A Cooperative Problem Solving System for User Interface Design

Andreas C. Lemke and Gerhard Fischer

Department of Computer Science and Institute of Cognitive Science
Engineering Center ECOT 7-7, University of Colorado, Boulder, CO 80309-0430
{andreas, gerhard}@boulder.colorado.edu

Abstract

Designing a user interface is an ill-defined problem making cooperative problem solving systems a promising approach to support user interface designers. Cooperative problem solving systems are modular systems that support the human designer with multiple, independent system components. We present a system architecture and an implemented system, FRAMER, that demonstrate the cooperative problem solving approach. FRAMER represents design knowledge in formal, machine-interpretable knowledge sources such as critics and dynamic specification sheets, and in semi-formal knowledge sources such as a palette of user interface building blocks and a checklist. Each of these components contributes significantly to the overall usefulness of the system while requiring only limited resources to be designed and implemented.[1]

Cooperative Problem Solving

Artificial intelligence research has traditionally focused on building systems that autonomously solve complex problems (e.g., R1/XCON (McDermott, 1982) and MYCIN (Buchanan, Shortliffe, 1984)). This approach is however not easily applicable in ill-defined problem domains, such as user interface design. Consistency (Grudin, 1989), learnability, and many other concepts of user interface design cannot be adequately formalized in a precise way.

Alternatively, one can design cooperative problem solving systems (Fischer, 1988) that work in conjunction with human problem solvers rather than replacing them. Cooperative problem solving systems are located between systems that design with human guidance (e.g., UofA*, (Singh, Green, 1989)) and passive CAD tools (e.g., MENULAY, (Buxton et al., 1983)).

A desirable characteristic of practical cooperative problem solving systems is a modular, incremental architecture with simple but extensible components. In contrast, many intelligent support systems that have been proposed carry a heavy weight of complex system components. For example, a natural language based help system requires a natural language understanding component, a problem solver, and a natural language generator. Each of these components is large and complex, and all three components have to exist for the whole system to function properly. A system with an incremental architecture, however, can be gradually improved by extending its components and by adding new components. There is a low threshold for creating a low-end system and quickly introducing it into practical use.

To build effective cooperative problem solving systems, the limitations of both autonomous expert systems and human problem solvers must be understood. This knowledge will enable us to complement intelligent machines where they are limited, and to augment the human intellect where it needs support. Contributions from the machine must enable the human to proceed in ways that were not possible without them and vice versa.

Among the limitations of autonomous intelligent machines are the difficulty of capturing a sufficiently complete store of domain knowledge, the opaqueness of expert decision making process, the specification problem, and issues of conflicting and subjective practice. Of these, the specification problem is one of the hardest to overcome. It refers to the fact that, for ill-defined problems (Eastman, 1969; Simon, 1973), specification and solution are developed hand in hand and not in sequence (Rittel, 1972; Swartout, Balzer, 1982). At the start of a design process, a specification may be lacking in at least two ways. First, the specification may be incomplete, i.e., certain characteristics of the artifact have been left unspecified although they are important. For example, the behavior of computer systems in exceptional situations is often left unspecified. Second, for some characteristics, the desired values may yet be unknown, for example, because their consequences have not been evaluated. For these problems, an interactive approach is necessary because the human is unable to specify all the relevant information and preferences in advance and because specifying the problem is itself a problem solving process.

Human problem solving is limited by fundamental cognitive limitations such as short term memory capacity, forgetting, and slow long term memory access. At a higher level, it has been found that inexperienced problem solvers

[1]This research was partially supported by grants No. DCR-8420944 and No. IRI-8722792 from the National Science Foundation, grant No. MDA903-86-C0143 from the Army Research Institute, and grants from the Intelligent Interfaces Group at NYNEX and from Software Research Associates (SRA), Tokyo.

do not consider and deliberate enough alternatives but rather use the first one they find (Jeffries et al., 1981). Humans do not search for information of whose existence they are unaware (Fischer, 1989), and they are unable to keep all relevant factors in mind when making decisions.

