A Compositional View of UserInterfaces - A Framework for Fully-Functional User Interface Design Tools

by

Srdjan Kovacevic

GIT-GVU-91-22
September 1991

Graphics, Visualization & Usability Center

Georgia Institute of Technology
Atlanta GA 30332-0280

A COMPOSITIONAL VIEW OF USER INTERFACES – A FRAMEWORK FOR FULLY-FUNCTIONAL USER INTERFACE DESIGN TOOLS

Srdjan Kovacevic

Georgia Institute of Technology
College of Computing
Atlanta, GA 30332-0280
srdjan@cc.gatech.edu

ABSTRACT
A compositional view of user interfaces identifies common user interface component types and structuring principles for assembling components into a coherent interface. Both the components and the principles are reusable across different interfaces. A user interface is represented as a composition of primitives, where selection of primitives and their structuring depend on the application for which an interface is built, and on a desired dialogue style. A model based on the compositional view can support a wide range of different designs and easy transitions from one design to another. We describe the model and a tool built on top of it, and give examples of structures built by the tool, to illustrate how primitives can be assembled to meet requirements of a specific application and a dialogue style.

KEYWORDS: UI models, UI design tools, UI design space, UI components, design tool functionality.

INTRODUCTION

User interface (UI) design, being an iterative, exploratory process [4], puts specific requirements on UI design tools. These requirements can be expressed in terms of a design tool's functionality, static and dynamic [15]. The static functionality of a tool is its capability to produce different designs, whereas the dynamic functionality is its capability to support change, going from one design to another. Thus, if we have a design tool with which we can produce both command language and direct manipulation based user interfaces, dynamic functionality pertains to the ease with which we can produce direct manipulation interface once we have the command language one, or vice versa.

If we think of all possible UI designs as forming a design space, then the coverage of this design space corresponds to a design tool's static functionality, whereas support for navigation in the design space, going from one point to another, corresponds to the tool's dynamic functionality.

The problem with contemporary UI design tools is that they suffer either from poor static, or poor dynamic functionality, or both. Adding to the problem is the fact that the design space is not fixed, but expanding – technology advances are opening new possibilities for human-computer interaction, making possible new user interface designs, and most tools cannot keep up with these advances.

The capabilities of a UI design tool are limited by its underlying model. Features of a UI recognized and explicitly represented by the model are also potentially manipulable by the tool. On the other hand, features not explicitly represented in the model are not manipulable by the tool either; in this case a particular solution is "hardwired" in all UIs produced by the tool, thus leaving it beyond direct designer's control.

Obviously, this affects a tool's functionality. Hard-wired solutions provided by the tool affect the static functionality, or coverage of the design space. Dynamic functionality is affected as well. Unless the tool has knowledge about a specific feature of a UI (e.g. command syntax), and what variations are possible (e.g. prefix, postfix, and nofix), it cannot assist in changing design with respect to this feature (e.g. from prefix to postfix command syntax), though design variations can be produced using lower-level support.

Navigational support includes not only assistance in moving from one point in space to another, but guidance in the process. It is especially important when designing complex UIs, where features interact and changing one aspect of a UI affects others. Again, unless the model captures all features and their interdependencies, the tool cannot offer this level of navigational support.

Also affecting a UI design tool's functionality is its access to application semantics. Early UI research was based on the premise that an application's UI can be isolated from the application's functionality, allowing for development of different UIs for an application without affecting its non-interactive part. The Seeheim model [11] is representative of this traditional approach. However, separation inherently limits the range of interfaces that can be produced; in particular, interfaces providing semantic feedback are not possible without access to the application semantics. Accordingly, the underlying model must capture enough application semantics for a class of UIs to be produced by a design tool.

We present a compositional view of UIs, which identifies common UI components and their relationships. Based on a compositional view, we have defined a model which meets the above requirements. It supports a wide range of different designs and easy transitions from one design to another, and it allows for integrating different kinds of knowledge needed for guiding a UI designer. The compositional model views a UI as a composition of primitives from a finite set, structured in a specific way; the exact structure and primitives used depend on the application for which a user interface is being built and on a desired dialogue style. It integrates the compositional view of UIs, which identifies major parts of a UI, their roles and their contribution to UI characteristic, and the UIDE model [7,8,9] which captures the application conceptual model and a notion of design transformations.

Examples we use are based on the Circuit Design application. It has two major classes of objects: *gates*, with properties *position*, *angle* (orientation), *fan-in* (maximum number of inputs), *fan-out* (maximum number of outputs), *free-in-lines* (number of available input lines), and *free-out-lines* (number of available output lines); and *connectors*, with properties *source-gate* and *destination-gate*. Subclass *NOT-gate* differs from other subclasses of gate in that its fan-in property is always 1. Gates can be created, deleted, moved, rotated, aligned, connected (creates a connector object), and disconnected (deletes a connector).

RELATED WORK

The compositional model is the successor to the UIDE model. User Interface Design Environment (UIDE) integrates a control and a data model of the application – the control model pertains more to the syntax, describing sequences of user actions needed to input control information and how this information relates to application commands and their parameters, while the data model describes the application-specific objects: their types, properties and relationships. They complement one another and each represents a natural way of expressing some aspects of the application conceptual model. Other systems which integrate the control and data models are GWUIMS [18] and Serpent [1].

Most other UI tools rely on models focusing on either control or data aspects, which is the reason why they support only a subset of dialogue styles. Conversational style interfaces (e.g. Mike [17], Cousin [12], the UofA UIMS [11], and Diction [19]), are typical for tools focusing on the control model. In contrast, direct manipulation interfaces feature rich interaction semantics and require the data model. The Higgens system [14], built around the data model, is tuned for direct manipulation interfaces. However, it lacks explicit control model and does not support other dialogue styles. Similarly, the Nephew system [21,23] supports direct manipulation interfaces, but also lacks full-fledged global control model and hence cannot support other dialogue styles.

