KNOWLEDGEABLE DEVELOPMENT ENVIRONMENTS USING SHARED DESIGN MODELS

Robert Neches, Jim Foley, Pedro Szekely, Piyawadee Sukaviriya, Ping Luo, Srdjan Kovacevic, Scott Hudson

USC / Information Sciences Institute and Georgia Institute of Technology

ABSTRACT

We describe MASTERMIND, a step toward our vision of a knowledge-based design-time and run-time environment where human-computer interfaces development is centered around an all-encompassing design model. The MASTERMIND approach is intended to provide integration and continuity across the entire life cycle of the user interface. In addition it facilitates higher quality work within each phase of the life cycle. MASTERMIND is an open framework, in which the design knowledge base allows multiple tools to come into play and makes knowledge created by each tool accessible to the others.

KEYWORDS: models, collaboration, design, development

INTRODUCTION

The challenge facing the research community is to provide the bass for an effective, integrated suite of tools to support the entire lifecycle of an interface. This means that the tools must be given a great deal more knowledge than they currently have about the product they are intended to construct. It means that this knowledge must be preserved and shared between tools across the software lifecycle.

In an effort to move our research in this direction, we at Information Sciences Institute and Georgia Tech have been collaborating on the design of a shared system called MASTERMIND, which is comprised of a knowledge base, a design-time environment, and a run-time environment. In MASTERMIND (which stands for ***M**odels **A**llowing **S**hared **T**ools and **E**xplicit **R**epresentations to **M**ake **I**nterfaces **N**atural to **D**evelop*), the knowledge base serves as an integrating framework that allows separate tools to integrate into the design- and run-time environments.

Part of our underlying thesis in MASTERMIND is that models of interface concepts need to be a shared community resource that drives the creation of an architecture and tool suite for design, development, and maintenance. If knowledge of these concepts can be built into the tools, then greater assistance can be provided earlier in the design process, individual tools will become much more interoperable, and it will become possible to build knowledge bases about particular designs which can greatly facilitate their maintenance and extension.

These benefits come at a cost -- modelling entails a certain degree of additional effort. However, our argument is that this cost can, and should, be paid primarily when creating tools and environments rather than when building applications. Creating knowledgeable development environments is the way to provide the benefits of a model-based approach to application developers without making modeling too burdensome to be practical.

We will develop our view of a community-resource knowledge base according to the following exposition. First, we will describe the issues that arise over the course of the software lifecycle for a user interface design. We wish to make two major points from that analysis: (1) each phase is facilitated if we can carry over knowledge from previous phases; and (2) it is possible to identify the nature of the knowledge that needs to be carried over.

Having argued generally that this carryover is beneficial, next we will point out specific complementary benefits that arise from using shared models to combine tools developed under two complementary model-based approaches: the HUMANOID effort ongoing at Information Sciences Institute and the UIDE work at George Washington University and Georgia Tech.

After reviewing the leverage that these tools provide each other, our next topic will be the mechanisms that will allow them to be combined. In particular, we will describe our progress toward a unified model that supports prototyping from partial specifications, design critiquing, context-sensitive control of presentations, and context-sensitive animated help and tutorials.

Once that unified model has been explained, we will then turn to a consideration of the practical issues that must be addressed in moving toward an open, extensible environment in which such a model can serve to bring together our tools. We will close by speculating about the possibilities that this approach opens up for integrating and disseminating the results of research in the HCI community.

AN ANALYSIS OF THE UI LIFECYCLE: WHY A MODEL-BASED APPROACH IS NEEDED

Development of a user interface starts with an existing system (computerized or manual) that must be analyzed in order to understand what users need to accomplish and where the bottlenecks lie in attempting to do so. This problem identification process, which relies on techniques for task analysis and user monitoring, leads to the definition of a specific design problem. Elements of that design problem, at

this point in the process, involve a description of the task and identification of requirements for improvements in quality, speed, and/or accuracy of particular task components. Today, that task description is rarely made explicit (although techniques exist to do so [10]). Little help beyond force of will is available to ensure that the design evolves in line with that description. Yet the task analysis deals in goals, operators, methods, and selectors -- elements that, as we will see, are part of the interface design representation. Properly modeled, task analyses could feed directly into the design.

