1 User Interface Management Systems: Present and Future

Michel Beaudouin-Lafon

ABSTRACT This report presents the current state of the art, ongoing work, and future research directions in the domain of User Interface Management Systems (UIMS). We first define what a UIMS is, a task made difficult by the overloading of this term over the past ten years. We then present two major categories of UIMSs: those based on a language approach, and those dedicated to direct manipulation techniques. Finally, we present the active areas of research.

1.1 Defining a UIMS

1.1.1 A Bit of History

The term User Interface Management System (UIMS) is now 10 years old; it first publicly appeared in an article entitled "A User Interface Management System" [54], published by Kasik in the July 1982 issue of ACM Computer Graphics. It also appeared in the summary of the workshop on Graphical Input Interaction Techniques (GIIT), held in Seattle in June 1982 [83, page 7]. The term was coined after Data Base Management System (DBMS); in the same way a DBMS is a tool that helps to create and access a database, a UIMS should be a tool that helps to create and utilize user interfaces. The following sentence is quoted from GIIT [83, page 17]:

> "The analogy was drawn between the UIMS and a database management system. The DBMS frees the application programmer from detailed concern with the management of data storage and retrieval, and provides for the specific skills of the database designer and administrator to be applied to this task."

Buxton et al. [16] draws the following comparison:

> "The DBMS mediates between programmer and data, enforcing consistency of technique among all programmers in accessing that data. It provides portability because only the lowest-level DBMS routines are hardware-dependent. Similarly, the UIMS mediates between the applications programmer and the input events, encouraging a consistency both of graphical layout representation and of input processing mechanisms."

The notion of UIMS is older than the word itself: the Seillac II workshop on the Methodology of Interaction, held in 1979 [40], already evidenced the need for a specific component of an interactive application, in addition to the graphics system and the application program. Buxton et al. [15], states that the run-time support of their UIMS was put together in 1979. The idea of a UIMS became natural as soon as the interest turned from *graphical* applications to *interactive* applications. Most graphical applications at that time were CAD applications concerned primarily with graphical output, ie. visualization. In effect, the standards for graphical applications, such as GKS [46] had little support for managing complex user input. Basically, user input was always requested by the application in a conversational mode: the user was prompted to enter a value when that value was needed by the application.

Moving from graphical to interactive applications meant giving more control to the user, and recognizing that computer-human interaction is a two-way process. Hence, it is

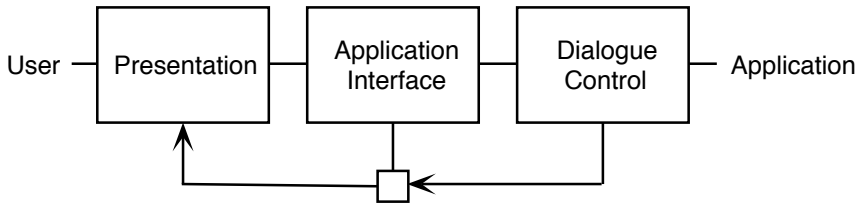

FIGURE 1.1. The Seeheim model, as presented by Green at Seeheim.

not surprising that the notion of UIMS emerged at a workshop on interaction techniques: issues like feedback for user actions and control of the application by the user are at the very heart of computer-human interaction.

Indeed the first people who tried to create interactive applications, in particular those who experimented with direct manipulation [78] were faced with difficult problems. Some were solved by introducing new primitives in the graphics systems (such as segment identification in GKS), but this was far from sufficient. It then became obvious that a new component of the application had to be devoted to managing the dialogue between the user and the application: this is known as the *Seeheim Model*.

1.1.2 The Seeheim Model

Seeheim, West Germany, was the site of a workshop on UIMS's held in November 1983. The results of this workshop were published in book form in 1985 [72]. The main contribution of this workshop was to define what is now known as the Seeheim Model. Until then, traditional graphical applications were built from a graphics system that ensured some degree of device independence, plus an application program that managed the display of objects, the user input and the application specific computation. Typical applications were CAD systems for mechanics, civil engineering, or VLSI.

The Seeheim workshop proposed a new model, which can be characterized by two principles:

- the application program shall be separated from the user interface;
- the user interface shall be decomposed into three components: the presentation component, the dialogue component, and the application interface component (see figure 1.1, from [37, page 9]).

The presentation component is responsible for the low-level input and output, and can be implemented with a graphics system. The application interface component appears because of the separation between the interface and the application; it provides the interface with a view of the application, and manages communication with the application. Finally the dialogue component manages the user dialogue and the mapping between user input and application commands on the one hand, application data and graphical objects on the other.

The three components of the Seeheim model were interpreted for some time as the lexical, syntactic and semantic components of a language analyzer. In effect, the presentation handles low-level aspects comparable to tokens (at least for user input), while the dialogue component ensures that proper commands are specified by a set of tokens, and the application interface describes the semantics of the application. This interpretation has been the motivation for using language description techniques to build user interfaces: we will describe a number of UIMSs which use such techniques. However, this interpretation falls short when trying to create direct manipulation applications. Hence, we can identify two trends in user interface construction: the language-based approach, which tries to describe

a user dialogue in terms of a (computer) language, and the approach which tries to use specific tools and techniques to describe the *whole* user interface, not just the dialogue.

