Multiple Perspectives in Interactive System Design

Jeffrey D. McWhirter*, David F. Redmiles† and Markus Stolze*

*Department of Computer Science
University of Colorado
Boulder, CO, 80309
{jeffm,stolze}@cs.colorado.edu

†Information and Computer Science
University of California, Irvine
Irvine, CA 92717
redmiles@ics.uci.edu

KEYWORDS Multiple perspective interfaces, user interface design.

ABSTRACT

The presentation of information in interactive systems plays a crucial role in the users’ ability to accomplish tasks. Many tasks, in fact, require multiple perspectives of the application data. However, designing and constructing systems that provide adequate support for multiple perspectives is challenging. This paper presents a characterization of the role of multiple perspectives in interactive systems, explores some of the many design issues involved and describes the authors’ experiences in this area.

INTRODUCTION

Direct manipulation interfaces support users’ interacting with application data through control components such as buttons and menus as well as through the graphical representation of the application data itself. The manner in which this data is presented strongly influences the end users’ ability to comprehend it and manipulate it in a specific task situation [19, 20, 35]. A single representation does not necessarily support all possible tasks (or all possible users). In this paper we will call systems that support more than one representation or view of the application data “multiple perspective systems” (see also [30] and [22] for alternative terminology). A few brief examples of systems and research in interactive systems motivate an examination of multiple perspectives in design.

First, an application may be used to accomplish a variety of tasks, each requiring some particular view of the application data. Norman [25] describes how flight schedules ideally would be represented differently depending on whether the user wants to compare flight duration or arrival times. In another example Norman shows that certain representations of the tic-tac-toe game make it rather hard to play, whereas the “normal” grid-representation makes it a straightforward game that does not require conscious effort for planning the next move.

Second, in their research in graphical systems, Wehrend and Lewis [37] describe a problem-oriented classification of representations. They identify nine distinct actions (or tasks) in which users may engage when trying to understand a body of information. These actions include identification, location, categorization, ranking, etc. These user tasks act on the same body of information. As a consequence, an interactive system that supports these tasks must provide multiple perspectives of the information. Similar dependencies between task and representation have been noted by other cognitive scientists, graphics designers, and computer scientists (e.g., [19], [34], [24]).

Third, an example of a commercial application that uses multiple perspectives is Microsoft Word. It enables users to work on their document in three different “Views”. Each of these views is customized for a certain class of tasks and represents the user’s document differently. The “Page Layout” view, for example, shows the document in a WYSIWYG mode, which is ideal for developing the final document layout. In contrast, the “Outline View” shows the hierarchical structure of the document which is more appropriate for the early writing phases.

Designing and implementing interfaces that support multiple perspectives can be a daunting task. The developer must arrive at a set of adequate representations, coordinate those representations, and possibly provide support for the simultaneous manipulation of the application data through the set of representations (i.e., multiple view editing). There are no simple solutions. However, we hope that by exploring the applicability of multiple perspectives and illustrating some of the implementation issues, we will help designers identify the kind and degree of support they need to provide for multiple perspectives in their applications.

MULTIPLE PERSPECTIVE SYSTEMS

One approach to supporting multiple tasks is to provide a single perspective of the application data which is a compromise between the different (and potentially conflicting) representational requirements of the different tasks [25]. However, the drawback of this approach is that usually only the most frequent tasks are well supported while many other tasks are much more difficult to perform because the application data representation does not fit these task. Thus, users which try to
perform many “atypical” tasks will find themselves hindered, rather than, supported by such a system.

To further complicate the issue, one single representation or perspective may not be adequate for a given task. It may be the case that the type of task or application domain requires the application data to be simultaneously displayed in a variety of ways. For example, one of the common tasks that requires multiple simultaneous perspectives is browsing through an information space. In this case, there may be both a presentation of an abstract view (such as a tree of hypertext nodes) and a detailed view of the information (such as text) [18].