In the next phase, conceptualization, *design policies* need to be set in order to provide for an interface which addresses the task analysis and requirements resulting from problem identification. Conceptualization, and the prototyping phase which follows it, can be viewed as a search through a space of alternative designs. This notion of search for a design that satisfices (rather than necessarily optimizing) multiple criteria is central to current research trends. Conceptualization formulates design policies that define regions in the space. Prototyping works within those abstractions to create a specific design specified at an executable level.

In particular, elements of a conceptualization describe design commitments. These include decisions about the choice and nature of application and interaction objects presented to users through the interface. Other commitments involve policy decisions about choices of interaction paradigms and dialogue techniques, as well as the general look-and-feel offered via input and output media. If we wish to express these commitments explicitly, then we benefit from having a model of tasks since the design policy commitments made during conceptualization build on our assumptions about the activities that the interface will support.

Many design commitments are made during these phases. It is only in the next phase, prototyping, that the design representation grows to include actual executable software. Unfortunately, the current generation of tools ignores the earlier phases. Interface builders and other interface programming aids really only help in creating code after the designer has a sense of what is wanted. As we have argued elsewhere [22], although some experimentation is possible, the cost of backing away from a commitment is quite high once much software is built.

A great deal is to be gained by maintaining an explicit declarative representation that covers both the design model and the code implementing it. Such a representation enables semi-automated design critics to evaluate the design with respect to issues such as usability and learnability. By providing higher levels of abstraction at which to specify the interface, it also empowers more rapid exploration of design alternatives and therefore faster arrival at a satisfactory design. A representation of the design goals allows us to provide help in managing the activities required to implement design policies.

As the software lifecycle proceeds into usage and maintenance phases, knowledge accumulated in the previous phases can be put to good use -- but only, of course, if there is a model that preserves it for use by tools in the run-time environment. In particular, knowledge of how the design mapped its model of the application onto its model of presentation methods is important, as is knowledge about tasks and goals. Carrying this knowledge over from design-time to run-time allows us to program systems that can make context-sensitive decisions about the best presentation technique to use for particular data. It allows us to define help and guidance systems that can help with how-to questions, that know enough about the presentation to be able to generate effective animations, and that maintain the accuracy of their help without extra programming effort because their help is generated from the design itself.

In summary, a declarative model-oriented approach allows separate tools, operating at very different times thoughout the lifecycle, to take advantage of knowledge collected by other tools and thereby build better interfaces with less effort. To accomplish this, we need a model capturing:

- task structure, and the goals, subgoals, operators, methods and selectors which comprise the means for accomplishing tasks
- conceptual design abstractions and policy decisions about structural and functional properties of the interface which constrain a particular design
- mappings of conceptual structure to uses of i/o media in system displays
- mappings of low-level, empirically-recordable user gestures onto higher-level semantics recorded in the design model

There are several advantages to this approach. The declarative model is a common representation that tools can reason about, and allows the tools that operate on it to cooperate. Because all components of the system share the knowledge in the model, the model promotes interface consistency within and across systems and reusability in the construction of new interfaces. Also, the declarative nature of the model allows system builders to more easily understand and extend the model.

CARRY-OVER OF KNOWLEDGE BETWEEN DESIGN-TIME AND RUN-TIME TOOLS AND ENVIRONMENTS

We have built a number of tools which operate at design time and at run time by making use of the kind of knowledge just listed.

ISI's model-based user interface development environment is HUMANOID [20, 21, 22]. Its contribution to interface design is that it lets designers express abstract conceptualizations in an executable form, allowing designers to experiment with scenarios and dialogues even before the system model is completely concretized. The consequence is that designers can get an executable version of their design

quickly, experiment with it in action, and then repeat the process after adding only whatever details are necessary to extend it along the particular dimension currently of interest to them.

