

How can the specifier be assured that the Add_A_Book transition meets the correctness requirements embodied in the invariant? It must be proved that if the invariant holds in the current (old) state, and the transition is applied, then the invariant will hold in the new state. That is,

$$\text{invariant' } \& \text{ entry' } \& \text{ exit} \rightarrow \text{invariant}$$

Note that in the antecedent, the invariant and the entry assertion are evaluated in the old state. The exit assertion and the consequent are evaluated in the new state.

ASLAN specifications can be made up of several levels of abstraction. Given a multilevel specification, the ALP generates additional correctness conjectures that ensure that types, variables, and transitions are properly refined, and that the correctness requirements are met at every level of abstraction. Details are found in [4].

Ideally, ASLAN should be used to specify increasingly concrete levels of abstraction. The resulting specification would be a high level specification defining the system as an abstract data type, followed by intermediate levels leading to a low level specification close to code level. In this most detailed specification level, the transitions' entry and exit assertions become the the pre- and postconditions of programming language level procedures which implement them.

II.3 A Specifier-friendly Feature

Expressions in ASLAN look like first order logic assertions for a simple reason: the techniques and expressive power of first order logic can be used to prove correctness conjectures.

Unfortunately for specifiers with a programming background, the semantics of first order logic are not the same as those of procedural programming languages such as Pascal and C. Consider an alternate version of the Return transition:

```
TRANSITION Return(B: Book)
 EXIT
    Checked_Out'(B) ->  Checked_Out(B) = FALSE
                      & Number_Books(Responsible'(B)) =
                           Number_Books(Responsible'(B) - 1)   ,
```

The above exit assertion is written in a purely 'logical' form. Recall that the ALP will construct a correctness conjecture whose proof ensures the invariant holds after the application of Return:

```
    FORALL b:Book (b ISIN Library' ->
                     Checked_Out'(b) & ~Available'(b)
                   | ~Checked_Out'(b) & Available'(b))
  & FORALL u:User (Number_Books'(u) <= Book_Limit)
  & FORALL u:User,b1, b2:Book(
                     Checked_Out_To'(u, b1)
                   & Checked_Out_To'(u, b2)
                   & Copy_Of(b1, b2)
                   -> b1 = b2)
&
    Checked_Out'(B) ->  Checked_Out(B) = FALSE
                      & Number_Books(Responsible'(B)) =
                            Number_Books(Responsible'(B) - 1)
->
    FORALL b:Book (b ISIN Library ->
                     Checked_Out(b) & ~Available(b)
                   | ~Checked_Out(b) & Available(b))
  & FORALL u:User (Number_Books(u) <= Book_Limit)
  & FORALL u:User,b1, b2:Book(
                     Checked_Out_To(u, b1)
                   & Checked_Out_To(u, b2)
                   & Copy_Of(b1, b2)
                   -> b1 = b2)
```

It is a simple paper and pencil exercise to show that the conjecture generated cannot be proved. In particular,

- The new values of **Available** and **Checked_Out_To** are mentioned in the consequent invariant. Nothing can be proved about the new values of **Available** and **Checked_Out_To**. Neither variable was mentioned in the exit assertion of **Return**, and the antecedent invariant only relates old values of the state variables.
- Because the exit assertion was written as a logical implication, we have not specified what will happen when Checked_Out'(B) is false. In particular, if the book B was not checked out, the new value of Checked_Out(B) could be true or false. Also, the values of Checked_Out for books other than B are unspecified!

- Similarly, we have not specified the new values of Number_Books for all users *not* responsible for the particular book B being returned. In particular, the new value of Number_Books for such users could be any integer.

These differences between the semantics of logic and those of programming languages have caught professional specification writers by surprise [14].

The ASLAN language provides constructs that operate like programmers tend to think logical operations *should* operate. Corresponding to logical implication is the ASLAN IF-THEN-ELSE-FI, corresponding to disjunction is the ALT (alternative) statement, and corresponding to equality is the BECOMES statement. Details are found in [3, 4].

In addition, ASLAN supplies the implicit 'no changes' for variables mentioned in the invariant and constraint, but not mentioned in a particular transition.

These language features allow specifiers to write specifications in a more natural way. Readers should compare the Return transition above with the version in the appendix.

Preliminary work on extending ASLAN to facilitate specification of real-time systems is documented in [2].