Even tools supporting different dialogue styles do not provide specific support for changing designs. An exception is UIDE [6,9], which provides a set of built-in transformations specifically aimed to make switching from one design to another easy. Diction [19] provides limited support for change; its model captures a notion of command syntax and the mapping between different syntaxes, allowing a designer to change command syntax easily. The compositional model extends the set of transformations provided by UIDE; it extended the UIDE model by explicit representation of a UI and allows for finer tuning of UI designs.

The idea behind the compositional view that a UI is a composition and that we can identify a set of UI building blocks is not new. Thus, Szekely [21,23] classifies the communication concepts characteristic for graphical interfaces, and shows that these concepts fall into a small number of categories. Based on this classification, Szekely proposes a standard for an internal interface between an application and a User Interface Management System, and a set of reusable building blocks that can be used to implement a class of graphical user interfaces. Hurley and Sibert [13] also focus on the internal interface. Their CREASE model uses its five modeling primitives – Conceptual Relation, Entity, Action, State, and Event – to model the information flow through the internal interface and how much knowledge about the application it requires.

Szekely [22] also addressed constructing presentation component of a UI from the graphical building blocks; he proposed a template-based method for composing widgets and tying them to application objects. Others have addressed input devices and interaction techniques. Card, Mackinley, & Robertson [5] propose a model that represents elementary devices as 6-tuples, and includes operators for composing elementary devices into more complex devices. Bleser and Sibert [3] propose the input model that also incorporates human factors criteria of uses of input devices, and serves as a basis for a tool for guiding a designer in selecting interaction techniques.

Sukaviriya [20] integrates into the UIDE model a framework for providing context-sensitive animated help, which also includes a model of interaction techniques. Tatsukawa [24] proposed a model consisting of a set of primitive components which can be manipulated graphically into an event-flow graph describing user interactions. However, his components do not provide direct support for handling application concepts (other than callbacks to application functions which handle various events), but are rather suited for defining interaction techniques.

The compositional model goes beyond past models by providing unified framework integrating the UI model and the application conceptual model. It does not focus on a set of primitives characteristic only for a specific type of interfaces, or a dialogue style, but it provides a set of generic component types common across different dialogue styles, or combination of dialogue styles, including multimedia, multimodal interfaces [16].

COMPOSITIONAL VIEW OF USER INTERFACES

UI Characteristics

A UI of an application is the part of the application in charge of the communication with a user. The rest of the application is a non-interactive part, providing the application functionality. Conceptually, we model this as separated parts, though the separation need not be clearly manifested in a physical implementation. A UI facilitates communication between the application functional part (from now on, referred to as the functional part) and a user. The communication consists of the information exchange in both directions: from the user to the functional part (user inputs), and from the functional part to the user (the application feedback). We consider characteristics of the communication on two levels: surface level, as seen by the user, and internal level, as seen by the application (i.e. the functional part).

On the surface level, the communication between the user and the application is characterized by its *syntax*, *interaction techniques*, and *feedback*. *Syntax* pertains to sequencing of inputs and outputs, that is, the order in which units of information are communicated from the user to the application and back. *Interaction techniques* are methods of using input devices that the user can employ to specify inputs. Finally, *feedback* pertains to the ways of presenting information to the user, that is, the way of encoding the output information. Differences on the surface level are relatively easy to spot; they are what makes UIs look and feel distinctive – the form that information exchanged takes is different.

Internally, regardless of surface differences, all UIs to the same functional part must communicate the same type of information. What concerns the application functionality is the content of the information exchanged, not its form.

A distinction between the information content and the information form is very important. The content of the information a UI communicates depends on the application functionality, not on the UI itself. The application's information needs determine *what* is to be communicated via the UI. In contrast, the UI establishes the way this information is actually communicated. All UIs belonging to the same application must be able to communicate the same information content, e.g. an object's type and properties, but they may differ in the information form.

There are two aspects of the communication that a UI can vary: *how* and *when* communication takes place. *How* corresponds to the mapping, performed by the UI, between internal representation of the information, used in communications with the application functional part, and external properties presented to and manipulated by the user. On the input side, *how* depends on the interaction techniques used to specify a piece of information. Thus, the user can specify a position either by typing its coordinates, or by selecting it by pointing. Similarly, the user can specify a tone by typing its specification, by creating a graphical symbol corresponding to the tone, or by playing it on a keyboard. On the output side, *how* depends on the choice of output medium and properties in that medium used to represent properties in the application domain. It is possible to represent properties in one domain via properties in another domain; what this requires is establishing a proper set of associations and conventions. For instance, a musical piece can be presented by playing it, e.g. using a synthesizer, or it can be presented graphically using musical notation; however, to understand the graphical representation, the user must be familiar with the music notations. A contrasting example would be representing graphical properties using sound, e.g. using a musical pattern (e.g. an earcone [2]) to represent object type, and volume and pitch to represent its scalar properties such as size and orientation.

The other aspect of communication, *when*, corresponds to syntax, or sequencing of the information exchange events. Similarly to *how*, *when* also pertains to both directions of the information exchange, input and output events. The sequencing has three parts to it: sequencing of user inputs, sequencing of output events comprising the feedback, and timing of feedback events with respect to the user inputs causing them.

Sequencing of user inputs corresponds to the order in which information units are entered. Variety here includes more than just a set of permutations of inputs on the level of individual actions, such as those corresponding to prefix, postfix, or some other syntax. It also considers default and global values of information units, which break into a direct correspondence between the information content that a user specifies and the information content the UI passes to the application functional part. If an information unit has a default value, that means that the user does not have to specify it (it is optional); if the user chooses not to, the UI will use a designer defined default value when passing information to the functional part. To the functional part, it is transparent whether a value comes from the user, or the UI has obtained it in some other way. In the case of global values, the UI designer does not specify the value at design time, rather the user does this at run time, using special actions provided by the UI designer. With default values, we have a situation when a UI passes the information to the functional part even when the user does not specify it. In the case of global values, we have the opposite situation as well – the UI does not pass to the UI the information the user did specify, but stores it internally. This is an important property of a UI, that it can store information and use it at some later stage, or even reuse it repeatedly.

In the same way a UI can buffer and reorder user inputs, it can buffer outputs when providing feedback. A situation when this may be useful is when an object goes through a sequence of changes, and interim states are not of interest, hence require no in-between feedback. A contrasting example is the animation of transition from one state to another when UI interpolates the interim states an object has to go through, thus generating more feedback than asked for by the functional part.