The purpose of our systems is to reduce the knowledge needed to design and to help less experienced designers achieve better results by providing external knowledge sources. Cooperative problem solving systems must be able to communicate design knowledge to the user. Typical AI knowledge representation formalisms, such as rules or frames, are designed to be efficiently executable by inference engines but are not necessarily applicable in cooperative problem solving systems where the knowledge must be interpreted by humans as well. Our approach is based on a combination of formal, machine-interpretable and semi-formal knowledge sources that can only partially used by the system to control its actions. The kinds of semi-formal knowledge structures we are employing are easier to acquire and modify than formal knowledge structures (Peper, MacIntyre, Keenan, 1989). Semi-formal knowledge structures are also useful in ill-defined problem domains where concepts and relationships cannot always be captured in a complete, executable way. Semi-formal knowledge structures alone, however, can not give users sufficient support—they have to do "all the work." Thus, we complement them with formal knowledge structures that allow the system to solve well-defined subproblems for the user.

In the following section, we describe a system architecture for cooperative problem solving systems that addresses these questions. The architecture employs system components that serve as formal and semi-formal knowledge sources.

Framer: A Cooperative User Interface Design Environment

Our research has focussed on devising methods and tools to support the above-mentioned design activities. We describe our results using the example of the FRAMER design environment. FRAMER (Figure 1) is a knowledge-based design environment for program frameworks, which are high-level building blocks for window-based user interfaces. Program frameworks consist of a window frame of nonoverlapping panes and an event loop for processing mouse clicks, keyboard input, and other input events. Program frameworks also manage the update of information displayed on the screen. The current Framer system and its architecture is the result of an iterative development process that has gone through three major stages: tool kits, construction kits, and knowledge-based design environments. In this sequence, each later stage is an extension of its predecessor. We describe version 2 of the FRAMER system, which is based on experience with FRAMER1.

Tool Kits

The first stage, tool kits, aims at providing domain-oriented building blocks, such as windows and menus. Examples of tool kits are Xlib, NextStep, and the Macintosh toolbox. Tool kits enable designers to work in terms of concepts of their domain of expertise rather than at the level of a general-purpose programming language. FRAMER uses the Symbolics user interface toolkit, specifically program frameworks and different kinds of windows and menus. Tool kits represent a limited amount of design knowledge that was used in the design of the building blocks.

Construction Kits

Toolkits provide domain-oriented building blocks, but they do not support the processes of finding and combining the blocks—designers have to know what blocks exist and how they are used. Construction kits address this problem by providing a *palette* and a *work area* (see Figure 1). The palette displays representations of the building blocks and thus shows what they are and makes them easily accessible. The palette provides an answer to the question what the possible components of the design are. The work area is the principal medium for design and construction in the FRAMER design environment. This is where the designer builds a window layout by assembling building blocks taken from the palette. Examples of user interface construction kits are the Symbolics FrameUp system, MENULAY (Buxton et al., 1983), the NeXt user interface builder, and WIDES and TRIKIT (Fischer, Lemke, 1988).

Design Environments

Knowledge-based design environments address shortcomings that we have found in construction kits. Construction kits support design of interfaces at a syntactic level only, and our experience with this class of systems has shown that it is easy to create a functioning interface, but creating a *good* interface requires a great deal of additional knowledge that is not provided by construction kits. Design environments provide additional design knowledge through critics, checklists, and other means described below.

Critics. Critics are a formal knowledge source in FRAMER. Critics (Fischer et al., 1990) are demons that evaluate the evolving artifact. When the system detects a suboptimal aspect of the artifact, it displays a message describing the shortcoming in the critic window entitled "Things to take care of" (Figure 1) The critics trigger as soon as the designer makes an inferior design decision and they update the critic window continuously.

FRAMER2 distinguishes between *mandatory* and *optional* suggestions. Mandatory suggestions must be carried out by the designer. They represent system requirements for the construction of a functioning program framework. For example, a frame must be completely covered with panes if correct LISP code is to be generated,

