

# Interactive Specification of Context-Sensitive Displays in Humanoid

*Pedro Szekely*  
*USC/Information Sciences Institute*  
*4676 Admiralty Way*  
*Marina del Rey, CA 90292*  
*Phone (310) 822-1511, FAX (310) 823-6714*

## INTRODUCTION

User interface design can be viewed as a search in a space of design alternatives. During the early conceptualization stages of application development designers are concerned with identifying the functionality that an application should deliver, the overall style of interactions provided by the interface, and global policies for generating displays. For example, designers might decide what objects and what attributes of those objects ought to be presented to the user, but not how. Later on, designers refine the design to identify the windows that the interface provides, and which objects are displayed in which windows. Finally, designers decide exactly how objects will be presented, how commands will be invoked, and how the various windows are sequenced. Typically, the design process does not proceed in a strict top-down fashion. Designers might work on the details of a presentation before finalizing the conceptual design of the application. The process is iterative [Buxton80], with redesigns based on feedback from representative end-users interacting with early prototypes.

Our goal in the Humanoid interface development environment is to provide tools that help designers search the design space quickly and effectively, and to generate high quality executable interfaces. Since iterative design plays such an important role in interface design, Humanoid provides tools that let designers work incrementally on any aspect of the design. In addition, Humanoid allows designs to be prototyped before they are complete in order to allow designers to collect feedback on the current state of the design.

This paper describes Humanoid's approach for assisting designers in the construction of the presentation component of a user interface (discussions of Humanoid's support for other aspects of interface design appear in [Szekely92]). The approach represents a compromise between two main approaches used in other systems: *power tools* for designers, and *automated designers*.

The *power tools* approach puts designers in control of the design, and gives them tools to easily and rapidly specify designs and investigate design alternatives. The most successful examples of the power tools approach are interface builders such as NextStep [NeXT90], OpenInterface [Neuron91], and many others [Myers92b].

The *automated designers* approach involves software that automates the design of presentations. Automated designers take as inputs information to be presented (e.g. tuples from a relational database) together with additional information characterizing the features of the information that should be emphasized (e.g. relationship between price and reliability), and automatically produce a display. The most successful examples of automated tools are APT [Mackinlay86] and SAGE [Roth90].

Humanoid bridges the power tool and automated designer approaches by taking a model-

based approach. The key element of the model-based approach is a model, i.e., a declarative description, of the characteristics of the displays of a user interface. The model contains all the knowledge needed to generate the displays at run-time given information to be presented.

Since the model is declarative, it enables the construction of a variety of power tools, i.e., tools that reason about the model to provide several levels of assistance to designers (e.g., editing tools to refine models [Szekely92], design critiquers [Braudes91], design guidance [Luo92]). The model also provides a mechanism for integrating automated designers with power tools. In the model-based approach, automated designers would work by automatically constructing model fragments that designers can refine using the power tools.

The rest of this paper is organized as follows. First comes a detailed analysis of the advantages and disadvantages of interface builders (the most widely used tools for presentation design) and automated designers. From this analysis we draw a set of goals for the design of the model-based approach. The next section gives a short overview of the complete Humanoid system. Then comes a description of the Humanoid model of presentations, followed by examples of how Humanoid can be used to specify presentations. We close with conclusions, a review of related work, and directions for future work.

## **INTERFACE BUILDERS AND AUTOMATED DESIGNERS**

Interface builders are the most successful representatives of the power tools approach. There are more than 20 commercially available interface builders, and surveys of user interface developers show that they are widely used [Myers92a].

### **Interface Builders**

Interface builders allow designers to draw the screens of an interface. They provide a palette of building blocks containing menus, buttons, labels and other building blocks commonly used in graphical interfaces. Designers can drag building blocks from the palette with the mouse, place them on a work area, and edit their parameters using a dialogue box. Once designers have drawn a screen they can go into test mode, and operate the various components as end-users would in the finished interface. Some interface builders [NeXT90] also provide facilities to connect the building blocks with application procedures that trigger application-specific computations.

The main advantages of interface builders are:

- *Ease of use.* The drawing paradigm of interface builders is intuitive to designers. There are no abstractions or programming language that designers need to learn.
- *Extensive control over details of the design.* Interface builders provide designers with extensive control over the visual appearance of the interface. Designers control layout by dragging the building blocks, and control fonts and other attributes of the building blocks via menus and dialogue boxes.
- *Immediate feedback to design changes.* Interface builders provide a What You See Is What You Get (WYSIWYG) interface. There is no costly compilation step to see the effects of design changes.

- *Efficient implementations*. Interface builders generate efficient code that implements the displays that designers draw.

Unfortunately, interface builders have serious shortcomings that result from the simple model of interface building they embody:

- *No support for dynamic displays*. Interface builders only support the construction of displays whose elements can be determined at design time. Interface builders do not provide any notion of conditionality or iteration to define presentation methods that can display the application data structures that change at run-time. This shortcoming renders interface builders useless to construct the main display of applications that shows the data structures that end-users construct and manipulate at run-time (e.g., the notes in a music editor).
- *No support for redesign at a policy level*. If designers want to make a change to the interface that affects many displays, designers must edit all the affected displays individually, which is time-consuming and error prone (e.g., changing displays so that command buttons are displayed at the bottom rather than on the right).

The challenge for more sophisticated power tools is to endow them with more sophisticated models (e.g., models that support conditionality and iteration), while retaining their ease of use. As will be seen in later sections, Humanoid represents a step in this direction. Humanoid's model of presentations is simple, and Humanoid provides a set of model visualization tools to make it easy to build the models.

But, before going into more detail on Humanoid, let us first finish describing the concerns we wish it to address. While interface builders have certain shortcomings, we next show that something other than automated designers is needed.

## Automated Designers

Automatic interface generation systems generate user interfaces based on a description of application functionality. These systems fall into two categories. The general purpose ones [Beshers89, Hayes85, Olsen86 and Singh89] attempt to generate all aspects of the interface for arbitrary application programs. The domain-specific ones [Feiner85, Feiner90, Kim90, Mackinlay86, and Roth90] can generate presentations for very restricted domains such as business presentation graphics (e.g., pie-charts and bar charts). The main shortcomings of the general purpose systems is that the interfaces they generate are of very poor quality, and often they can only generate the more static portions of the interface. The domain-specific systems generate very good quality presentations within their limited domain, but the domains are often too restrictive. The main advantages of the automatic generation systems are:

- *Ease of use*. Designers need only specify the information to be presented, together with some indication about the features of the information that should be emphasized, and the system generates the presentations automatically.
- *Support for dynamic displays*. Since presentations are generated from the data to be shown, they support the generation of displays of data generated at run-time by applications.
- *Support for redesign*. Automatic generation systems support redesign in principle, but not in practice. Since the system generates presentations automatically from the data,

changing the parameters or algorithms that the system uses results in re-designs of all application displays. However, this feature is often hard to use because it is indirect: designers must phrase their design goals in terms of features of the data (e.g. specify relationships between the data that the presentation should emphasize) rather than in terms of features of the display (e.g. use bar charts).