III. Specification of a Multiwindow User Interface

III.1 Overview

Appendix B contains an ASLAN specification of an interface commonly provided by window managers running on the X window system [9, 13, 15]. Windows can be created, deleted, opened, closed, resized, moved, brought to the foreground, and be made the target of user input.

The specification has one feature not usually found in multiwindow interfaces: dedicated, reserved ("special") windows that cannot be moved, closed, or covered. Displays proposed for shuttle ground support software will have such areas.

This specification was written to be an abstract description of the operations provided by a window manager to a user. Note that pixels and mice, usually associated with such interfaces, are not mentioned. It is sometimes hard to determine an appropriate level of abstraction for a top-level specification. A guideline is that the most abstract specification be such that critical functionality and correctness requirements can be expressed in a form that is readily understandable and easily manipulable. In addition, top-level specifications should not restrict possible implementations and refinements.

Although only a top-level specification of the interface is provided, it is clear that more detailed levels of refinement could introduce implementation details such as pixels and mice.

Each major syntactic unit of the specification will be discussed in turn. Sections III.2 through III.8.8 refer directly to Appendix B. The following lexical convention is used: constants and ASLAN keywords are uppercase, type identifiers begin with uppercase, variables and definitions are lower case.

III.2 Types

Six unspecified types are declared to represent classes of system objects that require no elaboration at this level. For example, `Processes` can be associated with `Windows`, however at this level of abstraction it is not important how either is implemented. Further, how windows look on the screen (their `Representations`, `Sizes`, and `Locations`) are deferred.

The `Display_Levels` type represents the stacking level of windows on the screen. It is tempting to define `Display_Levels` as a synonym for integer. This would restrict possible implementations. As discussed in III.3, it is only necessary that `Display_Levels` have a less-than-or-equal ordering.

The state of a window is a simple enumerated type. The layout of a window is a structure of three fields: a location, size, and a representation. As discussed in section III.4, windows have a layout for when they are open, and another layout (an icon) for when they are closed. Finally, the contents of the current screen is of type `Displays` – a set of window layouts. The current display along with current stacking levels for the active windows defines the look of the screen.

III.3 Constants

Constants are unchanging mappings. For example, `INITIAL_OPEN_LAYOUT` associates a de-

fault look for windows opened for processes. With INITIAL_CLOSED_LAYOUT and INITIAL_STATE, the window manager determines the look of a window when it is created for a process.

OVERLAPS is an important relation that maps two layouts to true or false. The intent is that refinements of OVERLAPS will check to see if any of the first layout overlaps the second. This constant relation is useful in determining if windows are on the screen (overlap the BACKGROUND) and if they would obscure a restricted window.

LESS_OR_EQUAL is similar to OVERLAPS. This boolean constant maps window stacking levels. If Display_Levels is subsequently refined to be the integers, this constant may turn into nothing more than $\leq$.

SPECIAL is a boolean function that determines if a window is restricted. It is made a constant in this specification so that the mechanisms for making windows restricted or not can be omitted. It is reasonable that SPECIAL could be changed to be a state variable and state transitions for its manipulation be added.

III.4 State Variables

The value of the state variables determine the state of the interface. These values are changed by the application of the state transitions discussed in section III.8.

Windows are created for processes. The mapping of processes to windows is represented in process. When a process is bound to a window, the window will inherit the initial open and closed layouts from the process. These layouts (open_layout, closed_layout) can be changed by the resize and move state transitions. Associated with each window is also a stacking level display_level.

The layouts currently active on the screen are in display. The window selected to receive input is determined by the value of input_focus.

III.5 Definitions (Macros)

ASLAN definitions are macros used to make state transitions more understandable. Eight definitions are given to represent mundane events such updating the display and changing layout fields.

to_top_level is interesting because it specifies that the stacking level of its argument is to be less than all other windows, and that the relationship between other windows should remain as it was before the argument was made uppermost window.

There are two things to note. First, to_top_level may be restricting future implementations. It is not necessary for the argument's stacking level to be strictly less than that of all other windows, just that it be less than the level of all windows in its stack. That is, the topmost windows of independent stacks on the screen could have the same display_level. It is an interesting exercise to rewrite the definition to allow this.

Second, although it is specified that the relationships of other windows remains as they were before the state transition, it is possible that the value of display_level for each window has changed! This allows refinements and eventual implementations flexibility in assigning display levels - any implementation that has the argument window ending up on top, and doesn't

rearrange the other windows meets this specification.

