N90-16687

1989 NASA/ASEE SUMMER FACULTY FELLOWSHIP PROGRAM

JOHN F. KENNEDY SPACE CENTER
UNIVERSITY OF CENTRAL FLORIDA

FORMALISMS FOR USER INTERFACE SPECIFICATION AND DESIGN

PREPARED BY:	Dr. Brent J. 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 Branch
NASA COLLEAGUE:	Mr. Les Rostosky
DATE:	August 28, 1989
CONTRACT NUMBER:	University of Central Florida NASA-NGT-60002 Supplement: 2

Acknowledgements

This work would not have been possible without the help of the following people — their assistance is sincerely appreciated. The NASA colleague for this project was Les Rostosky. Dave Roberts of NASA provided resources for word processing and access to the Internet.

Dr. Ramon Hosler guided me through NASA and UCF paperwork. His organization of the summer faculty program was conscientious and thorough. Kari Baird's enthusiasm was motivating at just the right times.

Finally, two academic colleagues were particularly helpful providing insights and references via electronic mail. I am grateful to Gary Perlman of the Ohio State University and Stephanie Doane of the Institute for Cognitive Science at the University of Colorado, Boulder for their help.

Abstract

This report describes the application of formal methods to the specification and design of human-computer interfaces. A broad outline of human-computer interface problems, a description of the field of cognitive engineering and two relevant research results, the appropriateness of formal specification techniques, and potential NASA application areas are described.

Table of Contents

I	**Introduction**
I.1	Organization of the Paper
I.2	Background
I.3	The Emerging Field of Cognitive Engineering
I.4	A Fallacy of Interface Design
I.5	The Contributions of Cognitive Engineering
I.6	The Inadequacies of Present Software Engineering Techniques
I.7	User Interface Management Systems
I.8	Independence
I.9	Disentangling Application and Interface
II	**Direct Manipulation Interfaces, Specification, and Verification**
II.1	The Challenge of Direct Manipulation Interfaces
II.2	Overviews of Interface Development Techniques
III	**NASA Applications**
III.1	Some NASA Related Goals
IV.	**A Discussion of Two Relevant Research Results**
IV.1	The Rooms Window Management System
IV.2	Specification of Direct Manipulation Interfaces
IV.2.1	Objections to Formal Specification and Verification
IV.2.2	Reasonable Expectations for Formal Specification
IV.2.3	Jacob's Formal Specification System for Direct Manipulation Interfaces
V.	**Correctness Constraints**
VI	**Concluding Remarks**
VI.1	Recommendations
VI.2	Goals Revisited

I. Introduction

I.1 Organization of the Paper

This report describes the application of formal methods to the specification and development of human-computer interface software.

This paper is organized as follows: problems of interface specification and design and the inadequacies of traditional software engineering techniques are discussed. The emerging field of cognitive engineering as a new approach to solving user interface problems is described. Second, direct manipulation user interfaces are defined and contrasted with traditional command language interfaces. Third, a NASA context of shuttle and space station software is given. Fourth, two relevant research results are discussed. Both results are influenced by the emerging field of cognitive engineering. Techniques of reducing the confusing jumble of windows [1], and of formally specifying direct manipulation user interfaces [2] are presented. The feasibility and appropriateness of formal specification techniques for user interfaces is discussed, and user interface correctness criteria from a NASA user interface document are described. Finally, recommendations are given for using formal methods of user interface development and specification on NASA projects.

I.2 Background

In the beginning, the primary users of software were software developers themselves, or highly motivated users willing to endure bizarre idioms to take advantage of a computer's power. A typical user today is usually not a software developer and has little tolerance for productivity impediments imposed by software that is hard to use, learn, or is dangerous.

User interface software takes a significant amount of the total effort involved in the development of a software system. Estimates of of "40 to 50 percent of the code and runtime memory" [3, p. 15] being devoted to the user interface, and "50% of the coding effort in a typical data base application is usually spent on the implementation of the user interface" [4, p. 481], are common.

I.3 The Emerging Field of Cognitive Engineering

Possible solutions to the user interface crisis come from the emerging field of cognitive engineering.

Cognitive engineering is the offspring of cognitive science and software engineering. This multidisciplinary approach to human-computer interfaces is well-founded in the fields of human factors and human-computer interaction (HCI). [5] is a description of the interaction between cognitive psychology and computer science that results in the field of HCI.

A new, interdisciplinary academic discipline has formed to address general questions of human and computer cognition. Norman defines this new field of cognitive science [6, p. 326] as "a fusion of many disciplines, all concerned with different aspects of cognition.

It combines psychology, artificial intelligence, linguistics, sociology, anthropology, and philosophy."

