

# Facilitating the Exploration of Interface Design Alternatives: The HUMANOID Model of Interface Design

*Pedro Szekely, Ping Luo and Robert Neches*

USC/Information Sciences Institute  
4676 Admiralty Way, Suite 1001  
Marina del Rey, CA 90292  
(213) 822-1511  
szekely@isi.edu, ping@isi.edu, neches@isi.edu

## ABSTRACT

HUMANOID is a user interface design tool that lets designers express abstract conceptualizations of an interface in an executable form, allowing designers to experiment with scenarios and dialogues even before the application model is completely worked out. Three properties of the HUMANOID approach allow it to do so: a modularization of design issues into independent dimensions, support for multiple levels of specificity in mapping application models to user interface constructs, and mechanisms for constructing executable default user interface implementations from whatever level of specificity has been provided by the designer.

**KEYWORDS:** Design Processes, Development Tools and Methods, User Interface Management Systems, Rapid Prototyping, Interface Design Representation, Dialogue Specification.

## INTRODUCTION

Interface design really begins much earlier than current tools recognize. Long before a designer is ready to experiment with presentation issues like the layout of widgets chosen from a widget library, designers have typically made (often implicitly and unconsciously) strong design commitments about conceptual issues such as the choice of application data structures and capabilities that will be presented, as well as the general nature of interaction techniques which will be used to present them. By and large, the tools that designers use at this point are whiteboards or pad-and-pencil because the conceptualizations are more abstract than interface drawing or mock-up tools support. For example, a designer may decide that a file directory structure needs to be presented in a window, without yet knowing whether to use indented text or a grapher. The fact that those early conceptualizations are not supported on-line inhibits exploration of design alternatives. It is too difficult to walk through scenarios and imagine dialogues when sketching by hand. There is too much work and too much commitment to particular details when using a display layout package.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

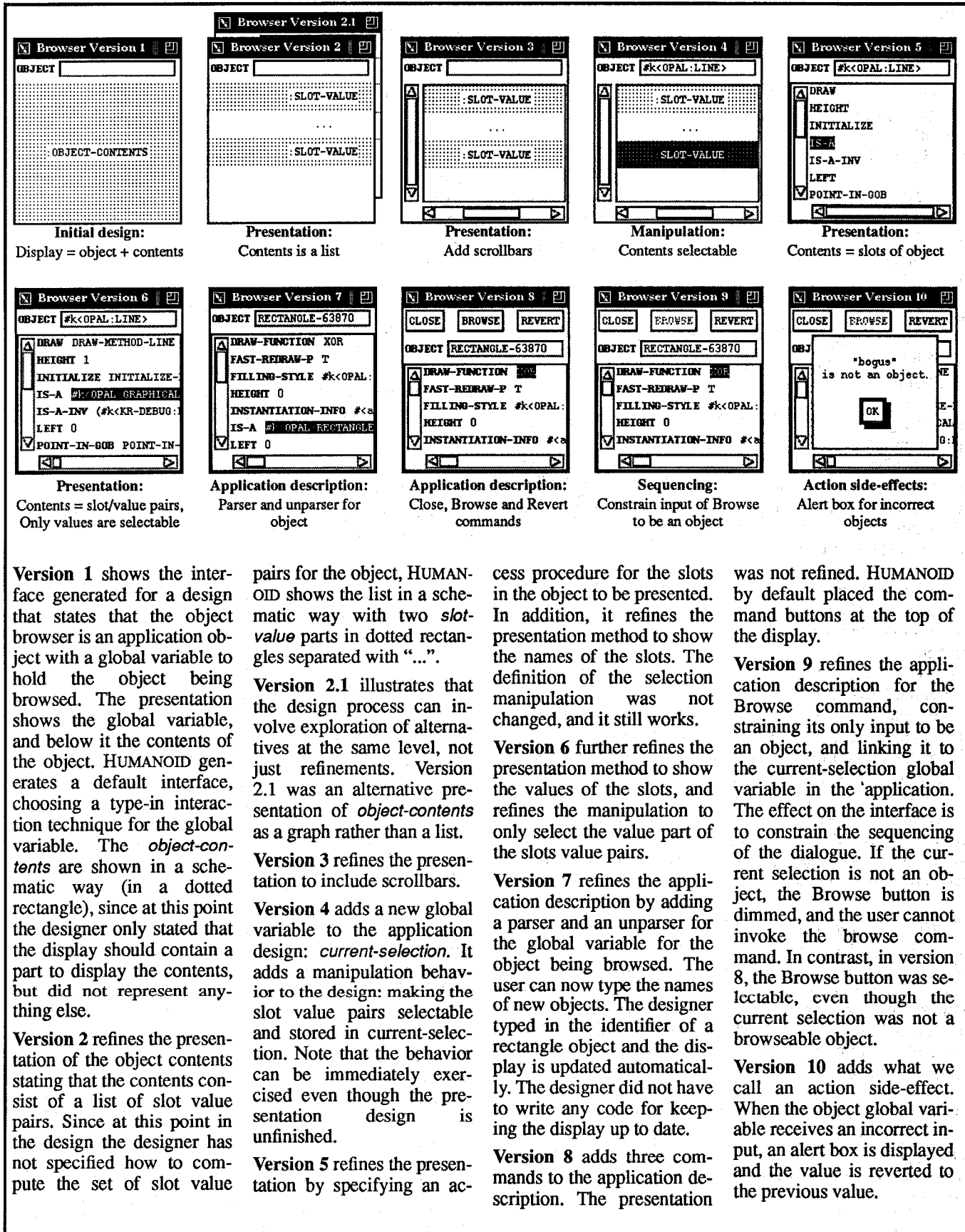
HUMANOID's contribution to interface design is that it lets designers express abstract conceptualizations in an executable form, allowing designers to experiment with scenarios and dialogues even before the application model is completely concretized. For example, designers can execute an interface after specifying only data types of command inputs, without having said anything about sequencing or presentation. The consequence is that designers can get an executable version of their design quickly, experiment with it in action, and then repeat the process after adding only whatever details are necessary to extend it along the particular dimension currently of interest to them.