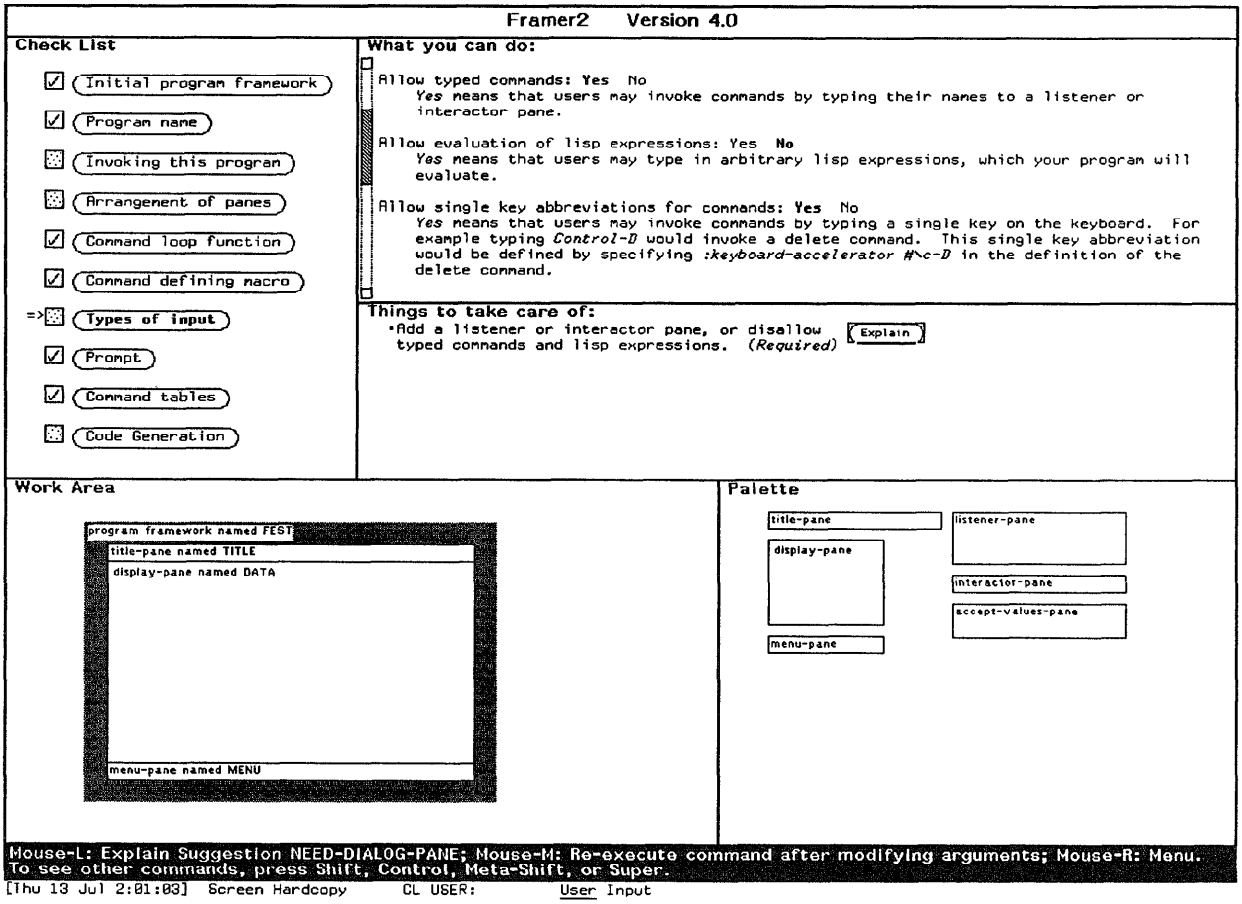

Figure 1: FRAMER

In the situation shown in the figure, the designer makes a decision about what types of user input should be supported in the interface. The system responds to this decision by displaying a critic message in the critic window entitled "Things to take care of." The critic message identifies a discrepancy between the specification sheet (entitled "What you can do") and the work area. The designer can either modify the window layout in the work area or change the specification sheet.

and the suggestion to take care of this is mandatory. Optional suggestions recommend typical design choices, but they can be ignored by the designer if desired. The *Explain* button accesses prestored explanations of why the system critiques and what the designer can do about it. Designers can indicate their intent to disregard the suggestion through the *Reject* operation. For some critic messages, a *Remedy* operation is available; that is, FRAMER can provide a default solution for a problem it has detected.

Critics provide heuristics to decide design questions and point out interactions between different subproblems. The critic knowledge base contains rules about naming the program, arranging window panes, specific knowledge about title panes, dialog panes, and menu panes, and knowledge about invoking a program and selecting interaction modes. These rules are based on a study of existing systems in our computing environment. We interviewed the system designers and elicited the rules they were using. Some of the rules represent system constraints, for example, that a window frame must be completely divided up into panes. Other rules concern the consistency among different applications and functional grouping.