The main disadvantage of automatic generation systems is the trade-off between quality of the generated presentations and generality of the domain of applicability. This trade-off seems inevitable. Generating good interface designs automatically is intrinsically difficult because the design space is large and contains many bad, as well as many good designs, and current technology is too primitive to allow the construction of evaluation functions to rank designs effectively. The following are other disadvantages of automatic generation systems.

- *Lack of designer control.* Automatic generation systems hinder designer control in two ways. First, the philosophy of these systems is to do the designer's job automatically, so their architecture provides limited options for designers to control the design. Second, to the extent that designs can be controlled, it is done by indirect methods, as discussed above. Often, designers know the kind of output they want (e.g. bar charts), but might not know what information to provide to "trick" the system into generating what they want.
- *Poor efficiency.* Most automatic generation systems are very slow, making them inadequate for the construction of direct-manipulation interfaces, which must provide instantaneous feedback to all user actions (e.g., on every mouse movement).

## **The Humanoid Approach**

The Humanoid approach strives to provide a balance between the interface builder and the automatic generation approaches, in order to have the strengths of both and the weaknesses of neither. The design goals of Humanoid are:

- *Support dynamic displays.* The application displays that show information that changes at run-time are the hardest and most time-consuming to construct with today's tools. The support for dynamic displays should include support for automatically updating the displays when the underlying data structures change.
- *Provide extensive designer control.* Interface design is an iterative refinement process during which designers modify and refine designs based on feedback from previous iterations. In many cases designers develop an understanding of what a program should do while constructing the program and studying users interacting with it. Designers must be able to control all aspects of the design in order to achieve the desired goals.
- *Allow designers to delegate design decisions to the system, while retaining control should they decide to override the system.* The goal is to let designers make the difficult design decisions, and let the system take care of the details.

The assumption in Humanoid is that difficult design decisions are best done by designers because they require extensive knowledge about the application domain, the end-users, the workstation hardware characteristics, and the environment in which the application will run. Humans are able to acquire such information quite effectively, and it seems difficult and time-consuming to model the same information in the

computer so that a system can make the appropriate decisions. In addition, if the design fails with the users, designers can often see, or be told, why and how, and can change the design appropriately. Modeling the information gathered during user feedback so that a system can change its decisions accordingly seems impractical as well.

- *Support design exploration starting with the early conceptualization phases of application development.* The goal in Humanoid is to let designers work on-line starting with the initial conceptualization stages where the general features of an application are defined, and throughout the refinement stages where all details of the interface are concretized. Designers should be able to execute the applications, even before the design is complete, in order to get a better feel for the interface, and in order to let users try out the application and provide early feedback.

Humanoid provides a framework for overcoming the disadvantages of automated designers. Automated designers should be built so that their output is not only a display of some information, but also a model of the presentation. Then, designers can inspect, understand, and modify the resulting model, thus overcoming the first shortcoming of automated tools. This approach also overcomes the efficiency problem because large portions of the design process would be done at design time, and not at run time.

## **OVERVIEW OF HUMANOID**

Humanoid belongs to the family of interface design tools and UIMSs that generate the user interface from a high-level specification of the application semantics. The application semantics are usually specified as a set of object types and procedure headers. The interface is specified by elaborating the semantic description by annotating it in a format that is understood by a run-time system component. The interface specification is used by the run-time system to produce the presentations, behaviors, and sequencing for interface objects and application data structures.

### **The Interface Model**

Humanoid's design model captures information about an application's functionality (objects and operations), and information about all features of the interface. Humanoid factors the model of an application and its interface into five semi-independent dimensions.

*Application semantics design.* The application semantics model represents the operations and objects that an application program provides.

*Presentation.* The presentation defines the visual appearance of the interface: major parts of a display, data to be displayed in each part, presentation methods for each part (recursive step), layout of the parts, and conditions for including parts in displays.

*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 application and the interface.

*Sequencing.* The sequencing defines the order in which manipulations are enabled, or equivalently, the set of manipulations enabled at any given moment. Most sequencing constraints follow from the data flow constraints specified in the application description (e.g., a command cannot be invoked unless all its inputs are correct). Additional

constraints can be imposed during dialogue design.

*Action side effects.* Action side-effects refer to actions that an interface performs automatically as side effects of the action of a manipulation. For example, a newly created object can become automatically selected, or closing a dialogue box can reset all its options to their default values.

This paper only describes the model of presentations, and its use in the model-based approach to interface construction. Details about the other aspects of the model appear in [Szekely92].

## **The Run-Time System**

The run-time system is a component of every application program that uses the model described above to execute the application interface. Application developers do not need to write code to execute the models, they only need to model the desired behavior.

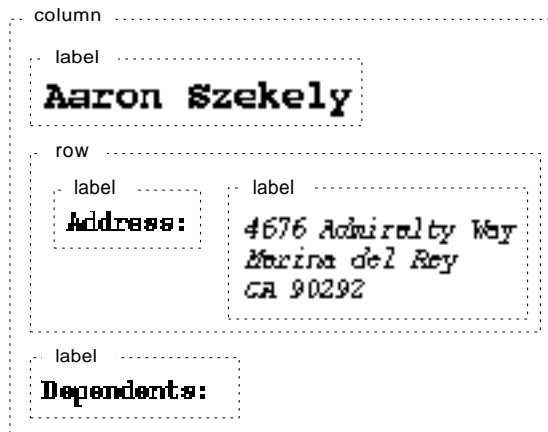
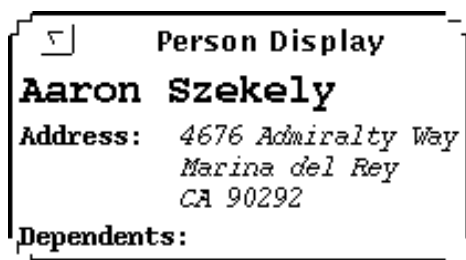
To interpret inputs, the run-time system uses the presentation model to map input events into application data references, and trigger the appropriate commands according to the application's behavior and sequencing model.

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. The presentation component obtained from the model might either be a default inherited from the generic part of the model, or a more specific presentation component specified by the interface designer.