Figure 1. illustrates the nature of this refinement process. The figure shows ten snapshots from the evolution of the design for a program to view the slots of an object, starting with an initially empty design and ending with a complete, working application. Each snapshot shows the interface that HUMANOID generates after one or two refinements to the design. The interfaces generated at each step are not just mock-ups of the presentation, but fully working interfaces, to the extent that they have been defined thus far. For example in Version 4 of the evolving design in Figure 1. the designer is able to explore dialogues involving selectable objects, despite not yet having defined how to display those objects. We believe that the design process is substantially enhanced by the opportunity this approach affords to put designs into action faster and earlier, and to test them before all the abstractions are concretized.

HUMANOID belongs to the family of interface design tools and UIMs centered around the notion of deriving the user interface from a high-level specification of the semantics of an application program [3, 6, 14, 15, 17, 18 and 20]. 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 with information used by an interface generating component. We recognize five dimensions along which interface designs can be varied and elaborated. We summarize them below, and expand on each dimension on the following sections.

*Application design.* The application design specifies the operations and objects that an application program provides. Variations involve changing the parameters of operations,

FIGURE 1. The evolution of the design of a program to browse objects.



deleting or adding operations, adding attributes to object types, etc.

Application design is the major direction of refinement in interface design. Our model of design views designers' exploration of design alternatives as a process of incrementally adding information to the application design, generating alternatives along the other four dimensions of variation as needed at each step.

**Presentation.** The presentation defines the visual appearance of the interface. Variations involve: data to present, 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. Many 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, closing a dialogue box can reset all the options to their default, typing return in a type-in field can automatically move the cursor to the "next" type-in field. Side-effects can add, set, delete values of command inputs and global variables, and can change the state of commands and inputs, which trigger changes in presentation and sequencing.

The sections following discuss each of these dimensions in detail, together with the benefits for interface design exploration that the variations provide. Then we discuss related work, and close with conclusions.

### DESIGN DIMENSION #1: APPLICATION DESIGN

An application design consists of a definition of commands, object types, global variables and the data flow constraints between these entities. The application design specifies the information about an application that is independent of how the objects are displayed, and how the operations are invoked. HUMANOID provides three kinds of objects to specify application designs:

**Commands.** A command is an object that describes all the information necessary to invoke an operation. The description of a command includes the call-back procedure that implements the command, a set of pre-conditions, and a description of each of the inputs to the command.

**Inputs.** An input is an object that describes all the information about a parameter of an operation needed to ensure that the interface will invoke the call-back procedure with correct arguments. The description of an input includes the *type* of the input value, a *predicate*, which is a procedure that does semantic validation on input values, the *minimum* and *maximum* number of values that the input can take, *alternatives*, which specify a set of values from which the input values must be chosen, and a *parser* and *unparser* for converting strings to input values and viceversa.

**Application Objects.** An application object groups together a set of commands and objects. An *application program* consists of one or more application objects. For example, a mail program could consist of an application object to manipulate folders and an application object to edit messages. At run time, a program can make multiple instances of its application objects. For instance, a mail program would have an application object instance to manipulate folders, and perhaps multiple instances of the application object for editing messages, so that the user would be able to compose multiple messages in parallel.