The term “multiple perspective” takes on a variety of meanings depending on the context of its use. In general, a large number of diverse systems make use of multiple perspectives. These perspectives may range from multiple windows showing different parts of the same representation to fully independent and editable representations of the same underlying application data. Multiple perspectives appear in domains ranging from video games to programming environments to exploratory data analysis. For example, Defender, a video game popular in the 1980’s, featured simultaneous perspectives of the game state. One was a detailed view of the current playing area and the other provided an abstract view of the entire game “world.” In programming environments one often encounters both textual and graphical perspectives (e.g., code and class hierarchy views). In this section we characterize implementations of multiple perspectives across three dimensions: presentation, manipulation, and domain.

Syntactic and Semantic Presentation

One can classify multiple perspectives as differing along syntactic and semantic lines. In [32] Kimura discusses syntactic and semantic zooming in graphical diagramming environments. Syntactic zooming refers to changing the level of detail presented within a perspective based on scaling and panning. Semantic zooming refers to changing the actual graphical presentation of an object irrespective of its scale or translation. We extend this terminology to differences in perspectives.

Syntactic perspective differences refer to differences based on scaling, translation and elision. Here, the basic presentation of an application object is the same abstract perspective. However, that object may be displayed at a higher or lower level of detail or may not be shown at all. For example, in Microsoft Word you can modify the “Normal” view to show (or not show) paragraph marks, and you can show or hide the ribbon and the ruler. Graphical environments often provide multiple windows in which the application data is displayed. Each window may maintain its own scaling and translation factor. Systems can also provide perspective differences over time where certain objects are hidden or accentuated by the user as their tasks and needs change.

Semantic perspective differences refer to differences in how the basic application objects are represented. For example, a graph may be displayed in both a “circles and arrows” form as well as an incidence matrix based tabular form. Another example of multiple perspectives that exhibit semantic differences is Word-count in Microsoft Word and Print-preview in Microsoft Excel. In these examples the representation is not mapped directly from the underlying application data. Rather, some complex transformation is applied to the application data to achieve the representation.

Support for Manipulation

One can also categorize multiple perspective systems based on their modes of use. For example, some systems solely support browsing and navigating where the underlying application data is fixed. On the other end of the spectrum are systems that support simultaneous editing and manipulation of the application data through each perspective.

Browsing and navigation

Many uses of interactive systems can be classified as browsing or navigating through some body of information through its graphical representation. Here the information is read only, the system need only support the presentation and navigation - not the creation and manipulation of the information. The prevalence of multiple perspectives in browsing tools is pointed out in a recent taxonomy of image browsers[27]. In this taxonomy the authors describe the role of global and detailed views (or perspectives) and the coordination between views of differing detail.

A fish-eye view (e.g., [14, 31]) allows the user to interactively highlight important components of the information being presented and to de-emphasize less important components. It provides a way in which a detailed view and an abstract global view are integrated and simultaneously shown to the user. Typically the user defines some point of interest. The perspective is changed based on the “nearness” of an object to this point of interest. This nearness measure may be, among other measures, a measure of actual geographic distance or distance in a graph. Here, the perspective is rapidly changed based on user input. The fish-eye view is an example of a multiple perspective mechanism in that it provides to the user the ability to change, over time, what is being shown and how it is being shown.

Multiple editable perspectives

Multiple editable perspectives allow users to manipulate the underlying application data through the different perspectives of the data. Relatively few systems support multiple editable perspectives in comparison to the number that provide multiple perspectives for browsing and navigation. As will be discussed in the following section the design and implementation of such systems can be a very difficult task.

Domain of Representation

The domain of representation refers to the idea that the data or objects of an application may be described using different terms depending upon context. Specifically, Bobrow and Winograd assert that “a description must be able to represent partial knowledge about an entity and accommodate multiple descriptors which can describe the associated entity from different viewpoints [7].” They use the example of a person being described in terms of a traveler (with attributes such as age and preferred airport) as well as in terms of a customer (with attributes such as billing address and credit rating).
In applications that support software engineering, namely, software development environments, the concept of one software design reflecting several domains is common. In an obvious sense, a design frequently integrates domains of communications, data storage, and data sensing or input [13]. However, in a more subtle sense, a design may be described in terms of a variety of abstraction ranging from a problem domain (such as furnace control) to a system’s implementation (such as in C, FORTRAN, or LISP). Software development environments seeking to provide support for users must support representations in these different domain perspectives [29, 15].