## **THE MODEL OF PRESENTATIONS**

Humanoid models displays by recursively decomposing complex displays into simpler ones. This recursive decomposition leads to modeling a complex display as a tree. The leaves of the tree consist of graphical primitives, such as lines, arcs and text, together with primitive building blocks of the underlying interface toolkit, such as menus and buttons. The internal nodes are grouping objects, which define the layout and sizes of their children. Each node of the tree is connected to application data-structures that contain the information to be displayed by the node.

The following figure shows the hierarchical decomposition of a display that was generated by Humanoid. The left hand side of the figure shows a snapshot of a window showing information about a person. The right hand side of the figure shows a hierarchical decomposition of the contents of the window, which consist of a column containing a label, a row of labels, and another label.



Humanoid's abstraction for modeling displays is called a *presentation template* [Szekely90]. A presentation template is an object that captures all the knowledge to create the display trees that represent the structure of a presentation. Presentation templates are represented as objects in a knowledge representation language that supports inheritance [Giuse90]. The different pieces of knowledge that define a presentation template are represented as slots of the presentation template object. New presentation templates can be defined in terms of existing ones by adding and modifying the slot values of existing presentation templates.

Humanoid's modeling language also supports the notion of one-way constraint formulas. A Humanoid *formula* is like a formula in a spreadsheet, which performs a computation based on the values of other cells in the spreadsheet. When the value of any of those cells changes, the formula is automatically recomputed. Humanoid formulas allow designers to specify computations that depend on the slot values of application data structures. When those values change, Humanoid automatically recomputes the value of the formula. The formula mechanism is used to tie presentations to application data structures so that when the data structures change, Humanoid is automatically notified of the change, and can update the display appropriately. Currently, the formula mechanism is integrated with KR [Giuse90] (the representation language used to model designs), Loom (a knowledge representation language used in artificial intelligence applications) and CLOS (the Common Lisp Object System). Humanoid provides a protocol for integrating other kinds of data structures into the formula mechanism.

Humanoid provides a rich library of presentation templates. The library contains templates to present primitive objects such as strings and numbers in various ways (e.g. multi-font labels, sliders and gauges); templates to present menus, buttons and other common interaction techniques for the inputs of the commands of an application; templates that automatically create dialogue boxes for commands; templates to present full application windows, including menu bars and other such controls, and templates for geometry management such as columns, rows, graphs and tables. Designers typically define their presentations based on a template from the library and add the knowledge needed to create a presentation for particular type of information.

The knowledge captured in presentation templates is described in detail below. The essence of templates is to capture the hierarchical decomposition of a display into parts, together with knowledge needed to select among several templates the most appropriate one to display some information, and to control the attributes of the presentation such as font, size, color, etc.

## Presentation Templates

The following is the knowledge represented in a presentation template:

*Input data.* The input data models the type of information that can be displayed using a presentation template. This information is used by Humanoid at run-time to match application data structures against templates in order to select the templates capable of presenting given application data structures. The algorithm for selecting among possible candidates is explained below.

*Applicability condition.* The applicability condition is a formula to test whether a template is applicable or not for presenting application data structures that satisfy the input data restrictions. Applicability conditions are used when designers need to encode complex conditions that cannot be captured in terms of the type of the input data.

*Widget.* The widget specifies the graphical object produced by a presentation template. It can be a primitive graphical object, such as a line or an arc, a building block from a widget library, such as a menu or a button, or a layout management widget, such as column or row layout managers.

*Widget parameters.* The widget parameters specify the parameters of the widget of the presentation template. For example, if the widget is a line, the parameters include line-style, color, position, etc. The values of the parameters can be constants, or can be formulas that compute the value based on the application data structures supplied as input data for the presentation template. For example, the color parameter of a widget of a presentation template to display the status of a valve in a mechanical system could be specified by a formula that returns “green” if the valve is open, and “red” if the valve is closed.

*Parts.* The parts of a template specify the decomposition of a complex display into simpler displays. Parts are modeled in terms of the following information:

*Inclusion condition.* The inclusion condition is a formula to determine if a part should be included in the display. For example, a presentation template for objects of type Person could have a part to display the name of the spouse. The inclusion condition would specify that the part should be included only if the person is married.

*Input data.* The input data of a part specifies the application data structures to be displayed in the part. The part input data is specified by formulas that compute their values based on the input data of parent nodes of the display tree. The part input data thereby provides a mechanism to break-down complex application data structures associated with parent nodes of the display tree into smaller chunks that the child nodes can display. For example, the dependents part of a person display would compute the list of person objects it needs to display by accessing the “dependents” slot from the person object displayed in its parent node.

*Presentation template.* The presentation template specifies the starting point for the search for the “best” presentation template for displaying the part. The search algorithm is explained below under the heading *Substitutions*.

The presentation template of a part can be a single template or a list of condition/template pairs. When presenting a part, Humanoid will evaluate the conditions in sequence until one returns true, and use its associated presentation template.

*Replication data.* The replication data is an input data slot that is used as the iteration index in Humanoid's iteration facility. When a part of a template has a replication data it means that the part will be instantiated multiple times. The instantiation process works as follows. Suppose that Humanoid is asked to display a `Person` object using a template called `Person-Template`. The `Person-Template` has multiple parts to display the various slots of a `Person`, such as the `name`, `address` and `dependents`. Since a person can have multiple dependents, the `dependents` part must be replicated, once for each dependent. This means that the generated display tree will have a node to display the name, a node to display the address, and multiple nodes to display each of the dependents.

When a part has a replication data, Humanoid expects that its value at run-time will be a list. Humanoid automatically makes an instance of the part for each member of the list. Each element of the list will be displayed using the presentation template associated with the part. Even though each replication is displayed using the same presentation template, the template search mechanism is applied separately to each replication. So the generated display might be different for each replication. For example, dependents of age less than two can be displayed differently from other dependents.

*Substitutions.* The substitutions of a presentation template are essentially a list of condition/presentation template pairs. The condition of each pair is a formula that characterizes the contexts in which it is appropriate to substitute a different presentation template. When displaying a part, Humanoid tests the conditions of substitutions of the part's presentation template until one returns true. The presentation template associated with the successful conditions is considered a better template to display the part. Then Humanoid will test the substitutions of the substitutions, and so on, until the selected template has no substitutions. Since all presentation templates can have substitutions, the substitutions form a hierarchy, called the *substitution hierarchy*.

Substitution hierarchies provide a powerful and modular mechanism to specify context sensitive presentations. When specifying the presentation template to display a part, it is not necessary to worry the different displays that might be needed to display the part depending on the characteristics of the data to be presented. The different displays are specified as substitutions of the part's presentation template.