Input objects can also be defined for application objects. In this case they are called *global inputs*, because they are similar to global variables in a program. The value they hold can be accessed from any command in the application.

Application designs can be refined in three ways: by editing the command, input and application objects, by defining data flow constraints, and by defining command and input groups.

**Command, input and application modifications.** Commands, inputs and applications are organized in an inheritance hierarchy. Designers can define new versions of these objects, inheriting properties from existing ones. Designers can also add, delete and modify any of the properties of the command, input and application objects described above.

**Data flow constraints.** The data flow constraints specify constraints between the properties of inputs, command and application objects. One can, for instance, constrain the type, value, alternatives or other property of an input to be a function of any property of a command or an input. The constraints are enforced automatically by the underlying representation system [4]. Whenever the value of any of the slots changes, the constrained values are recomputed, and HUMANOID automatically reconstructs the affected portions of the display, enforces the relevant sequencing constraints, and performs the relevant side-effects.

The constraints between command inputs and global inputs can be used to implement selection-based interfaces, and other interface features where commands get their inputs from a global variable. For example, these constraints are used to support the factoring transformations in UIDE [3].

Version 9 of the Object Browser in Figure 1. illustrates the use of data-flow constraints. The *object-to-browse* input of the *browse* command is constrained to get its value from the

application global input called *current-selection*. Whenever a value is selected in the *object-contents* part of the display, the value is stored in the *current-selection* global input. HUMANOID propagates the value to the *object-to-browse* input, and enforces a sequencing constraint by disabling the *browse* button if the selected value is not an object.

*Command and input groups.* Command and input groups allow designers to add more structure to the definition of an application by grouping command and inputs into named objects. Command and input groups can be installed in application, command and input objects. The groups do not specify any interface feature by themselves, but designers can refine them to specify presentation, sequencing and side-effect interface features.

The application design represents information about an application program's semantics in a central place. This information is shared by the other four dimensions of interface design, and separating it out allows the other dimensions to be varied more independently. Also, since the application design describes the "objects of discourse" for the interface independently of other design dimensions, it represents a start towards explicitly capturing issues that designers are concerned with in the early, conceptual phases of design.

The following sections discuss the presentation, manipulation, sequencing and action side-effect dimensions of interface design refinements, which can be applied to application designs to yield interfaces with particular features.

## DESIGN DIMENSION #2: PRESENTATION

The presentation component of HUMANOID is designed to allow designers to specify presentations in stages. The goal is to let designers specify just the amount of information that they can or want at any given time during the design process, and to let them refine the presentation design iteratively as they understand the design better. HUMANOID can prototype the interface given any amount of information.

The presentation component of the user interface for a program is defined via *templates* [21]. A template is an object that specifies a method for constructing the presentation of a data structure. Designers construct presentations by refining the templates in HUMANOID's library.

HUMANOID's template library contains default templates to display application objects, commands and input objects, as well as templates to display lists of objects in columns, rows or tables, to display graphs and trees, and several flavors of scrolling windows.

The default template for application objects can generate menu bars of pull down menus, panels of command buttons, and panels of global inputs using radio buttons, check boxes, and other traditional interaction techniques. This template illustrates the use of command and input groups to specify presentations. For example, to create the application window in Version 8 of the Object Browser example in Figure 1., the designer defined a group called *panel-commands*,

with the *close*, *browse* and *revert* commands. The default template generates the panel of command buttons. Menu bars could have been generated by defining the *menu-bar-commands* command group, and the input panels are generated from the *panel-inputs* input group.

The templates mechanism supports the following kinds of presentation refinements:

*Adding parts to existing templates.* Designers can start by specifying what data should be displayed in a part, and later on refine the part to specify how the data should be displayed. For example, Version 1 of the Object Browser shown in Figure 1. was created by adding a part called *object-contents* to the default application template. Designers can initially provide some hints about the size and proportions of the presentations of parts so that when HUMANOID prototypes the interface it can generate presentations that approximate the presentations the designers have in mind.

*Adding inclusion conditions.* Designers can add an inclusion condition to the definition of a part so that it is only included in the display when the conditions are met. The conditionals can be constraint expressions that depend on application data. The constraints are automatically maintained so that when the application information changes, HUMANOID automatically updates the display to exclude or include the part as appropriate.

*Adding template applicability conditions.* Designers can add applicability conditions to templates to define the situations when the use of a template is appropriate. The applicability conditions can also be constraints. When the data they depend on changes, HUMANOID will automatically re-search the template hierarchy to find a new template to display the corresponding portion of the display.

*Refining parameters.* Designers can refine the parameters of a template to override default values (e.g., change the font of a class of labels). It is possible to put a constraint in the parameters so that the values will be computed at run-time based on application information. When the application information changes, HUMANOID automatically regenerates the appropriate parts of the display.