Norman argues that the application of the disciplines of cognitive science to the "cognitive aspects of human-computer interaction" describes a new multidisciplinary approach of *cognitive engineering* [6, p. 326]. Throughout this paper, the term cognitive engineering will be used to include the field of HCI.

The important contributions of cognitive engineering are insights into what makes computers hard to use and learn. Norman argues [6, p. 331] that

> Computers can add to task difficulty when they stand between the person and the task, adding to the issues that must be dealt with to complete the task. When computer stand between, they act as an intermediary, requiring that the user be proficient in both the task domain and with the intermediary (the computer).

I.4 A Fallacy of Interface Design

A goal of cognitive engineers is to make the intermediary nature of the computer disappear allowing the user to exert their entire effort on the task.

Making the computer transparent was until recently not a concern of software developers. In the beginning, software developers themselves were the primary users of software. That is, the users of software had both computer and task knowledge. This naturally lead to what Landauer calls the "egocentric intuition fallacy" [7, p. 906]:

> One part of the egocentric intuition fallacy is the compelling illusion that one knows the determinants of one's own behavior and satisfaction. ... Even when a system is obviously hard [to use], intuition may not reveal the true reasons.

In addition to having flawed intuition about what makes a good interface, developers have mental models of software systems that are often very different from the operational mental model that users have of the same system [8]. A designer's natural tendency is to design to their mental model and not to the users' cognitive models.

I.5 The Contribution of Cognitive Engineering

The contribution that cognitive engineering makes to solving human-computer interface problems can be summarized as follows: cognitive engineering provides the sound experimental basis of human cognition necessary to build human-computer interfaces that enhance rather than hinder problem solving.

I.6 The Inadequacies of Present Software Engineering Techniques

Traditional software engineering techniques for interface design supported the development of command language interfaces. Command language interfaces consist of a typed dialogue between the system and the user. These software engineering tools were based on state machine and context free grammar models of command languages. Traditional tools, however, are inadequate for developing mouse-window interfaces. Fischer is convinced that "current life-cycle models of software engineering are inadequate for most of today's computing problems," [9, p. 44] particularly for user interfaces.

Recent developments in rapid prototyping, knowledge-based support tools, and specification systems are being applied to the development of user interfaces. Many of these second generation interface development tools are coming from cognitive science and engineering laboratories.

I.7 User Interface Management Systems

The class of interface development tools called *user interface management systems* (UIMSs) is rapidly maturing. Perlman defines a UIMS as "a set of software tools for developing and managing user interfaces" and elaborates [10, p. 823]:

> The fundamental role of user interface tools is to act as mediator: (1) to simplify the interface between users and the applications with which they interact; and (2) to simplify the task of the user interface developer to create the user-application interface.

Some UIMSs allow designers to compose interfaces with respect to a body of constraints or guidelines and have their designs critiqued [9], or design interfaces by demonstration, rather than coding [11]. In general, UIMSs support the development of good user interfaces by [12, p. 33]:

- Providing a consistent user interface between related applications
- Making it easier to change the user interface design when needed
- Encouraging development and use of reusable software components
- Supporting ease of learning and use of applications

Meyers further categorizes user-interface tools into user-interface *toolkits* and user-interface *development systems* (UIDS) [3, p. 16]. A user interface toolkit is a collection of templates for common interaction idioms such as windows, mice, or sliders. Application source code calls high level routines in toolkit libraries to use common services. The most well know toolkit is the X Windows system [13]. Meyers further defines a UIDS as "an integrated set of tools that help programmers create and manage many aspects of interfaces" [3, p. 16]. A UIDS provides both development and run-time support for user interfaces.

I.8 Independence

An important idea underlying user interface toolkits is that of independence. The X window system, for example, provides device independence while supporting sophisticated windowing and mouse input environments. Perlman explains [10, p. 823]:

> The fundamental concept behind the creation of useful tools is *independence*; the ability to independently insert new users, devices, and applications into information processing tasks without requiring major efforts by the application developer.

Hartson and Hix also point out that a primary motivation for UIMSs is independence. Hartson and Hix stress the importance of dialogue independence — using the same language to communicate with diverse applications or objects. They explain: "in UIMS, the concept of dialogue independence is explicitly recognized and supported. Most UIMS are based, at least to some extent, on dialogue independence" [14, p. 17].

I.9 Disentangling Application and Interface

The notion of independence has been taken further as the basis of disentangling applications and user interfaces [15]. Figure 1 shows an application in which the user interface is entangled in the application. Figure 2 shows application and user interface functionality clearly separated. This separation has some cost, however. Common, independent interfaces sometimes are the lowest common denominator [10] and may have performance problems [12, p. 36].