HUMANOID models the functional capabilities of the system as a set of objects and operations, and partitions the model of the style and requirements of the interface into four dimensions that can be varied independently:

1. Presentation. The presentation defines the visual appearance of the interface.
2. Manipulation. The manipulation specification defines the gestures that can be applied to the objects presented, and the effects of those gestures on the state of the system and the interface.
3. Sequencing. The sequencing defines the order in which manipulations are enabled. Many sequencing constraints follow from the data flow constraints specified in the system functionality model (e.g., a command cannot be invoked unless all its inputs are correct). Additional constraints can be imposed during dialogue design.
4. Action side-effects. Action side-effects refer to actions that an interface performs automatically as side effects of the action of a manipulation (e.g., a newly created object can become automatically selected).

HUMANOID provides facilities to incrementally refine the system functionality model and to refine any of the dimensions of interface style to allow the exploration of a large set of interface designs, while allowing the design to be executed at any time.

In addition to supporting design exploration, HUMANOID's model allows it to construct displays whose characteristics depend on the runtime values of system data structures. HUMANOID reasons about the values of the data structures and the presentation policies defined in the presentation dimension of interface style to determine the resulting presentation. HUMANOID's model also allows it record the dependencies between displays and system data structures, enabling it to automatically update the displays when the data structures change.

Georgia Tech's model-based user interface development environment is UIDE, the User Interface Design Environment [3, 4, 6, 7]. UIDE's models support rich descriptions of the application. The basic elements of the model are: the class hierarchy of objects which exist in the system, properties of the objects, actions which can be performed on the objects, units of information (parameters) required by the actions, and pre- and postconditions for the actions.

A variety of run-time and design-time uses have been made of the representation. For design time, tests have been developed for certain aspects of completeness, consistency and command reachability [4, 1]. UIDE can automatically organize menus and dialogue boxes [11], including use of style-guide knowledge encapsulated in a rule base [2]. It can automatically create an interface to the application, using menus, dialogue boxes, and direct manipulation [6]. It has been extended to evaluate the interface design with respect to speed of use, using a key-stroke model type of analysis which accounts for different interaction techniques and action sequences [16].

At run-time, UIDE can explain why a command is disabled (based on false predicates in its preconditions), and partially explain what a command does (based on the semantics implied by its preconditions, postconditions, and action class [4]. It can provide procedural help, via animation of a mouse and keyboard on the screen, taking into account the current application context [18, 19]. Specifically, the sequence of commands which must be executed to carry out a (potentially disabled) command is animated, based on back-chaining from the target command. Finally, it can control actual execution of the application, including enabling and disabling of menu items, as well as display of menus, dialogue boxes, and windows [6, 8].

PROGRESS TOWARD A UNIFIED MODEL

Both our groups start from a base of implemented software, which is written in terms of their own current generic model, and which processes declarative user interface design specifications written in the terminology defined by their generic model. Our work therefore begins with aligning the models, producing an initial knowledge base that merges the best representational approaches of each. For example, the ISI model has a richer and more flexible approach to specifying interactive dialogues, while Georgia Tech's is stronger when describing the effects of commands.

Our call for explicit user interface design models is an interesting application of the DARPA Knowledge Sharing Effort's development methodology for large knowledge-based systems [12]. In the Knowledge Sharing Effort's methodology, sharing and reuse of software is greatly facilitated by adopting a common ontology: i.e., a set of agreements about how to model the topic area. Their work is developing tools to facilitate the evolution of such ontologies, so there are compelling opportunities for that line of work to leverage user interface research and vice versa.

The problems in defining an ontology of user interface designs are to structure the design space into relatively orthogonal dimensions, and to provide a characterization of implications and interdependencies between design commitments. Structuring the design space organizes design tools so that any aspect of a design can be revised with minimal necessity to recode other aspects. Modeling implications and interdependencies lets design spaces be pruned more quickly, by using knowledge to restrict the search to alternatives consistent with current design commitments.

The MASTERMIND Generic Model

As it stands now, our models for interface development

contain the following kinds of information.

Application Semantics. The application semantics is a description of the functional capabilities of the system as a set of *objects* and *commands*. In building a model of the application semantics for an interface design, the designer is making explicit what we earlier called the conceptual design of the system. That is, without making commitments about the appearance or behavior of the interface, the designer's model of application semantics captures abstract commitments about the capabilities that the interface will offer and the type of information it will allow users to see and manipulate. The MASTERMIND generic application semantics model defines the vocabulary in which these commitments can be expressed.