A UI can control timing of feedback events with respect to the user inputs causing them. Feedback can be provided immediately after each user input (we denote this as *dynamic* feedback), or it can be delayed until after a sequence of user inputs is completed (*static* feedback). Thus, when moving an object, feedback can be given for each new position, or only after the final position is selected. More specifically, in a graphical editor, when moving a shape, the shape can follow the cursor, or simply jump to a new position. Again, these variations are the result of different UIs; the functional part remains the same, but the UI can buffer and reorder input and output events.

Building Blocks of UIs – Functional Components
As we have seen, a UI is not just a passive transducer of information between a user and the application functional part. It performs tasks falling into three major categories:

- Map user inputs into internal representation and internal representation to external representation;
- Buffer information, both that coming from the user and from the functional part;
- Control information flow, where information goes to and comes from.

These tasks are common for all UIs – communication is inherent to an interactive application; buffering occurs even if default and global values are not used, as information communicated from a user must be stored somewhere while all pieces of information needed for carrying out an action are collected; controlling information flow is necessary since there are multiple sources and destination of information. These components are necessary parts of a UI, because they perform mandatory UI tasks. A UI can also include other components, for performing optional tasks, such as error-recovery and help. These tasks belong to the UI because they do not contribute to the application's functionality, but to the quality of its UI. On the other hand, they are optional since a UI can function without them, though a user will probably perceive such a UI as of lesser quality. Figure 1 represents a UI as a composition of three kinds of components – communication, buffering, and control – performing the mandatory tasks mentioned above, and *other* component for optional UI tasks. In the rest of this section we further subclass the mandatory component types.

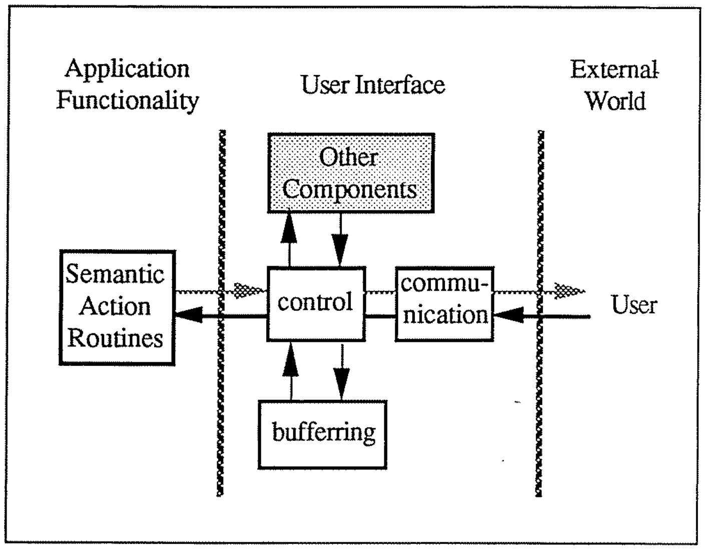

Figure 1 – A Compositional View on a User Interface

Communication Components. While the role of a UI as a whole is to exchange information between a user and an application, the communication components perform the part of the exchange involving the user. While doing this, they typically transform the information: input is converted into internal form which can be stored in the UI itself before passing it to the application; and, internal form is transformed into the form used to present information in the external world. Each instance of a communication primitive acts as a converter, mapping one information form into another, and is characterized by the mapping it performs. A primitive is selected based on its characteristic mapping.

The communication components category has two subclasses:

- Interaction techniques
- Presentation objects

Interaction techniques are defined as methods of using input devices for producing desired types of input [10]. They map external input into internal form. An interaction technique can be simple, such as "push button", or composite, such as "drag: push button, move mouse, release the button". Each interaction technique is characterized by (1) the input device it uses, (2) the interactions it takes upon these devices, and (3) outputs it produces – what interaction tasks it is good for. Based on these characteristics we can select an interaction technique based on available input devices and the required interaction task, and also detect possible conflicts – whether two techniques that can be enabled simultaneously have (leading) interaction steps in common.

Presentation objects are defined as a set of presentation routines which map given internal information form into desired presentation properties, such as color, size, shape, or tone pitch.

The set of communication primitives (instances of communication components category) is open – as technological advances are constantly bringing new possibilities for communication exploiting new external information forms, new primitives are being added to the set. The primitives, together with descriptions of their characteristics, are organized as a catalog, or a library. They are selected from the catalog based on requirements of application semantics (primitives must support a specific information type) and desired dialogue styles (must have a specific external form).

Buffering components. Primitives belonging to this category maintain the context of a UI. Contrary to the communication primitives, the buffering primitives do not perform any conversion of information, but serve as passive repositories of information. The buffering components category has three subclasses, each maintaining a different aspect of the overall UI context:

- Action-object – for the context pertaining to individual application actions,
- Object-prototype – for the context pertaining to application objects, and
- Blackboard – for the context pertaining to the UI state.

During the activation of an action, the *action-object* primitive keeps all (known) pieces of information needed to perform the action, decoupling the order in which these pieces of information are specified from the order in which they are passed to the application functional part. Between the activations, the *action-object* instance keeps the information which is reusable across multiple activations of the action and specific to that action, such as local default values and partially-global values for action parameters. These default and global values may have corresponding object attributes, but are kept as the part of action's context because the action may use different default or global values. In other situations there may not be a corresponding object attribute; then keeping default or global values in the action-object primitive is the only alternative. For instance, *vertical-alignment* parameter in *align* command does not have corresponding object attribute.

The *object-prototype* primitive keep global properties for each application object class. These include default attribute values and reusable (factored) attribute values. Both the action-object primitives and the object-prototype primitives are instantiated to match their corresponding application actions and objects. For instance, action-object primitives must have a slot for each unit of information it uses, i.e. for each action parameter. Additional slots may be needed for default and global values defined specifically for its corresponding action. Similarly, object-prototype primitives must have a slot for each default or global value its properties may have.