`set_location` and `set_size` manipulate one field of a particular window's current layout. Care is taken that other fields for this window, and layouts for other windows are unchanged.

The `update_display` definitions specify the addition and deletion of layouts to the current display.

III.6 Initial Conditions

The `INITIAL` assertion describes the state of the interface when the system is first brought up. An informal reading of the assertion is "nothing is on the screen, and all windows are inactive, and windows that are created for processes will be on somewhere on the screen."

III.7 Critical Correctness Requirements

The critical correctness requirements are expressed in the `INVARIANT`. This assertion is to be true when the system is started, and continue to hold in every state the system can reach starting at the initial state and using the state transitions described in III.8. The assertion consists of three conjuncts. The first says that every layout on the screen has to be associated with an active window. The second asserts that every current layout must be at least partially on the screen. The third states that restricted windows are not covered.

III.8 State Transitions

The following subsections describe the eight state transitions. Since none of the transitions have explicitly stated `ENTRY` assertions, there are no restrictions on when the transitions can be applied. This corresponds to typical window managers – it is possible, for example, to attempt to close windows at any time.

Although there are no restrictions on when the transitions can be applied, it is not always the case that applying them has any effect on the state of the system. For example, most window managers will allow a user to close an already closed window. From the user's view, there is no change in the state of the display.

The specifications for the state transitions are written with this in mind. The style used is as follows: an exit assertion is a disjunct of two clauses joined by the ASLAN `ALT` operator. The first clause specifies the effect of the state transition when variables are changed (the closing of an open window, for example) and the second clause specifies that no variables change.

The `ALT` operator is logical disjunction augmented by statements specifying that unmentioned state variables do not change. These statements are generated automatically by the ALP [3, 4].

When reading the transitions it is important to pay careful attention to the use of the old-value operator (apostrophe). The following sections will focus on the first disjunct of each transition's exit assertion.

III.8.1 Window Closing (Iconifying)

To close a window `w`, it is necessary that `w` be open, that it is not a restricted window, that `w`'s new state is closed (and that the states of other windows are unchanged), that `w`'s open

layout is taken off the screen and replaced by its icon, and that the icon not be hidden.

III.8.2 Window Opening

`open_window` is symmetric to `close_window`.

III.8.3 Window Destruction

To destroy a window `w`, it is necessary that `w` active before the state transition and unused afterwards, that `w` is not a restricted window, and that the layout of `w` be removed from the screen.

III.8.4 Window Creation

Windows are created for and associated with processes. To create a window for process p it is necessary that there exists a window `w` that was inactive before the state transition and will become active. This window will inherit its initial state and layouts from p. `w` will be associated with p, and `w`'s current layout will be added to the display uncovered by other windows or icons.

III.8.5 Shifting Input Focus

This transition assumes that only one window at a time can be the target of user input. It is a simple transition that checks that only active windows can receive input.

III.8.6 Moving Windows

The `move` transition looks more complicated than it is. There are two symmetric cases for when the window to be moved, `w`, is open and closed. In either case, `w` cannot be a restricted window and the current display is modified. If `w`'s state is open then its location is changed, `w` must still be on the screen, and `w` cannot overlap any special windows. The case when `w` is closed is symmetric.

III.8.7 Window Resizing

This transition states that only ordinary, open windows can be resized. In addition, a window cannot be resized to overlap a restricted window.

III.8.8 Window Restacking

To bring an active window `w` to the foreground, the display must be changed, and the act of bringing `w` to the foreground must not overlap a restricted window.

IV. Concluding remarks

This paper has discussed formal specification of user interfaces. The particular approach taken was to construct an abstract, state-machine model of the interface using the ASLAN specification language. Emphasis was placed on defining essential functionality and critical correctness requirements without introducing implementation details.

The resulting specification defines the functionality of a typical window manager (Appendix B). The specification can be the foundation of several further activities:

- The correctness conjectures generated by the ASLAN language processor could be proved. Successful proofs would show that the specification satisfies its critical correctness criteria.

  Failed attempts to prove correctness conjectures have lead to new insights into the system being specified. Failed proofs can show misunderstandings in functionality, inconsistency, and incompleteness. These insights can be especially valuable to software engineers as they work toward defining essential functionality and correctness of a system.