Figure 1 shows the part of that model representing objects, a fusion of the models in [21, 22] and [4, 6, 7]. The model contains a superset of the information contained in the definition of a class in typical object-oriented programming languages. Object class definitions typically state only the slots of an object and the types of values that each slot can contain. The additional knowledge represented in our model, in attributes such as *formatter, slot-class* and *validator,* is used by various components of the design and run-time tools.

For example, the *formatter* attribute contains knowledge that the interface software needs to translate between the internal representation of an object and textual forms (e.g., to construct the labels of menus that allow the user to choose from a set of objects). *Parsers* contain knowledge to convert from a textual representation of an object to its internal form, which is used by interfaces that allow the user to type in the identifier of an object. *Validators* attached to the object model tell how to check consistency of values supplied when a user attempts to input an instance of that class. Organizing knowledge in this fashion facilitates prototyping of partial designs, because it allows the system to use class inheritance to fill in parsers and formatters from the generic model for use during execution of designs for which more application-specific methods have not yet been provided.

Two unusual pieces of knowledge in the model of object slots are the *slot-class* and the *validator*. The slot-class contains knowledge about the semantics of the slot that the presentation component can use to aid in the design of displays. For example, one kind of slot-class in our model is called *Part-Of;* it indicates that the values of the slot are in a part-of relationship with respect to the object. Such knowledge can be used to pick out certain presentatation methods and rule out others.

The unique aspect of validators is that they contain, in addition to a procedure to test a condition (*predicate*), a specification of the error messages to show the user for the different error conditions that the validator can detect (*error-conditions*). Storing the error messages with the validator separates the representation of the error messages from the presentation techniques used to communicate them to the user. This gives the presentation component the flexibility to choose a presentation technique appropriate to the current situation.

The object model, which comes mostly from UIDE, together with the presentation model, which comes mostly from HUMANOID, enables MASTERMIND to provide capabilities unavailable in UIDE or HUMANOID. For example, the object model provides design-time information that DON, the automatic dialogue-box generation component of UIDE uses to group and select the interaction techniques in a dialogue box. Similar uses of the object model could be incorporated into HUMANOID, to increase HUMANOID's ability to automatically design displays, while conserving the context-sensitive presentation capabilities of HUMANOID.

Figure 2 shows the MASTERMIND *command* model, derived from HUMANOID's and UIDE's. Commands model the operations that can be performed on objects.

The command model contains knowledge about the *inputs* of a command, the conditions under which the command can be executed (*preconditions, exceptions, validator*), and the effects of the command (*post-conditions* and *side-effects*). The run-time environment uses some of this knowledge to acquire values for nputs from the user: the legal values of the inputs (*type, validator, alternatives, min, max*), default values, parsers and formatter. Knowledge from the command model is also used to control the sequencing for acquiring the input values from the user.

The preconditions, postconditions, exceptions and side-effects provide knowledge about the semantics of an operation that can be used by many tools. For example, the animated help generation system uses preconditions and postconditions to figure out the sequence of actions that a user needs to perform to carry out a task. The presentation component enables and disables menu-items when the preconditions of commands change. The help system can explain why a command is disabled based on unsatisfied preconditions and whether the values of inputs are incorrect or missing.

Presentation and Behavior. The presentation model describes the visual appearance of the interface, and the behavior model defines the gestures that can be applied to the objects presented, and the effects of those gestures on the state of the system and the interface.

Figure 3 shows MASTERMIND's merger of the presentation and manipulation models in HUMANOID and UIDE. A presentation is modeled as a composition of simpler presentations called *parts*. In addition to the parts, the model contains knowledge about the *layout* of the parts, the kind of *data* that the presentation can display, the contexts in which the presentation is appropriate (*applicability-condition*), the input *behaviors* associated with the presentation, and other presentations that might be more

appropriate in certain contexts (*refinements*).

Each part of a presentation contains knowledge about conditions when the part should be included in the complete presentation (*inclusion-condition*), knowledge that allows a part to be replicated when the data to be presented is a list (*replication-data*), and knowledge about different *choices* of presentation methods for displaying that part .

The model of behaviors is based on the Garnet Interactors Model. Briefly, a behavior describes the area of a presentation where it is active, the events that invoke it and stop it, and the action to be executed (see [Garnet-Interactors] for more details).