Both the action and object related contexts implicitly describe some aspects of the UI state. The context not captured there is maintained in a different set of primitives, the *blackboard* primitives. They have in common that they post "public" information accessible to all, which is not the case with the action-object and object-prototype primitives. The posting include state information for maintaining the sequencing and information flow control, and properties of sets, e.g. sets of currently selected objects, non-selected objects, and clipboard objects. Additional primitives may be used to maintain history context (that is, not only what the current context is, but how we got here), but we are not modeling the history context for now.

Control Components. Primitives belonging to this category tie in all parts of a UI in a coherent whole. They link communication and buffering primitives and application semantic routines and maintain and direct information flow between all these primitives. Control primitives not only pass information, but decide where each piece of information goes to and where it comes from. Their role spans three major tasks and there is a subclass corresponding to each of the three tasks:

- Information flow control,
- Sequencing control, and
- Event propagation subclass.

The first subclass, *information flow control,* maintains information flow among primitives, thus integrating functional parts of a UI into a single structure. Its primitives act as intermediaries in obtaining information required by the action and they know where to look for each piece of information, whether to get it from a communication primitive, or from a buffering component, and which one. They are specialized according to the nature of the information they pass and what it is used for. The information user provides may be (1) a value for an action's parameter, or (2) an event aimed to change the action's state. Action parameters can pertain to an object selection (either object class or instance), or a data value (typically corresponding to an object attribute); the corresponding specialized primitives are *select-object* and *get-attribute*. We distinguish the following events for changing an action's state: *select-action*, *cancel-action*, *invoke-action*, *suspend-action*, and *resume-action*; these are also the names of corresponding specialized primitives handling those events. There are no explicit events for enabling/disabling an action because the user does not enable/disable actions directly, but it is the side effect of other actions. Whether the user has to explicitly signal events for the primitives listed here or not, depends on a specific UI design. For instance, invoke-action event is specified only if confirmation is required, otherwise this primitive will "fire" automatically as soon as all necessary information is collected and will pass the information to an application's semantic action routine. Also belonging to this category is *bundle* specialized primitive.

It connects other information flow control primitives to a single communication primitive.

The second subclass, *sequencing control*, maintains the relevant UI context; it updates the context whenever something potentially affecting information flow control primitives happens, and it constantly evaluates the context to enable/disable those primitives. The two tasks are performed by specialized primitives: postconditions are updating the context, while preconditions are evaluating it.

Finally, the third subclass, *event propagation*, propagates events of interest, possibly performing relation detection and enforcement. While pre- and post-conditions also propagate events when enabling or disabling other primitives, they do it indirectly through context updates. Event propagation primitives do this directly – they monitor an event of interest and, when it happens, propagate it right to a desired target. Doing so, they effectively link other primitives and establish a flow of information between them. In that sense they complement information flow control primitives – they are specialized to monitor specific events, which do not have to originate from the user inputs.

Other Components. This subclass includes primitives which go beyond the minimal UI in embodying additional knowledge about the UI domain. They include common interface objects and actions, functional, and "advanced" primitives. Interface objects and actions are comparable to application objects and actions, but in the UI domain. They allow reuse of commonly used interface objects, such as windows, and buttons, and actions manipulating them, such as scroll-window, push-button, and are closely related to communication primitives which use the same interface objects [20].

Functional primitives perform standard functions commonly needed in different UIs and applications, such as name management, clipboard management, data-base management, and set management. Primitives for name management are used to provide a unique name or identifier for an entity. Clipboard has become common way of information exchange between different processes, or between parts of the same process, and this exchange is facilitated by cut, copy, and paste primitives. Data base management encompasses instantiating and deleting object instances, and modifying, inspecting, and saving attribute values. Set management primitives include primitives for selecting and deselecting objects, for adding objects to the set, and for removing objects from the set. Finally, the advanced primitives include primitives for maintaining history, providing various forms of help, and for error recovery. UIDE [8,9,20] demonstrated some possibilities in providing automatically generated help and history mechanism, and we plan to expand work in this direction

Compositional View and Reusability

The compositional view offers several important characteristics which give us a leverage for building UI design tools. One is that there is a finite set of component types that can be found in any UI. However, the set of primitives (instances of those component types) is not fixed — it is open to expand to accommodate technology advances opening new ways of communicating information and allowing for creation of new instances of communication components.

Another important characteristics is that these components are reusable across different interfaces and different applications. While a user interface as a whole is not generally reusable across applications, because we cannot just plug any UI into any application, its parts are. The user interface as a whole is application semantics dependent, which is why it cannot be easily reused. Its building blocks are also application dependent, but to a lesser extent. Selection of primitives is driven primarily by the application semantics, that is, the types of information they must handle. But, the same primitive, handling specific information type, can be reused across different user interfaces and applications.

Structuring of components is also application dependent, and it makes up for the rest of the application specificity that a UI has, but is not captured in its components. Accordingly, the resulting structure is not directly reusable across applications and UIs, but basic structuring principles are. This reusability of components and structuring principles is what makes the compositional view an excellent framework for developing highly functional design tools.

EXAMPLES

Different UIs are produced by replacing individual primitives and changing the way the composed parts are structured. By replacing one instance of the communication component with another, handling the same information type, we can change the external form of information as seen by the user. Different syntaxes can be achieved by changing the control and buffering components. Thus, we can control *when* and *how* information is communicated by selecting different instances of components and structuring them differently, resulting in different dialogue styles. We illustrate this with examples showing how to compose UI structures for two representative commands from our sample application, move-gate and create-gate.