The substitution mechanism is modular because designers can enrich a substitution hierarchy, and the effects will be visible to all templates that use the root of the substitution hierarchy. The substitution mechanism also gives designers a range of control over the characteristics of a display. Designers can delegate decisions to Humanoid by using templates close to the root of substitution hierarchies, and they can exercise more control by selecting templates close to the bottom of substitution hierarchies.

Each substitution of a presentation template is specified in terms of the following information:

*Condition.* The condition is a formula that specifies the conditions when it is appropriate to substitute one template for another.

*Presentation template.* This is the template to substitute when the condition is met.

*Input data.* The input data of a substitution has the same meaning as the input data of

a part definition. It specifies additional data to be passed to the substitute template.

*Widget parameters.* The widget parameters of a substitution specify values for the widget parameters of the substitute.

*Policy.* The substitutions of a presentation template can also be tagged with a policy marker. Designers can define any policy markers they wish, and then use them to control the search for presentation templates. Humanoid maintains a stack of current policies, and when searching the substitution hierarchy below a presentation template, Humanoid ignores all templates whose policy does not correspond to a policy in the stack. The policies provide a convenient mechanism to enable and disable large portions of substitution hierarchies in a simple way. For example, if the “novice-user” policy is pushed onto the stack, Humanoid will consider in its search for presentation templates those tagged with the “novice user” policy. This will result in a display that takes into account the presentation features that a designer has defined for novice users.

In addition, templates also allow the specification of policies to be pushed onto the policy stack when a template is used for creating a portion of the presentation. The result is that the presentation templates for the parts will take into consideration the new policy. For example, the template to display `Person` objects can push the “summary view” policy, so that the dependents of a person are displayed in a summarized way (e.g., only showing their names, rather than recursively using the `Person` presentation template). So, the `Person-Template` changes the context to encourage Humanoid to present the dependents differently. Without the context change the dependents would be displayed as any other person, because dependents are persons. Of course, the dependents will be displayed differently only if there is a substitution for `Person-Template` for the given policy.

## Benefits of Presentation Templates

Humanoid’s model of presentations uses a small set of primitives with simple structure, yet powerful enough to generate a large variety of context-sensitive presentations. The simplicity of the model enables the construction of interactive tools to help designers construct models of presentations and explore design alternatives (see section on Humanoid Environment). The interactive environment provides many of the features of interface builders, making Humanoid accessible to designers with no programming experience.

The Humanoid modeling language has several features to support the specification of very complex context-sensitive displays. These features allow Humanoid to support important capabilities missing from interface builders.

- The replication-data facility provides an iteration capability to compactly specify displays of data structures that contain varying numbers of values.
- The inclusion condition of parts, template substitutions, and template applicability conditions provide a direct way to specify displays whose contents depend on the data to be displayed.
- The policy mechanism provides a direct way to make global design changes that affect many displays.

- The input data, widget parameters, inclusion and applicability conditions, and substitution conditions are defined via formulas. When the information on which a formula depends changes, Humanoid recreates the display tree as necessary. Humanoid does all the book-keeping so that only the portions of the display that are different are updated. This leads to greater efficiency, and smoother display transitions for the end user.

Humanoid provides a generic model of presentations which designers can specialize for their particular applications. The generic model provides both simple building blocks of the kind that all user interface toolkits provide (e.g. menus and buttons), and much larger components such as tables and graphs. The generic components are extensible in that designers can change any part of the model that they wish (e.g. add parts, change inclusion and applicability conditions, etc.).

In addition, some of the generic building blocks come with substitution hierarchies. For example, the template used to display interaction techniques to acquire inputs from users has a sophisticated substitution hierarchy that automatically selects between radio-buttons, menus and a variety of other techniques based on the characteristics of the input to be acquired (e.g. simple vs. multiple value, number of alternatives from where values must be chosen, policy for allocating screen space). When designers are constructing a display that acquires inputs from the user they need only model the characteristics of the values to be acquired. Humanoid automatically selects the technique for prompting for the value. If designers do not like the choice, they can refine the design in two ways:

- If the case at hand is unique, they can directly specify the presentation template to be used.
- If the case at hand is an example of many similar cases in the given application, but one that was not envisioned in Humanoid's generic model, they can specify a new substitution of `Input-Template`, modeling the conditions that characterize the cases when the different choice should be made.

Humanoid provides facilities to let designers easily make an arbitrary decision (like they would with an interface builder), and also to enhance the Humanoid model to cover a new class of cases (like they would when adding knowledge to a automatic generation system). So, the Humanoid modeling language supports many of the capabilities of automatic presentation generation systems, while allowing designers to retain complete control over the design.

#### **EXAMPLE OF A PRESENTATION MODEL**

This section illustrates how the Humanoid presentation modeling language can be used to specify the presentations for a simple application. The example illustrates the knowledge modeled in a presentation template, and the ease with which can be refined.

The example is based on a database of information about people. Suppose the database contains the following kind of information:

```
(Person Pedro (name "Pedro Szekely")
              (address "4676 Admiralty Way
                       Marina del Rey
                       CA 90292")
              (spouse Claudia))
```

```

(dependents Aaron Ana))

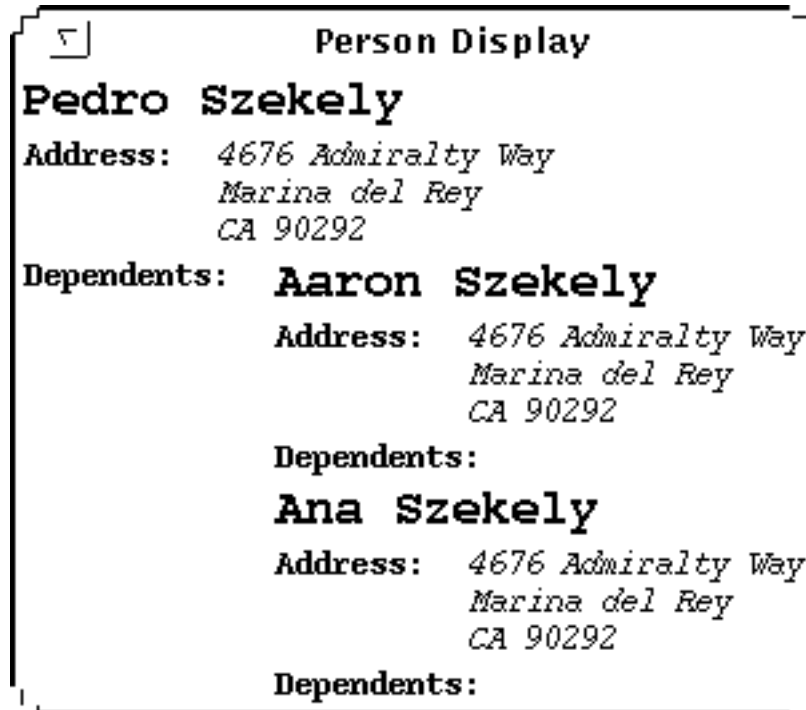
(Person Claudia (name "Claudia Garcia"))

(Person Aaron (name "Aaron Szekely"))

(Person Ana (name "Ana Szekely"))

```

Suppose the task is to present information about a person in a display such as the one shown below. The display consists of a column of the name, the address and the dependents of a person. Each dependent is recursively displayed in the same way.