This notion of domain of representation is different from the semantic zooming as used in the literature and discussed above. The latter refers to a transformation of the same data for presentation whereas domain perspectives may require entirely separate sets of descriptors.

DESIGN AND IMPLEMENTATION ISSUES

There are a variety of issues pertaining to the design and implementation of multiple perspective systems. In this section we discuss two major issues - what are the “right” perspectives to provide to the user and how to support the user interactions with those perspectives.

Presentation: Finding the Right Set of Perspectives

Finding the right set of presentation perspectives is one of the central tasks in designing multiple perspective systems. On the one hand, designing individual perspectives to fit specific user tasks will usually be easier than designing a single perspective that compromises between the requirements of different tasks. However, there is a limit to the number of perspectives that users can manage. No experimental results are known to us, but a possible heuristic is to limit the number of semantically different editable perspectives in a system and if necessary limit the semantically different browsing perspectives. The reasoning behind this heuristic is that users have to learn to edit and to interpret each of these perspectives and must understand how operations in one perspective affect the other perspectives. This learning may place an undue burden on the user and negate any possible benefits of the multiple perspectives.

One way to reduce the number of semantically different perspectives is to supply different ways in which perspectives can be “enriched.” These enriched perspectives show additional properties of the application data which are crucial for specific tasks without gross changes to the underlying perspective. This is an example of a syntactic perspective, where extra elements may be shown or hidden. As mentioned before, Microsoft Word annotates the “Normal” view with visible and manipulatable markers of paragraphs, page-breaks and tables.

Perspective “enriching” is one way to derive new perspectives from already existing ones. But how do we determine good base perspectives? In our opinion, already existing methods for interface design can be used here. User task analysis (e.g. [8]) and an analysis of tools, materials, diagrams and representations already used for performing the tasks the system should be supporting can help to determine initial candidates for base representations. These candidates can then be tested and refined using such methods as the Cognitive Walkthrough[28] and paper-based mockups (e.g., [23]). Often, however, it will also be necessary to develop working prototypes to enable users to experience the interactive behavior of single or multiple perspectives. The behavior of paper mockups is often not representative of the highly interactive system behavior typically shown by direct-manipulation perspectives.

Manipulation: Editable Versus Browsable Perspectives

Another decision in designing multiple perspective systems is determining which perspectives will editable and which will be only browsable. From a user’s viewpoint it might seem that all perspectives should be editable. If users see an element of the application data in a perspective they may feel that they should be able to manipulate that element in situ without having to go to an editable perspective.

However, the decision is not straightforward. First, as mentioned above, a system should only have very few editable base perspectives. Thus making all system perspectives editable might add more confusion than benefits. Another important reason for keeping the number of editable perspectives low is that their implementation can be very hard and might also involve high runtime costs for mapping the changes between perspectives. Finally, some perspectives might show the application data at a very high level of abstraction. Thus when the user edits these data abstractions directly it is far from clear how these operations on the data-abstraction should be mapped to the original application data.

While choosing a method for keeping views consistent is mainly an implementation decision it also can have considerable effects on the response time the system and thus becomes an issue that has to be considered already during system design. Different approaches have been proposed to reduce the complexity involved in mapping between different perspectives and keeping them consistent. The easiest but computationally most expensive approach is to completely regenerate each perspective when a change in the underlying application data has occurred. A more efficient approach is to limit the changes in a perspective to only those objects that need updating. Event propagation mechanisms are used in such systems as the Model-View-Controller(MVC) framework[1] and the Zeus algorithm animation system[3] to manage the change notifications.

Different proposals have been made to improve system response times while keeping the propagation mechanisms. The traditional approach is to propagate updates immediately after each edit operation. This approach is satisfactory if the related perspectives can be updated rapidly. However, if too many perspectives need to be updated or if the updates are computationally expensive system performance will degrade to intolerable levels. An alternative is update perspectives only when no current user input is pending. Thus if users perform operations very rapidly, only the perspective the user is currently working in will be updated immediately. The update of other perspectives will be delayed until no further
user input needs to be processed. A third alternative is the "update on demand" approach where inconsistent perspectives are marked and are only updated after an explicit update request - for example by clicking on the perspective.