Move-gate command has two parameters, object to be moved and position where to place the object. The initial design we consider has confirmation mode and cancel option enabled, and only type-in interaction technique enabled. Figure 2 shows the resulting structure. On the left hand side are *information-flow-control* primitives instantiated for action parameters (*cm-so-move-object* for object parameter, and *cm-ga-move-position* for position parameter), and for changing the action's state (*cm-sa-move* for selecting the action, *cm-ca-move* for canceling, and *cm-ia-move* for invoking its semantic action routine). All information flow control primitives are connected to *aoi-move* buffering primitive for maintaining the action's con-

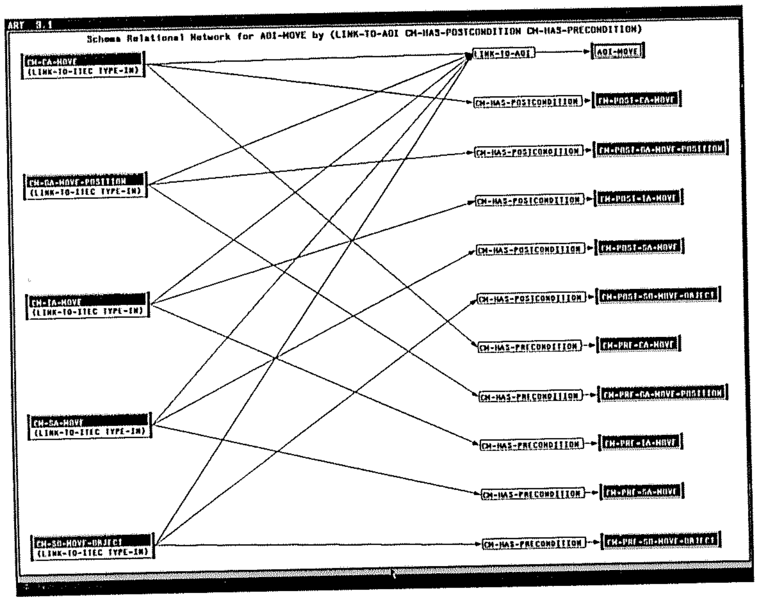
ART 3.1
Schema Relational Network for ADI-MOVE by (LINK-TO-ADI CM-HAS-POSTCONDITION CM-HAS-PRECONDITION)
CM-CA-MOVE (LINK-TO-ITEC TYPE-IN)
CM-GA-MOVE-POSITION (LINK-TO-ITEC TYPE-IN)
CM-IA-MOVE (LINK-TO-ITEC TYPE-IN)
CM-SA-MOVE (LINK-TO-ITEC TYPE-IN)
CM-SO-MOVE-OBJECT (LINK-TO-ITEC TYPE-IN)
LINK-TO-ADI
ADI-MOVE
CM-HAS-POSTCONDITION
CM-POST-CA-MOVE
CM-POST-GA-MOVE-POSITION
CM-POST-IA-MOVE
CM-POST-SA-MOVE
CM-POST-SO-MOVE-OBJECT
CM-HAS-PRECONDITION
CM-PRE-CA-MOVE
CM-PRE-GA-MOVE-POSITION
CM-PRE-IA-MOVE
CM-PRE-SA-MOVE
CM-PRE-SO-MOVE-OBJECT

Figure 2

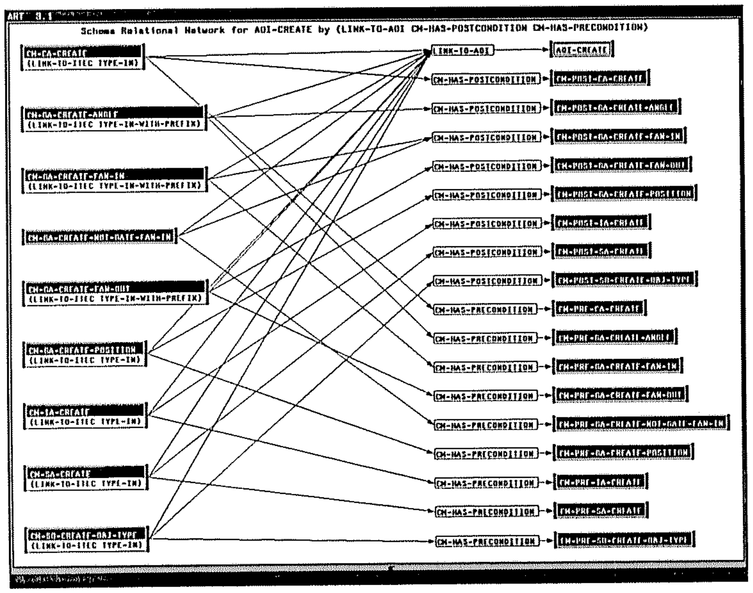
ART 3.1
Schema Relational Network for ADI-CREATE by (LINK-TO-ADI CM-HAS-POSTCONDITION CM-HAS-PRECONDITION)
CM-CA-CREATE (LINK-TO-ITEC TYPE-IN)
CM-GA-CREATE-ANGLE (LINK-TO-ITEC TYPE-IN-WITH-PREFIX)
CM-GA-CREATE-FAN-IN (LINK-TO-ITEC TYPE-IN-WITH-PREFIX)
CM-GA-CREATE-NOT-GATE-FAN-IN
CM-GA-CREATE-FAN-OUT (LINK-TO-ITEC TYPE-IN-WITH-PREFIX)
CM-GA-CREATE-POSITION (LINK-TO-ITEC TYPE-IN)
CM-IA-CREATE (LINK-TO-ITEC TYPE-IN)
CM-SA-CREATE (LINK-TO-ITEC TYPE-IN)
CM-SO-CREATE-OBJ-TYPE (LINK-TO-ITEC TYPE-IN)
LINK-TO-ADI
ADI-CREATE
CM-HAS-POSTCONDITION
CM-POST-CA-CREATE
CM-POST-GA-CREATE-ANGLE
CM-POST-GA-CREATE-FAN-IN
CM-POST-GA-CREATE-FAN-OUT
CM-POST-GA-CREATE-POSITION
CM-POST-IA-CREATE
CM-POST-SA-CREATE
CM-POST-SO-CREATE-OBJ-TYPE
CM-HAS-PRECONDITION
CM-PRE-CA-CREATE
CM-PRE-GA-CREATE-ANGLE
CM-PRE-GA-CREATE-FAN-IN
CM-PRE-GA-CREATE-FAN-OUT
CM-PRE-GA-CREATE-NOT-GATE-FAN-IN
CM-PRE-GA-CREATE-POSITION
CM-PRE-IA-CREATE
CM-PRE-SA-CREATE
CM-PRE-SO-CREATE-OBJ-TYPE