*Specifying layout.* Designers can specify the layout for the parts of a display separately from the specification of the parts. Designers can refine the layout once the design of the presentation of the parts is complete, in order to achieve a pleasing layout.

*Specifying a replacement hierarchy.* A replacement hierarchy is a decision tree for selecting the most appropriate templates for displaying an object. Each node in the tree is a template. Child templates construct more specific presentations than their parent, and are chosen only if their applicability condition is satisfied in the current context. When HUMANOID is directed to display an object with a template, it will search the replacement hierarchy below that template to find the lowest template in the hierarchy whose applicability condition is satisfied. By calling HUMANOID with a

template close to the top of a replacement hierarchy, designers delegate to HUMANOID the selection of presentation method. By specifying a template closer to the bottom of the hierarchy, designers exercise more control.

For example, HUMANOID has a template replacement hierarchy to choose between different presentations of input objects, such as check buttons, radio buttons or type-in buffers. The applicability conditions are based on attributes of the input object such as whether there is a set of alternatives from which the value is to be chosen, the size of the alternatives, the number of values that the input accepts, etc. Designers can build similar replacement hierarchies for the templates for their application data structures.

HUMANOID's template mechanism for constructing presentations has the following benefits:

- Designers can refine the presentation step by step, always seeing the effects of the current design, even when it is only partially specified.
- The part inclusion conditions, and template applicability conditions provide a natural way to create conditionalized displays whose characteristics depend on the run-time values of the data to be presented.
- The template replacement hierarchy provides a convenient way to organize and reuse presentation methods.
- HUMANOID automatically reconstructs displays when the data being presented changes.

### DESIGN DIMENSION #3: MANIPULATION

Manipulation specification involves specifying the input gestures that users can perform to manipulate presented information, along with the actions that should be invoked when the appropriate gesture is detected.

Manipulations are specified by adding to templates one or more *behavior* specifications. A behavior specification consists of a specification of the gesture that invokes the behavior (*e.g.*, mouse click, mouse drag), a specification of the parts of the presentation where the gesture applies (*e.g.*, over the widget generated by a template, or over all the parts of a template), a specification of the application data on which the gesture operates, and the actions to be taken at interesting points during the gesture (*e.g.*, for a dragging gesture the interesting points are "mouse press", "mouse move" and "mouse release").

The actions of behaviors can contain arbitrary Lisp code. However, typical actions are very simple. They only set the value of an input object, or change the status of an input, a command, or a group (see the following section on sequencing for an explanation of input, command and group status). Designers do not need to include code to, for instance, activate or deactivate other behaviors, highlight or dim presentations, etc. These are subsidiary actions to setting the value of an input or changing a status, and so are specified in the sequencing and action side-effect dimensions.

Version 4 of the Object Browser shown in Figure 1, defines a mouse-click behavior to select slot-value pairs. The *start-where* slot is the list of all the elements of the *object-contents* part (making all the slot-value pairs selectable), and the action sets the *current-selection* global input to the value presented in the slot-value pair that the user buttons. The action does not contain code to highlight the *current-selection*, or to enable or disable any commands that might use the *current-selection*. These features are specified in the presentation and sequencing aspects of the design. When commands are added later on in Version 8, there is no need to come back to this behavior and edit the actions in order to enable or disable the relevant commands.

The behaviors are implemented on top of the Garnet Interactors package [11]. The library of behaviors includes behaviors for type-in, dragging and moving, button and menu selection, angle specification, two point specification. These behaviors cover most of the gestures used in direct manipulation, mouse-based interfaces [11].

HUMANOID's model of manipulation has several benefits:

- Separates the specification of *what* the behaviors do, *where* they apply, and *when* they are applicable. The separation of what and where derives from the Garnet model of interactors [11]. The separation of when derives from HUMANOID's model of sequencing (see next section).
- When designers refine the presentation of the objects, it is not necessary to modify the definition of the manipulations. The manipulations will continue to work with the refined presentations of the objects.
- Behaviors are easy to specify because their actions are simple: they either set the value of an input, or change the status of an input, a command or a group. Interface designers need not program in order to specify the actions.

### DESIGN DIMENSION #4: SEQUENCING

Sequencing design involves specifying the order in which different displays appear on the screen, and the set of behaviors that are enabled at any given moment. In HUMANOID designers do not specify sequencing by directly adding instructions at appropriate places to enable or disable particular behaviors. Instead, HUMANOID computes the set of enabled behaviors at any time based on the data flow constraints in the application design, and by applying a fixed set of policies to a model of the states of individuals or groups of commands and inputs. The sequencing of the displays is computed in a similar way, and is explained in the next section on action side-effects.