The model of presentation and behavior is used by the run-time system to generate context-sensitive presentations by matching the types in the slots of objects with the types and predicates in the data attributes of presentations.

Together, the presentation and command models let MASTERMIND-based interfaces provide animated help for free. The animation generation works from the command model to figure out the sequence of steps to animate, and from the presentation model to construct the contents of the animation. Animation generation is a compelling example of the benefit of the MASTERMIND approach because it piggybacks on knowledge that is in the model for other purposes.

Sequencing and Action Side-Effects. Sequencing defines the order in which input behaviors are enabled. Action side-effects refer to actions that an interface performs automatically as side effects of the action of a manipulation (e.g., a newly created object can become automatically selected).

Our model of sequencing and side-effects is described in detail in [HUMNAOID] and [UIDE]. The main feature of the model, that distinguishes it from the models used in other UIMSs, is that sequencing is not represented explicitly, either as a finite state machine or an event system. Instead, sequencing constraints are derived from the data flow constraints specified in the system functionality model (e.g., a command cannot be invoked unless all its inputs are correct) and from the preconditions, postconditions and exceptions of a command. Additional sequencing constraints (e.g. that certain inputs should be prompted for in sequence) are defined by annotating groups (Figure B) with declarative descriptions of the sequencing desired.

Planned Extensions to the Model. A number of design issues are not currently covered in our model. Our future plans include coverage of: policies that define global style characteristics of the interface, characteristics of the delivery platforms, end-user characteristics and preferences, and user tasks.

Design and Execution in MASTERMIND

In our model-based approach, interface developers specify interfaces by modeling the desired features declaratively in terms defined in the generic knowledge base. Unlike the traditional approach to interface construction, where programmers spend most of the effort writing and debugging procedural code, our goal is for developers using MASTERMIND to spend the bulk of their effort writing declarative specifications that extend and specialize the generic model. As these specifications evolve, the tools that we described in the previous section can interpret those specfications to provide assistance in critiquing the design, executing and evaluating partially specified designs, and managing the activities necessary for extending the specifications.

The run-time environment of an application developed with MASTERMIND consists of a standard software module that is a component of every application program. The run-time component module uses the model of the application and its interface, along with knowledge about the state of the application program's run-time data structures, in order to generate and control the interface of the application, interpret inputs, and provide help to the end user.

To interpret inputs, the run-time system uses the presentation model to map the input event into the application data referenced by it, and triggers the appropriate commands according to the application's model of behavior and sequencing.

To produce or update the display of an application data structure, the run-time system queries the model for a presentation component capable of displaying the data structure. The model returns the most specific presentation component suitable for displaying the data structure in the given context (e.g. taking into account data type congruence and size restrictions), and the run-time system uses it to produce or update the display. Note that the presentation component obtained from the model might either be a default inherited from Mastermind's generic knowledge base, or a more specific presentation component specified by an interface's designer. This mechanism is the key to two valuable properties of MASTERMIND: (1) built-in support for context sensitive presentation; and, (2) the ability to generate default behavior that fills in for deferred design commitments, thereby making even incomplete specifications executable and testable.

RELATED WORK

Other user interface management systems which derive the user interface from a high-level specification of the semantics of a program are MIKE [15], and UofA* [17], which are able to generate a default interface from a minimal application description, and provide a few parameters that a designer can set to control the resulting interface.

Our model of commands allows designers to exert much finer control over dialogue sequencing. In addition, we provide a library of command groups that allows designers to very easily specify the dialogue structures that MIKE and

UofA* support. We also provide finer control over presentation design, and offer richer descriptions of application semantics that can be used to support more sophisticated design tools.

Interface builders such as the Next Interface Builder [14], and OpenInterface [13] are a different class of tools to aid in the design of interfaces. These tools make it very easy to construct the particular interfaces that they support, but are very poor for design exploration. Designers have to commit to particular presentation, layout and interaction techniques early in the design. Making global policy changes, such as changing the way choices are presented, is difficult because it requires manually editing a large number of displays. A model-based approach handles both these problems.