II. Direct Manipulation Interfaces, Specification, and Verification

Mouse-window user interfaces are examples of direct manipulation user interfaces. Jacob explains differences between direct manipulation and command interfaces [2, p. 283]:

> With a direct manipulation interface, the user seems to operate directly *on* the objects in the computer instead of carrying on a dialogue *about* them. Instead of using a command language to describe operations on objects that are frequently invisible, the user "manipulates" objects visible on a graphic display

II.1 The Challenge of Direct Manipulation Interfaces

Direct manipulation interfaces have been a challenge for traditional software engineering methods of specification and design. Fischer asserts that "static specification languages have little use in HCI software design," [9, p. 50] and DeMillo et al. also doubt that formal software engineering techniques can be applied to user interfaces [16, p. 277]:

> It has been estimated that more than half the code in any real production system consists of user interfaces and error messages - ad hoc, informal structures that are by definition unverifiable.

II.2 Overviews of Interface Development Techniques

[10] and [3] overview software tools for user interfaces, [4] describes formal methods for interface development, and [14] reviews the management of user interface development.

III. NASA Applications

The motivation of this research is the development of a new *Core* of test, control, and monitor software for the space station and shuttle. This system consists of three parts: a generic core, new checkout, control, and monitor software (CCMS-II) for shuttle, and test, control, and monitor software (TCMS) for the space station.

The hardware architecture of the project is highly distributed. Multiple processors are dedicated to data acquisition, application processing, data bases, data archival and retrieval, and display. These processors communicate through high speed networks.

Two things are significant in this effort. First, each user will have a powerful display processor (DP) as their interface to the system. A DP is capable of multiple sessions with remote applications and supports a mouse-and-window (direct manipulation) environment. Second, the system is required to use commercial off-the-shelf (COTS) hardware and software where feasible. Independence from particular hardware and software is emphasized throughout the Core system.

Casual use of COTS can result in a jumble of interfaces. To prevent this problem, the software must meet the Space Station Information Systems (SSIS) User Support Environment Interface Definition (USEID). USEID defines a common user interface (CUI), user interface development tools, and CUI standards.

The CUI is the most relevant to this research. CUI consists of direct manipulation interface (DMI) services, user interface language (UIL) services, and an intermediate language (IL) to describe interapplication communication.

Schoaff describes four scenarios for use of TCMS: monitoring applications (such as programs to sample pressure), controlling applications (such as opening and closing valves), testing applications (creating combinations of monitoring and controlling applications), and operating the interface (allowing users to customize their interface) [17].

III.1 Some NASA Related Goals

The preceding section produces motivates several goals. The remainder of this report should be read with the following in mind.

- Can present UIMS's support a direct manipulation, highly distributed system such as Core?
- What formalisms are available to support the specification of Core user interfaces? In particular, can a formalism for the use of COTS software be developed as a means of maintaining a common user interface?
- Given a formal specification of portions of a user interface, can the specification and implementation by verified with respect to some correctness criteria?

IV. A Discussion of Two Relevant Research Results

Two systems in particular will be examined: the Rooms UIMS [1], and a UIMS built around a formal specification system for direct manipulation user interfaces [2]. Rooms is an example of a system with roots in cognitive engineering, while the foundation of Jacob's work is traditional software engineering. Both systems are of the new generation of interface tools and are heavily influenced by cognitive engineering.

IV.1 The Rooms Window Management System

Computer scientists know that running programs access memory nonuniformly. That is, a plot of memory accesses shows distinct clumping of references. This characteristic is exploited in virtual memory operating systems which allow programs to run when physical memory is much smaller than the program's range of memory accesses. Virtual memory operating systems shuffle programs' code and data between a small physical memory and a larger secondary store giving each program the illusion of a large physical memory. Operating systems take advantage of programs' locality of memory references to perform this shuffling between physical and secondary memory in ways that tend to maximize performance.

Optimal memory shuffling is in general not attainable, and in the worst case a running program will experience thrashing - repeatedly referring to memory locations that are not in physical memory and must be shuffled in from secondary storage.

Henderson and Card applied the concepts of locality of reference and thrashing to direct manipulation interfaces. Their Rooms UIMS was designed as a solution to the problem of *window thrashing.* Window thrashing occurs because the small size of computer displays. Although mouse-and-window interfaces are often described as a metaphor for a physical desktop, "a standard office desk has the area of 22 IBM PC screens, 46 Macintosh screens, or even 10 of the "large" 19-inch Xerox 1186 or Sun-3 screens" [1, p. 212].