We discuss the model of command states below. The model of states for inputs and groups of commands and inputs is similar, and is not discussed in this paper.

The state of a command is defined by the following slots:

*Idle/active/running.* Commands are *idle* by default. The *idle* state specifies that the command is not being interacted with. The *active* state specifies that the command is being interacted with in order to obtain the inputs needed before it

can run. The *running* state specifies that the call-back of the command is executing. HUMANOID automatically returns the command to its default state once the call-back returns.

HUMANOID's policies for determining enabled behaviors from these states are as follows. When a command is *idle*, only the behaviors that set it *active* are enabled. When the command is *active*, the behaviors that set inputs for the command are enabled (subject to one exception, as explained below), and the behaviors that set the command to *idle* are also enabled. The behaviors that set the state to *running* are enabled only if the command is ready to run, as defined by the *ready/not-ready* slot. When the state is *running*, no behaviors are enabled.

*Disabled/enabled.* When a command is *disabled* it cannot be made *active*, and the behaviors that set the command to *active* or *running* are disabled. The *disabled/enabled* slot can be defined with a constraint so that a command disables or enables itself by testing the state and value of any command or input in the application. By default, commands where one or more inputs are tied to a global input are automatically *disabled* when the value of the global input does not satisfy the *type*, the *minimum* and *maximum* restrictions, and the *predicate* defined for the command input. Commands whose preconditions are not satisfied are *disabled* by default.

*Ready/not-ready.* Indicates whether a command is ready to run, *i.e.*, whether it is *active*, all its inputs are correct, and no preconditions are violated. Behaviors that change the command to *running* are enabled only if the command is *ready*.

The *disabled/enabled* and the *ready/not-ready* act as guards: they specify the conditions under which the *idle/active/running* slot can change value. Designers control sequencing indirectly by defining additional constraints on the guard slots, and by triggering actions that change the state of other commands and inputs when the value of any of the three slots changes. HUMANOID propagates the effects of the state changes by enabling and disabling behaviors as explained above.

The actions triggered on state changes are specified via methods of commands (similarly, the sequencing constraints of inputs and groups are specified via methods of input and group objects). Whenever a state slot of a command changes from A to B, the method *a-to-b* is called on the command and all command groups to which the command belongs. For example, when a command changes from *active* to *running*, the method *active-to-running* is called; when a command is no longer ready to run, the method *ready-to-not-ready* is called. The method can then change the status of other commands in the group, or call arbitrary procedures.

The state transition methods provide a general mechanism for designers to control the states of individuals and groups of commands and inputs, and thus a general way to control sequencing. In addition HUMANOID provides a library of objects called *attributes*, which define packages of methods that implement commonly-used sequencing features. For example, HUMANOID provides the following attributes for

command sequencing (similar attributes are provided for input sequencing):

- *Only-One-Active.* This attribute specifies that only one command in a group can be *active* at any given time. When a new command in the group is made *active*, the previously *active* command is made *idle*. When no command is *active*, a pre-designated command in the group is made *active*, if one is defined.
- *Only-One-Enabled.* This attribute specifies that when a command in a group is made *active* or *running*, the other commands in the group are *disabled*.

These two command sequencing attributes can be used to implement familiar interface features. *Only-One-Active* can be used to specify the sequencing of the palette of drawing commands in a MacDraw-like drawing program, where the user selects a tool to draw. Only one tool is selected at any given time, and only the behaviors for the selected tool are enabled over the drawing area. *Only-One-Enabled* can be used to disable the menu bar of pull down menus while one of the commands is either executing, or prompting for inputs in a dialogue box. Note that in both examples the attributes are defined for command groups that the designer would define to control the presentation.

To incorporate sequencing constraints into a design, the designer simply lists the relevant attribute in the *attributes* slot of individuals or groups of command and inputs. If an attribute that packages the desired sequencing constraints does not exist, the designer first has to define it, by defining the appropriate methods. This mechanism makes simple, commonly used features easy for designers to use, but provides enough generality so that complex sequencing constraints can also be implemented.

The *browse* command in Version 9 of the Object Browser shown in Figure 1. illustrates the sequencing model. The *object-to-browse* input of the *browse* command is defined to be of type *Object*. The default definition of the *disabled/enabled* slot specifies that if the *object-to-browse* is incorrect, the command is *disabled*, causing the behavior that activates the command to be disabled so that clicking on the Browse button has no effect. So, when the user selects an object in the *object-contents* part of the display, the button is enabled, but if the user selects a non-object such as the constant XOR, the button is disabled.