1.1.3 What is a UIMS ?

Within the context we have just outlined, a UIMS is a tool, or a set of tools, which helps to specify, implement, test and maintain a user interface. However, it turns out that each author of a UIMS has their own definition of what a UIMS is (or should be). Instead of trying to synthesize a definition from these different authors, we will outline the definitions that were given in several working group reports.

According to the Graphical Input Interaction Techniques workshop [83, page 16]:

> "The role of a UIMS is to mediate the interaction between a user and an application; satisfying user requests for application actions, and application requests for data from the user. It accepts as input a dialogue specification, describing the detailed structure of the interaction."

The same workshop identifies the following advantages of constructing user interfaces with a UIMS:

- a UIMS improves the efficiency with which user interface designers can use their skills;
- a UIMS speeds the incremental process of creating a user interface;
- a UIMS makes it possible to create prototypes that can be discussed with the end user;
- a UIMS can adapt to different user profiles;
- a UIMS makes the integration of new application functionalities easier;
- a UIMS allows the application to be portable, while the user interface can be tailored to the particular environment;
- a UIMS can ease the debugging of interactive applications.

In his paper [54], Kasik defined the *architecture* of a UIMS as follows:

> "The user interface management system contains two components: a special purpose, application-independent dialogue specification language and a run-time interpreter that provides a number of interaction extensions not possible with procedure libraries."

The Seeheim workshop tried to identify models to be used for defining a UIMS. Kasik's UIMS uses a language-based model, but this is not a basic requirement. The most basic requirement is that the UIMS is a tool that takes as input a description of the user dialogue and produces an output used by the run-time component of the UIMS to control the execution of the interaction. In his definition, Kasik puts a slight restriction on the role of the UIMS in that it is only concerned with the dialogue. The Seeheim workshop extended this to the presentation and application interface components, although it recognized that the dialogue component models had the most highly developed notations [72, page 14]. Nevertheless a large number of UIMSs did take Kasik's approach instead of Seeheim's. This had the effect that the dialogue was considered to be modeled by a language, and it took a long time to realize that this was not appropriate. This had been quoted at Seeheim [72, page 6]:

> "The popular linguistic model, which distinguishes a lexical, a syntactic and a semantic level, is not considered appropriate for this purpose. Its main problem stems from the fact that in graphical interactions commands are not necessarily "entered in full" before feedback or action starts."

A more recent workshop gives a better idea about what a UIMS should be: the ACM SIGGRAPH Workshop on Software Tools for User Interface Management, held in November 1986, and reported in the April 1987 issue of ACM Computer Graphics, defines a UIMS as follows [8, page 73]:

> "A User Interface Management System (UIMS) is a tool (or tool set) designed to encourage interdisciplinary cooperation in the rapid development, tailoring and management (control) of the interaction in an application domain across varying devices, interaction techniques and user interface styles."

The domain of a UIMS is much wider in this definition. According to the report, a UIMS must provide a user interface with the following characteristics:

- consistency;
- support for a range of users from novice to expert; and
- support for tailorability and extensibility of the application.

It encourages:

- a consistent user interface between related applications;
- development and use of reusable software components;
- insulation of applications from the complexities of environments;
- shielding of applications from the effects of the end user actions; and
- support for ease of learning and use of applications.

A UIMS is now supposed to address the complete life-cycle of a user interface: requirements, specification and design, implementation, testing, and maintenance. In other words, a UIMS is a CASE tool (Computer-Aided Software Engineering tool) specialized for interactive applications. Applying a software engineering approach to user interfaces is a good idea. However, extending the role of a UIMS to the whole life cycle is questionable, since there are several life cycle models corresponding to at least the same number of design methodologies [9]. Moreover, this view of a UIMS is not of great utility since there does not exist, to our knowledge, any UIMS that really handles all of these aspects. For these reasons, we will restrict our definition of a UIMS in this report to the models and tools used to specify, implement and test the user interface of an interactive application. In a recent article, Myers [64] captures this notion by the term User Interface Development System (UIDS).

1.2 First Epoch: Dialogue Specification

The first UIMSs to be designed focused on the dialogue component of the Seeheim model. In effect, here lies the heart of user interface management: the presentation layer provides the dialogue layer with tokens which must be analyzed and grouped together to construct application commands. During this construction, some feedback must be sent back to the user.

At the beginning of the eighties, the state of the art of graphics systems with respect to user input did encourage viewing the user dialogue as a language. For instance, GKS [10] provides the client application with logical input devices: locator, valuator, choice, pick,

string, stroke. These logical devices are accessed in a request, sample or event mode. *Request mode* means that the user is asked to enter a value with a given device, while *sample mode* makes it possible to retrieve the current state of an input device, for instance, to track the position of a stylus on a tablet. *Event mode* means that the device output is stored as events in a queue so that the application can retrieve them later. GKS's event mode, however, is quite primitive and cannot be compared to events as used in modern window systems like the X Window System [75]: GKS events are triggered by another device, which puts a lot of responsibility on the application.

Given the assumption that the user was seen as an entity sending input tokens, and that the control of the dialogue was mostly in the application, the language-based model of dialogue management was the natural way to proceed. Three models have been particularly popular: context-free grammars, several kinds of finite state automata, and event-response languages.