Figure 2 shows a typical critic rule. This rule contains knowledge about the relationship of interaction mode and configuration of window panes in the interface. If the *mouse-and-keyboard* interaction mode is selected, then the rule suggests adding a dialog pane. A *Remedy* action is also defined. Invoking the *Remedy* operation associated with this rule causes the system to add a listener pane at the bottom of the window frame.

The critics in FRAMER1 were passive, i.e., had to be explicitly invoked by the designer. FRAMER1 was tested in a video-taped thinking-aloud study, which showed that the critics substantially improved the performance of user interface designers when compared to a construction kit. But the passive critics failed to be effective in some cases. Subjects invoked the critics only after they thought they

```
;; A critic rule named need-dialog-pane.
(define-critic-rule need-dialog-pane
  ;; Applicability condition. This rule is applicable if the
  ;; interaction mode is mouse-and-keyboard.
  :applicability (equal $interaction-mode
                        mouse-and-keyboard)

  ;; The rule is violated if there is no pane of type dialog-pane
  ;; in the set on inferiors of a program framework.
  :condition
      (not (exists x (type x dialog-pane)))

  ;; The Remedy operation adds a listener-pane.
  :remedy
   (let ((pane(make-instance 'listener-pane
                :x (+ x 20) :y (+ y 184)
                :superior self)))
          (add-inferior self pane)
          (display-icon pane))

  ;; Text of the suggestion made to the user if critic is
  ;; applicable.
  :suggestion
  "Add a listener or interactor pane, or
   set the interaction mode to mouse-only."

  ;; Text for Praise command.
  :praise
  "There is a listener or interactor pane."

  ;; Text for Explain command.
  :explanation "Since the interaction mode
   is mouse-and-keyboard, a dialog pane is
   required for typing in commands.")
```

Figure 2: An Example of a Critic Rule

This is a slightly paraphrased FRAMER critic rule that applies to program frameworks. The rule suggests adding a listener or interactor pane if the interaction mode *mouse-and-keyboard* was specified.

had completed the design. Thus, the critics were not activated early enough to prevent designers from going down garden paths. In FRAMER2, the system described here, an active critiquing strategy has been chosen and has proved to be much more effective.

Specification Sheets. The window layout of an interface has a natural graphical representation as shown in the work area. This is, however, not true of all characteristics of an interface. Behavioral characteristics, for instance, must be described in a different way. In the FRAMER system, these other characteristics are described in a symbolic way as fillers in the fields of a specification sheet (see the "What you can do" window in Figure 1). Through the sheet, the system brings design issues and their possible answers to the user's attention. Associated texts explain the significance and consequences of the different design choices.

In the example of Figure 1, the designer makes a decision about what types of user input should be supported in the interface. The system responds to this decision by displaying a critic message in the critic window entitled "Things to take care of." The system can also respond by modifying the construction situation in the work area. This is accomplished through procedural demons attached to the fields of the specification sheet. The specification sheet is dynamic in that the set of fields in the sheet is dynamically determined based on information that the designer has previously specified.

Catalog. The catalog is a collection of predesigned artifacts illustrating the space of possible designs in the domain. Rather than starting from scratch, the designer starts the design process by invoking the catalog (Figure 3) and selecting a suitable program framework in the catalog. The selected framework is inserted into the work area, and the designer modifies and adapts it to fit the requirements of the problem. Our experiments have shown that use of the catalog can substantially reduce the difficulties in using the design environment. The catalog provides design knowledge in the form of concrete examples that allow reuse and case-based design.

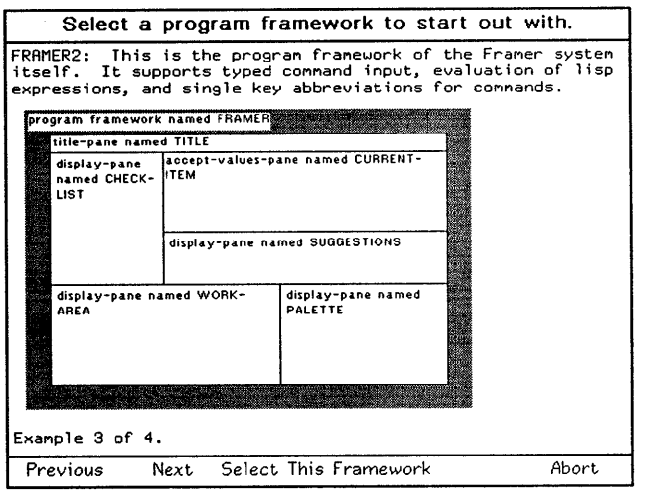

Figure 3: The FRAMER2 Catalog

Users of FRAMER1 tended to design frameworks from scratch without using the catalog. In FRAMER2, the use of the catalog has been made mandatory, which eliminated many low-level tasks. Making the use of the catalog mandatory is not really a restriction because designers can choose very generic frameworks that are almost equivalent to starting from scratch. However, subjects did use more complex examples from the catalog.

