A Knowledge-based Design Environment for Graphical Network Editors

Scott Henninger, Andrea Ignatowski, Christian Rathke, David Redmiles

Department of Computer Science and Institute of Cognitive Science
University of Colorado, Campus Box 430
Boulder, CO 80309

ABSTRACT

Design systems for graphical network editors are general purpose tools that capture common characteristics of network like structures. As a consequence, these systems support their users only as far as the network functionality is concerned. While this is important, it is not enough to effectively support users in designing specific network viewers or editors for specific applications. Without knowledge about the application domain, for which the editor is designed, there is little potential to guide the user or make suggestions for a better design.

We have developed a graphical editor design environment that incorporates and applies knowledge about application domains. Our goal is to boost a design environment closer towards its application. As an example of this new generation of design support systems we have developed a design environment for graphical editors in the domain of object-oriented inheritance networks. In addition to the general knowledge about graphs, the system knows about inheritance mechanisms in object-oriented systems, it knows about the nodes being classes and the links representing the superclass relation. This knowledge is used to provide guidance, critique, and constraints.

Introduction

Graphs are visualizations of aspects of application systems. Nodes represent entities and connections between nodes represent relations. Whether they display telephone networks, dependencies between events, functional or structural decompositions, points in problem spaces, nodes in a search tree, or family relationships: the list of possibilities for applying graphs is endless. Even in the more restricted domain of software systems, graphs play a major role in their being used as flowgraphs, callgraphs, inheritance graphs, and rule dependency graphs.

Graphical representations of software become even more useful when they can be generated and manipulated on a display screen. On paper, graphs serve as a means for communication between humans; on a display screen, graphs can also serve as a way of communicating information between a software system and its human user. These graphs can be generated automatically from the software system. Direct manipulation [8] of graphs can be translated into modifications to the software system (see [2] for more examples of the usage of graphs for programming).

Graphical representations are specific to their functions. A flowgraph differs significantly from an inheritance hierarchy graph or a data dependency graph. Appearance as well as functionality are specific to the respective application. It is therefore no surprise that most graph representations are specifically designed for the purposes of the respective systems. Despite these differences, there are many commonalities among different graph representations. Problems of shape, layout, node, links, connectivity, and interaction are present in any graphical system. Some generic graph packages try to capture these issues by providing a set of abstractions out of which specific graph applications can be built [1; 9; 11; 14]. Application designers use these packages as modules in their programs.

Often, graph packages can only be used on the programming language level. In order to adapt them and link them to a specific application, knowledge about the syntax of the programming language and the semantics of the graph package is required. Not only does a special "language" have to be learned but also an "understanding" of the way abstractions are used in the graph systems has to be acquired. The latter is often the more difficult part because it includes knowledge about the range and structure of the available modules and knowledge about when to use which part. Usually it takes some experience to make efficient use of the full power of such a generic graph system.

In our research we investigate methods and techniques to make useful systems (e.g., generic graph packages) more usable by devising *design environments* that support their users in applying them to *their specific problems* [4]. Fischer and Lemke [5] describe the TRIKIT system which illustrates a design environment for graphical network editors.

TRIKIT is a design environment for combining components of a generic graph package with a specific application. It presents itself to the user as a collection of interaction sheets as shown in Figure 1. Using these interaction sheets the designer specifies the interface to the application. The designer specifies in terms of the application what it means to create and delete a node or to insert and remove a link. Here the designer chooses the desired graphical representation for the nodes of the graph and controls the creation of the user interface. This design process is carried out above the code level. TRIKIT hides from the designer, program code that is generated when the network editor is created or modified.

Design environments for graphical network editors such as TRIKIT are general purpose tools that capture common characteristics of network-like structures. As a consequence, these systems support their users only as far as the network functionality is concerned. While this is important, it is not enough to effectively support users in designing *specific* network viewers or editors for *specific* applications. This paper describes an augmentation to design environments by including *domain specific knowledge* about the design task. Without this knowledge of the application domain, there is little potential to guide the user or make suggestions for a better design.

A graphical editor design environment called NODE (Network

0073-1129/89/0000/0881$01.00 © 1989 IEEE

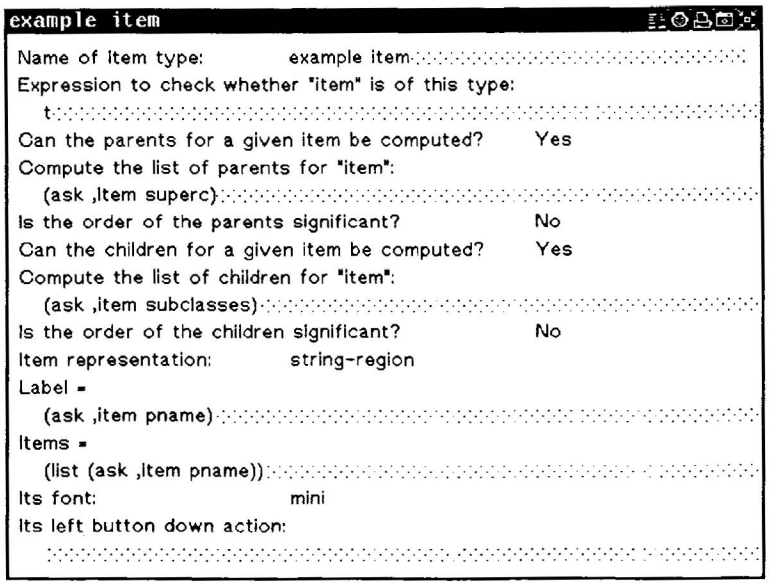


Figure 1: Initial State of the Node Form Describing Properties of Individual Nodes.

Oriented Design Environment) has been developed. It incorporates and applies knowledge about application domains. *The environment deliberately sacrifices generality in exchange for domain specific support.* It is believed that however well-designed general purpose editors for graphs might be, they inadvertently fail to bridge the gap to specific applications. *Design environments* support a designer by helping him construct a specific system out of mostly predefined, general building blocks. The communication between the designer and the design environment is as general or as specific as these building blocks. The design environment described below uses domain knowledge to allow communication in domain specific terms.