The template is described here in pseudo-formal syntax. The lines are numbered for easy reference in the commentary that follows. In Humanoid, templates are defined interactively using the Humanoid template editor, and the designer would not have to be concerned with the syntax. The section on the Humanoid interactive design environment contains a description of the interactive template editors. It takes only a few minutes to model this presentation template using the Humanoid environment.

The presentation template that constructs this display is called `Person-Template`:

```

1 Presentation-TemplatePerson-Template
2 input-data: a-person, type = Person
3 is-a: Column-Template
4 parts: name-part
5           . presentation-template= Label-Template
6           . widget-parameters label = get-value (a-person,
name)
7           .                               font = Large-Bold-Font
8           address-part

```

```

9      . presentation-template = Labeled-Object-Template
10     . parts: object-part
11     . .      presentation-template = Label-Template
12     . .      widget-parameters label = get-value (a-
person,
13     . .
address)
14     .      font = Italic-Font
15     . widget-parameters label = "Address: "
16     depends-part
17     presentation-template = Labeled-Object-Template
18     parts: object-part
19     .      presentation-template = Column-Template
20     .      parts: entry
21     .      input-data:
22     .      a-person = get-value (a-
person,
23     .
dependents)
24     .      replication-data a-person
25     .      presentation-template = Person-
Template
26     widget-parameters label = "Dependents: "

```

### Commentary

- Line 1        Introduces a new presentation template called `Person-Template`.
- Line 2        Specifies that the template displays objects of type `Person`.
- Line 3        Specifies that the display should be a column.
- Line 4        Introduces the parts of `Person-Template`. The first part is called `name-part`, and it shows the name of the person.
- Line 5        The presentation template for the name part is `Label-Template`, which adds a label to the display.
- Line 6        Specifies the parameters for the label widget to be created. The label is the value of the name slot of the person to be displayed (`a-person`).
- Line 7        Specifies that the font to be used for the label should be `Large-Bold-Font`.
- Line 8        The next part of `Person-Template`, called `address-part`, displays the address of the person.
- Line 9        The presentation template for `address-part` is a `Labeled-Object-Template`. This template is used to annotate the presentation of a template with a label. In this case we are annotating the display of a person's address with the label `"Address: "`.
- Line 10       The `Labeled-Object-Template` has a part called `object-part`, which specifies the object part of the label/object pair to be displayed.
- Line 11       The presentation template for `address-part` is `Label-Template`, which adds a label to the display.
- Lines 12-13   Specifies the parameters for the label widget to be created. The label is the value of the address slot of the person to be displayed.

- Line 14        The address should be displayed with italic font.
- Line 15        The label of the label/object pair displaying the address part is the string "Address: ".
- Line 16        The next part of `Person-Template`, called `dependents-part`, displays the dependents of the person.
- Lines 17       The presentation template for `dependents-part` is again `Labeled-Object-Template`.
- Line 18-19     The presentation template for the `object-part` of the `Labeled-Object-Template` presentation is a column.
- Line 20        The column has a part called `entry`, which is to be replicated for each dependent (see Line 24).
- Lines 21-23    The data to be displayed in each entry is `a-person`, whose value is the value of the `dependents` slot of the person being presented.
- Line 24        The entry part is to be replicated for each of the values of the `a-person` data, i.e., `Humanoid` will generate an instance of the `entry` part for each dependent.
- Line 25        The presentation template to be used for each dependent is `Person-Template`. This a recursive definition.
- Line 26        The label of the label/object pair displaying the dependents part is the string "Dependents: ".

Even though the person display is simple, interface builders cannot be used to specify it for two reasons. First, interface builders do not provide an iteration facility other than simple text strings, so it would not be possible to generate the display of the dependents, which requires a recursive application of the `Person-Template`. Second, interface builders do not support the definition of building blocks that can be used recursively.

It is conceivable that the display could be generated automatically by a tool similar to APT. However, as we develop the example further, we will introduce features that would be difficult to generate automatically.

In the next sections we show two refinements that designers might wish to make to display generated from the specification shown above. The first refinement is an example of a general class of refinement where designs need to handle boundary conditions in a special way (e.g., a person with no dependents). The second refinement shows how to construct context-sensitive presentations with `Humanoid`. The two refinements are easy to make in `Humanoid`, but are difficult or impossible to make with interface builders, automated designers, or other model-based interface design tools.

### Refinement #1

A bad feature of the person display is that if a person has no dependents, the label "Dependents:" is displayed, giving the impression that there is something missing. Two possible solutions to this problem are:

- Do not include the dependents part in the display, if there are no dependents. This effect can be achieved by defining an inclusion-condition for the `dependents-part` that returns false if there are no dependents. Adding the following line after line 16 modifies the design accordingly:

```
inclusion-condition (get-value a-person,  
dependents)
```

The call to get-value returns NIL if there are no dependents, making the inclusion condition false.

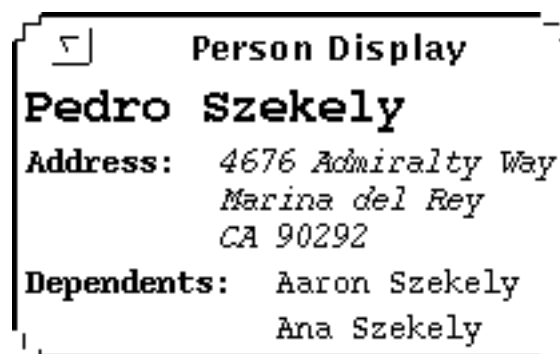
- Display the string “none” if there are no dependents. This effect can be achieved by inserting a conditional selection of presentation templates in line 16. The lines marked with “New” represent the new parts of the model. Line 16a introduces a conditional presentation template selection that selects the previously defined column template when there are dependents to display. Lines 26 to 28 have the second branch of the conditional, which displays the label “none” when a person has no dependents.

```
16a  condition = get-value (a-person, dependents)           New  
17    . presentation-template= Labeled-Object-Template  
18    . parts: object-part  
19    .       presentation-template= Column-Template  
20    .       parts: entry  
21    .       input-data:  
22    .           a-person = get-value (a-person,  
23    . dependents)  
24    .       replication-data a-person  
25    .       presentation-template= Person-  
Template  
26    condition = otherwise                               New  
27    presentation-template= Label-Template              New  
28    widget-parameters label = "none"                  New
```

It is conceivable that an automatic generation tool would have chosen one of the two solutions to create the person display. However, allowing designers to choose the alternative they want requires that the presentation model be accessible and editable by designers. Current automatic generation tools do not give designers such level of control.