HUMANOID's sequencing model has the following benefits:

- Provides a much less cumbersome means of specifying sequencing than event-based systems [5], or state transition networks [7]. Rather than specifying sequencing at the level of gestures/behaviors, or a potentially large number of states, HUMANOID derives the sequencing constraints on behaviors by applying a fixed set of policies to simple state model of commands, inputs and groups.
- Provides a framework (states and methods) for designers to express complex sequencing constraints, and provides abstractions (attributes) that make it easy to express commonly used sequencing constraints.

- Provides good support for design exploration because much sequencing behavior falls out from the data flow constraints expressed in the application design, with no extra effort needed from the designer. Additional sequencing constraints are expressed as annotations to individuals and groups of commands and inputs that are often used for presentation purposes too.

#### DESIGN DIMENSION #5: ACTION SIDE EFFECTS

Action side-effects are actions performed automatically as side effects of the actions triggered by user inputs. For example, a newly created object can become automatically selected, closing a dialogue box can reset all the options to their default, typing return in a type-in field can automatically move the cursor to the "next" type-in field.

Action side-effects are expressed using the command, input and group state transition methods described in the previous section. Whenever a behavior sets the value of an input, or changes the state of a command or an input, methods indicating the change are called.

Designers can specify side-effects by writing methods for the appropriate state transitions, or can define attributes similar to the sequencing attributes. For example, to cause a dialogue box to appear in response to a menu selection, designers can write a method for the *idle-to-active* state transition that calls *present-object* with the command and *dialogue-box-template* as its parameters. Since showing dialogue boxes is a common case, HUMANOID provides a command attribute called *Show-Dialogue-Box*.

HUMANOID provides the following attributes for commonly used side-effects on inputs. A similar library for command and group side-effects is also provided, but it is not discussed in this paper.

*Revert-When-Incorrect.* When an input is set to an incorrect value, the previously correct value is automatically restored.

*Message-When-Incorrect.* When an input is set to an incorrect value, an alert box is posted.

*Beep-When-Incorrect.* When an input is set to an incorrect value, the interface beeps.

*Prompt-Ring.* This attribute is defined for input groups. When an input in the group is set, the behaviors for the next input in the group are automatically activated.

Version 10 of the Object Browser shown in Figure 1. is an example of side-effect specification. It uses the *Revert-When-Incorrect* and the *Message-When-Incorrect* action side-effect attributes on the *object* global input, where users can enter the object to be viewed. If the user types in an incorrect object, the user will be notified with an alert box, and the value of the object is reverted to the previous (correct) value.

HUMANOID's model of side-effects has several benefits, which derive mostly from linking side-effects to command,

input and group state transitions, rather than specifying them in the actions of behaviors:

- Makes behaviors easier to reuse. Since the side-effects of behaviors are separate from the behaviors, the same behavior can be used in different contexts that require different side-effects.
- Increases modularity. Side-effects are represented centrally in the command, input or group objects, rather than being spread out in the possible multiple behaviors that act on these objects.
- Provides good support for design exploration. The manipulations and side-effect dimensions can be explored independently because the side-effects depend on the effects of the action of a behavior rather than on the behavior itself.

#### RELATED WORK

The most sophisticated of the UIMSs centered around the notion of deriving the user interface from a high-level specification of the semantics of a program are MIKE [14], UofA\* [18] and UIDE [3]. MIKE and UofA\* are able to generate a default interface from a minimal application description, and provide a few parameters that a designer can set to control the resulting interface. MIKE allows designers to define the interaction techniques for prompting for inputs, the structure of the menus, and actions to be executed when presentations are selected. However, MIKE has a built-in prefix dialogue structure that cannot be changed. UofA\* supports prefix, post-fix and no-fix dialogue structures, supports current selected objects, and open-ended, and close-ended command invocation. Both systems allow designers to refine the layout.

HUMANOID's general model of commands allows designers to exert much finer control over dialogue sequencing. In addition, HUMANOID provides a library of command groups that allows designers to very easily specify the dialogue structures that MIKE and UofA\* support. HUMANOID also provides finer control over presentation design, and supports the construction of the "main window" of application programs, which MIKE and UofA\* do not support. UIDE's application description is much richer than those used in MIKE and UofA\*. Such richer descriptions can be used to support more sophisticated design tools [3] (help generation, consistency and completeness checking, automatic dialogue box and menu design, transformations). Even though we have not constructed such sophisticated design tools, it should be possible to construct them, since our application description provides the necessary knowledge.

HUMANOID's application description is similar to UIDE's. HUMANOID improves on UIDE by providing a richer model of command states, enabling designers to exert finer control over dialogue sequencing. For example, HUMANOID's action side-effect mechanism subsumes UIDE's post-condition one, because it allows commands to assert side-effects on any of the state transitions of a command, not just on the successful execution of a command. In addition, HUMANOID provides more sophisticated facilities for refining the pre-

sentation and manipulation dimensions of the interface.