Figure 4

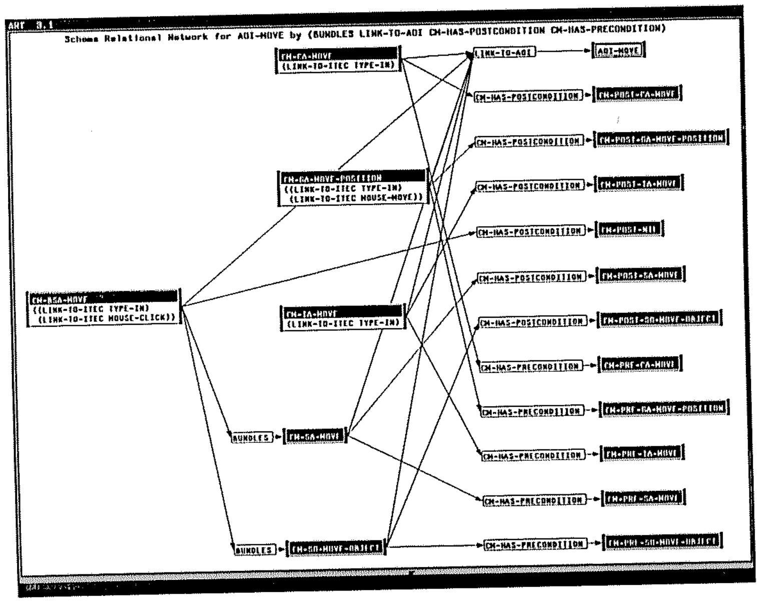
ART 3.1
Schema Relational Network for ADI-MOVE by (BUNDLES LINK-TO-ADI CM-HAS-POSTCONDITION CM-HAS-PRECONDITION)
CM-CA-MOVE (LINK-TO-ITEC TYPE-IN)
CM-GA-MOVE-POSITION ((LINK-TO-ITEC TYPE-IN) (LINK-TO-ITEC MOUSE-MOVE))
CM-GSA-MOVE ((LINK-TO-ITEC TYPE-IN) (LINK-TO-ITEC MOUSE-CLICK))
CM-IA-MOVE (LINK-TO-ITEC TYPE-IN)
BUNDLES
CM-SA-MOVE
CM-SO-MOVE-OBJECT
LINK-TO-ADI
ADI-MOVE
CM-HAS-POSTCONDITION
CM-POST-CA-MOVE
CM-POST-GA-MOVE-POSITION
CM-POST-IA-MOVE
CM-POST-NIL
CM-POST-SA-MOVE
CM-POST-SO-MOVE-OBJECT
CM-HAS-PRECONDITION
CM-PRE-CA-MOVE
CM-PRE-GA-MOVE-POSITION
CM-PRE-IA-MOVE
CM-PRE-SA-MOVE
CM-PRE-SO-MOVE-OBJECT

Figure 3

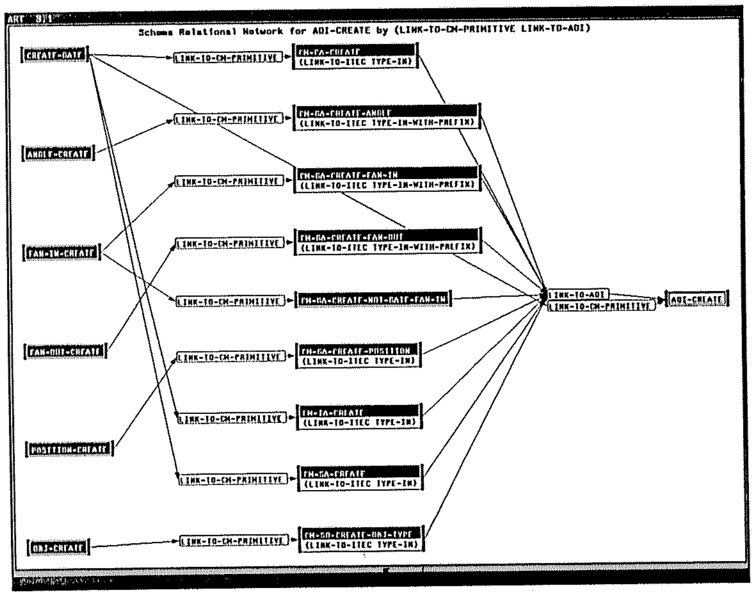
ART 3.1
Schema Relational Network for ADI-CREATE by (LINK-TO-CM-PRIMITIVE LINK-TO-ADI)
CREATE-GATE
ANGLE-CREATE
FAN-IN-CREATE
FAN-OUT-CREATE
POSITION-CREATE
OBJ-CREATE
LINK-TO-CM-PRIMITIVE
CM-CA-CREATE (LINK-TO-ITEC TYPE-IN)
CM-GA-CREATE-ANGLE (LINK-TO-ITEC TYPE-IN-WITH-PREFIX)
CM-GA-CREATE-FAN-IN (LINK-TO-ITEC TYPE-IN-WITH-PREFIX)
CM-GA-CREATE-FAN-OUT (LINK-TO-ITEC TYPE-IN-WITH-PREFIX)
CM-GA-CREATE-NOT-GATE-FAN-IN
CM-GA-CREATE-POSITION (LINK-TO-ITEC TYPE-IN)
CM-IA-CREATE (LINK-TO-ITEC TYPE-IN)
CM-SA-CREATE (LINK-TO-ITEC TYPE-IN)
CM-SO-CREATE-OBJ-TYPE (LINK-TO-ITEC TYPE-IN)
LINK-TO-ADI
LINK-TO-CM-PRIMITIVE
ADI-CREATE

Figure 5

text. Each of them is also connected to a pair of *sequencing control* primitives, a pre- and a post-condition. Boxes *link-to-aoi*, *cm-has-postcondition* and *cm-has-precondition* symbolize these relations. Each information flow control primitive is associated with an interaction technique – in this case *type-in* – represented by a *link-to-itec* relation in the white box below the primitive's name. To keep the diagram simple, we did not show buffering primitives not directly related to this action, or *event-propagation* primitives monitoring the state of aoi-move primitive.