An interesting design issue is whether perspectives which show application data at a high level of abstraction should be made editable or not. A tension exists between providing potentially useful facilities to the user and implementing those facilities in the system. The implementation of a mapping between the operations in the abstract perspective and the application data may be difficult. However, he benefit for the user might also be considerable. As users only specify in the abstract what operation they want to perform on the application data some additional information needs to be provided to achieve a complete mapping to the application data. In principle, heuristic approaches could be used to fill in the missing information. However, usually the system leaves the task of completing the information to the user and only generates frames which need to be specified further. This approach is used, for example, by the MViews system [17], for mapping between an abstract graphical view of an object oriented program and a more detailed textual perspective of the same program. Making abstract perspectives "semi-editable" is an alternative approach. Here users are allowed to change elements in the abstract perspectives, but these changes are not propagated directly to the application data. Instead the changes are interpreted as requests for help on what to do to make this change a reality. Thus, for example, instead of changing the underlying application data the system might come up with a list of five things the user should modify in the more concrete perspectives to achieve the intended change in the abstract perspective. "Cross perspective selection" is another popular way to support users in determining which elements in the editable perspectives need to be changed to achieve a desired change in a non-editable perspective. A perspective that supports cross-perspective selection highlights corresponding elements in other perspectives when an element gets selected. This guides users in determining which elements in the editable views should be changed to achieve a change in the non-editable perspective.

There is no one solution to the above mentioned design issues. Rather, these issues open up design space that has to be explored individually for each new application domain. The dynamic nature of many of the perspectives will make it a goal to achieve a desired change in a non-editable perspective. A perspective that supports cross-perspective selection highlights corresponding elements in other perspectives when an element gets selected. This guides users in determining which elements in the editable views should be changed to achieve a change in the non-editable perspective.

There is no one solution to the above mentioned design issues. Rather, these issues open up a design space that has to be explored individually for each new application domain. The dynamic nature of many of the perspectives will make it inevitable to develop rather elaborate prototypes to investigate these issues. Current prototyping tools are often not well suited for developing editable perspectives of application data. Better tools are needed to support the investigation of these design issues in individual cases.

**Domain: Bridging the Gap between Problem and Solution**

Little has been said so far about content of the application data that is to be presented and potentially edited. While the content will vary from system to system, there are some general guidelines.

The discussion in this paper assumes a general class of applications for supporting problem solving or design. Within this scope, researchers in cognitive science have studied the role of mental representations or mental models. For example, Kintsch and Greeno studied the role of mental models in children solving word algebra problems and in particular the sets of strategies to transform problem statements to problem solutions [36]. Pennington adapted this model for explaining programmers’ behavior in understanding example programs [26]. In research in design environments Fischer noted that the biggest obstacle to design was bridging the conceptual gap between the situation model a person develops about a design problem and the system model that supports its implementation, such as in a programming language [11].

These observations indicate the need for varying levels of abstraction in representing data. However, these different levels extend beyond the syntactic or semantic presentation of data discussed above. They are not simply increasing degrees of refinement. Instead they often require different terms for description. For instance, in the domain of software design, a problem domain might be to implement the control mechanism for a commercial furnace. The system model might depend on communications and sensing domains. Representations of all these domains are essential to understand or, conversely, to design a problem solution.

As cited earlier, Bobrow and Winograd identified one mechanism for representing alternative domain perspectives of an object and others have made related suggestions [5]. What remains problematic is the method of arriving at different sets of descriptive terms. One general guideline is that the different domain perspectives must serve design. This goal is adapted from work in design rationale systems [12]. Together with the observations on system and situation models, it suggests a goal-subgoal decomposition; varying domains are related by their fulfillment of a need in the problem solution (e.g., communications substrate). A second set of guidelines come from the area of requirements engineering and software design. Namely, there are a variety of problem decomposition methods to identify the domain objects as nouns and domain actions as verbs used to describe problem settings (e.g., [2]). Finally, a third guideline is to allow descriptions to evolve out of a variety of information sources. To support this approach, heuristics for making suggestions about descriptors might be implemented in a design tool [10].