Four techniques of dealing with the small size of computer screens are: 1) alternating screen usage, 2) distorted views, 3) large virtual workspaces, and 4) multiple virtual workspaces. The present CCMS system use the first technique, allowing users to flip among screens. The second technique is commonly used - shrinking an entire window to an icon is an example of a distorted view. The second technique can be taken to an extreme by using fish-eye techniques in which objects are distorted based on some priority (e.g. most recently used or most important). The third technique uses windows as viewports into a large virtual world. Henderson and Card cite the NASA Ames virtual reality helmet as an example of the third technique taken to an extreme. Rooms is based on the fourth technique - a hypertext-like organization of users' worlds. A problem with hypertext applications is navigation. The authors state that [1, p. 238]

Our experience suggests that navigation tends to be easier in a multiple-virtual-workspace system than in either a large single workspace or a hypertext system.

Henderson and Card use locality of window reference as the basis of Rooms. The authors postulate that just as programs in execution exhibit locality of reference, use of windows also occurs in related chunks. In the same way that locality of reference makes virtual memory feasible in that only a small amount of physical memory is necessary to run a program in what it thinks is a large virtual address space, locality of window reference should make multiple virtual workspaces feasible.

Henderson and Card elaborate [1, p. 221]:

> The design of the Rooms system is based on the notion that, by giving the user the mechanism for letting the system know he or she is switching tasks, it can anticipate the set of tools/windows the user will reference and thus preload them together in a tiny fraction of the time the user would have required to open, close, and move windows or expand and shrink icons. A further benefit is that the set of windows preloaded on the screen will cue the user and help reestablish the mental context for the task.

Thus, task-specific sets of windows aid in navigation.

IV.2 Specification of Direct Manipulation Interfaces

A formal specification of a system is a state of the functionality that a system is to perform. Formal specifications have been used to specify the behavior of complex system such as secure operating systems and terminals.

A formal specification can be used in several ways. First, if the specification is highly abstract, it is possible to reason about essential qualities of the system before it is implemented. The essence of functionality can be explored without superfluous details. Second, abstract specifications can be refined to more detailed specifications. The transformations, or mappings, between an abstract specification and a refinement are also formally defined and can be reasoned about. Third, if correctness constraints can be formally expressed, it is possible to verify that the specification is correct with respect to the correctness criteria.

It is possible to formally specify, possibly through multiple levels of abstraction, a system down to pre and post assertions corresponding to entry and exit assertions for code. If the specification is proved correct with respect to the correctness criteria, we are assured that the a program implementing the specifications will meet the correctness criteria.

Figure 3 shows the specification and verification process.

Although formal specification and verification is theoretically possible, its use has been limited to critical applications such as secure message systems. A typical correctness criteria for such systems is that a user cannot read an object having security level higher than theirs, and cannot write to an object that has security level lower than theirs. These properties are called the simple security property, and the star-property. In the ASLAN formal specification language [18, 19, 20] they would be expressed as

```
INVARIANT

FORALL u: user, o: object (
      (reading(u, o) -> level(u) >= level(o))
    & (writing(u, o) -> level(u) <= level(o)))
```

where -> is logical implication.

The disadvantages of formal specification and verification are many and well known. DeMillo et al. argue that a social process much like that for mathematical theorems must be applicable [16]. That is, much of our faith in mathematical theorems comes from the wide dispersion and perusal of conjectures and proofs. The authors argue that verifications are tedious and dull, lacking insight and that no one would read through an entire specification or verification proof. More recently, Fetzer has reexamined the DeMillo arguments and refined them further [21], concluding that DeMillo et al. "arrived at the right general conclusion for the wrong specific reasons" [21, p. 1062].

IV.2.1 Objections to Formal Specification and Verification

The major objections to formal specification and verification are summarized as follows:

1. Interesting constructs of modern programming languages are not formally defined, and are therefore not verifiable.

2. The volume of information that formal specification and verification develop make it infeasible to prove all but the smallest programs.

3. Specifications are not interesting reading and do not embody insights that would draw human reviews to peruse them.

4. To be useful, a specification must correspond to implementation code: high level specifications are worthless.

These arguments are too pessimistic and will be examined in turn. Our responses will be with respect to the user interface software. Our general assertion is that formal specification, when applied to isolated pieces of a system, with specific correctness constraints in mind, can be productively applied.

1. Hoare [22] has specified the semantics of Pascal, and Mallgren [23] has developed techniques for specifying graphics programming languages.

2. Although to specify and verify even a small program takes volumes of information, care must be taken to specify only well defined pieces. Also, the specification process does not have to be taken to code to be worthwhile.

3. High level specifications embody the essential functionality and criticality of a software system. The abstract specifications are often quite insightful, and not difficult to read. Also, Just as programmers usually do not read the assembly languages produced by compilers of higher level languages, specifiers don't have to reading all the output of specification and verification systems.

4. High level specifications are useful in isolating critical aspects, something physical that can be discussed in groups (the social aspect that DeMillo was looking for), and can serve as the foundation for user documentation and for traceablity of requirements in design.