The example selected to demonstrate this new generation of design support systems is a design environment for graphical editors in the domain of *object-oriented inheritance networks.* In addition to the general knowledge about graphs, the system knows about inheritance mechanisms in object-oriented systems, it knows about the nodes being classes and the links representing the superclass relation. This knowledge is used to provide guidance, critique, and constraints. Guidance consists of suggestions on how to proceed; e.g., what are basic elements to construct a class browser. Critique reacts to choices; e.g., that a node or link icon is too small or perhaps too complex. Constraints prevent conflicting choices; e.g., a parent relation is not usually bidirectional.

The design environment is itself object-oriented. General knowledge about graphs is specialized by domain specific knowledge about browsers, classes and their properties. Constraints can be more restrictive for a given application domain, suggestions can be more helpful and critique can be more knowledgeable. The object-oriented architecture allows general knowledge to be represented at the more general levels in the inheritance hierarchy of classes, whereas domain specific knowledge is located in specialized classes. The mechanisms in NODE for guidance, critique and constraints were designed to reflect these levels of expertise.

The next section describes how the design environment presents itself to its users. Section Conceptual Background describes the conceptual background with respect to design methodologies and the role of design knowledge. Section Implementation of Knowledge Structures in NODE provides some information about the implementation aspects. The conclusion evaluates the system building effort and points out future research directions.

An Example: Creating a Browser with NODE

NODE is a design environment (implemented on the Symbolics LISP machine) for designing network-oriented application programs. It guides the designer toward the implementation of his application by providing the means to select and combine capabilities that will be used in a direct manipulation interface for network applications. The designer makes choices through a set of pop-up windows that are filled in much like forms used in various business practices. Filling in fields on these forms results in changes to an application tool that can display network-like data structures whose graphical representations are manipulated on the screen.

The overall strategy of NODE is that a designer will create the user interface by interacting with a series of forms. These forms elicit information about the designer's desired application tool, and serve as a specification of how objects in the application program are represented graphically and are operated on. During this specification process, NODE is monitoring the designer's input and utilizing its knowledge base to provide guidance about design choices. When the designer is satisfied with the state of the design, he requests that the application be instantiated. NODE then uses the selections entered in the forms to combine existing code and create the application.

An ObjTalk browser was selected as the application for showing how NODE can be used to design a network-based program. ObjTalk [12] is an object-oriented programming language. Applications in ObjTalk are made of classes that are arranged in inheritance hierarchies. Browsers can be used to inspect properties that are related to classes. Besides the usefulness of such a tool, the domain of browsers is typical

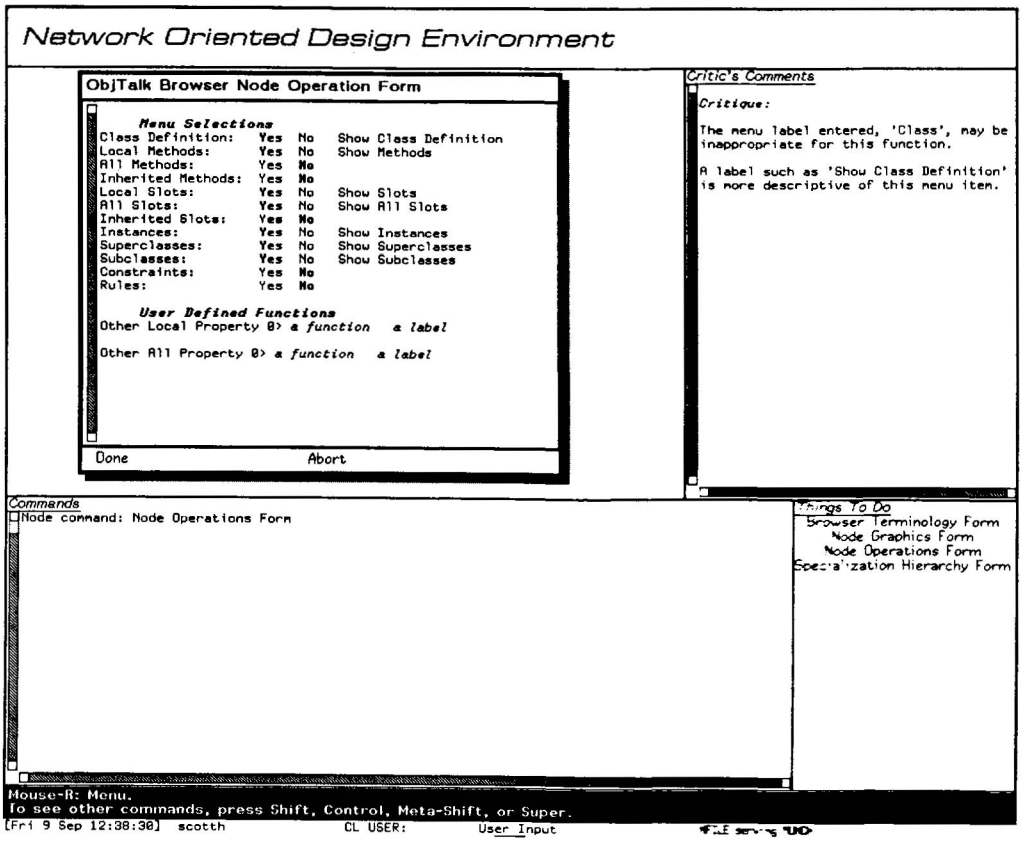


Figure 2: The NODE User Interface

The NODE user interface consists of a display pane where input forms are displayed, a listener pane which allows the designer to communicate with the operating system, a command pane through which the user chooses which form to display, and a critic pane, where constraint and critic information is displayed. Note that the user has changed the "Show Class Definition" label to something less meaningful and a criticism was triggered.

and well-suited to the types of problems NODE is concerned with. Making a browser for ObjTalk has provided valuable insight into the process involved in constructing an application with NODE, and has demonstrated the practicality of NODE.

There are two parts to NODE: the design environment and the browser generated by it. The design environment consists of a window containing several panes (Figure 2).

The display pane is used to present the forms through which the designer of the browser inputs data. Through the *node operations form*, the designer determines what parts of the network can be browsed (e.g., subclasses, superclasses, slots, methods, instances). The form displays some basic functionality that the final browser can have; the designer chooses which of these features are to be included in the final browser. The designer can also input names of his own functions to give the resulting browser special capabilities which are not part of the basic browser. Additionally, the designer uses the node operations form to input a menu label for each of these functions. These labels will eventually make up a menu through which the browser user will invoke the functions. Other forms allow the designer to set various labels used in presenting the final browser and to specify graphical actions on the nodes to be manipulated by the browser.