The three main language-based models have been compared by Green in [39]. Other studies can be found in [50] and [72]. The following sections present these models, with a common example: the interactive specification of a polyline with the mouse, providing rubber-band feedback to the user.

1.2.1 Context-Free Grammars

Context-free grammars have been used for many years in language theory. Most, if not all programming languages have their syntax described by a context-free grammar. Because the area of programming languages was so crucial to computer science, context-free grammars have a very sound theoretical basis. A context-free grammar describes the syntax of a language by a set of *production rules*. The left hand of a production rule is a non-terminal, while the right hand can contain tokens or non-terminals. In particular, it is possible to use recursive rules, for instance, to specify a list of items of arbitrary length. The theory provides the necessary results to check that a given grammar is well-behaved, and that an analyzer can be built for it. Moreover, well-known techniques exist to generate such analyzers automatically. Under Unix, YACC [51] is such a generator of syntax analyzers.

Within the context of user interfaces, the tokens are user inputs, and the rules describe commands of the system. For instance, assuming the tokens BUTTON for a click and MOVE for a mouse move, the grammar for a command consisting of clicking, dragging the mouse, and clicking again can be described as follows:

line → BUTTON end-point
end-point → MOVE end-point
end-point → BUTTON

The rule for end-point is recursive, which is, unfortunately, not very intuitive compared to the description of the command. Moreover, such rules do not specify any feedback. For instance, to draw a rubber-band line, we need to add some actions to the rules (the actions are written between braces):

line → BUTTON {register start-position} end-point
end-point → MOVE {draw line from start-position to current position} end-point
end-point → BUTTON {trigger command Line(start-position, current position)}

This approach has several drawbacks:

- the production rules are difficult to write;
- the actions are even more difficult; and
- to work correctly, the feedback must be carried out at the time the tokens are produced by the user.

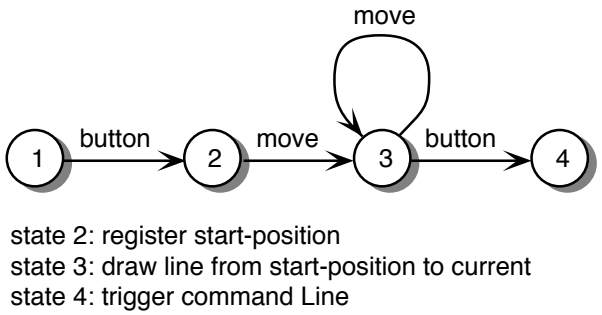

FIGURE 1.2. Specifying a rubberband with a finite state automaton.

The latter problem is the most important. It means that most of the technology developed for programming languages analyzers does not apply, because most parsers need some look-ahead tokens. The only parsers that work are descent parsers, which can analyze only a small subset of grammars called LL(1) grammars.

To describe the complete polyline example, we need the following grammar and actions:

polyline → BUTTON {register start-point} next-point {add end-point} polyline
polyline → BUTTON {trigger command Polyline(poly)}
next-point → MOVE next-point
next-point → BUTTON {register end-point}

It is not obvious from this specification that the polyline is finished by two successive clicks without intervening mouse moves. Moreover, this grammar is not suitable for a descent parser, although some transformation techniques could be used to make it an LL(1) grammar.

If we were to add a new token, say BACKSPACE, to allow backtracking in the specification of a polyline, the grammar would become unreadable: there would be many more rules, because a BACKSPACE can happen anywhere, and the actions to undo the polyline would be hard to specify.

Syngraph [69] is one of the first and the most well-known of UIMSs that used context-free grammars. Mike [70] also uses production rules, which are generated from a higher level description of the application commands and functions. Edge [55] provides a graphical notation for the production rules. The examples above reveal the limitations of this model, which nevertheless can still be used for the conversational parts of a user interface.

1.2.2 Finite State Automata

A finite state automaton is a logical machine composed of *states* and *transitions*. States are usually represented graphically by circles, and transitions by arrows between states. The machine works as follows: if it is in state s_1 and token t arrives, it goes into state s_2 if there is a transition labeled t from s_1 to s_2. There cannot be more than one transition leaving a state with the same label. If there is no transition, depending on the models, the token is ignored, or the machine goes into an error state.

Like context-free grammars, finite-state automata can only specify the valid sequences of input tokens. But the sequence of actions is explicit in an automaton, while it is implicit with a grammar. Feedback and commands still need to be handled by actions. There are two classes of automata with respect to actions. Traditional automata have their actions associated to a state; when that state is entered, the action is executed. Figure 1.2 shows the specification of the line example with this kind of automaton.

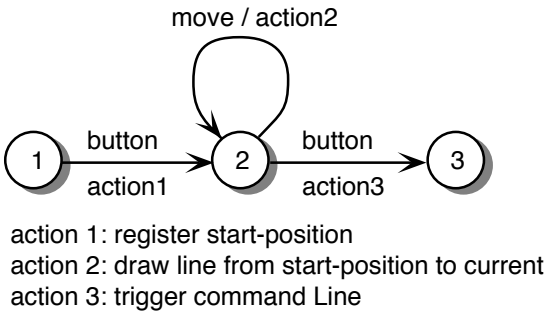

FIGURE 1.3. Specifying a rubberband with an ATN.