IV.2.2 Reasonable Expectations for Formal Specification

Our view is that formal specifications can be productively applied to software development processes. In addition, formal specification is worthwhile even if formal verification is not the ultimate goal. For a formal specification effort to be worthwhile there must be 1) well defined areas of functionality to be specified, 2) well defined constraints on system behavior that can be used as high level correctness criteria, 3) abstraction techniques that can refined through multiple levels of abstraction, and 4) personnel able to read and evaluate specifications.

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.

IV.2.3 Jacob's Formal Specification System for Direct Manipulation Interfaces

The creed of formal specification is "describe what the system does, not how". When specifying user interfaces it is easy to be trapped into specifying the entire system behavior - after all, what a user sees is the essential functionality of the system. As noted, for a specification to be successful it is essential to break up the system into interacting parts.

Thus, user interface specifiers should limit their work to defining only users methods of interaction. An application's functionality can be specified, and verified if necessary, separately.

That is, given separately specified user interface and applications, and a definition of their interaction, one can inductively argue that the system as a whole is specified.

Until recently, user interface specification languages, and their associated UIMSs have addressed the problem of dialogue specification, design, and support. These techniques have commonly been based on state machine models or formal grammars.

Although easier to use, direct manipulation user interfaces have proven to be much harder to specify than keyboard-and-screen based dialogues.

Jacob [2] has developed a collection of specification methods and a language for direct manipulation user interfaces. Jacob's specification language is object oriented and executable, allowing prototype interfaces to be rapidly developed.

Jacob uses a three level specification technique: the lexical level, syntax level, and semantic level. Tokens, such as button clicks, highlighting, and dragging, are defined at the lexical level. Sequences of tokens representing plausible user input make up the syntactic level. Finally, the functional requirements of the system are embodied in the semantic level. The semantics of an application's functionality can be specified separately using conventional specification techniques. Jacob's techniques, therefore, emphasize the lexical and syntactic levels and the way functions are invoked.

Jacob defines an interaction object as "the smallest unit in the user interface that has a state" [2, p. 290]. He then gives three general steps to specifying a direct manipulation user interface [2, p, 290]:

1. define a collection of interaction objects,
2. specify their internal behaviors, and
3. provide a mechanism for combining them into a coordinated user interface

A key point of Jacob's work is that user interfaces can be described as interacting state machines. However, direct manipulation interfaces have often been described as stateless, or modeless. Jacob argues that in direct manipulation user interfaces are "highly moded, but they are much easier to use than traditional moded interfaces because of the direct way in which the modes are displayed and manipulated" [2, p. 288]. By separating the specification into three levels and allowing objects to essentially inherit state machines from other objects, Jacob has brought the power and simplicity of state machines to the specification of direct manipulation user interfaces.

In particular, Jacob proposes that an interface be modeled as a high level executive process that invokes coroutines corresponding to individual dialogues. Such a user interface is highly moded (input has different meaning depending on its context) and can be modeled by state machines that interact in well-defined ways. The user interface as a whole is viewed as a simple executive and multiple independent dialogues.

V. Correctness Constraints

Formal specifications are particularly valuable when they are analyzed with respect to formally state correctness criteria. Correctness constraints can be categorized as being properties that must hold in every state (safety properties), or properties that must eventually be true (liveness properties).

Formal specification and verification is particularly well suited to ensuring that high-level safety properties are maintained. Liveness techniques are much harder to specify and prove.

A space station user interface requirements document [24] contains many safety properties. These properties could be used as correctness constraints for formal specifications. The following are example correctness constraints:

- 3.3.1.4 Graphics Panels shall be no larger than the maximum size of the Workspace Panel and no smaller than TBD. [24, p. 9]
- 3.3.2.1 Selection of a Radio Button shall invoke the highlight of the selected button and the removal of the Radio Button highlight from the previously selected Radio Button. [24, p. 10]
- 3.3.2.2.1 If the selected Momentary Button is not released and the cursor is moved outside the boundaries of the button, the highlight shall be removed from the button and the switch action canceled. [24, p. 10]
- 3.4.2.1 The Alarm Classification area shall contain an alphabetic identifier that signifies to the operator the classification of the alarm ... when no alarm is present, this area shall be left blank. [24, p. 16]

VI. Concluding Remarks

This report argues that the development of NASA human-computer interfaces can benefit from recent cognitive engineering research. It has also shown that formal specification techniques can be used to specify direct manipulation user interfaces.

In particular, a study on window use and management that combined results from cognitive psychology and operating systems theory [1] is applicable to designing interfaces for ground and flight software. These interfaces can automatically present users with familiar environments for specific tasks.

Second, a formal specification system for direct manipulation user interfaces [2] was described. The system is object oriented and breaks down the task of specification into three levels: lexical, syntactic, and semantic.