- The specification can be expanded. It would be useful to refine the top-level specification into lower, more detailed specifications. A challenge is to refine the specification down to an implementation level at which objects such as pixels, mouse clicks, and scroll-bars are used. New techniques would have to be developed to maintain readability and understandability while handling the amount of detail at low levels.

- The specification could serve as inspiration for a specification of a particular part of the proposed shuttle/space station ground software. This is a promising area for further research. After informal requirements for, say, protected alarm areas on screens are developed, an effort should be made to formally specify their actions and correctness requirements.

- The specification could be tested. A symbolic execution tool for ASLAN specifications should be constructed.

Formal specification of user interfaces is not cost effective for most projects. However, for highly structured interfaces whose performance is critical (such as the NASA interfaces being developed) formal specification can play a valuable role in unambiguously defining functionality and providing confidence in meeting correctness requirements. There is considerable interest in formal techniques and proofs of correctness among developers of critical interfaces.

References

[1] B. Auernheimer. Formalisms for user interface specification and design. In NASA CR-166837, NASA Kennedy Space Center (October 1989).

[2] B. Auernheimer and R. A. Kemmerer. RT-ASLAN: a specification language for real-time systems. *IEEE Transactions on Software Engineering,* SE-12, 9 (September 1986).

[3] B. Auernheimer and R. A. Kemmerer. Procedural and nonprocedural semantics of the ASLAN formal specification language. *Proceedings of the nineteenth annual Hawaii international conference on system sciences* (January 1986).

[4] B. Auernheimer and R. A. Kemmerer. ASLAN users manual. Technical report TRCS84-10. Department of Computer Science, University of California, Santa Barbara (March 1985).

[5] J. Carroll and M. Rosson. Usability specifications as a tool in iterative development. In *Advances in human-computer interaction*, vol. 1. H. Hartson, ed. (1985).

[6] G. Fischer. Human-computer interaction software: lessons learned, challenges ahead. *IEEE Software* 6, 1 (January 1989).

[7] R. J. K. Jacob. A specification language for direct-manipulation user interfaces. *ACM Transactions on Graphics* 5, 4 (October 1986).

[8] R. J. K. Jacob. An executable specification technique for describing human-computer interaction. In *Advances in human-computer interaction*, vol. 1. H. Hartson, ed. (1985).

[9] O. Jones. *Introduction to the X window system.* Prentice-Hall (1989).

[10] R. A. Kemmerer. Analyzing encryption protocols using formal verification techniques. *IEEE Journal on Selected Areas in Communications*, 7, 4 (May 1989).

[11] R. A. Kemmerer. Testing formal specifications to detect design errors. *IEEE Transactions on Software Engineering*, SE-11, 1 (January 1985).

[12] P. G. Neumann. Flaws in specification and what to do about them. *ACM SIGSOFT Engineering Notes*, 14, 3 (May 1989).

[13] O' Reilly & Associates, Inc. *X Window System Series*, vols. 0-7.

[14] R. Platek and D. Sutherland. The semantics of the Freiertag MLS information flow tool and its impact on design verification: some SCOMP examples. Unpublished report, Odyssey Research Associates, Inc., Ithaca (December 1983).

[15] R. W. Scheifler and J. Gettys. The X window system. *ACM Transactions on Graphics* 5, 2, (April 1986).