**MULTIPLE PERSPECTIVES IN EXPLAINER**

The Explainer tool (see Figure 1) was developed to help programmers solve tasks in computer graphics by allowing them to explore previously worked-out examples. By our characterization, Explainer implements multiple perspectives for presentation as well as for differing domains. Specifically, Explainer presents four views of a program example: *code, execution, text, and diagram*. Within the text and diagram views, information may be described from any of four domains: *problem, graphic features, programming features, and programming language*.

The Explainer interface is similar to a hypermedia tool. This implementation allows minimal information about the example to be initially presented. Programmers can then decide which specific features of the example they want to explore, presumably choosing those most relevant to their
current task. Information is accessed and expanded through command menus (see Figure 3). Almost all items presented on the screen are mouse sensitive and may be expanded. In both the diagram and text presentations, “more” and “less” cues suggest to users where the detail of information may be increased or decreased. These cues encourage the use of syntactic perspectives defined earlier. Having the four different presentations available by default enable the user to benefit from semantic perspectives. The perspectives menu (Figure 3b) enables expanded information to be presented according to different domain perspectives.

Thus, the Explainer interface allows programmers to study a programming example from several different perspectives and to study the interrelationship of those perspectives. Figure 1 is the actual state of the Explainer interface as left by a test subject. We refer to the example illustrated in this screen as the Modulo Addition Example. The example program generates a visualization of addition modulo 100, illustrated the modulo operator by the circle and the numbers 1 - 99 by a partial labeling around the perimeter of the circle. The domain perspectives include Modulo-Addition, Cyclic-Operations, Plot-Features, Program-Features, and LISP. The test subject was exploring this program as an example related to a new task of drawing a clock face with its numerals. The screen shows that the subject had

- expanded the diagram (upper right pane) to explore the components of labels in the Plot-Features perspective (by clicking on “more” cues);
- redrawn the initial diagram from its Plot-Features perspective to a LISP perspective (only a portion is visible in the middle of the upper right pane – menu action “Diagram”);
- retrieved a text description (lower right) about the concept of labels in the Program-Features perspectives (menu action “Text Story”);
- generated a text description (lower right) of how labels are implemented in the LISP perspective (menu action “How”); and
- highlighted the concepts common across several different perspectives which pertain to labels (menu action “Highlight”).

This specific information enabled the test programmer to identify the LISP function called to draw the labels, the assignment function that calculated the positions, and what variables the position calculation depended on. The programmer could then apply the same problem decomposition, program structure, and functions in the solution of the clock task, or in this case, simply modify a copy of the example to place the labels inside the perimeter of the circle.
Explainer’s knowledge is encoded in a semantic network (see Figure 2). The nodes of the semantic network represent concepts in an example program code. Concept are instantiated with one specific perspective. Instances of nodes are connected by three kinds of links: components, roles and perspectives. The component links connect one instantiation to zero or more instantiations which comprise it in the same perspective. In the Modulo Addition Example, a plot consists of a coordinate-space, a circle, and labels. The roles link serves as an inverse to components. Finally, through the perspectives link, concepts can have analogous counterparts in other perspectives. For example the operator and elements components of the Cyclic-Operations perspective are equivalent to the circle and number-labels components of the Plot-Features perspective. In sum, concept instances in different perspectives are organized into networks along the components/roles relation. The networks in different perspectives are interrelated through the perspectives links of individual nodes. Thus, as a whole, an example program is a single, structured network that may be interpreted according to various perspectives, syntactic, semantic, and representation domain.

The number and selection of perspectives for representing an example or any object is then clearly an issue. As with any knowledge-based application, the choices are dependent on their intended use. Explainer was postulated as an aid to programmers accessing library functions, in this case, graphic functions. In this sense, it serves as a documentation substitute as opposed to a general-purpose reuse library or as an intelligent tutoring system. This choice clarifies the need for LISP and plot-features perspectives and the mapping between the two. Commonality between problem domains would be serendipitous. In contrast, for Explainer to serve as a general-purpose reuse library, scalability would be a major concern. To serve as an intelligent tutoring system, examples would not be chosen for coverage of software library features, but rather to show incrementally complex concepts in a domain. Which perspectives are needed depend on the nature of the domain and the range of context expected in tasks and subtasks within that domain.