Upon invoking NODE, the designer proceeds to create the browser by selecting each form one by one through the command pane and filling in their fields with the appropriate information. The order in which the forms are filled in is not important - the designer inputs the data in any convenient order.

The designer may first choose to display the node operations form (shown in figure 2). As described above, this form prompts the designer for three types of information: an indication of all of the basic capabilities the final browser is to have, the names of any special functions the designer may add to customize this particular browser, and the menu labels to invoke these functions in the generated browser program. The designer indicates which of the listed functionality he wants to include in the final browser by selecting "Yes" or "No". He inputs the names of special functions in the appropriate fields at the bottom of the form.

As the designer fills out the form, his input and choices are guided by the constraints, critics, and suggestions built into NODE. For example, if the designer decides he wants the browser to show the local methods for a class, he selects "Yes" for that field in the form. Immediately upon making this decision, a label appears next to this field: "Show Methods". This is being suggested by NODE as the possible menu label

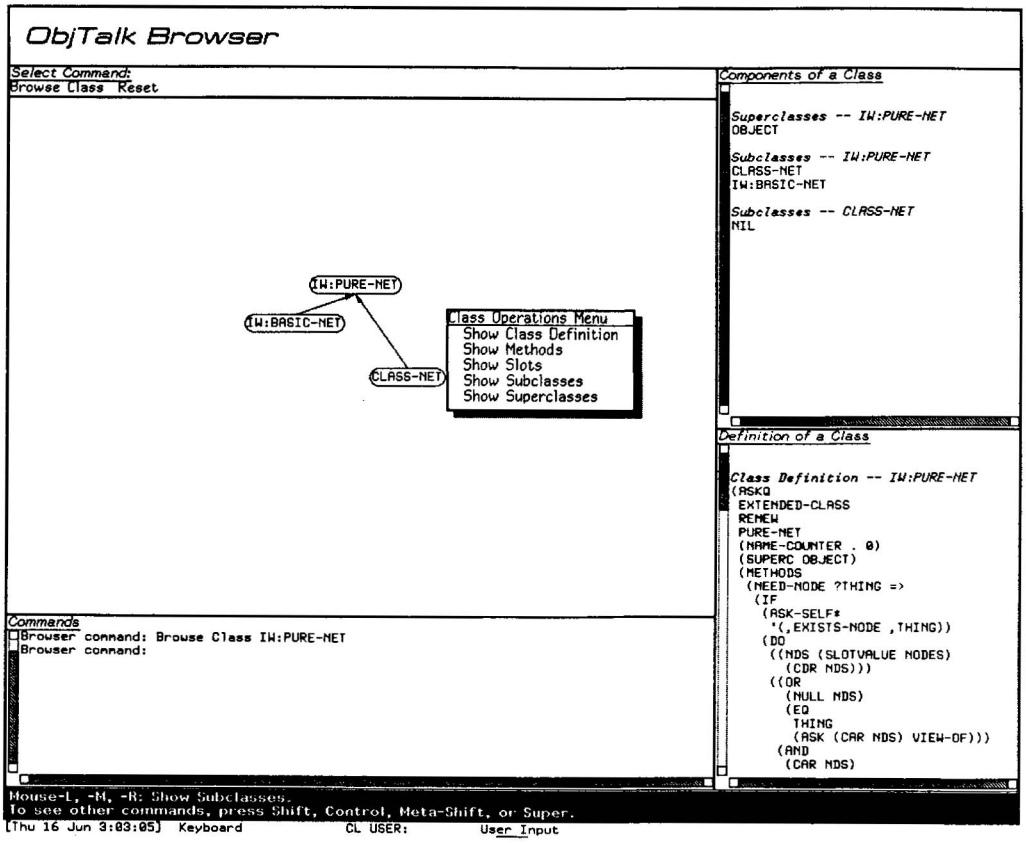


Figure 3: The NODE Browser User Interface

The NODE browser user interface consists of the following panes: a display pane on which the ObjTalk classes are displayed graphically as a network of nodes; a listener pane which allows the user to communicate with the operating system; a command pane through which the user can request the display of initial nodes (classes); a components pane, where class properties can be displayed textually; and a definition pane, where the ObjTalk definition of a class can be listed.

for this particular capability. NODE makes suggestions such as this for every function field listed on the form. The designer can either choose to use this label or change it to one of his own.

Suppose the designer chooses the "Class Definition" functionality and, as shown in Figure 2, changes the default label "Show Class Definition" to simply "Class". NODE will criticize the new label for not being very meaningful. However, NODE does not require that the designer change it - it will accept that value if that is really what the designer wants. It is just using its knowledge of what comprises a good browser to assist the designer.

The designer may wish to add a function beyond the basic browser capabilities offered in the checklist. He enters a function name in one of the "Other Property" fields at the bottom of the form. If the function is not yet defined, a constraint comment appears in the critic pane. At any point, the designer can replace the name he initially entered with that of a function which does indeed exist or he can bring up an editor window and define the function. He may then return to NODE and continue to fill in the form.

When the designer exits a form, NODE analyzes the combination of form entries as a whole. Criticisms and constraints will react to the combination of features. For example, if the designer had chosen to activate "All Slots" but not "Local Slots", the NODE critic would point out that it would be difficult for the user of this browser to determine the local slots of a class, thus suggesting the importance of the "Local Slots" capability.

Once the designer is satisfied with his input on the different forms, he may instantiate the browser he has specified through the form interaction. Combining the specifications, code supplied by the designer, and an existing code substrate, NODE instantiates a browser such as the one in Figure 3.

The created browser may now be invoked. When the browser window comes up, its title is whatever the designer defined it to be when he filled out the browser terminology form. The user then selects the "Browse Class" item in the command pane. A "Browse Class" prompt appears in the listener pane, and the user inputs the name of the first class node to be displayed, such as "iw:pure-net". An icon with the label "iw:pure-net" appears in the display area, and the user can then use the mouse to move the icon to an appropriate place in the display area. When the user selects that node with the mouse, the class operations menu appears. It contains five entries: "Show Class Definition", "Show Methods", "Show Slots, "Show Superclasses", and "Show Subclasses". These entries are those specified by the designer with the node operations form.

The browser user might select "Show Subclasses", at which time a list of all of the subclasses of the chosen class appears in the class information pane. Several such actions are illustrated in the figure.