The structure supports the following type of interaction. A default design assumes the prefix syntax, therefore all information flow control primitives are disabled when cm-sa-move is disabled (the action must be selected first). Once it is enabled and the user specifies required input, it (i.e. its postconditions and other primitives' preconditions) enables other information flow control primitives (except cm-ia-move – semantic action routine cannot be invoked before all necessary information is provided); after that they can accept the user input as well. Once both object and position are specified, the user can complete action by providing input required by cm-ia-move. Or, the user can cancel the action at any time.

Now we modify this design to use different interaction techniques (position can also be specified by pointing). We also bundle primitives for selecting the move action and its object parameter. This means that we don't have to explicitly and separately specify an event for selecting the action, but when we select object, the action is selected as well. As a result, syntax changes and is no longer prefix. Figure 3 illustrates a structure for this new design. Among changes we can see is a new primitive, *cm-bsa-move*, which is an instance of *bundle*. Also, bundled primitives are no longer linked to any interaction technique, but to their bundling primitive instead; it is in turn linked to interaction technique which can deliver information for all bundled primitives. Move command in this design works simply by selecting object and a new position for the object.

We can further modify a design, e.g. by changing feedback style of cm-ga-move-position primitive (corresponding to parameter position) to *dynamic*, attaching a continuous interaction technique which sense each mouse move. Move command would now give impression of direct manipulation: once we select a gate, it will follow the mouse until we confirm the action or cancel it. Confirmation may in turn be associated with mouse click; if the object is selected by push-button, then we get interaction where we push button over the object, start dragging it, and release the gate by releasing the button. Another way to achieve the same effect is to bundle the primitive for position parameter together with already bundled primitives and attach the resulting bundle to *dragging* composite interaction technique. This example is not shown here.

Figure 4 shows the structure for *create-gate* command. It is similar to *move* structure, but more complex, with more primitives, since create command has more parameters.

COMPOSITIONAL MODEL

A compositional view offers a potential for a rich static functionality. As shown in the previous section, different designs can be produced by replacing individual primitives and rearranging their structure; these designs range from conventional command language interfaces to direct manipulation interfaces, and, with adequate set of communication primitives, can be extended to virtual reality interfaces as well. While this is powerful, there is still a problem of assembling interfaces from the the set of primitives identified in the compositional view.

It is possible to generate a UI from these primitives directly – either graphically or by writing the code "by hand", without additional assistance from any tool. The latter is what we were doing in our pilot version of the tool based on the compositional view. We were modifying designs by changing specifications of individual primitives. Compositional rules also identify structuring principles and how changes in the structure affect the resulting UI. Using these principles, a UI can be generated automatically, and later changed in a systematic way to conform to requirements of an application and of a desired dialogue style. This is done in the compositional model, which integrates the compositional view with the UIDE model. It thus unites the UI model with the application conceptual model, enabling utilization of those structuring principles – the application conceptual model captures knowledge about the application semantics and can therefore drive rules for generating UIs, which are in turn based on the structuring principles we identified using compositional view of UIs.

The compositional model has four major parts:

- Modeling primitives,
- Composition rules,
- Transformation rules, and
- Conflict detection and resolution rules.

Modeling primitives include both UI primitives we discussed earlier, and the application modeling primitives based on the concepts in the UIDE model [7,9,10].

Composition rules establish relationship between the application modeling primitives and the UI primitives. They capture structuring principles and automatically generate a (default) UI based on the application conceptual model. Figure 5 illustrate this process. It shows how *create* application action and its parameters are mapped into their corresponding UI primitives. Boxes *link-to-cm-primitive* represent this relation. We see that the action is mapped into its state-changing primitives and its buffering primitive. Each action parameter is mapped into a information flow control primitive. *Fan-in-create* parameter is a special case. It corresponds to *fan-in* attribute of *gate* and its allowable values depend on which object is selected. It is mapped to two information flow control primitives which handle different possible values for *not-gate* and for all other *gate* objects. Their preconditions capture dependencies between parameters and will enable the right primitive once object type is known (a value of parameter *obj-create*). Note that *cm-ga-create-not-gate-fan-in* primitive, which handles fan-in value for NOT-gate, is not linked to any interaction technique – it does not need any input since the value is predefined (equal 1). The example illustrates that the compositional model is powerful enough to capture complex dependencies between UI components and, what is also important, the composition rules can handle those dependencies automatically.

Transformation rules complement composition rules by allowing easy access to other (non-default) designs. They are a composition model's vehicle for navigation in the design space, adding to its dynamic functionality. For instance, design changes described in the previous section are easily made applying appropriate transformations.

Finally, conflict detection and resolution rules utilize knowledge about individual UI primitives and their interaction to detect potentially invalid design. Currently, we detect ambiguous situations in which the same user's input is mapped to different interpretations. This can be extended to detect conflicts on the output side as well – for instance, mapping different internal values into the same external value. The system also offers the assistance in resolving detected conflict, typically by changing the interaction techniques, or modifying dialogue syntax [16].

IMPLEMENTATION

We are currently working on the third version of our tool based on the compositional view, and later the compositional model. The first one was the pilot version, featuring only UI primitives without accompanying composition and transformation rules. Assembly of a UI and its modifications had to be done by changing the original specification. Based on the experience with this version we refined the representation of UI primitives, and integrated the UIDE model, as well as automatic composition, transformation rules, and conflict detection and resolution rules. The first two versions were done in Inference Corp's Art3.1 and Common Lisp, on Sun 3/60 and HP9000 platforms. Figures 2-5 are for instance created by this version of our tool; all structures shown are automatically generated. We are now porting our system to Art4 and CLOS, running on SparcII.

Using Art enables us to combine rule-based and object-oriented programming. All primitives are defined as Art schemata, which makes subclassing easy, and also makes the resulting structure visible to pattern-matching rules which check design for conflicts and inconsistencies. Behavior of a primitive is implemented by methods attached to its corresponding schemata, and by active values and pattern-matching rules. *Event propagation* subclass of primitives is completely implemented via active values and pattern-matching rules.