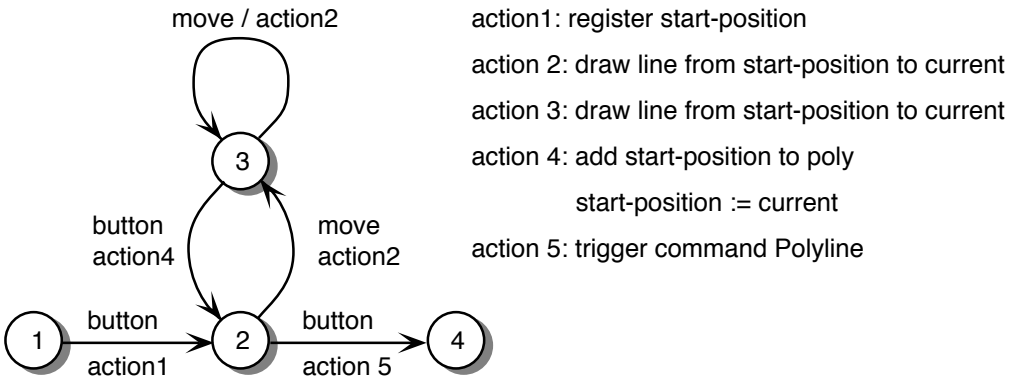

FIGURE 1.4. Specifying a polyline with an ATN.

More specialized automata, called Augmented Transition Networks (ATN) associate the actions with the transitions. They have been used in Rapid/USE [87]. Figure 1.3 describes the line creation example with an ATN, while figure 1.4 describes the polyline example. The transition can also have a predicate associated with the token: the transition can be traversed only if the predicate returns true. This makes it possible to specify more complex dialogues by augmenting the expressive power of the automata. For instance, the polyline example could be extended to include transitions with the BACKSPACE token. A predicate would use the number of points already entered to decide whether backspacing is allowed.

Another extension of finite state automata, Recursive Transition Networks (RTN) have been used [27]. The University of Alberta UIMS [38] also uses RTNs as a specification formalism, but they are translated into an event-based form. With such networks, a transition can be described by a separate network; if this network reaches a final state, the transition can be fired. The augmented expressiveness comes from the possibility of referencing a network from itself. This recursive aspect makes it possible to specify the backspace feature of the polyline example, without needing an external predicate as required by an ATN. Unfortunately, such recursive definitions are complex and can be difficult to understand.

Finally, Statecharts [41], which also provide sophisticated hierarchical automata, have been used for user interface specification in Statemaster [88].

All these automata share the same problem: the feedback to the user is specified in the actions, that is, independently of the automaton. When the application grows, the size of the automaton becomes difficult to manage, and proving it becomes impossible.

1.2.3 Event-Response Languages

Both context-free grammars and finite-state automata suppose that the set of valid input sequences has been predefined by the designer so that, in some sense, the user has no real control over the application. This is not compatible with the advent of event-driven applications and event-based graphics systems: in such systems, the activity of the user is not requested or sampled by the application or by the user interface, but instead, all user activity is reflected by events, stored in an event queue. It is the responsibility of the user interface to handle all of these events.

Event-response languages have been introduced to support event-driven applications. Unlike the other models presented in the previous sections, event-response languages have no theoretical background. This is a problem if their semantics are not precisely defined, which, unfortunately, is often the case.

An event handler is a process that can handle a number of event types. This process is described by a program which contains local variables and the different procedures to handle events of the different types. For instance, the program below is an event handler for the single line creation command:

```
EVENT HANDLER line;
    TOKEN Button, Move;
    VAR
    int state;
    point first, last;
    EVENT Button DO {
        IF state = 0 THEN
            first := current position;
            state := 1;
        ELSE
            last := current position;
            deactivate (self);
    }
    EVENT Move DO {
        IF state = 1 THEN
            draw line from first to current position
    }
    INIT
        state := 0;
END EVENT HANDLER line;
```

A complete program contains several event handlers, which can activate and deactivate each other. In the example above, the handler deactivates itself when the final click has been processed. Several event handlers can be active at the same time, and several active event handlers can process the same event in parallel. Because the data accessible by an event handler is local, there is no conflict. Thus, it is possible to describe multi-threaded dialogues. This is especially useful to handle several devices in parallel, as demonstrated by the Sassafras UIMS [44]. Event response languages have also been used in ALGEA [29]. They are well-adapted to modern graphics systems, which have an input model based on events. For instance, the University of Alberta UIMS [38] uses an event-based language internally, although the dialogue can be specified with RTNs.

An advantage of event handlers and the event model is that new event types can be defined. Event handlers can send synthesized events, which provides a means of communication between event handlers. This is useful in multi-threaded dialogues to coordinate

several event handlers. For instance, an event handler can send an event to indicate that he wishes exclusive access to a device, ie. that no other event handler should process events of a given type.