## Refinement #2

Another feature that the designer might want to change is to hide the detailed information about the dependents. The refined presentation would show only a summary view of the dependent (e.g. only the name), and allow the user to display the detailed information via a command (e.g. by double-clicking on the name).



We show below how the presentation model can be changed to add the summary view.

The command to display the detailed information can also be added very easily in Humanoid by elaborating the application and manipulation models (see [Szekely92] for more details on how this would be done).

In order to add the summary view to the presentation, three simple changes are required.

- Define a `Summary-Person-Template` that displays a person in summary view:

```
Presentation-TemplateSummary-Person-Template
  input-data: a-person, type = Person
  is-a: Label-Template
  widget-parameters label = get-value (a-person, name)
```

- Define a substitution for `Person-Template` that selects `Summary-Person-Template` when the policy `summary-view` is active:

```
Template substitution forPerson-Template
  policy = summary-view
  presentation-template = Summary-Person-Template
```

- Specify that when the dependents of a person are to be displayed, the `summary-view` policy should be activated. This is achieved by inserting the following line after line 19, that specifies the entry part of the column that displays the dependents:

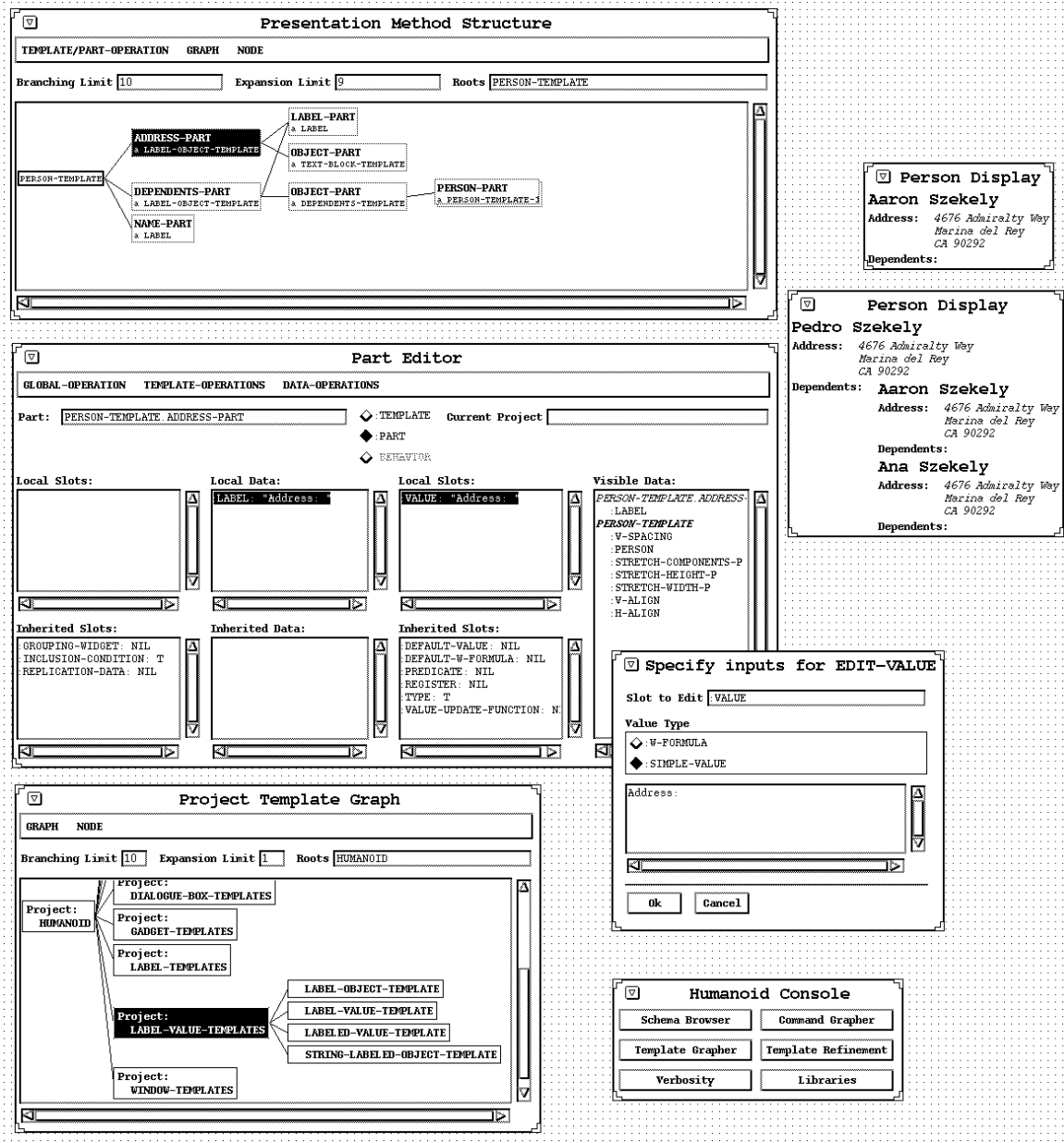
```
push-policy: summary-view
```

This kind of design exploration is not supported in either interface builders or automatic generation tools. Interface builders do not have the expressive power to represent the conditionality involved in this design. Automatic generation tools do not give designers the control to effect these changes. In Humanoid, these changes can be made via the interactive environment in a few minutes.

### The Humanoid Interactive Design Environment

The Humanoid interactive design environment provides a suite of tools to help designers construct and refine all aspects of a user interface, to help them understand the consequences of design decisions, and to help them manage an agenda of design tasks that arise during the design process [Luo92]. This section presents a brief overview of the tools to help designers construct and refine presentation models.

The presentation modeling tools are designed to support a very quick refine/prototype/evaluate design cycle. The standard mode of operation is for designers to have open one or more windows showing various aspects of the design, and simultaneously have open one or more windows that show examples of displays produced from the design for different sets of information.