Finally, appropriate use and reasonable expectations of formal specification of user interfaces is discussed. The use of formally expressed correctness constraints is examined, and examples of such constraints from a NASA user interface document are given.

VI.1 Recommendations

The following are recommendations for development of NASA user interface software. These recommendations are realistic and can be used in addition to other user interface development techniques.

- Build data collection capabilities into user interface software. These capabilities should not be used to track specific individuals' work, but should be used to identify areas in which mistakes are frequently made, or to find task specific localities of interface use. The Rooms user interface system was a direct result of being able to monitor users' needs for interface resources.

- Use rapid prototypes of interfaces for early usability tests. Early useability tests frequently bring out interaction areas which are difficult to users but simple to the interface developers. Jacob's system for specification of direct manipulation user interfaces is not only a formal specification system, but also a rapid prototyping tool. Using executable formal specification languages has the advantage of both formal specification and rapid prototyping.

- Identify high level correctness constraints for interfaces. Identification of correctness criteria is useful even without formal verification. Although NASA user interface documents contain many examples of low level correctness criteria, quantitative high-level correctness criteria are rare.

- Consider a pilot study using formal techniques on a small, well-defined piece of a user interface. A good example to formally develop would be an alarm area of a screen. The function of alarm areas is typically straightforward, and high level correctness criteria are apparent.

VI.2 Goals Revisited

Finally, the three goals identified in section III.1 can be addressed. The first goal asked whether present UIMSs can support direct manipulation interfaces that NASA requires. At this point the answer is no, although there are excellent interface development tools that can be applied to pieces of interface development efforts.

The second goal was to seek formalisms to support the specification of user interfaces. Again, the tools are not mature, but are potentially useful for well defined pieces of interface development. The second part of the second goal concerned the use for formalisms for use of COTS software. This remains an open research question.

The third goal concerned the formal specification of interfaces and correctness criteria. We believe that with reasonable expectations, formal specification can be productively applied to the development of user interfaces.

References

[1] D. A. Henderson, Jr. and S. K. Card. Rooms: the use of multiple virtual workspaces to reduce space contention in a window-based graphical user interface. *ACM Transactions on Graphics* 5, 3 (July 1986).

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

[3] B. A. Meyers. User-interface tools: introduction and survey. *IEEE Software* 6, 1 (January 1989).

[4] M. U. Farooq and W. D. Dominick. A survey of formal tools and models for developing user interfaces. *International Journal of Man-Machine Studies* 29 (1988).

[5] B. Auernheimer and J. L. Dyck. Human-computer interaction: where computer science and psychology meet. *Proceedings of the Twenty-second Annual Hawaii International Conference on System Sciences,* IEEE Computer Society Press, 1989.

[6] D. A. Norman. Cognitive engineering - cognitive science. In *Interfacing Thought* J. M. Carroll, ed. The MIT Press, 1987.

[7] T. K. Landauer. Research methods in human-computer interaction. In *Handbook of Human-Computer Interaction, M. Helander, ed. North-Holland, 1988.*

[8] S. Doane, W. Kintsch, and P. Polson. Action planning: producing UNIX commands. *Proceedings of the Eleventh Annual Meeting of the Cognitive Science Society.* Ann Arbor, 1989.

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

[10] G. Perlman. Software tools for user interface development. In *Handbook of Human-Computer Interaction, M. Helander, ed. North-Holland, 1988.*

[11] B. A. Meyers. *Creating User Interfaces by Demonstration.* Academic Press, 1988.

[12] J. Löwgren. History, state and future of user interface management systems. *SIGCHI Bulletin* 20, 1 (July 1988).

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

[14] H. R. Hartson and D. Hix. Human-computer interface development: concepts and systems for its management. *ACM Computing Surveys* 21, 1 (March 1989).

[15] M. J. Muller. Disentangling application and presentation in distributed computing: architecture and protocol to enable a flexible, common user interface. *IEEE Transactions on Systems, Man, and Cybernetics.* 18, 4 (July/August 1988).

[16] R. A. DeMillo, R. J. Lipton, and A. J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM* 22, 5 (May 1979).

[17] W. D. Shoaff. The design of an intelligent human-computer interface for the test, control and monitor system. Technical report, grant NASA-NGT-60002, August 1988.

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

[19] 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).

[20] 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).

[21] J. H. Fetzer. Program verification: the very idea. *Communications of the ACM* 31, 9 (September 1988).

[22] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica* 2 (1973), p. 335-355.

[23] W. R. Mallgren. *Formal Specification of Interactive Graphics Programming Languages.* The MIT Press, 1983.

[24] A. D. Cohen. User interface requirements document (DR SY 45.1). Work Package No. 2. McDonnell Douglas Space Systems Company (July 1989).