Appendix A
Specification of A Library

```
SPECIFICATION Library
LEVEL Top_Level

TYPE
    User,
    Book,
    Book_Title,
    Book_Author,
    Book_Collection IS SET OF Book,
    Titles IS SET OF Book_Title,
    Pos_Integer  IS TYPEDEF i:INTEGER (i>0)

CONSTANT
    Title(Book):Book_Title,
    Author(Book):Book_Author,
    Library_Staff(User):BOOLEAN,
    Book_Limit:Pos_Integer

DEFINE
    Copy_Of(B1,B2:Book) : BOOLEAN ==
             Author(B1) = Author(B2)
          & Title(B1) = Title(B2)

VARIABLE
    Library:Book_Collection,
    Checked_Out(Book):BOOLEAN,
    Responsible(Book):User,
    Number_Books(User):INTEGER,
    Never_Out(Book):BOOLEAN,


DEFINE
    Available(B:Book):BOOLEAN ==
         B ISIN Library & ~Checked_Out(B),
    Checked_Out_To(U:User,B:Book):BOOLEAN ==
            Checked_Out(B)
         & Responsible(B)=U
```

```
INITIAL
       Library = EMPTY
    & FORALL u:User (Number_Books(u) = 0)
    & FORALL b:Book (~Checked_Out(b))

INVARIANT
      FORALL b:Book(b ISIN Library ->
                        Checked_Out(b) & ~Available(b)
                      | ~Checked_Out(b) &  Available(b))
    & FORALL u:User(Number_Books(u) <= Book_Limit)
    & FORALL u:User,b1,b2:Book(
            Checked_Out_To(u,b1)
         & Checked_Out_To(u,b2)
         & Copy_Of(b1,b2)
              -> b1=b2)

TRANSITION Check_Out(U:User,B:Book)
  EXIT
        Available'(B)
      & Number_Books'(U) < Book_Limit
      &  IF FORALL B1:Book (Checked_Out_To'(U,B1) -> ~Copy_Of(B,B1))
         THEN
          Number_Books(U) BECOMES (Number_Books'(U) + 1)
            &  (Checked_Out(B) BECOMES TRUE)
            &  (Responsible(B) BECOMES U)
            &  (Never_Out(B) BECOMES FALSE)
         FI

TRANSITION Return(B:Book)
  EXIT
    ( IF Checked_Out'(B)
         THEN Checked_Out(B) BECOMES FALSE
              & Number_Books(Responsible'(B))
                  BECOMES (Number_Books(Responsible'(B)) - 1)
      FI)

TRANSITION Add_A_Book(U:User,B:Book)
  EXIT
    ( IF Library_Staff(U)
       & B ~ISIN Library'
        THEN Library = Library'  UNION  {B}
             &  Checked_Out(B) BECOMES FALSE
             &  Never_Out(B)  BECOMES TRUE
```

```
      FI)


TRANSITION Remove_A_Book(U:User,B:Book)
  EXIT
    (IF Library_Staff(U)
       & Available'(B)
             THEN Library = Library' SET_DIFF {B}
    FI)


END Top_Level
END Library
```