At a conceptual level, Explainer’s use of perspectives (both for varying presentation and domain) corresponds to research in learning, understanding, and designing software [26, 6, 33]. This research focuses on people’s development of mental
models when studying existing program codes. In programming, mental models are referred to as programming plans. In this research, breakdowns in comprehension and subsequent problem solving are attributed to the inability of a person to develop a “correct” mental model. In Explainer, this research is interpreted as a motivation to make more apparent to programmers, the various aspects of a programming plan underlying an example. It is through the perspectives mechanism that this approach succeeds. In particular, perspectives capture the plan for translating elements of a task into elements of a solution. Explanation consists of various techniques to visualize the different perspectives and interrelationships. In a sense, the representation of the example tries to capture the many perspectives of the mental model the author of the example had when originally working out the example task.

MULTIPLE PERSPECTIVES IN ESCALANTE

Escalante [21] is a development environment for dynamic interactive graphical systems. Escalante allows for the construction of applications that support both syntactic as well as semantic multiple perspectives. Escalante provides a variety of mechanisms to support multiple perspective applications, ranging from simple multiple windows with zooming to fully editable multiple representations of the underlying application data. The underlying application data model of Escalante is the graph. However, the graphical representation one can derive of a graph is not restricted to “circles and arrows”. In general, one can define a large number of representations for an underlying graph structure.

Figure 4 shows the runtime architecture of applications constructed using Escalante. Applications are centered around an underlying non-visual graph and any number of visual graphs. The non-visual graph serves as the underlying canonical form. Each of the visual graphs provides a semantic perspective of the underlying non-visual graph. Each of the graphs consists of a set of elements. A one-to-n relationship exists between the elements of the non-visual and visual graphs. This structure provides a simple and regular way in which multiple representation applications are built, allowing for a wide variety of approaches in its use. For example, a system could initially be designed around a non-visual graph, thus capturing the underlying semantics of the application. Any number of different perspectives, in the form of visual graphs, could then be defined. The design process could also begin by focusing on a particular perspective to first explore representational issues. No one approach is sufficient for all problem
domains and tasks. Rather, Escalante provides a flexible environment that supports a variety of design approaches.

An attribute propagation mechanism is used to bidirectionally map attribute values between the underlying non-visual graph and its set of perspectives. This provides the ability to not only have different representations of the overall graph structure but also to have different representations of the internal state of the elements. Editing tasks in an application take the form of operations on a graph or changes to the internal attribute state of the graph elements. These changes can occur to either a visual graph or a non-visual graph.

Operations applied to one visual graph are also applied to the underlying non-visual graph. This change is then propagated to the set of other visual graphs. At the user’s discretion, certain tasks (e.g., movement, scaling, elision) can be applied to either a single visual graph or to all related visual graphs. This allows for both simple graph editing as well as showing the related elements in each perspective (i.e., the user can move an element and view the movement of the related elements). The attribute propagation mechanism can be used to provide cross-perspective selection as discussed in Section and more generally to provide shared information across perspectives, enabling the user to understand the relationships that exist. For example, in Figure 4, the textual label of an element may be directly changed through any perspective. This change is propagated to the related elements in the other perspectives, thus maintaining consistency across perspectives. The user can understand the relationships among the perspectives by seeing the similar labels (and the changes of state).

Each visual graph is displayed in any number or views or windows which provide the interface between the user and the application. Escalante provides to the user of an application the ability to configure a wide range of syntactic perspectives. For example, the user can bring up any number of windows to view a particular visual graph. Each of these windows maintains its own scaling and translation factors. The user can also interactively elide from view different components of a graph based on type or attribute state. An ObserverView is provided that allows the user to arbitrarily link the display of a source view with the display of another view. The displayed area of the source view is shown as a rectangle in the ObserverView. The scaling of the ObserverView is fixed as some percentage of the scaling of the source view. As the user
zooms and pans in the source view the ObserverView updates its rectangle and zoom level. This facility allows one to maintain an overview of an entire graph in the ObserverView while focusing in on particular areas in the source view. The rectangular display in the ObserverView provides a reference as to what area of the graph is being displayed in the source view. Furthermore, an ObserverView can itself be the source view of another ObserverView, allowing for an even more abstract overview of the graph.