Conceptual Background

This section will explain the conceptual background that led to the behavior of NODE as described in the previous section.

Designing Software Systems

Adapting software systems to users and tasks is becoming increasingly important. The flexibility of software *in principle* has not yet reached the level of the unsophisticated user. As a first and necessary step, users should be given the possibility to *adapt* systems to their needs in restricted application domains. In these domains the programming level oriented generation of software is replaced by design activities which are analogous to those performed by architects and other mechanical engineers.

The user of NODE is assumed to be interested in solving problems within his domain of expertise (e.g., ObjTalk programming, Unix file systems manipulations, editing rule dependencies, etc), but not within the domain of network-based graphical interfaces. He uses the system produced in cooperation with NODE as a tool for solving problems in his domain of expertise. Since the goal is to create systems that can be easily customized to the users' needs without the need for intervention by a specialist, the distinction between the designer and the end-user becomes one of usage. It is anticipated that an end-user will use NODE to customize his environment, thereby becoming the designer in the process (Figure 4).

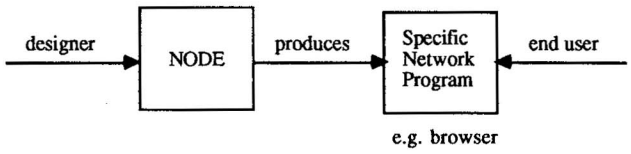


Figure 4: Roles of the Designer and End-users

Designers create systems for end-users. Also, users can adapt systems to their specific needs and adopt the role of a designer temporarily. End user and designer can be the same person.

In [4], a taxonomy for *constraint design processes* is presented. It is claimed that in many cases the full generality and power of a programming language are of little value because they cannot be exploited by the unsophisticated users. Users want to apply a tool to their specific problem, but they are not interested in going through a long learning process of how to use the tool.

Selection is one of the most basic design methods. "Design" is reduced to selecting from a fixed set of alternatives. Although this has some problems if the number of alternatives becomes big, it can be applied in many cases. In the browser specific part of NODE (Section Specialization of Design Knowledge) users choose between attributes they want to have appear in the BROWSER menus. Selection requires the enumeration of all possible alternatives, thereby restricting the design space significantly. On the other hand, selection puts the least burden on the designer/user because only recognition of alternatives is required.

From the system's point of view, selection allows all of the design knowledge to be made explicit. Only meaningful design choices need to be offered. Help facilities such as "mouse documentation lines" and textual explanations can be associated with all the alternatives. In NODE, selection often replaces more general design methods if the domain is close to NODE's expertise (see Section Specialization of Design Knowledge).

Simple combination is a design methodology that is applied if the design space can be characterized by the immediate components of the target systems. The large number of possible systems that can be constructed by combining parts prohibits a simple selection methodology. The design space is less constrained and the issue of good design vs. bad design becomes relevant. In NODE, design by simple combination is applied when users combine the various operations that can be accessed by menu selection.

Experienced network designers are able to evaluate a design based on their knowledge about legal, possible and good combinations. NODE's knowledge about good combinations includes the more general graph level as well as the more specific browser level.

Instantiation as a design methodology can be compared to parameterizing a class of systems. It opens up the design space because it is impossible to list the entire set of parameter values. The number of possible systems that can be generated by instantiating system parameters is (for all practical purposes) infinite. The issue of legal vs. illegal parameters becomes relevant. In NODE, strings like menu labels, names for operations, the size of nodes, etc. instantiate a specific graph application.

For inexperienced designers, the design methodology of instantiation can pose many problems. Designers have to provide information to the system instead of just selecting or combining what the system already offers. Intelligent design support systems make use of their knowledge about the application domain by providing defaults or by making suggestions for parameter values. In NODE's domain of expertise, all parameters have reasonable defaults and user supplied values are checked for legality and even appropriateness.

Selection, simple combination and instantiation are the design methodologies used in NODE. They do not require actual modification of the underlying system. However, the object-oriented architecture allows existing classes to be *specialized*, yielding new classes with modified properties. This activity opens the design space tremendously and is currently beyond the grasp of the NODE system.

Design Space for NODE

Graph representations for different application areas share many concepts and properties. Nodes, links, icons, arrows, curves, association of operations to nodes and links, movement, overlapping, display handling, texts, and labels are part of almost any graphical representation system. Together, these objects and their properties define the graph domain. They characterize the design space for any specific application. The design task consists of *selecting* the appropriate components, *combining* them in a meaningful way and *instantiating* them with parameters specific to the application domain.

In NODE, the components of graphs are represented as classes and objects of ObjTalk. Figure 5 shows parts of the inheritance hierarchy for the graph system being used. The available building blocks offer unlimited possibilities for selection, combination and instantiation. The object-oriented architecture provides for easy extension of predefined classes.