The figure above shows a snapshot of a workstation screen while the designer is working on the Person-Template example discussed in this paper. The figure shows the main windows of the Humanoid design environment. The windows on the left belong to Humanoid, while the two windows on the right, titled Person Display are the displays generated from the model.

The window labeled Presentation Method Structure shows the structure of Person-Template as a tree of the hierarchical decomposition of the display into parts. This window provides menus to add and delete parts, to rearrange the structure of the parts, and a set of commands to prototype the design and to help designers understand designs. For example, there is one command that lets designers select a part of a presentation template (e.g. address-part), and ask Humanoid to highlight all the

portions of the prototype displays generated from the selected part. This command lets designers quickly see the portions of a display controlled by each part of a presentation template.

The window titled **Part Editor**, allows designers to edit all the slots of a presentation template. This window shows all the slots of currently selected part in the **Presentation Method Structure** window. The designer can bring up dialogue boxes such as the one titled **Specify inputs for EDIT-VALUE** to change any of the slots of a template. Whenever designers modify a design in any way, no matter how small the change is, all prototype displays are immediately updated to reflect the new design. In our example, the **Person Displays** are always updated to reflect any changes to the presentation templates.

The generated displays are updated even when the presentation templates are not fully specified. For example, if the designer adds a new part to a template, the prototype displays are immediately updated to show the new part. If the designer has not yet specified the presentation template for the part, Humanoid will show the part as a shaded area, to indicate that the relevant portion of the design is not yet complete.

The window labeled **Template Library Browser** allows designers to browse Humanoid's library of templates, or custom template libraries that designers might have created for their applications. Designers can inspect the attributes of templates, and when they find a template they want to use, they can drag it to the **Template-Part-Editor**. This interface feature is similar to the way in which the palette of building blocks is used in interface builders.

Humanoid also provides a set of commands that designers can apply to a prototype display (e.g. **Person Display**) to inspect the relationships between it and the presentation model that generated it. For example, designers can point with the mouse at a prototype display and ask Humanoid to highlight the parts of the presentation template that generated the graphics elements under the mouse cursor. This command helps designers identify the parts of the model they need to change by pointing at the portions of the prototype displays they wish to change.

The Humanoid design environment represents a first step towards a powerful presentation design tool that can support the generation of dynamic displays, provides extensive designer control, allows designers to delegate some design decision to the systems, and supports all aspects of interface design (application, presentation, manipulation, sequencing and side-effects), and is also easy to use.

We are actively working on improving Humanoid's usability in two respects. First, tools like the **Presentation Method Structure** and the **Part Editor** allow designers to construct and change presentation templates easily. These kinds of tools address the syntactic aspects of template construction. Second, tools like the facilities to highlight the relationships between a template and the displays it generates help designers understand templates. These facilities address an area where designers have many difficulties: there is a semantic gap [Hutchins85] between what designers must tell Humanoid (i.e. input-data, parts, conditions, etc.) and what designers want (i.e. certain display characteristics). This gap, or indirectness, is inevitable due to the conditional and iteration constructs. Humanoid attempts to bridge the gap by using a small and simple set of primitives for modeling presentations, and by providing visualizations (e.g. the highlighting of relationships) to help designers understand the models. Much research

remains to be done in this area. An interesting approach is to use demonstrational techniques [Myers87, Myers89 and Singh90] to let designers specify portions of a presentation model by showing the system examples of the desired displays. The demonstrational techniques would provide interfaces to model-based design tools that resemble the interfaces of today's interface builders.

It is worth noting that the Humanoid design environment is built using Humanoid. The environment windows make extensive use of the iteration and conditional constructs of Humanoid. They illustrate that the capabilities of Humanoid are powerful and convenient to build interfaces for real applications.

All windows use iteration: the two graph windows use iteration to generate the nodes in the graphs, the **Part Editor** uses iteration to generate the scrolling lists of label value pairs, and the dialogue box **Specify inputs for EDIT-VALUE** uses iteration to generate one block of the dialogue box for each input that the end-user can supply.

All windows use conditionals too. The **Presentation Method Structure** window uses conditionals to select the presentation for each node: the root node only shows a label, the nodes corresponding to the parts of the root template show the name of the part and the template to be used to display the part, replicated parts are shown with double borders (e.g. **Person-Part**). The most striking example of conditionality appears in the dialogue box. The dialogue box prompts for three inputs, and Humanoid selected different interaction techniques for each of them. The first input is displayed as a simple type-in area because the only information that Humanoid has about that input is that it is a symbol. The second input is displayed as a set of radio buttons because the input has a small set of alternatives, and only one should be selected. The third input is displayed as a multi-line type in area because the type of value requested (a formula) has a long printed representation.

## **RELATED WORK**

This section compares Humanoid to other systems that use an explicit design model.

The II [Arens88] system encodes the design model as rules. The left hand side of the rules captures the conditions under which presentation techniques can be used, and the right hand side specifies the presentation techniques to be used, and their parameters. Designers customize the behavior of the system by adding new rules. The shortcoming of II is that it only addresses the output part of interface generation, its rule language is complicated and hard to use, and the displays produced are not updated when the presented data changes. The main advantage of II over Humanoid is that rules provide a more flexible control mechanism to choose presentations than the top-down scheme used in Humanoid.

The ITS system [Bennett89] is another system that uses rules. In ITS, interactive programs are defined using three kinds of abstractions: data definitions, to define the data manipulated by the program, content trees, to define the dialogues in an abstract way, and style rules, to map the content trees to graphical objects. ITS constructs interfaces by first refining the content tree until it represents all the data to be presented to the user, and then applying the style rules to the refined content tree to obtain the graphical representation of the data. The main difference between ITS and Humanoid is that Humanoid uses the templates both to decide what information to present (content tree refinement) and to format the data (style rule application). Hence, Humanoid allows designers to control

what information to show in different contexts, where as ITS does not.

UIDE [DeBaar92, Foley91a] uses a model-based approach to control all aspects of user interface design (presentation, manipulation, sequencing and side-effects). UIDE emphasizes sequencing and side-effects, and provides tools to analyze design models for completeness and consistency [Braudes91]. UIDE also shows how the model can be used to automatically generate animated help showing how to execute the commands of the application [Sukaviriya88, Sukaviriya90]. UIDE's presentation model is much simpler than Humanoid's. It does not support the notion of nested presentations, conditionality and iteration, so it cannot be used to specify complex context-sensitive presentations.

## **CONCLUSIONS**

Humanoid is a model-based user interface design environment. Interface designers specify interfaces by representing declaratively the knowledge that characterizes the features of the interface. A run-time system or module uses the interface model together with run-time data structures of an application to generate the application displays and interpret inputs.