The expressive power of an event language depends on the language used in the procedures, which is typically a general purpose programming language. Although this looks good, this also means that in general it is impossible to prove anything on such a program. On the contrary, context-free grammars and automata make it possible to prove that a given sequence of tokens is or is not recognized. Some languages with well-defined semantics have been created to overcome this problem: Squeak [19], for instance, is dedicated to user interfaces; Esterel [7], on the other hand, is a general purpose reactive language, which has been used successfully for describing interactive objects [21]. Unfortunately, such efforts are more the exception than the rule.

1.3 Second Epoch: Direct Manipulation

The term "direct manipulation" was introduced by Shneiderman in 1983 [78] as a syntactic and semantic model for user interfaces that has the following properties:

- continuous representation of the objects of interest;
- physical actions instead of complex syntax;
- rapid, incremental, reversible operations whose impact on the object of interest is immediately visible; and
- layered or spiral approach to learning that permits usage with minimal knowledge.

Although the term was coined by Shneiderman, the concept of direct manipulation already existed and some commercial products such as Visicorp's Visicalc already used it. At the same time, Xerox PARC was developing the STAR [80], the first workstation with a graphical user interface, featuring direct manipulation through icons. In 1984 the Apple Macintosh came out, bringing the concept of direct manipulation to the non-specialist through three revolutionary applications: the Finder, MacPaint, and MacWrite.

Since then, direct manipulation has become the standard paradigm for creating natural interfaces. But developing direct manipulation interfaces is still a hard task: even on the Macintosh, most applications are developed directly on top of the Macintosh Toolbox [1]. There are very few tools to aid the development: MacApp, an application framework [77], and Prototyper [79], an interface generator, cannot be considered full-fledged UIMSs.

Although direct manipulation interfaces have been around for almost as long as the concept of UIMS, UIMSs supporting direct manipulation have not been developed until recently. The most likely reason is that direct manipulation breaks the "dialogue as a language" model that was the main motivation for creating UIMSs. Also, because direct manipulation interfaces require the presentation on the screen of application objects, a UIMS that supports direct manipulation must be able to describe the data of the application. Again, the main focus of UIMSs of the first period was on the dialogue, not on the application interface. Finally, direct manipulation works well only if there is a close feedback from user actions on the screen objects. This kind of feedback is often called *semantic feedback* because it is highly dependent on the application.

Semantic feedback breaks the Seeheim model and poses performance problems. Here is an example of semantic feedback: under the Macintosh Finder, when dragging an icon around, some icons highlight when the cursor passes over them while others do not. For instance, when dragging the icon of a document, the icon of the trash and the icons of the folders highlight. The feedback of the icon being dragged is a lexical feedback

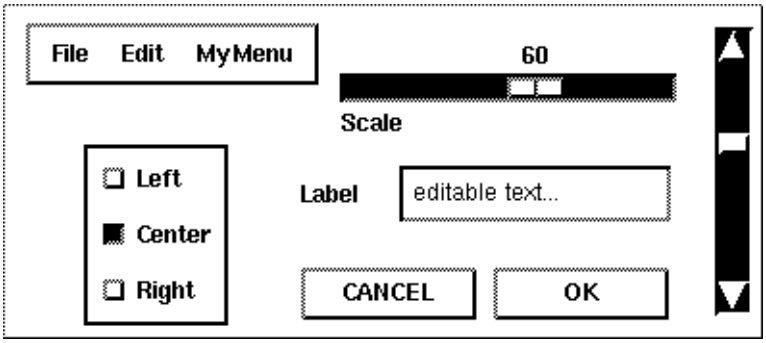

FIGURE 1.5. Sample MOTIF widgets.

similar to the echo of characters, but the highlighting of icons is a semantic feedback, because the presentation cannot guess which icons must highlight: this depends on the objects represented by the icon being dragged and the icons being highlighted, which are known only to the application. Hence semantic feedback breaks the Seeheim model because there is a close relationship between the presentation and the application. The dialogue component becomes mixed in with the presentation layer. Moreover, this poses performance problems because asking the application whether an icon must highlight each time the mouse changes position can be very expensive.

1.3.1 User Interface Toolkits

The first tools to help build direct manipulation applications were *user interface toolkits*. Some managers of the Macintosh Toolbox [1] (menu manager, control manager, dialogue manager), as well as the X Toolkit [62] and InterViews [59] illustrate what a toolkit is: a collection of object types that can be used by applications. Objects of the toolkit are called "widgets" in the X toolkit, "interactors" in InterViews; we call them *reactive objects*. They are defined by an aspect on the screen (their presentation), which can be parameterized by a set of attributes, and a behavior in response to user input. Figure 1.5 shows some MOTIF widgets: menus, buttons, scrollbars, etc.

Reactive objects correspond to the principles of direct manipulation. For instance, a scrollbar is a reactive object with an aspect on the screen, made of two arrows and a thumb. When the user clicks on the arrows, the thumb moves in one direction or the other; the user can also drag the thumb to a different position. A toolkit provides a means for notifying the application of the user's action on the reactive objects. This can be in the form of *callbacks* (most often), or *logical events* (more rarely).

Callbacks are a primitive way of communicating with the application: the application registers a function in a reactive object, to be called back (hence the name) by the interface when something interesting happens to the object, such as being activated by the user. This makes it possible for an application to monitor the manipulation of a scrollbar by a user.