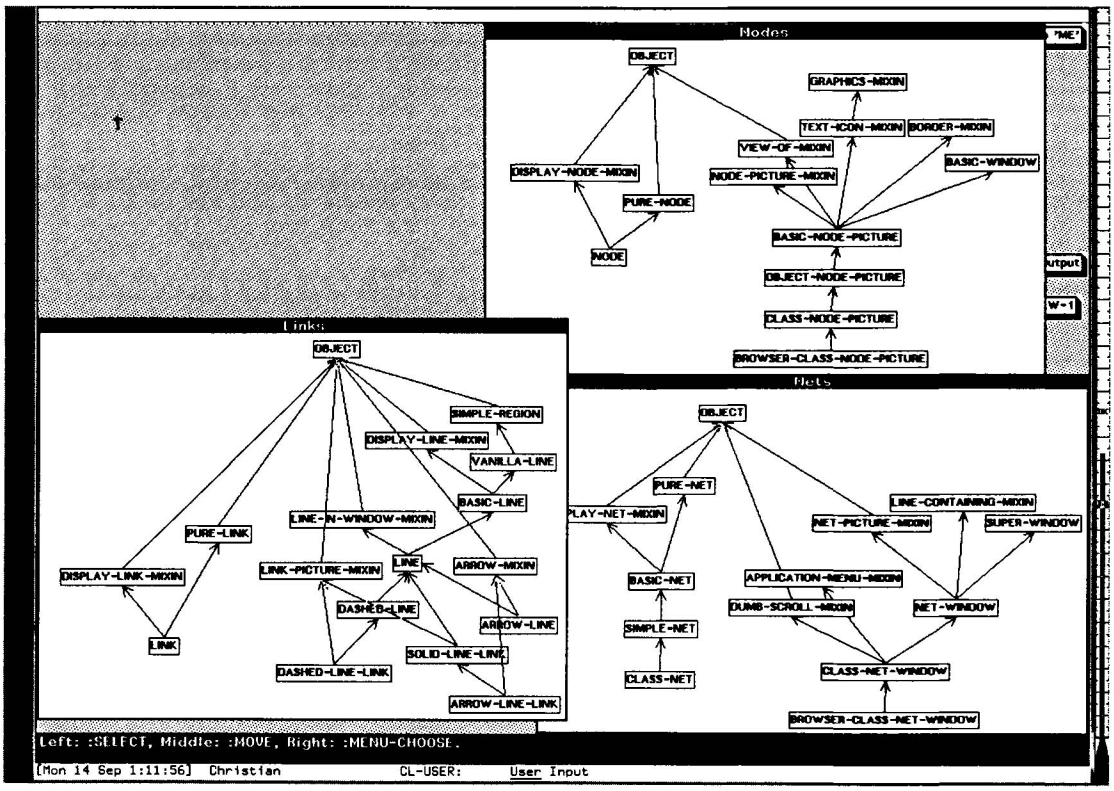


Figure 5: Inheritance Hierarchies for the Graph Classes

Thereby a large variety of graphical applications can be constructed.

The application that was concentrated on was that of a hierarchical network browser for ObjTalk-based systems. Browsers have become increasingly valuable for understanding and changing systems. They displace program listings because they present a program as a multidimensional structure being generated and filtered dynamically. In SMALLTALK [7], the browser is the main interface to the system and is used both for finding and analyzing existing pieces of software and for modifying and creating new software. It replaces the file system and the editor of conventional systems. The interaction style is one of moving and searching through an information space rather than directly accessing the space through names or descriptions.

The WLISP-BROWSER [13] is an example of a browser that is based on the graphical display of inheritance networks as directed graphs. The interconnections between classes and their components can be analyzed in more detail by selecting from menus that are associated with each of the nodes (Figure 6).

The WLISP-BROWSER had been developed previously without using a design environment. It has served as a guideline for the kind of system NODE's users should be able to build.

With the help of the WLISP-BROWSER, users can inspect and search through the inheritance hierarchy of classes. Inspection of classes is achieved through textual display of the class definition and other class properties such as methods and slots. Superclass relationships are visualized by the links of the network graph. Each node displayed in the network graph represents an ObjTalk-class. Users of NODE should be able to choose the visual effect (circle, rectangle, etc) of the representation. This representation should also provide the means to inspect the components of a class. Designers should be able to affect the set of actions that can be taken on a class (i.e., what can be inspected).

When the user clicks on the node icon, a menu of actions allowed on the node is displayed. Each selection provides the means to inspect a particular aspect of the class. The designer should be given the ability to specify which properties will be made available, and hence, what items appear on the menu. Since a specific nomenclature may be inherent to the browser's application, the designer should be able to specify the label that is displayed on the menu.

The ability to select or create the access functions that retrieve ObjTalk properties and definitions has not been a part of the NODE effort. Although such an ability is not applicable to ObjTalk browsers (there is little distinction between access methods for ObjTalk), it may be inadequate for general applications [10]. For such cases, similar methods must be used to link application code to the interface such that specific access functions can be chosen by the designer.

Specialization of Design Knowledge

Design environments around today are fixed at a particular knowledge level. Their ways of communicating are formulated in certain terms. The primitives and alternatives they present are fixed. The designer is locked into a certain level of generality. As he gains experience, he becomes better able to

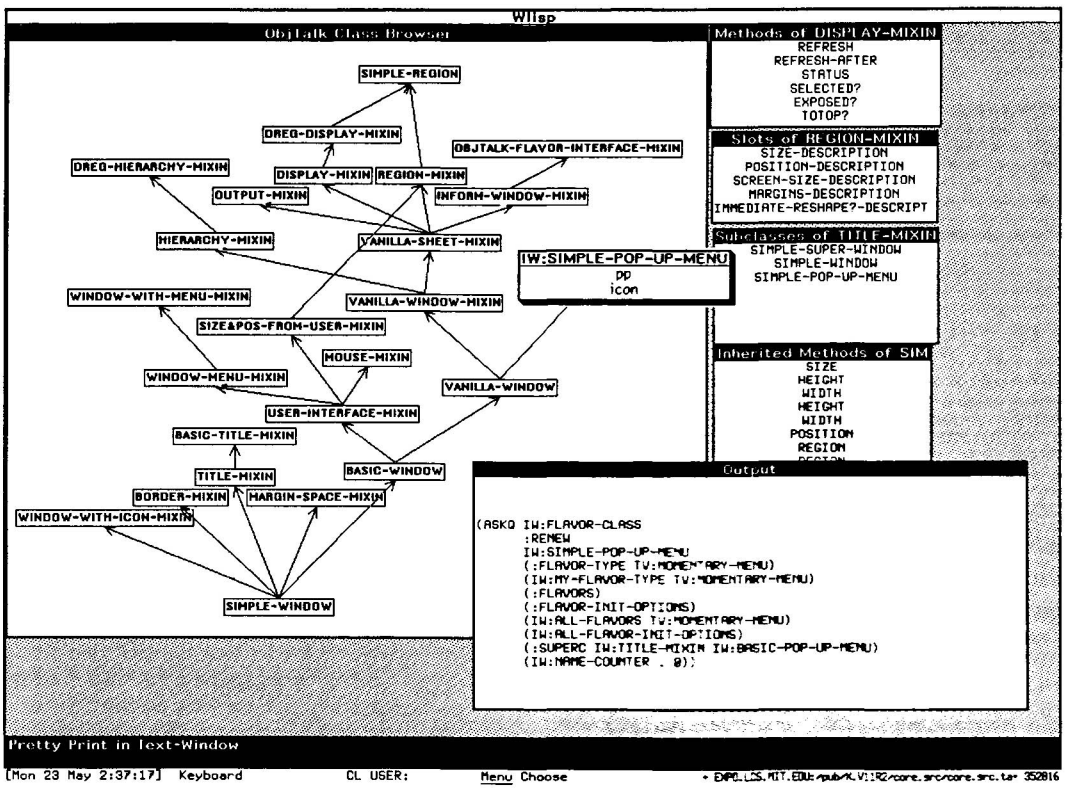


Figure 6: The WLISP-BROWSER

manipulate the design tool to create the particular program he has in mind.