SUMMARY

The existence of the finite set of primitives, or building blocks, and of the basic set of structuring principles is what makes UI design tools possible. We have presented the compositional view of user interfaces which identifies the set of common UI primitives and structuring principles and is a significant step in building the UI domain knowledge needed for developing fully-functional UI tools. We showed how UI primitives can be used to build various UI designs, and illustrated how it can be easily modified to produce different dialogue styles.

We have integrated the compositional view into the UIDE framework; the resulting compositional model unites the UI model and the application conceptual model. The UI model features common UI primitives which, with adequate library of communication primitives, can represent a wide variety of different UIs. Add to this composition and transformation rules for automatic generation and easy modification of UIs, and we get a framework for a fully functional UI design tool capable of delivering rich static and dynamic functionality.

We are in the process of further enhancing the set of transformations, to include high-level transformation that would capture knowledge of differences between typical dialogue styles and enable designer to go directly between such designs. We are also improving navigational support, to recognize and provide assistance for more conflicts and design inconsistencies.

ACKNOWLEDGEMENTS

The Graphics and User Interface Research Group at the George Washington University provided an intellectual environment in which this work originated. I am grateful to James Foley and other members of the UIDE research team for their discussions concerning some of these ideas. Special thanks to Piyawadee Sukaviriya who gave me valuable comments on earlier versions of the paper. Financial support was provided by National Science Foundation grant IRI-8813179, Inference Corporation, Software Productivity Consortium, Siemens Corporation, the Department of EE & CS Industrial Liaison Program at GWU, and the Graphics, Visualization, and Usability Center at the Georgia Institute of Technology.

REFERENCES

[1] Bass, L., Hardy E., Hoyt K., Little M., and Seacord R. Introduction to the Serpent User Interface Management System. Software Engineering Institute,

Carnegie-Mellon University, Pittsburgh, PA, March 1988.

[2] Blattner, M.M., Sumikawa, D.A., and Greenberg, R.M. Earcons and Icons: their Structure and Common Design Principles. Human-Computer Interaction, Vol. 4, pp. 11-44.

[3] Bleser, T., and Sibert J. Toto: A Tool for Selecting Interaction Techniques. In Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology. Snowbird, Utah, October 1990, pp. 135-142.

[4] Buxton, W.A., Sniderman R. Iteration in the Design of the Human-Computer Interface. In Proceedings of the 13th Annual Meeting of the Human Factors Association of Canada, pp.72-81.

[5] Card, S.K., Mackinlay J.D., and Robertson G.G. The Design Space of Input Devices. In Proceedings of CHI'90 (Seattle, Washington, April 1-5, 1990). ACM New York, 1990, pp. 117-124.

[6] Foley, J., Gibbs C., and Kim W. Algorithms to Transform the Formal Specification of a User-Computer Interface. In Proceedings INTERACT '87, 2nd IFIP Conference on Human-Computer Interaction. Elsevier Science Publishers, Amsterdam, 1987, pp. 1001-1006.

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

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

[9] Foley, J., Kim W., Kovacevic S., and Murray K. UIDE – An Intelligent User Interface Design Environment. In Sullivan, J. and Tyler, S. (eds.), Architectures for Intelligent Interfaces: Elements and Prototypes, Addison-Wesley, 1991.

[10] Foley, J., Gieskens, D., Kim W.C., Kovacevic S., Moran, L., and Sukaviriya, P. A Second-Generation Knowledge Base for the User Interface Design Environment. Technical Report GWU-IIST-91-13, Dept. of EE&CS, The George Washington University, Washington, D.C. 20052, 1991.

[12] Hayes, P., Szekely P., and Lerner R. Design Alternatives for User Interface Management Systems Based on Experience with COUSIN. In Proceedings of CHI'85, ACM, New York, 1985, pp. 169-175.

[13] Hurley,, W. D., and Sibert, J. L. Modeling User Interface-Application Interactions. IEEE Software, 6(1), January 1989, pp. 71-77.

[14] Hudson, S. and King R. Semantic Feedback in the Higgens UIMS. IEEE Transactions on Software Engineering 14(8), August 1988, pp.1188-1206.

[15] Johnson, J., and Richard J. B. Styles in Document Editing Systems. IEEE Computer, January 1988.

[16] Kovacevic, S. A Compositional Model of Human-Computer Dialogues. Technical Report GWU-IIST-90-36, Dept. of EE&CS, The George Washington University, Washington, D.C. 20052, 1990.

[17] Olsen, D. MIKE: The Menu Interaction Kontrol Environment. Transactions on Graphics 5(4), October 1986, pp. 318-344.

[18] Sibert, J., Hurley D., and Bleser T. Design and Implementation of an Object Oriented User Interface Management System. In R. Hartson and D. Hix (eds.), Advances in Human-Computer Interaction. Vol.2, Ablex, 1988, pp.175-213.

[19] Singh, and Green M. A High-Level User Interface Management System. In Proceedings of CHI'89 (Austin, Texas, April 30-May 4, 1989), ACM New York, 1989, pp. 133-138.

[20] Sukaviriya, P., and Foley J. Coupling a UI Framework with Automatic Generation of Context-Sensitive Animated Help. In Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology. Snowbird, Utah, Oct. 1990, pp. 152-166.

[21] Szekely, P. Standardizing the Interface Between Applications And UIMS's. In Proceeding of the ACM SIGGRAPH Symposium on User Interface Software and Technology, Williamsburg, Virginia, ACM, New York, 1989, pp. 34-42.

[22] Szekely, P. Template-Based Mapping of Application Data to Interactive Displays. In Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology. Snowbird, Utah, Oct. 1990, pp. 1-9.

[23] Szekely, P. Using Classification and Separation to Build Intelligent Interfaces. In Sullivan, J. and Tyler, S. (eds.), Architectures for Intelligent Interfaces: Elements and Prototypes. Addison-Wesley, 1991.

[24] Tatsukawa, K. Graphical Toolkit Approach to User Interaction Description. In Proceedings of CHI'91 (New Orleans, Lousiana, April 27-May 2, 1991), ACM New York, 1991, pp. 323-328.