Entangled Interface and Application

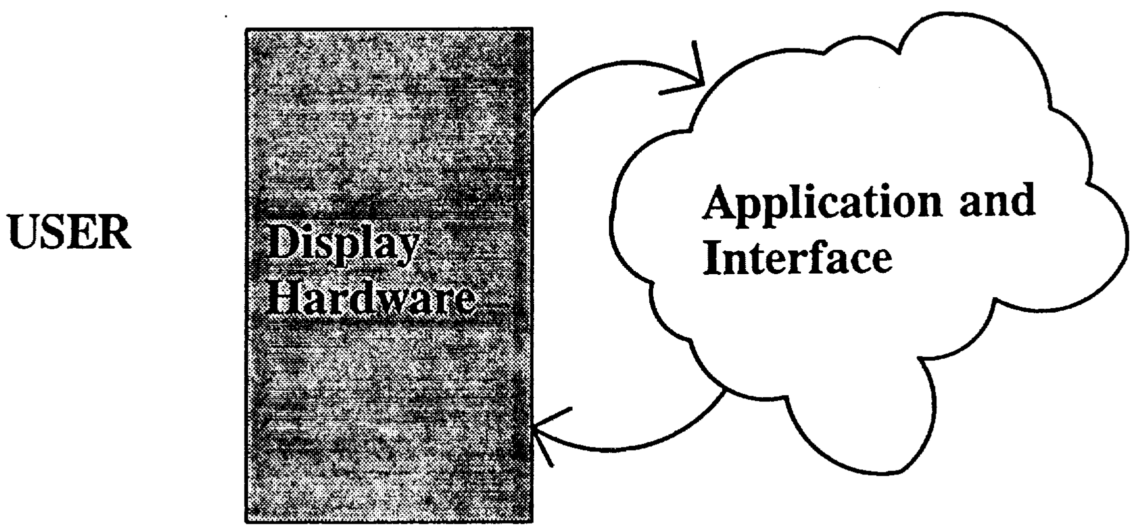


Figure 1. Entangled User Interface and Application

Disentangled Interface and Application

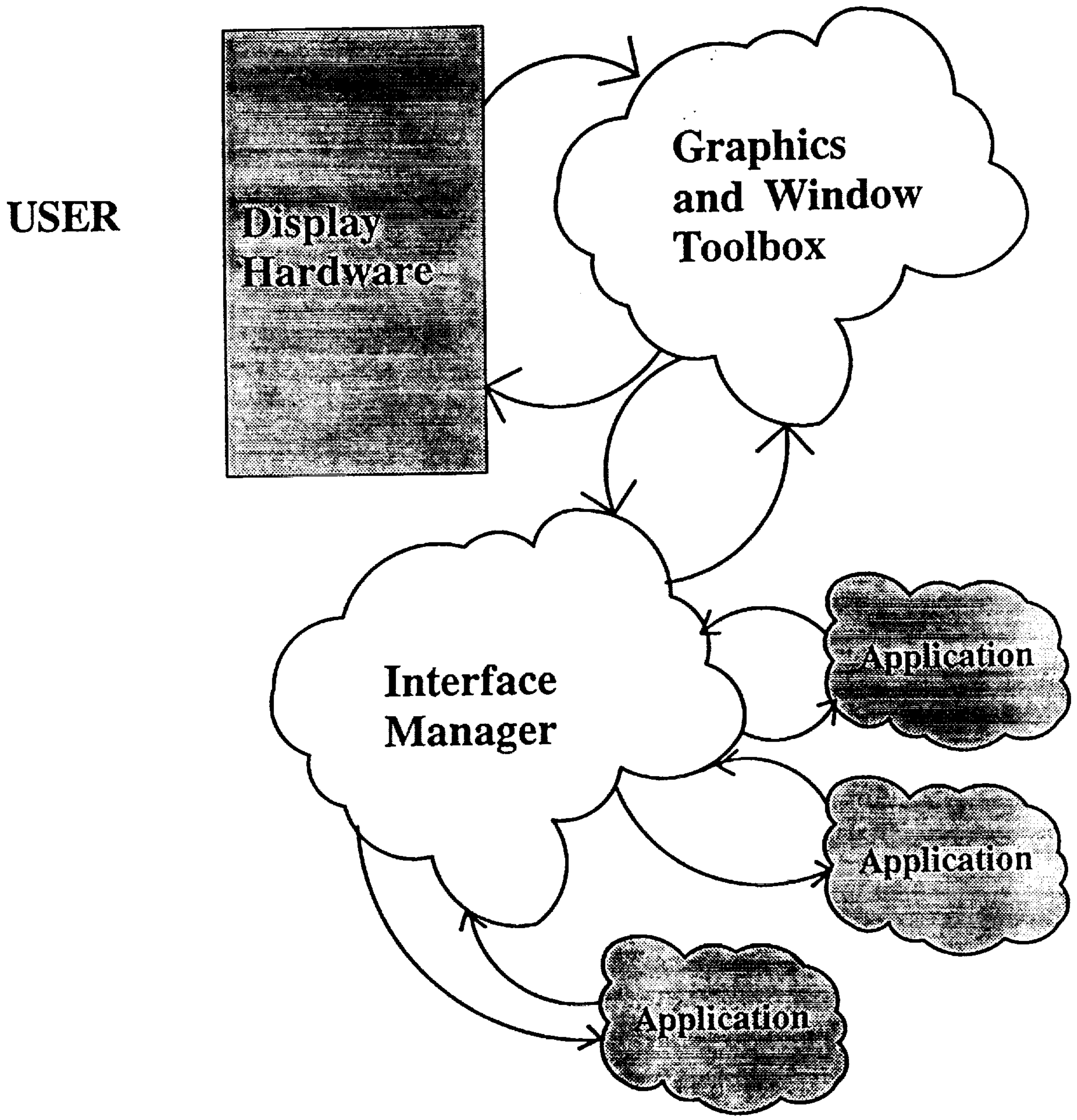
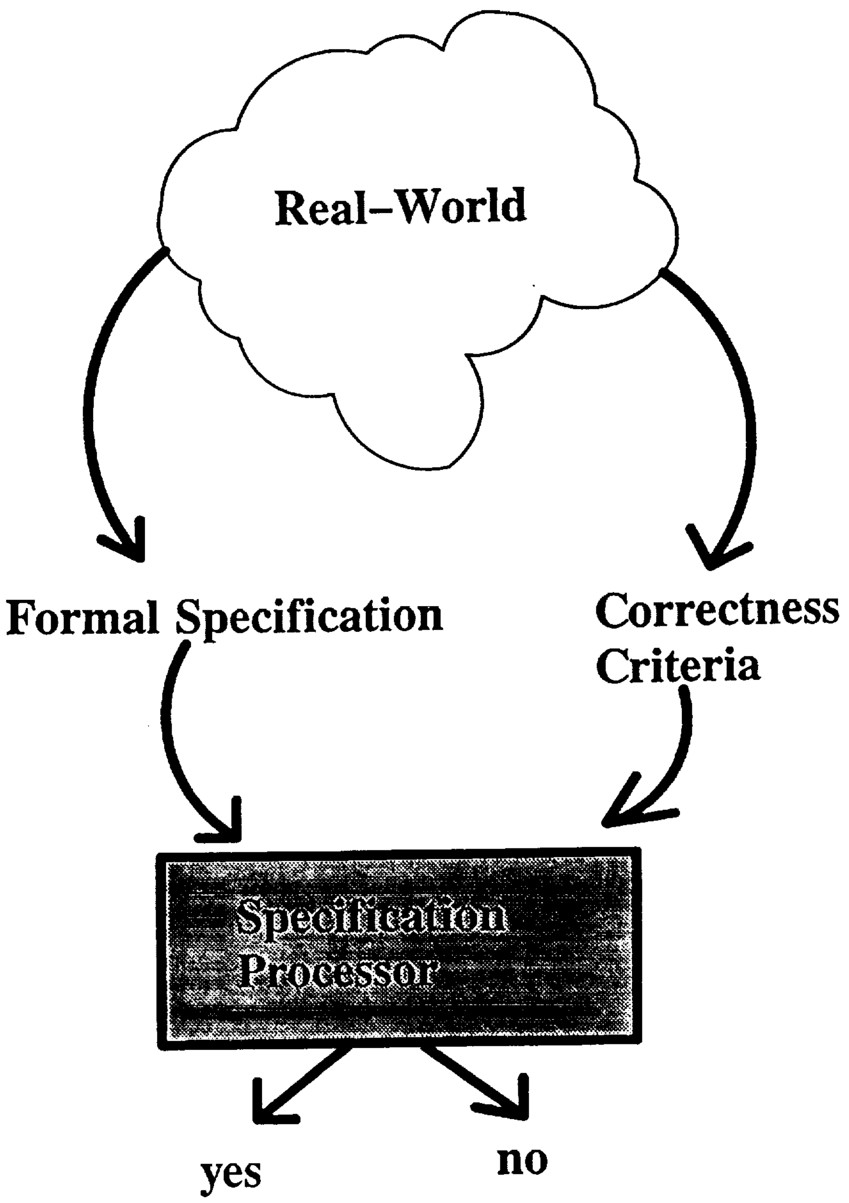


Figure 2. Disentangled User Interface and Applications

Formal Specification and Verification



Is the specification correct with respect to the correctness criteria?

Figure 3. The Formal Specification and Verification Process