Logical events are high level events sent by the reactive object themselves, instead of being sent by physical devices. For instance, a scrollbar can send events when it is scrolled up, down, or to a given position. The reactive object is then considered as a logical device, so that event handling techniques can be applied.

A typical toolkit contains a dozen different basic types of reactive objects: scrollbars, push and toggle buttons, menus, labels, editable texts, scrolled windows, confirm boxes, prompt boxes, etc. A toolkit also provides facilities to compose reactive objects, by means of container objects. Such containers can impose some constraints on the layout and behavior of their components; for instance, a container can ensure that its components

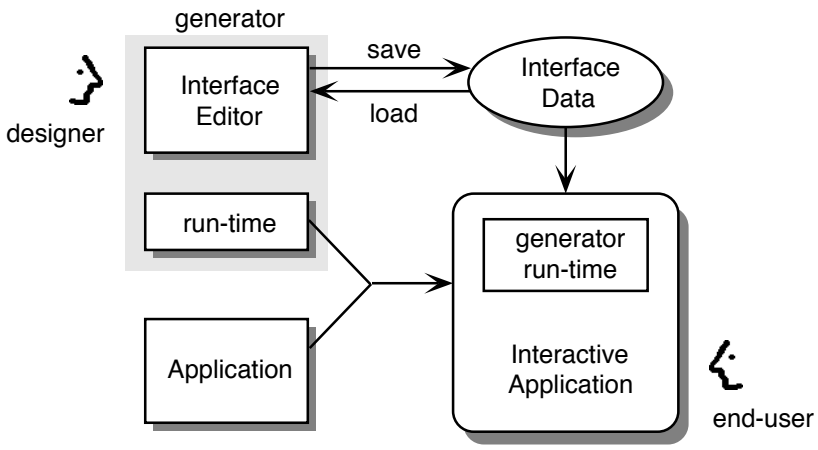

FIGURE 1.6. Interface generator: sample architecture.

be of the same size and aligned in rows.

An application that uses a toolkit has to create the reactive objects it needs, and to register the necessary callbacks. This works as long as the toolkit offers the reactive object types needed by the application. But if this is not the case, then the application has to go down one level and use the graphics layer directly. For instance, on the Macintosh, the toolbox does not offer any support for icons like those managed by the Finder. Hence, it is very difficult to create iconic applications on the Macintosh. Similarly, it is impossible to create a drawing tool with an X toolkit such as OSF/MOTIF [71] or Open Look [81] without using the underlying graphics system.

Toolkits are now a necessary, but insufficient, component of a user interface development environment; higher level components are also needed. The next two sections present two families of such components: interactive tools that help to create interfaces from the object classes found in a toolkit, and tools to create new reactive objects.

1.3.2 Interface Generators

Programming with a toolkit is quite tedious because it means writing programs that use a (usually large) library of procedures. This does not encourage modifying the interface. Interface generators have appeared for several years now: SOS Interface [48], Graffiti [6], and Dialogue Editor [18] were the first research prototypes. Many tools are now commercially available, such as UIMX by Visual Edge, or NeXT's Interface Builder.

Such tools are made of an interactive editor that makes it possible to "draw" the interface much like a drawing tool makes it possible to create drawings. Instead of drawing circles and rectangles, the editor of an interface generator draws reactive objects. Once the interface has been built with the editor, it is saved in a data file. The application can then load this file, which will create the interface at run-time. The interface can be tested from within the editor, saving alot of turnaround time. This process is illustrated in figure 1.6.

This approach is very satisfying because it uses the very principle of direct manipulation to create direct manipulation interfaces. However, specifying everything by direct manipulation can become a real challenge. FormsVBT [2] overcomes this problem by providing two editable views of the interface under construction: a graphical view and a textual view.

Nevertheless, the problems inherent in the toolkit approach still exist: you will not be able to create an interface for an application if the toolkit does not offer the reactive

objects you need. To compare with a drawing tool, you will not be able to draw circles if you only have rectangles and lines in the palette. Moreover, the interfaces constructed with a generator are *static*: there is usually no way to describe the dynamic modification of the interface at run-time, such as a menu containing one item for each opened window. These two problems leave user interface generators ill-adapted to the presentation of the application data, which is by nature domain specific and dynamic. On the other hand, these generators can be used for specifying the control part of the interface, ie. the set of menus, palettes and dialogue boxes that decorate the application data windows.

Another main weakness of interface generators stems from the fact that these tools are oriented towards the presentation part of the interface. Hence, the model for the application interface is usually very poor, mostly limited to callbacks. This does not encourage a clean separation between the interface and the application. Some interface generators such as Graffiti [6] and XFM [68] provide active values, a concept also investigated in Apogee [43]. Active values are shared data between the interface and the application. An active value can be bound to one or several presentation objects, but this mapping is not known to the application. This dramatically increases the independence between interface and application.

1.3.3 Architectural Models

Smalltalk was one of the first programming environments with a graphical user interface. All the components of the graphical user interface as well as the applications that may be created by the user are built on the Model-View-Controller model (MVC) [35].

Each reactive object is made of three components: the *view*, similar to the presentation, the *controller*, similar to the dialogue, and the *model*, similar to the application interface. Although MVC allows the model to directly communicate with its view, it is comparable to the Seeheim model. The new aspect is that the interface is not monolithic with one presentation component, one dialogue component and one application interface component, but instead made of a set of small MVC triplets.