The command and input groups attributes provide a compact mechanism to specify dialogue sequencing similar to Statecharts [23], that avoids the explosion of states that occur in state transition networks. In fact, command and input group sequencing attributes can be used to emulate all the dialogue structures supported in the UofA\* UIMS [18], plus provides the framework to define others.

Interface builders such as the Next Interface Builder [13], and OpenInterface [12] are a different class of tools to aid in the design of interfaces. These tools allow designers to draw interfaces consisting of check boxes, radio buttons, labels, type-in areas and other such interface building blocks. These tools make it very easy to construct the particular interfaces they support, but they are poor for design exploration. Designers have to commit to particular presentation, layout and interaction techniques early in the design. Making changes to the dialogue structure is difficult. For example, changing an input prompted in a dialogue box to a global input is difficult because all dialogue boxes that prompt for that input have to be manually edited. Also, making global policy changes such as changing the interaction technique to present choices requires manually editing a large number of displays. Achieving the same results that HUMANOID enables by using an interface builders, if it can be done at all, requires a level of programming sophistication beyond the reach of the designers for whom these tools are intended.

Systems such as ITS [1] and HUMANOID do not have this problem because to change a global policy it is enough to change a rule in ITS, or a template in HUMANOID. HUMANOID and ITS provide similar facilities for constructing presentations, but ITS lacks the facilities to do interface design along the other dimensions that HUMANOID supports.

Interface builders are currently easier to use than application description-centered systems like HUMANOID, MIKE, UofA\* and UIDE, for constructing simple displays like dialogue boxes. However, this shortcoming can be overcome. APT [9], SAGE [16] and DON [8] are examples of systems that automatically generate high quality displays from the design knowledge base. APT and SAGE generate high quality charts, and DON, which is based in UIDE, is an initial attempt to generate high quality dialogue boxes.

HUMANOID currently lacks an interactive interface to construct the application description, which MIKE and UIDE have, and an interactive layout editor, which MIKE and UofA\* have. We are currently working to remedy this shortcoming. Perhaps the ultimate interactive interface for design should also build on demonstrational systems like Lapidary [10] and Druid [19]. These systems allow the designer to specify the presentation and the behavior of an interface by example. Designers draw the interface as the user will see it, and then demonstrate the actions that users can perform, by graphically manipulating the presentation. These systems generalize the examples, and generate code that implements the general case. The attractiveness of these systems is in

their claims for ease of use. We view these systems as potentially complementary to knowledge-based systems like HUMANOID. For instance, one could imagine a Lapidary-like interface to specify some of the design changes illustrated in our Object Browser example. To specify that the slot-value pairs should be selectable, and highlighted in reverse video, the designer could draw the black, xored, rectangle, and the Lapidary-like tool would make the appropriate generalizations. It is an open research issue, however, whether demonstrational tools can be made sophisticated enough to design complex interfaces.

Our work only partly addresses issues of task analysis and user centered design: HUMANOID facilitates creating designs that act upon realizations obtained through these design approaches, but does not address these methods directly.

### CONCLUSIONS

HUMANOID is an interface design system that lets designers express abstract conceptualizations of an interface design in executable form, allowing designers to experiment with scenarios and dialogues before the application model is completely concretized. The novel features of HUMANOID are:

- Supports top-down design. Designers can refine interfaces step by step. At any step designers can ask HUMANOID to generate the interface in order to try it out, or can refine it further.
- Allows designers to delay committing to specific interface features until they want to. Designers do not have to fully specify any aspect of the design in order to prototype it or to refine another aspect of it. For example, designers can specify manipulations for a presentation that has not been fully specified.
- Allows designers to explore the design space in any order. HUMANOID supports both breadth-first and depth-first design strategies, or any combination in between. In the breadth-first strategy, designers can specify the complete interface at a high level before refining any aspect of it. In the depth-first strategy designers can fully specify one aspect of the design (presentation, manipulation, sequencing) before working on a different aspect.
- Supports the specification of all aspects of an interface. HUMANOID supports the iterative specification of the application functionality, its presentation, input behavior and sequencing. HUMANOID supports the construction of the interface for the "main windows" of programs, not just the menus and commands to control the program.
- Provides an extensive library of presentation methods, and defaults for choosing presentation methods based on the types of objects to be displayed. For example, HUMANOID can construct dialogue boxes automatically, choosing check boxes, radio buttons, etc. automatically based on the types of the inputs to be requested. Designers can provide various hints that steer the dialogue box constructions in several directions.
- Defines a novel partitioning of interface design spaces into relatively independent dimensions, allowing designers to refine designs along these dimensions. The exam-

ple in Figure 1. illustrates how designers can refine the presentation, application description, interactive manipulation and sequencing aspects of the interface in a rather independent manner.