Checklist. Another problem in FRAMER1 was that designers who were not familiar with the program framework abstraction were unable to decide what steps had to be done to create a complete functional program framework. The checklist in FRAMER2 addresses this problem by providing the designers with an explicit problem decomposition that is appropriate for the design of program

System Component	User communicates to system	System communicates to user (uninterpreted knowledge)	System communicates to user (interpreted knowledge)
checklist	current focus of attention	how to decompose design problem	raise subproblems depending on information from designer
palette	What primitive components are used in the artifact?	What are the primitive components?	-
specification sheets	User symbolically specifies answers to design issues.	System brings design issues to the designer's attention. System presents potential answers. System explains significance and consequences of different design choices.	System raises design issues depending on information from the designer. System updates artifact according to specified information.
critics	User may reject the system's critique.	-	System points out suboptimal design decisions. System explains why it objects. Critics provide heuristics for making decisions.
catalog	User selects an artifact to reuse and modify.	System provides design knowledge in the form of examples, allows "case-based" design.	-
code generator	-	-	System generates an executable representation of the designated artifact.

Table 1: Components of Cooperative Problem Solving Systems

frameworks.

The checklist serves as the main organizing tool for the interaction with FRAMER. With the checklist, the system indicates to the user how to decompose the problem of designing a program framework, and it helps to ensure that designers attend to all necessary issues, even if they do not know about them in advance. Each item in the checklist is one subproblem of the total design process. By selecting a checklist item, designers tell the system their current focus of attention in the design process.

When the designer selects an subproblem in the checklist, the system responds by displaying the corresponding options in the specification sheet shown in the neighboring "What you can do" window and, thus, provides further detail about the subproblem. The critics are grouped according to the checklist items. The critic pane always displays exactly those critic messages that are related to the currently selected checklist item.

When designers believe that the topic of one checklist item has been completed, they indicate this fact to the system by checking off the associated check box. This causes the system to verify whether all constraints represented in the active critics are satisfied. Only then does the system insert a check mark into the check box. By showing check marks for completed subproblems, the checklist is also a tool for the designer to keep track of which issues have or have not been resolved.

The exact set of checklist items displayed depends on the designer's previous design decisions. The system displays only those items that are currently relevant (i.e., it is context-sensitive); for example, the prompt item is only displayed if command-based interaction is specified.

Code Generator. The ultimate goal of user interface design is the generation of an executable program code, and the design activity supported by FRAMER can be viewed as creating a specification for the code. The code generator component of FRAMER is an formal knowledge source that takes care of creating syntactically correct, executable code.

An Architecture for Cooperative Problem Solving Systems

FRAMER cooperates with the user in a structured dialog mediated through the following system components: checklist, palette, specification sheets, critics, catalog, and code generator. Table 1 shows how these different components contribute to the cooperative problem solving process.

The cooperative system architecture of FRAMER was designed to cope with the ill-structured nature of the user interface domain. Most cooperative design support systems operate in well-defined domains. For example, PRIDE (Mittal, Araya, 1986) operates in the well-defined domain of paper path design for copiers. In this domain, the design problem can be completely specified and decomposed in advance, and for each design question there is a well-known set of possible answers. These

premises are not true for the user interface domain. The challenge for the FRAMER system was to define an architecture that can support designers effectively even if the system's knowledge is incomplete.

Conclusions and Ongoing Research

The goal of this work is to build a cooperative support system for user interface design. For cooperative systems, not only internal representation and reasoning mechanisms but, in particular, the external presentation and communication of that knowledge to the user is of crucial importance. The proposed architecture provides a migration path from simple tool kits to sophisticated design environments. By incrementally adding relatively simple components such as critics and checklists, the utility of a support system can be significantly improved.