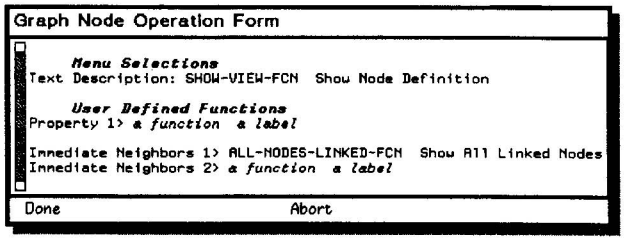
For example, users of TRIKIT (see Introduction) specify their systems by filling in forms that solicit information on the descriptive and functional aspects of the program being designed (Figure 1). For each different application, the forms have to be filled in appropriately. After repeated use, a designer becomes aware of how which fields relate to which aspects of his application. The designer would be better assisted if the design environment knew more about the program he was trying to design. This added knowledge is especially important if the designer is not greatly experienced at using the design tool nor an expert at the particular task. Specifically, specialized knowledge about the task domain allows queries and explanations to be phrased in terms of the domain. Forms can be tailored to the task domain, showing the user where to begin and suggesting what to include, thus simplifying the use of the design tool.

Consider the progression of forms in Figure 7. Any one of them could be a part of a form-based design tool like TRIKIT. All of the forms fall under the domain of graph-oriented programs; i.e., programs that will display and manipulate node-link structures. All of the forms concentrate on the node object; i.e., they query the designer for functions that will manipulate nodes in the final program. The progression represents an "instantiation" of the node form at different levels of specialization, i.e., different levels of knowledge. The levels progress from the general Graph level to the very specific ObjTalk level. A design tool at any of the levels of knowledge could be used to build a browser for class hierarchies for the ObjTalk object-oriented programming environment. However, a design tool at the ObjTalk knowledge level makes this task the simplest. In fact, the default values in the form would themselves create a valid Browser.

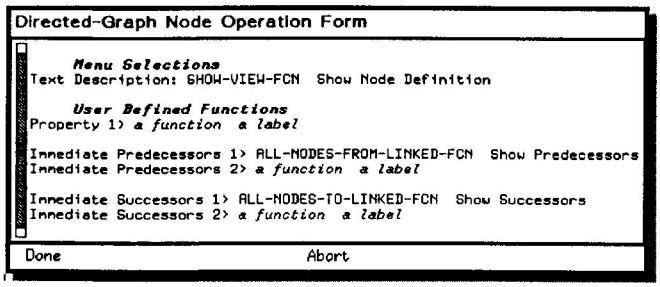
The form at the Hierarchical-Graph level corresponds roughly to the level of knowledge used by TRIKIT. At this level, the system knows it is dealing with specifications for a program that deals with hierarchical data structures, i.e., it understands inheritance. Thus it makes the distinction between a "local property", which is something searched for against only a given node, and an "all property", which is searched for against both the immediate node and nodes from which the immediate node may inherit that property. The designer is thereby reminded that in inheritance systems, such a distinction exists. He may choose to include one and not the other. The distinction was not made at the previous Directed-Graph level because, at that level, the system does not "know" about the particular relation, inheritance, involved in the graph. Also at the transition from the Directed-Graph level to the Hierarchical-Graph level, "immediate predecessors" becomes "parent" and "immediate successors", "child." Again, the basis for the change is a more specialized understanding, the concept of browser. The purpose of the change is clarity to the designer/user. This class of knowledge is defined to be the *form entry class*.

Another knowledge aspect to the forms is the kind of feedback provided to the designer. As entries are made, different degrees of reactions are possible. Three classes of knowledge are defined for the varying degrees of feedback: constraint, critic, and suggestion.

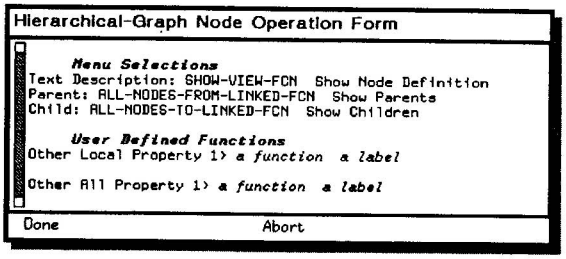
1. The *constraints class* defines restrictions on the form entries. The purpose is to prevent a design choice that will create erroneous results. For ex-



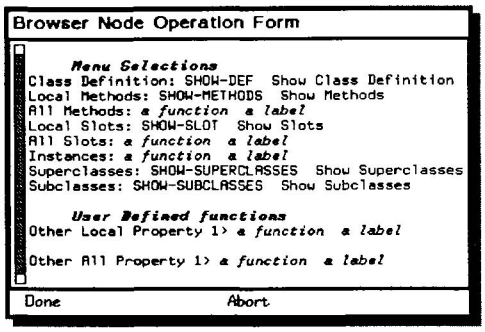
(a) Graph Level (Most General)



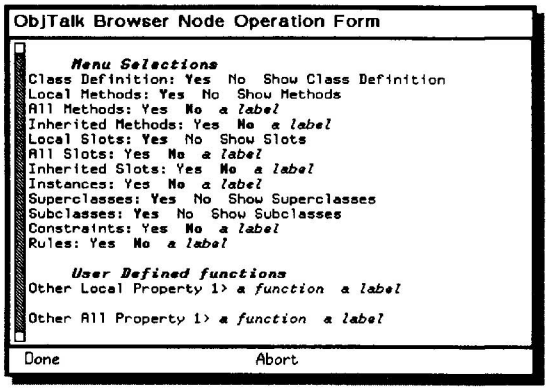
(b) Directed-Graph Level



(c) Hierarchical-Graph Level



(d) Browser Level



(e) Objtalk Level (Most Specific)

Figure 7: Progression of Forms From Least to Greatest Knowledge Level

ample, an integer may have been provided where a function was needed.