Taking the example of the scrollbar again, the view would contain the necessary information to display the scrollbar and to capture user input; the model would contain three values: the low and high bounds of the scrollbar and its current value. Finally, the controller would ensure that the current value of the model corresponds to the position of the thumb.

The MVC model is a good framework for creating interactive applications where specific interactive objects are needed. However, the correct use of the MVC model requires much experience, although the existing classes provide a good starting point. No interactive system has been built to date for the creation of new reactive objects: the MVC model requires programming by hand.

Another model, called PAC (Presentation, Abstraction, Control), has been developed by Coutaz [23, 3]. PAC provides a framework to decompose a user interface into a set of *agents* that can communicate together in several ways. Unlike MVC, PAC is not bound to an implementation language. It is more a logical model that helps to identify, structure and design interactive applications.

Like MVC, PAC and other user interface tools and models are based on an object-oriented model which has long since been proven to be well-suited to user interface construction. The many variations of the object paradigm apply to the domain of user interfaces as well. Prototype-based approaches can be more appropriate than the usual class-based approach, as will be illustrated with the description of Garnet in the next section; actor-based system can be an implementation model for the agents of PAC; etc.

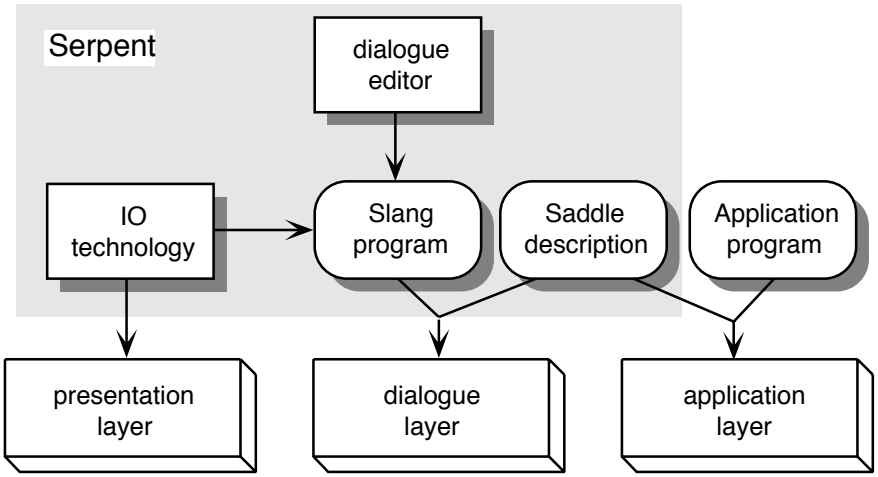

FIGURE 1.7. The architecture of Serpent.

1.4 Two Examples

Interface generators are now common tools, widely available on the commercial market. They do, however, solve only a small part of the problem of user interface construction, because they only deal with the part of the interface that can be represented by a set of predefined reactive objects. Many systems have been developed in the research community, and many are under development as well, to overcome this limitation and to address the construction of the whole interface. We now present two of them which are particularly significant and representative of the current state of the art.

1.4.1 Serpent

Serpent [76, 4] is a UIMS developed at the Software Engeneering Institute, Carnegie Mellon University. It features:

- A language designed for the specification of user interfaces (Slang);
- A language to define the interface between the application and Serpent (Saddle);
- An interactive editor for the specification of dialogues and for the construction and previewing of displays; and
- Provision to integrate new input/output subsystems.

The architecture of Serpent is based on the Seeheim model (see figure 1.7). The presentation layer is managed by an *IO technology*, ie. a collection of interaction objects visible to the user. The only IO technology currently available is a subset of the Athena or MOTIF Widgets (form, text, button, label), under the X toolkit. The dialogue layer is described by a mapping between application and presentation layers, by means of application-shared data. For instance, the dialogue tells the presentation the position of a button and its contents, and responds when the button is selected. The application layer consists of the description of the shared data, using a data declaration language called Saddle. An interactive dialogue editor makes it possible to build the presentation of the interface interactively, but most of the work has to be done by programming the dialogue in Slang.

The dialogue component is the most interesting part of Serpent. The dialogue is described by a set of *view controller* templates. A view controller is an object that maps application data into display objects. Presentation objects are described by a set of attributes

(e.g. position) and methods (e.g. notify). View controllers can be instantiated and deleted dynamically. They can be nested, and multi-threaded dialogues are possible. Instantiation and deletion of view controllers are triggered by conditions; these conditions are boolean expressions which can use attributes of objects and local variables, as well as state changes in the shared data, like the creation of a new application object. For instance, a condition can instantiate a view controller template when a new data is added to the shared data base. Conversely, the application has only to interact with this shared database to add, modify, and delete objects. When the shared database is changed as a consequence of a user action, the application is notified; this provides a means for sending commands to the application.

The original idea behind Serpent is the fact that it is centered around the application interface, represented by the shared database, instead of the dialogue, as is usually the case in UIMSs. The dynamic aspect of the description is also very powerful; by describing templates instead of particular instances, the user of Serpent can describe how objects are created and deleted. Most often, UIMSs require that the set of objects in the interface is statically defined, so that it is only possible to hide or show presentation objects to allow dynamic behavior.