CONCLUSIONS

An overall architecture centered around an all-encompassing design model would provide integration and continuity across the entire lifecycle of a user interface in addition to enabling more powerful results within each phase. Today's interface development environments are primitive with respect to what is needed.

The state of the art today is an architecture consisting of a library of low-level objects like menus and buttons, and specification and prototyping tools consisting of aids for drawing what individual displays should look like. Prototyping in today's environment really means that -- if enough code is also written -- you can test the interface before the application is done. However, it does not mean that it is easy to experiment with different interface designs or easily see how a partially conceptualized design might look.

The opportunity exists to go far beyond this, not by throwing away that architecture, but by building on it. MASTERMIND is a first step in that direction.

MASTERMIND is best thought of as a framework that others can build on, with some pieces instantiated. The framework supports design, execution, help, and maintenance for well-designed user interfaces to advanced applications. We have identified certain design tools that fill missing needs: visual aids for developing design models, tools for managing and automating multi-step design refinements, and critics based on design policies. Because the framework is open, other tools can be added later. For example, if the pychological research on analyzing the usability of proposed designs matures to the the point where it enables creation of automated design usability critics, the extensible nature of the MASTERMIND design makes it feasible for other researchers to add those tools.

Similar observations apply to the MASTERMIND run-time environment. For example, as psychological research progresses on identifying bottlenecks in the use of implemented designs, it should be easily possible to augment the run-time environment in order to collect and analyze performance data from users interacting with the interface system. In fact, the user task models that we plan to build in order to support more sophisticated help and interactive guidance may well contribute to such an extension.

For these reasons, MASTERMIND offers a valuable path toward a comprehensive, interoperable suite of tools, what the recent ISAT study on intelligent interfaces referred to as a *knowledgeable development environment*. [9]

Although MASTERMIND only instantiates a portion of that comprehensive framework, it has significant merit in its own right. Among other innovations, it will represent a major step toward explicit representation and support for early, conceptual phases of design. Designers can partially describe their designs, by providing descriptions of application functionality and data structures or by using abstractions about presentation, manipulation, or sequencing. Because this approach allows execution and testing of partially-specified designs, because it also facilitates exploration of design alternatives, and because it allows stating and enforcing high-level design policies, MASTERMIND will facilitate rapid production of much more thoughtfully designed user interfaces.

MASTERMIND uses the knowledge created in the design process to provide useful run-time services, such as context-sensitive presentation and help, which would not be possible without a design model.

An integrated set of easy-to-use tools with the above properties would provide a much faster and cheaper path to the creation of usable, maintainable, and better-adapted interfaces. Knowledgeable development environments would dramatically change the nature of interface system development. It would ease the task of initial design. It would let design and evolution extend throughout the lifecycle, and it would soften unhealthy boundaries between designers and end users.

The result would be improvements in the quality, cost, and production time of advanced user interfaces. Quality would increase because first-pass interface designs would be better, because there would be more opportunity to iteratively refine the designs, and because end users would have greater participation and influence in ensuring that their needs and limitations were addressed. Cost would decrease because interfaces could be developed and tested much more quickly, because better adaptivity to task requirements would simplify training and because better design would enhance user productivity. Production time would be speeded because prototyping would be faster and more complete, because the distinction between prototypes and deployed systems could be blurred/eliminated, and because generation of informational materials would not entail extra effort.

Thus, we believe that much more powerful systems can be built much more quickly in the future -- if two conditions are met:

- we organize our development and maintenance tools around explicit models
- we begin, as a community, to work towards sharing common models

Doing so will allow the research community to compose our tools together to create development and maintenance environments far superior to what any of us could build alone.

ACKNOWLEDGEMENTS

The research reported in this paper was supported by DARPA through Contract Numbers NCC 2-719 and N00174-91-0015 at ISI, and by grants from SUN, Siemens, and the State of Georgia at Georgia Tech.

REFERENCES

[1] Braudes, R.E., and J.L. Sibert, "ConMod: A System for Conceptual Consistency Verification and Communication," *SIGCHI Bulletin* 23(1), Jan. 1991, pp.92-94.

[2] DeBaar, D, K. Mullet, and J. Foley. Coupling Application Design and User Interface Design, Proceedings CHI'92 - SIGCHI 1992 Computer Human Interaction Conference, ACM, New York, NY, 1992, in press.