Humanoid's model-based approach to presentation specification allows it to provide many benefits that other approaches cannot provide. Humanoid

- supports the generation of dynamic displays,
- provides extensive designer control,
- allows designers to delegate design decision to the system,
- supports all aspects of interface design (application, presentation, manipulation, sequencing and side-effects), and,
- provides an environment with an array of tools to help designers refine their designs.

Humanoid also provides a framework to integrate power tools such as interface builders and automated assistants. An interface builder can be seen as a demonstrational interface to specify models that do not have conditionals and iteration. An automated assistant is another tool that works on a shared model, by automatically constructing model fragments based on application descriptions and little other information.

## **CURRENT STATUS AND FUTURE WORK**

Humanoid is implemented in CommonLisp, X Windows and Garnet [Myers92c]. Humanoid has been operational since early 1991, and is being used by three people other than its original designers to construct user interfaces. Humanoid has been used to construct the interfaces to DRAMA, and inventory control tool for the Defense Logistics Agency, to SHELTER, a knowledge-base development environment, and Humanoid itself (all the screen snapshots shown in this paper were generated using Humanoid).

We are actively working on the Humanoid design environment. We are refining the tools to make it more convenient to create and refine templates, and we are working on additional tools to visualize the relationships between models and the displays generated from them. We are also working on a tool to help designers manage the design process [Luo92]. This new tool will allow designers to state high level goals representing

features that the interface should have (e.g. objects in a window should be selectable). Once a goal is posted, the tool will decompose it into simpler goals that can be achieved by directly editing the model. The tool will keep an agenda of all the design goals and subgoals that are currently posted, but not met, and help designers achieve them.

## ACKNOWLEDGEMENTS

The research reported in this paper was supported by DARPA through Contract Numbers NCC 2-719 and N00174-91-0015. I also want to thank Peter Aberg, Yigal Arens, Ping Luo and Robert Neches for their help with this paper.

## REFERENCES

- Arens88 Y. Arens, L. Miller, S. Shapiro and N. Sondheimer. Automatic Construction of User Interface Displays. In AAI 88, 1988, pp. 808-813.
- Beshers89 C. M. Beshers and S. Feiner. Scope: Automated Generation of Graphical Interface. In Proceedings of ACM SIGGRAPH 1989 Symposium on User Interface Software and Technology (UIST '89). pp.76-85.
- Bennett89 Transformations on a Dialog Tree: Rule-Based Mapping of Content to Style. In Proceedings of ACM SIGGRAPH 1989 Symposium on User Interface Software and Technology (UIST '89). pp. 67-75.
- Braudes91 Braudes, R.E., and Sibert, J.L., "ConMod: A System for Conceptual Consistency Verification and Communication," SIGCHI Bulletin 23(1), Jan. 1991, pp.92-94.
- Buxton80 W. Buxton and R. Sniderman. Iteration in the Design of the Human-Computer Interface. In Proceedings of the 13th Annual Meeting of the Human Factors Association of Canada. 1980, pp. 72-80.
- DeBaar92 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.
- Feiner85 Feiner, S.K. APEX: An Experiment in the Automated Creation of Pictorial Explanations. IEEE Transactions on Computer Graphics and Applications, November 1985.
- Feiner90 Feiner, S.K. and McKeown, K. R. "Generating Coordinated Multimedia Explanations," Proceedings of the 6th IEEE Conference on Artificial Intelligence Applications, pp. 290-303, 1990.
- Foley91a 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.
- Giuse90 D. A. Giuse. Efficient Knowledge Representation Systems. Knowledge Engineering Review 5(1). pp. 35-50, 1990.

- Hayes85 Hayes, P. , Szekely, P. and Lerner, R. Design Alternatives for User Interface Management Systems Based on the Experience with COUSIN. In Proceedings of CHI'85, April 1985.
- Hutchins85 E. L. Hutchins, J. D. Hollan and D. A. Norman. Direct Manipulation Interfaces. Human-Computer Interaction, Vol. 1, 1985, pp. 311-338.
- Kim90 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.
- Luo92 P. Luo, P. Szekely and R. Neches. Management of Interface Design in Humanoid. Internal Memo. USC/ISI, 4676 Admiralty Way, Marina del Rey, CA 90292, 1992.
- Mackinlay86 J. Mackinlay. Automating the Design of Graphical Presentations of Relational Information. ACM Transactions on Graphics, pp. 110-141, April 1986.
- Myers87 B. A. Myers. Creating Dynamic Interaction Techniques by Demonstration. Proceedings of ACM CHI+GI'87 Conference on Human Factors in Computing Systems and Graphics Interface. 1987. pp.271-178.
- Myers89 B. A. Myers, B. Vander Zanden and R. B. Dannenberg. Creating Graphical Interactive Application Objects by Demonstration. Proceedings of ACM SIGGRAPH 1989 Symposium on User Interface Software and Technology (UIST '89), 1989, pp.95-104.
- Myers92a B. A. Myers and M. B. Rosson. Survey on user interface programming. In Proceedings of CHI'92, The National Conference on Computer-Human Interaction, May, 1992, pp. 195-202.
- Myers92b B. A. Myers. State of the Art in User Interface Software Tools. In H. Rex Hartson and Deborah Hix, Ed. , Advances in Human-Computer Interaction, Volume 4, Ablex Publishing, 1992.
- Myers92c B. A. Myers, et. al. The Garnet Reference Manuals. Technical Report CMU-CS-90-117-R2, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213. May 1992.
- Neuron91 Neuron Data, Inc. 1991. Open Interface Toolkit. 156 University Ave. Palo Alto, CA 94301.
- NeXT90 NeXT, Inc. 1990. Interface Builder, Palo Alto, CA.
- Olsen86 D. Olsen. MIKE: The Menu Interaction Kontrol Environment. ACM Transactions on Graphics, vol 17, no 3, pp. 43-50, 1986.
- Roth90 S. Roth and J. Mattis. Data Characterization for Intelligent Graphics Presentation. In Proceedings SIGCHI'90. April 1990, pp. 193-200.
- Singh89 G. Singh and M. Green. A High-level User Interface Management System. In Proceedings SIGCHI'89. April 1989, pp. 133-138.

- Singh90 G. Singh, C. H. Kok, and T. Y. Ngan. Druid: A system for Demonstrational Rapid User Interface Development. Proceedings of ACM SIGGRAPH 1990 Symposium on User Interface Software and Technology (UIST '90), 1990, pp.167-177.
- Sukaviriya88 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.
- Sukaviriya90 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.
- Szekely90 P. Szekely. Template-based mapping of application data to interactive displays. In Proceedings UIST'90. October 1990, pp. 1-9.
- Szekely92 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.