Appendix B
Specification of a Multiwindow Interface

```
SPECIFICATION window_interface
INHIBIT /* do not produce correctness conjectures */
LEVEL Top_Level

/* Brent Auernheimer -- July 1990

   This is a high-level specification written using the Aslan
   specification language of a window-based
   user interface. Mice are not explicitly mentioned.

   Note that a ' ('prime') is the old-value operator.  That is,
   is x is a variable, then x' represents its value before the
   application of a transition. An unprimed x represents the
   new-value of x.

   This user interface is typical of window managers running
   on X. One added feature is SPECIAL windows
   which cannot be closed (iconified), moved, or covered by
   other windows or icons.

   Notational conventions --  alphanumeric tokens are lowercase
   except for the following:

      * Keywords and constants are uppercase.
      * Type identifiers begin with uppercase.

*/

TYPE
    Windows, Processes, Locations, Sizes, Representations, Display_Levels,
    States IS (OPEN, CLOSED, UNUSED),
    Layouts IS STRUCTURE OF
      (location: Locations, size: Sizes, rep: Representations),
    Displays IS SET OF Layouts

CONSTANT
    NULL_PROCESS: Processes,
    INITIAL_OPEN_LAYOUT(Processes): Layouts,
```

```
    INITIAL_CLOSED_LAYOUT(Processes): Layouts,
    INITIAL_STATE(Processes): States,

    /* OVERLAPS is to be  true if first argument overlaps the second */
    OVERLAPS(Layouts, Layouts): BOOLEAN,
    BACKGROUND: Layouts, /* windows must overlap the background */
    SPECIAL(Windows): BOOLEAN, /* some windows cannot be covered */

    /* the smallest display_level is the window closest to the top,
       the largest is the window buried the deepest */
    LESS_OR_EQUAL(Display_Levels, Display_Levels): BOOLEAN

VARIABLE
    process(Windows): Processes,
    open_layout(Windows): Layouts,
    closed_layout(Windows): Layouts,
    state(Windows): states,
    input_focus(Windows): BOOLEAN,
    display: Displays,
    display_level(Windows): Display_Levels

DEFINE
   /* DEFINitions are macros used to make state transitions easier to read */

   to_top_level(w: Windows): BOOLEAN ==
      /* w becomes the topmost window ... */
      FORALL w2: Windows (
         (w ~= w2)
             -> LESS_OR_EQUAL(display_level(w), display_level(w2))
              & display_level(w) ~= display_level(w2))

      /* all other windows maintain their previous relationship */
    & FORALL w1, w2: Windows (
         (w1 ~= w & w2 ~= w) -> (
               (LESS_OR_EQUAL(display_level'(w1), display_level'(w2))
                   -> LESS_OR_EQUAL(display_level(w1), display_level(w2)))
             & (LESS_OR_EQUAL(display_level'(w2), display_level'(w1))
                   -> LESS_OR_EQUAL(display_level(w2), display_level(w1))))),

    /* note that square brackets are used to select fields from
       structure typed variables */

    set_location(w: Windows, s: States, l: Locations): BOOLEAN ==
```

```
  ((s = OPEN)
     -> FORALL w1: Windows (
           (w = w1 -> open_layout(w)[location] = l
                    & open_layout(w)[size] = open_layout'(w)[size]
                    & open_layout(w)[rep] = open_layout'(w)[rep])
         & (w /= w1 -> open_layout(w1) = open_layout'(w1)))
      & NoChange(closed_layout))

& ((s = CLOSED)
     -> FORALL w1: Windows (
           (w = w1 -> closed_layout(w)[location] = l
                    & closed_layout(w)[size] = closed_layout'(w)[size]
                    & closed_layout(w)[rep] = closed_layout'(w)[rep])
         & (w /= w1 -> closed_layout(w1) = closed_layout'(w1)))
      & NoChange(open_layout)),

set_size(w: Windows, s: States, s1: Sizes): BOOLEAN ==
  ((s = OPEN)
     -> FORALL w1: Windows (
           (w = w1 -> open_layout(w)[size] = s1
                    & open_layout(w)[location] = open_layout'(w)[location]
                    & open_layout(w)[rep] = open_layout'(w)[rep])
         & (w /= w1 -> open_layout(w1) = open_layout'(w1)))
     & NoChange(closed_layout))

& ((s = CLOSED)
     -> FORALL w1: Windows (
          (w = w1 -> closed_layout(w)[size] = s1
                   & closed_layout(w)[location] = closed_layout'(w)[location]
                   & closed_layout(w)[rep] = closed_layout'(w)[rep])
         & (w /= w1 -> closed_layout(w1) = closed_layout'(w1)))
     & NoChange(open_layout)),

update_display_close(w: Windows): BOOLEAN ==
  (display = display'
         SET_DIFF {SETDEF l: Layouts (l = open_layout'(w))}
         UNION    {SETDEF l: Layouts (l = closed_layout'(w))}),

update_display_open(w: Windows): BOOLEAN ==
  display = display'