[3] Foley, J., C. Gibbs, W. Kim, and S. Kovacevic, A Knowledge Base for a User Interface Management System, Proceedings CHI '88 - 1988 SIGCHI Computer-Human Interaction Conference, ACM, New York, 1988, pp. 67-72.

[4] Foley, J., W. Kim, S. Kovacevic, and K. Murray, Designing Interfaces at a High Level of Abstraction, IEEE Software, 6(1), January 1989, pp. 25-32.

[5] Foley, J., A. van Dam, S. Feiner, and J. Hughes, Computer Graphics – Principles and Practice, Addison-Wesley, Reading, MA, 1990.

[6] Foley, J., W. Kim, S. Kovacevic and K. Murray, UIDE - An Intelligent User Interface Design Environment, in J. Sullivan and S. Tyler (eds.) Architectures for Intelligent User Interfaces: Elements and Prototypes, Addison-Wesley, Reading MA, 1991, pp.339-384.

[7] Foley, J., D. Gieskens, W. Kim, S. Kovacevic, L. Moran and P. Sukaviriya, A Second-Generation Knowledge Base for the User Interface Design Environment, GWI-IIST-91-13, Dept. of Electrical Engineering and Computer Science, The George Washington University, Washington DC 20052, 1991.

[8] Gieskens, D. and J. Foley, Controlling User Interface Objects Through Pre- and Postconditions, Proceedings CHI'92 - SIGCHI 1992 Computer Human Interaction Conference, ACM, New York, NY, 1992, in press.

[9] Intelligent User Interfaces. ISI/RR-91-288, USC/ISI, 4676 Admiralty Way, Marina del Rey, CA 90292, September 1991.

[10] John, B. E., Extensions of GOMS Analyses to Expert Performance, Requiring Perception of Dynamic Visual and Auditory Information, Proceedings of ACM CHI'90 Conference on Human Factors in Computing Systems, pp. 107-115.

[11] Kim, W. and J. Foley, DON: User Interface Presentation Design Assistant, Proceedings SIGGRAPH Symposium on User Interface Software and Technology, ACM, New York, 1990, pp. 10-20.

[12] R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, and W.R. Swartout. Enabling Technology for Knowledge Sharing. AI Magazine, Volume 12 No. 3 , Fall 1991, pp.36-56.

[13] Neuron Data, Inc. 1991. Open Interface Toolkit. 156 University Ave. Palo Alto, CA 94301.

[14] NeXT, Inc. 1990. Interface Builder, Palo Alto, CA.

[15] D. Olsen. MIKE: The Menu Interaction Kontrol Environment. ACM Transactions on Graphics, vol 17, no 3, pp. 43-50, 1986.

[16] Senay, H., P. Sukaviriya, L. Moran, Planning for Automatic Help Generation, Proceedings of Working Conference on Engineering for Human Computer Interactions, IFIP, August 1989.

[17] G. Singh and M. Green. A High-level User Interface Management System. In Proceedings SIGCHI'89. April 1989, pp. 133-138.

[18] Sukaviriya, P., Dynamic Construction of Animated Help from Application Context, Proceedings of ACM SIGGRAPH 1988 Symposium on User Interface Software and Technology (UIST '88), 1988, ACM, New York, NY, pp. 190-202.

[19] Sukaviriya, P and J. Foley, Coupling a UI Framework with Automatic Generation of Context-Sensitive Animated Help, Proceedings of ACM SIGGRAPH 1990 Symposium on User Interface Software and Technology (UIST '90), ACM, New York, 1990, pp. 152-156.

[20] P. Szekely. Standardizing the interface between applications and UIMS's. In Proceedings UIST'89. November 1989, pp. 34-42.

[21] P. Szekely. Template-based mapping of application data to interactive displays. In Proceedings UIST'90. October 1990, pp. 1-9.

[22] P. Szekely, P. Luo, and R. Neches. Facilitating the Exploration of Interface Design Alternatives: The HUMANOID Model of Interface Design. In Proceedings of CHI'92, The National Conference on Computer-Human Interaction, May, 1992, pp. 507-515.