We believe that HUMANOID substantially enhances the iterative design process required for constructing good user interfaces [2, 22] by allowing designs to be put into action faster and earlier than current design tools allow, and by allowing designers to refine designs along multiple, relatively independent dimensions.

#### ACKNOWLEDGEMENTS

We wish to thank Peter Aberg, David Benjamin, and Brian Harp for helpful comments on earlier drafts of this paper. This work was supported by DLA and DARPA under contracts #MDA972-90-C-0060 and #N00014-91-J-1623. Contents represent the opinions of the authors, and do not reflect official positions of DLA, DARPA, or any other government agency.

#### REFERENCES

- 1 W. Bennett, S. Boies, J. Gould, S. Greene and C. Wiecha. Transformations on a Dialog Tree: Rule-Based Mapping of Content to Style. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pp. 67-75, November 1989.
- 2 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.
- 3 J. D. Foley, W. C. Kim, S. Kovacevic and K. Murray. UIIDE: An Intelligent User Interface Design Environment. In J. S. Sullivan and S. W. Tyler, editors, *Intelligent User Interfaces*. pp. 339-384. ACM Press, 1991.
- 4 D. Giuse. Efficient Frame Systems. In J. P. Martins and E. M. Morgado, editors, *Lecture Notes in Artificial Intelligence*, Springer Verlag, Sep, 1989.
- 5 M. Green. Report on dialogue specification tools. In *User Interface Management Systems*. G. E. Pfaff, editor, Spring-Verlag, 1983, pp. 9-20.
- 6 P. J. Hayes, P. Szekely and R. Lerner. Design Alternatives for User Interface Management Systems Based on Experience with COUSIN. In *Proceedings SIGCHI'85*. April 1989, pp. 169-175.
- 7 R. K. Jacob. A specification language for direct manipulation interfaces. *ACM Transactions on Graphics* 5, 4. (October 1986), 283-317.
- 8 W. C. Kim and J. Foley. DON: User Interface Presentation Design Assistant. In *Proceedings UIST'90*. October 1990, pp. 10-20.
- 9 J. Mackinlay. Automating the Design of Graphical Presentations of Relational Information. *ACM Transactions on Graphics*, pp. 110-141, April 1986.
- 10 B. Myers, B. Vander Zanden and R. Dannenberg. Creating Graphical Interactive Application Objects by Demonstration. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pp. 95-104, November 1989.
- 11 B. A. Myers. A New Model for Handling Input. *ACM Transactions on Information Systems* 8, 3. (July 1990), pp. 289-320.
- 12 Neuron Data, Inc. 1991. *Open Interface Toolkit*. 156 University Ave. Palo Alto, CA 94301.
- 13 NeXT, Inc. 1990. *Interface Builder*, Palo Alto, CA.
- 14 D. Olsen. MIKE: The Menu Interaction Kontrol Environment. *ACM Transactions on Graphics*, vol 17, no 3, pp. 43-50, 1986.
- 15 D. Olsen. A Programming Language Basis for User Interface Management. In *Proceedings SIGCHI'89*. April 1989, pp. 171-176.
- 16 S. Roth and J. Mattis. Data Characterization for Intelligent Graphics Presentation. In *Proceedings SIGCHI'90*. April 1990, pp. 193-200.
- 17 K. J. Schmucker. MacApp: An application framework. In R. M. Baecker, W. A. Buxton, editors, *Readings in Human-Computer Interaction*. pp. 591-594. Morgan Kaufmann Publishers, Inc. 1987.
- 18 G. Singh and M. Green. A High-level User Interface Management System. In *Proceedings SIGCHI'89*. April 1989, pp. 133-138.
- 19 G. Singh, C. H. Kok and T. Y. Ngan. Druid: A System for Demonstrational Rapid User Interface Development. In *Proceedings UIST'90*. October 1990, pp. 167-177.
- 20 P. Szekely. Standardizing the interface between applications and UIMS's. In *Proceedings UIST'89*. November 1989, pp. 34-42.
- 21 P. Szekely. Template-based mapping of application data to interactive displays. In *Proceedings UIST'90*. October 1990, pp. 1-9.
- 22 W. Swartout and R. Balzer. On the Inevitable Intertwining of Specification and Implementation. *CACM* 25, 7 (July 1982), pp. 438-440.
- 23 P. Wellner. Statemaster: A UIMS Based on Statecharts for Prototyping and Target Implementation. In *Proceedings SIGCHI'89*. April 1989, pp. 177-182.