         SET_DIFF {SETDEF l: Layouts (l = closed_layout'(w))}
         UNION {SETDEF l: Layouts (l= open_layout'(w))},
```

```
  update_display_create(w: Windows): BOOLEAN ==
     (state(w) = OPEN & (display = display'
                         UNION {SETDEF 1: Layouts (1 = open_layout(w))}))
    | (state(w) = CLOSED & (display = display'
                         UNION {SETDEF 1: Layouts (1 = closed_layout(w))})),

  update_display_destroy(w: Windows): BOOLEAN ==
     (state(w) = OPEN & (display = display'
                        SET_DIFF {SETDEF 1: Layouts (1 = open_layout'(w))}))
    | (state(w) = CLOSED & (display = display'
                        SET_DIFF {SETDEF 1: Layouts (1 = closed_layout'(w))})),

  update_display_move (w: Windows): BOOLEAN ==
     (state'(w) = OPEN & (display = display'
           SET_DIFF {SETDEF 1: Layouts (1 = open_layout'(w))}
           UNION    {SETDEF 1: Layouts (1 = open_layout(w))}))
    | (state'(w) = CLOSED & (display = display'
           SET_DIFF {SETDEF 1: Layouts (1 = closed_layout'(w))}
           UNION    {SETDEF 1: Layouts (1 = closed_layout(w))}))


INITIAL /* the following assertion defines the initial state of the system */
  display = EMPTY
& FORALL w: Windows (
     state(w) = UNUSED
   & process(w) = NULL_PROCESS
   & input_focus(w) = false)
& FORALL p: Processes (
     OVERLAPS(INITIAL_OPEN_LAYOUT(p), BACKGROUND)
   & OVERLAPS(INITIAL_CLOSED_LAYOUT(p), BACKGROUND))

INVARIANT
/* the following assertion is the critical correctness requirements
   that must hold in every state (including the initial state */

  FORALL 1: Layouts (
     1 ISIN display ->
      EXISTS w: Windows (
            (state(w) = OPEN &  1 = open_layout(w))
            | (state(w) = CLOSED & 1 = closed_layout(w))))
 & FORALL 1: Layouts (
       1 ISIN display -> OVERLAPS(1, BACKGROUND))
 & FORALL w: Windows (
```

```
        SPECIAL(w) & state(w) ~= UNUSED
      -> EXISTS l: Layouts ((l ISIN display
                & (l = open_layout(w) | l = closed_layout(w)))
                   & FORALL l1: Layouts ((l ~= l1) & (l1 ISIN display)
                                         -> ~OVERLAPS(l1, l))))

/* the transitions are written to have NoChange to the state variables
   if they shouldn't be applied. These NoChange clauses could be
   rewritten to specify error notification and processing */

TRANSITION close_window(w: Windows) /* iconify */
  EXIT
    state'(w) = OPEN
  & ~SPECIAL(w)
  & state(w) BECOMES CLOSED
  & update_display_close(w)
  & to_top_level(w)

ALT NoChange

TRANSITION open_window(w: Windows)
  EXIT
    state'(w) = CLOSED
  & state(w) BECOMES OPEN
  & update_display_open(w)
  & to_top_level(w)

ALT NoChange

TRANSITION destroy_window(w: Windows)
  EXIT
    (state'(w) = OPEN | state'(w) = CLOSED)
  & state(w) BECOMES UNUSED
  & ~SPECIAL(w)
  & update_display_destroy(w)

ALT NoChange

TRANSITION create(p: Processes)
  EXIT
    EXISTS w: Windows (
      state'(w) = UNUSED
    & state(w) BECOMES INITIAL_STATE(p)
```

```
    & open_layout(w) BECOMES INITIAL_CLOSED_LAYOUT(p)
    & closed_layout(w) BECOMES INITIAL_OPEN_LAYOUT(p)
    & process(w) BECOMES p
    & update_display_create(w)
    & to_top_level(w))

 ALT NoChange

TRANSITION shift_focus(w: Windows)
/* assumes that only one window at a time has input focus and
   that closed windows can have input_focus */
  EXIT
     (state'(w) = OPEN | state'(w) = CLOSED)
   & FORALL w1: Windows (input_focus(w1) = (w1 = w))

ALT NoChange

TRANSITION move(w: Windows, l: Locations)
  EXIT
   (~SPECIAL(w)
 & update_display_move(w)
 & (((state'(w) = OPEN)
   & set_location(w, state'(w), l)
   & OVERLAPS(open_layout(w), BACKGROUND)
   & ~EXISTS w1: Windows (
        SPECIAL(w1) & state'(w1) ~= UNUSED
      & OVERLAPS(open_layout(w), open_layout'(w1))))

  |  ((state'(w) = CLOSED)
   & set_location(w, state'(w), l)
   & OVERLAPS(closed_layout(w), BACKGROUND)
   & ~EXISTS w1: Windows (
        SPECIAL(w1) & state'(w1) ~= UNUSED
      & OVERLAPS(closed_layout(w), open_layout'(w1))))))

 ALT NoChange


 TRANSITION resize(w: Windows, s: Sizes)
   EXIT
         (state'(w) = OPEN) & ~SPECIAL(w)
        & ~EXISTS w1: Windows(SPECIAL(w1) & (state'(w1) ~= UNUSED)
                  & OVERLAPS(open_layout(w), open_layout'(w1)))
```

```
        & set_size(w, state'(w), s)

ALT NoChange

TRANSITION to_foreground(w: Windows)
  EXIT
   to_top_level(w)
 & state'(w) ~= UNUSED
 & ~EXISTS w1: Windows(SPECIAL(w1) & (state'(w1) ~= UNUSED)
      & ((state'(w) = OPEN) & (OVERLAPS(open_layout'(w), open_layout'(w1)))
       | (state'(w) = CLOSED) & OVERLAPS(closed_layout'(w), open_layout'(w1))))

ALT NoChange

END Top_Level
END window_interface
```