Our approach was driven by the needs of designers, i.e., their needs for support in decomposing the problem, finding applicable building blocks, and understanding the effects of design decisions. Knowledge-based design environments are unique in addressing these needs with a rich set of semi-formal and formal knowledge sources.

The FRAMER system is an object of ongoing research in several directions. The existence of the knowledge sources in FRAMER does not guarantee that users find and take advantage of them, and the control of the user's attention to the great variety of available information becomes a problem. We are investigating ways to control attention using a cognitive modeling approach using the construction integration model of cognition (Kintsch, 1989).

Another active research area is the design of generalizations of the checklist and the specification sheets. These two components taken together represent a two level hierarchy of design issues. We are extending this to an unlimited number of levels by using the concept of issue-based information systems (IBIS) in the form of (McCall, 1987). Issue-based information systems represent argumentative design knowledge as hierarchies of issues, answers, and arguments for or against choosing those answers. To make an IBIS component more responsive, we are adding active mechanisms similar to the ones found in the checklist and the specification sheets.

References

B.G. Buchanan, E.H. Shortliffe (1984). *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project.* Reading, MA: Addison-Wesley Publishing Company.

W.A.S. Buxton, M.R. Lamb, D. Sherman, K.C. Smith (1983). Towards a comprehensive user interface management system. *Computer Graphics, 17*(3), 35-42.

C.M. Eastman (1969). Cognitive Processes and Ill-Defined Problems: A Case Study from Design. *Proceedings of the International Joint Conference on Artificial Intelligence,* 669-675. Los Altos, CA: Morgan Kaufmann Publishers.

G. Fischer (1988). Cooperative Problem Solving Systems. *Proceedings of the 1st Simposium Internacional de Inteligencia Artificial (Monterrey, Mexico),* 127-132.

G. Fischer (1989). Human-Computer Interaction Software: Lessons Learned, Challenges Ahead. *IEEE Software, 6*(1), 44-52.

G. Fischer, A.C. Lemke, T. Mastaglio, A. Morch (1990). Using Critics to Empower Users. *Human Factors in Computing Systems, CHI'90 Conference Proceedings (Seattle, WA),* 337-347. New York: ACM.

G. Fischer, A.C. Lemke (1988). Construction Kits and Design Environments: Steps Toward Human Problem-Domain Communication. *Human-Computer Interaction, 3*(3), 179-222.

J. Grudin (1989). The Case Against User Interface Consistency. *Communications of the ACM, 32*(10), 1164-1173.

R. Jeffries, A.A. Turner, P.G. Polson, M. Atwood (1981). The Processes Involved in Designing Software: In J.R. Anderson (Ed.), *Cognitive Skills and their Acquisition* (pp. 255-283). Hillsdale, NJ: Lawrence Erlbaum Associates.

W. Kintsch (1989). The Representation of Knowledge and the Use of Knowledge in Discourse Comprehension: In R. Dietrich, C.F. Graumann (Eds.), *Language Processing in Social Context* (pp. 185-209). Amsterdam: North Holland.also published as Technical Report No. 152, Institute of Cognitive Science, University of Colorado, Boulder, CO.

R. McCall (1987). PHIBIS: Procedurally Hierarchical Issue-Based Information Systems. *Proceedings of the Conference on Architecture at the International Congress on Planning and Design Theory.* New York: American Society of Mechanical Engineers.

J. McDermott (1982). R1: A Rule-Based Configurer of Computer Systems. *Artificial Intelligence.*

S. Mittal, A. Araya (1986). A knowledge-based framework for design. *Proceedings of AAAI-86,* 856-865. Los Altos, CA: Morgan Kaufmann.

G. Peper, C. MacIntyre, J. Keenan (1989). Hypertext: A New Approach for Implementing an Expert System. *Proceedings of Expert Systems ITL Conference.*

H.W.J. Rittel (1972). On the Planning Crisis: Systems Analysis of the First and Second Generations. *Bedriftsokonomen*(8), 390-396.

H.A. Simon (1973). The Structure of Ill-Structured Problems. *Artificial Intelligence*(4).

G. Singh, M. Green (1989). A high-level user interface management system. *Human Factors in Computing Systems, CHI'89 Conference Proceedings (Austin, TX),* 133-138. New York: ACM.

W.R. Swartout, R. Balzer (1982). On the Inevitable Intertwining of Specification and Implementation. *Communications of the ACM, 25*(7), 438-439.