2. The *critics class* defines guidelines on form entries and their overall combination. The purpose is to guide the designer toward a quality application tool by criticizing the design choices in the context of the given domain [3; 6]. A choice, or lack thereof, may be suboptimal according to some particular style, and a modification suggested. The choice is valid in the sense that it does not violate any constraints, but another solution is believed to be better. For instance, all of the forms in Figure 7 will react to the failure of the designer to provide a function to give a description of a node. If the designer tries to exit a form without one, he will be reminded that a good program would allow its user a means of seeing a description of the node. The comment can be ignored by the designer at his discretion.

3. The *suggestion class* defines ways to provide defaults to form entries. The purpose is to expedite and simplify the use of forms by automatically filling in entries. Inputing one entry might cause the program to react by filling in another entry automatically. For example, supplying a function, may result in its corresponding name field being filled in with a "pretty" version of the

name, perhaps with the beginning of words capitalized. This name field usually contributes to a menu in the program being designed.

Just as form entries can be specialized, so can feedback. Consider the critic variety of feedback. It was noted above that every level of knowledge would react to the failure of the designer to provide a function that gives a description or definition of a node. Though the message exists at all levels of knowledge, its wording may vary. At the more general Graph, Directed-Graph, and Hierarchical-Graph levels, the message would be:

> Shouldn't you have a description function to provide a general view of the object?

At the more specific Browser and ObjTalk levels, the message would be:

> Shouldn't you have a definition function to provide a general view of the class?

Some messages make sense only at more specialized levels:

> If you have an 'all-property', you should have a corresponding 'local-property'.

This criticism assumes that inheritance of properties is possible and that showing the value of a property as determined by inheritance should be complemented by showing the value defined locally by the node itself. Use of the notion of inheritance implies that this criticism cannot start before the Hierarchical-Graph level of specialization. At the Browser and ObjTalk levels, the message would read:

> If you have an 'all-methods', you should have a corresponding 'local-methods'.

Finally, other messages are inherently very general and remain unchanged:

> Will usage be frequent enough to justify including this feature?

Thus the knowledge contained in the design environment influences both the form entries and the feedback. Form entries serve to guide the designer by reminding him of features commonly included in the type of program he is designing. Feedback reacts to how the entries are filled in. Feedback from constraint, critic, and suggestion classes, guide the designer in making correct and good choices.

Both form entry and feedback knowledge are incrementally specialized. The environment can operate at the level of knowledge desired for the design task. If the designer is creating a browser program, he would operate the environment at the Browser level. If the browser happens to be for the ObjTalk object-oriented system, the designer can operate at the ObjTalk level in which the system not only knows about the pieces of a browser program, but also how to implement many of them in the ObjTalk and LISP programming languages. The default values in the forms at this level of the design environment implement an ObjTalk browser. This degree of automation is possible only at this level because only at this level does there exist knowledge about mapping to a programming language.

It is believed that specializing the knowledge in the design environment provides better guidance and understanding to the designer. It is in the designer's interest to operate the design environment at the most specialized level known to the environment and general enough for the task at hand.

Implementation of Knowledge Structures in NODE

The NODE design environment is implemented in the ObjTalk object-oriented programming language. Knowledge is represented in class instance values and methods.

Specialization is accomplished by inheritance. Specifically, every form has a corresponding class. That class generally has five immediate superiors or superclasses as illustrated in Figure 8. Four are classes that correspond to one of the kinds of knowledge in the system; i.e., form entry, constraint, critic, and suggestion. The fifth class is common to all form classes; it supplies the code necessary to display a form and review the users entries. This display and review is based on the information in the knowledge classes.

Each knowledge class is the product of an inheritance hierarchy. For instance, Figure 8 shows the inheritance scheme for the node operations form. At the bottom of the tree is a class, ObjTalk-Node-Operations-Form. This class corresponds to the ObjTalk level form of Figure 7; it combines through inheritance all the knowledge to operate the design environment at the ObjTalk and Browser levels of specialization. Similarly, the node form could have been instantiated at a more general level.

Consider the critic branch in the Figure. A critic that applies to all the levels would be defined in the most general class, e.g. graph-critics. The process of inheritance allows this critic to apply at all levels. Inheritance also allows critics to be redefined or introduced at more specialized levels.

The classes that constitute the knowledge are defined by a program expert. No interface yet exists to the knowledge base, so the knowledge classes must now be coded by hand into ObjTalk definitions. The program expert is someone who has experience with a particular class of programs, e.g. inheritance browsers. Different domains can be entered into the hierarchy. For example, flow graphs are a specialization of directed graphs. Knowledge classes that specialize the design environment to flow graphs would branch off from the classes at the Directed-Graph level, refer again to Figure 8.

The object-oriented model leads to a convenient separation of the categories of knowledge in the design environment. It allows specialization through inheritance. It allows knowledge of new applications to be incorporated with the addition of new modules.

Conclusions and Future Directions

The goal of the research was to enhance design environments by incorporating knowledge about the application domain. The design task considered was the design of a browser for an object-oriented language. The browser was based on graphical representations of inheritance hierarchies.

The work succeeded in several ways:

1. The notion of *constrained design processes* has proven useful for describing programming activities that are closely related to application domains. Application domains such as graphical representations for inheritance hierarchies significantly restrict the design space and allow the design environment to operate on a level which can be understood and handled by its users.

2. *Suggestions, constraints* and *critics* were identified as operationalizations of knowledge bearing concepts. It has been shown how they can support the user in making design decisions.

3. It was shown how knowledge about application domains can be organized hierarchically. Mechanisms have been developed for representing this knowledge in classes of the