Verdi application

Figure 5 is taken from a prototype of Verdi[16], a visual language and environment for the design of distributed systems. Figure 5(a) shows the Verdi visual language as described in [16]. The basic Verdi constructs include Team elements that spatially contain a set of Role elements. Role elements contain a set of elements that serve to define the control flow and synchronization of a computation. Among these elements include Box elements, sets of which make up an N-Way Interaction.

Other perspectives have also been investigated for the Verdi language including a tree and a tabular based view of the Team and Role elements and a view that focuses on the Box and Interaction elements. Figure 5(d) is a tabular view of the Team and Role elements; each row of the table represents a Team and each column represents a Role. A relation between a Team and a Role is represented as a bar entry in the table. Figure 5(b) shows a view that focuses on the N-way interaction elements and their related Box elements.

Each perspective provides to the user a context in which they can focus their attention. For example, the tree and tabular views focus on the interactions between Team and Role elements. The perspective shown in Figure 5(b) focuses on the N-way interaction elements, constructs that do not explicitly appear in the other views. The system architecture provided by Escalante supports the ready addition of new perspectives as an application evolves and the developer gains a better understanding of the requirements placed on the system by the users. This facilitates the iterative design and development of an application. For example, our original Verdi prototype consisted of the perspective shown in Figure 5(a). From this original design the need to focus on certain constructs in certain ways became apparent and the other three perspectives were developed and integrated into the application.

In the prototype, all four views in Figure 5 are different perspectives of the same underlying application data. One can manipulate the underlying data through each of these four views. For example, one can add a relation between a Team and Role element in the view of Figure 5(c) and appropriate relations are automatically added between the corresponding Team and Role elements in the other views. Likewise one can delete some element through one view and the corresponding elements in the other views are also deleted. Changes to the labels are also propagated among the different views using the attribute propagation mechanism.

Multiple Perspective Mobile Application

Sharing attribute values among the perspectives can aid the user in understanding and manipulating the information that is being presented. For example, Figure 6 shows Mobile, a prototype environment for the modeling and simulation of workflow systems. Mobile was built to explore extensions to the Information Control Net(ICN) model [9] in the context of FlowPATH, a commercial workflow system [4]. This application consists of two perspectives of an ICN. The perspective shown on the left contains all of the constructs of the ICN. The perspective on the right shows the subset of the ICN that deals with control flow. The components of the ICN have a token attribute, the image of which differs in the two perspectives. On the left, tokens are shown as a sequence of dots. On the right they are shown textually. On the right, edge tokens are shown as line width. The user can directly change the token count through a menu command as well as directly through the textual representation. Changes in one perspective (e.g., token increment, graph editing) are propagated to
the underlying non-visual graph which then propagates the change to the elements in the other perspective. Also note, in the figure not all elements are shown in the left perspective. Rather, the user has collapsed a Role node, to elide from view some of the detail. This elision does not affect the elements in the right perspective.

CONCLUSION
Modern computer workstations have the capacity to support users interacting with application models and data at fine degrees of granularity and in various presentation formats. As a result, a variety of systems have emerged under a general class of interfaces referred to as multiple perspective interfaces. We have sought to disambiguate the various meanings of this term by a characterization along three dimensions: syntactic and semantic presentation, support for manipulation, and domain of representation. In so doing, we have explored the technical challenges and design implications for building such interfaces, including detailed views of systems by the authors.

Corresponding to each of the three characterizations is a major open research issue. With respect to presentation, more work is required to guide the selection of and assess the consequences of multiple presentations. While cognitive models and experience highlight the limits of human attention, assessment of specific combinations of techniques still relies heavily upon evaluating prototypes. Support for manipulation would benefit from knowledge representations that provide more automated support for creating and, most importantly, maintaining multiple descriptions. “Canonical” forms are still application dependent. With respect to the domain of representation, some insight is provided by design methodologies especially in software development. However, traditionally, these approaches have been limited in their scope of application.

Multiple perspective systems have met with initial success and popularity. Their future promise encourages further exploration of the challenges set forth above.

REFERENCES