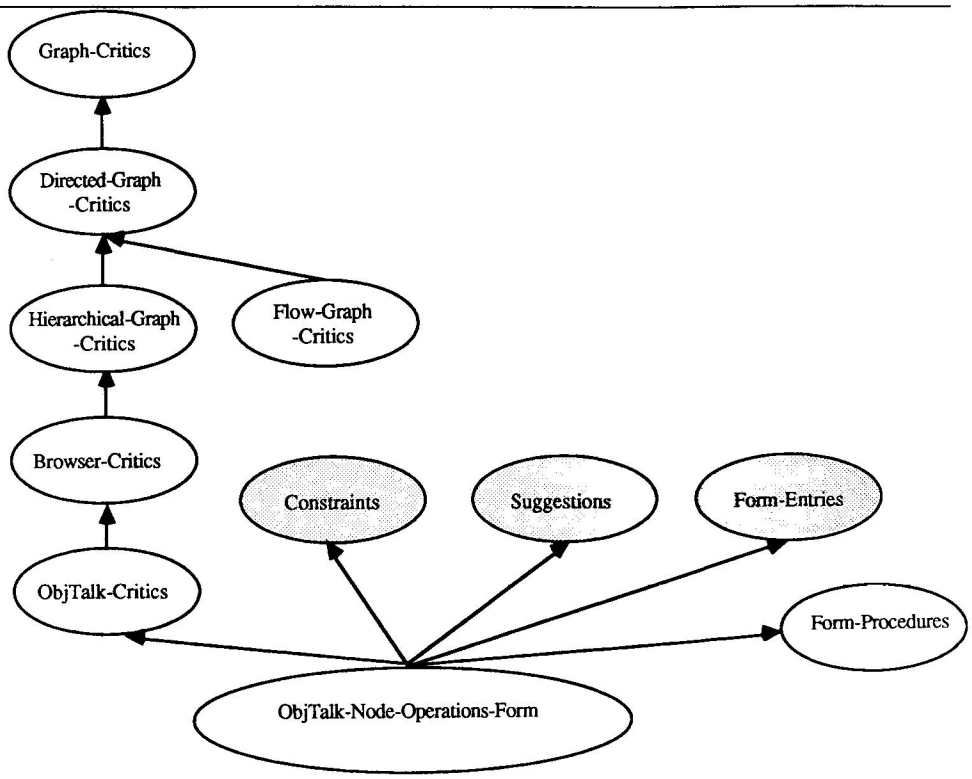


Figure 8: Forms Inherit from Knowledge Classes

Each kind of knowledge that contributes to a form is implemented as an object class and incrementally specialized along an inheritance chain. The inheritance for critic knowledge is filled in. The shaded ovals imply that constraint, suggestion, and form entry knowledge also have inheritance chains. The flow-graph critic class illustrates that domain knowledge can branch off in new directions.

object-oriented knowledge representation language, ObjTalk. They are organized for the purpose of serving as suggestions, constraints and critics. The organizational principles of the knowledge are independent of application domains.

One might feel a little awkward about the fact that a "general" programming environment for designing graphical applications was not provided. One might argue about the fact that NODE can only be used for a very restricted application domain. While this is all true, it is hard to believe that such a general tool will ever exist *without* any domain specific knowledge. What makes an application programmer a *successful* or *good* programmer? Certainly, to a large extent the knowledge about the programming tool, but also the knowledge of how the tool can be best used for representing the application specific concepts.

In the concrete example, the expertise of the *browser designer* has been very important. Some of us, who had not designed a browser before, were unable to formulate the knowledge needed to contribute to NODE's knowledge base. Of course, they were also unable to come up with a good browser design in the first place. This is a further indication of the importance of domain-related knowledge.

With restricting the design space in order to support the design process, questions of opinion and style play an increasingly important role. In its current form NODE is biased towards its notion of good or bad browsers, and other experts might disagree. Rather than being disturbed by this fact, one should see it as a further indication of the knowledge-based strategy used in NODE. If everybody's opinion about browsers were to be incorporated, the net result would be simply the initial state of affairs -- a general tool -- too general to be used for anything.

In the current stage, NODE only implements the knowledge and functionality necessary to support the thesis. It needs to be enhanced in many ways. Currently, it does not make extensive use of direct manipulation techniques. In many cases, the form oriented approach should be replaced by pointing actions on iconic representations of objects and commands.

Furthermore, the graphical representation of the nodes and links in the final browser are pre-determined by NODE; the designer has no control over it. Future versions of NODE will offer separate node and link forms, through which the designer may select appropriate representations from a variety of choices.

Also planned is the representation of knowledge about the domain of task dependencies. Users will be able to specify task dependency networks and associate functionality to nodes and links much like they do now in the browser domain. The object-oriented architecture of the knowledge-bearing entities allows specialization of the knowledge classes that are common to both applications. It is believed that only those aspects that are specific to the new application domain need to be defined.

Acknowledgment
The authors would like to thank Gerhard Fischer and Andreas Lemke for criticizing earlier drafts of this paper. The research was supported by a grant from US-West Advanced Technologies.

References

1. B.Bell. *The BOOGIE Manual, An Object Oriented Graphical Network Editor.* University of Colorado, 1987.

2. . Computer. Special Issue on Visual Programming, Vol. 18, No. 8, IEEE Computer Society.

3. G. Fischer. A Critic for LISP. Proceedings of the 10th International Joint Conference on Artificial Intelligence (Milan, Italy), Los Altos, CA, August, 1987, pp. 177-184.

4. G. Fischer, A.C. Lemke. Constrained Design Processes: Steps Towards Convivial Computing. In R. Guindon, Ed., *Cognitive Science and its Application for Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1988, Chap. 1, pp. 1-58.

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

6. G. Fischer, A. Morch. CRACK: A Critiquing Approach to Cooperative Kitchen Design. Proceedings of the International Conference on Intelligent Tutoring Systems (Montreal, Canada), June, 1988, pp. 176-185.

7. A. Goldberg. *Smalltalk-80, The Interactive Programming Environment.* Addison-Wesley Publishing Company, Reading, MA, 1984.

8. E.L. Hutchins, J.D. Hollan, D.A. Norman. Direct Manipulation Interfaces. In *User Centered System Design, New Perspectives on Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1986, Chap. 5, pp. 87-124.

9. P. Kahlhoefer, T. Manz, J. Seitz, H.-D. Stenger. EDGRAPH. Ein objektorientierter Graphikeditor. Institut fuer Informatik, Universitaet Stuttgart, 1984.

10. W. Mark. Knowledge-Based Interface Design. In *User Centered System Design, New Perspectives on Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1986, Chap. 11.

11. H. Nieper. TRISTAN: A Generic Display and Editing System for Hierarchical Structures. Department of Computer Science, University of Colorado, Boulder, CO, 1985.

12. C. Rathke. *ObjTalk: Repraesentation von Wissen in einer objektorientierten Sprache.* Ph.D. Th., Universitaet Stuttgart, Fakultaet fuer Mathematik und Informatik, 1986.

13. C. Rathke. The Browser: An Exploration Tool for ObjTalk Inheritance Structures. CU-CS-331-86, Department of Computer Science, University of Colorado, Boulder, CO, May, 1986.

14. G. Robins. *The ISI Grapher.* Information Sciences Institute, Marina del Rey, CA, 1987.