On the other hand, the current IO technology used by Serpent is very limited, offering only form-based interaction; hence it is difficult to tell how well-suited it is, for instance, to create an iconic interface or a drawing tool. Although the mechanisms can clearly support this, it is not obvious how difficult it is to actually build such interfaces.

1.4.2 Garnet

Garnet [66, 32], is a User Interface Development Environment (UIDE) developed at Carnegie-Mellon University under the direction of B. Myers. The Garnet system can be separated into two parts: the Garnet toolkit and the Garnet tools (see figure 1.8). The Garnet toolkit is built on top of CommonLisp and the X Window System. It contains an object system, a constraint system and a graphics system. The Garnet tools include an interface builder, a dialogue box creation system and a spreadsheet. We are now going to present these components in more detail.

The heart of Garnet is an object-oriented system called KR built on top of CommonLisp. This system uses a prototype-based approach instead of the usual class-instance model. Instead of defining classes, a programmer creates *prototype* objects, defined as a set of untyped *slots*. The value of a slot can be any Lisp object, including a Lisp expression. New objects are created from existing prototypes. An object inherits the slots (ie. their name and value) of its prototype; it can override inherited slots and define new ones. The advantage of this model is that it provides inheritance of values; if the value of a slot is changed in a prototype, all objects that inherit this slot inherit the new value.

The object model of Garnet is complemented by a constraint system, which is heavily used throughout the system. When the value of a slot is an expression that references other slots (called dependents), the constraint system reevaluates the expression whenever the value of a dependent slot is changed. This is not a full-fledged constraint system, as these constraints are one-way relationships (they are actually called *formulas* in Garnet), whereas a real constraint system such as ThingLab [12] uses multiway constraints. The authors claim that one-way constraints are sufficient in virtually all cases for user interfaces. Indeed, one-way constraints are much easier to satisfy than general constraints; they require exactly one traversal of the dependency graph.

The originality of the graphics system of Garnet lies in its model for input handling. This model is based on six types of input techniques (called *interactors*): the menu interactor,

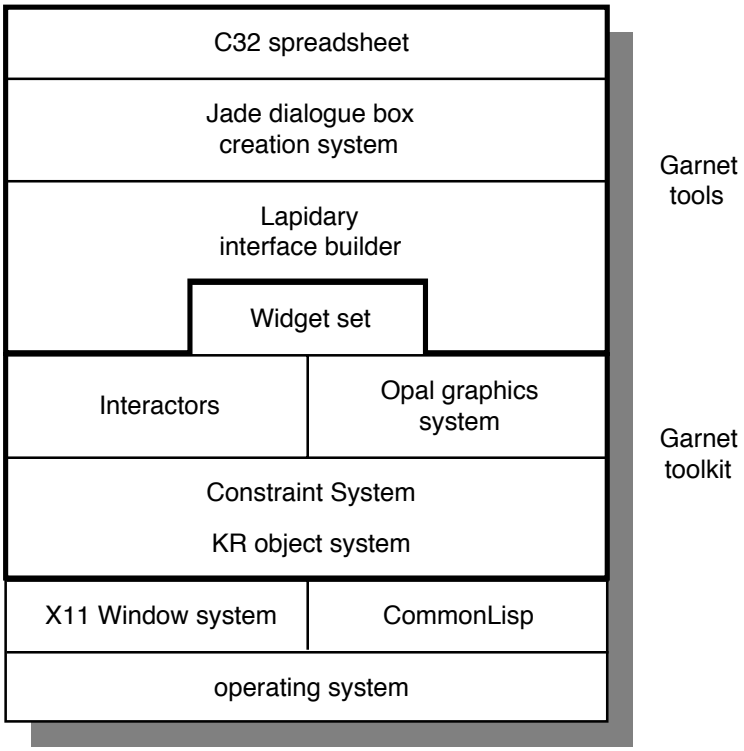

FIGURE 1.8. The architecture of Garnet.

the move-grow interactor, the new-point interactor, the angle interactor, the trace interactor, and the text interactor. These interaction techniques are independent of the actual graphical objects they use: a menu interactor can be used to pick a command in a menu as well as for radio buttons, graphical palettes, etc. The authors claim that these six interactors cover virtually all possible interaction styles in graphical user interfaces. This is probably true as long as one only considers keyboard and mouse input, but may fall short when considering new input devices like speech recognition, gesture input, eye tracking, etc. as well as multimodal input, ie. the combination of several input devices. Nevertheless, the approach of Garnet is much more open than most conventional toolkits, which tightly bind input techniques to graphical objects.

The first Garnet tool is *Lapidary*, an interface builder. It is similar to other interface builders in that it allows the user to create the interface interactively, by placing graphical objects and specifying their attributes. The differences come from the possibility to define constraints (formulas) between objects, and to bind input techniques to graphical objects freely. This is possible thanks to the underlying constraint system and input model.

The second Garnet tool is *Jade*, a system that automatically creates dialogue boxes from a simple textual description. This description contains the names and types of the fields of the dialogue box. Jade uses a database of presentation rules to decide where to place the fields and buttons, and which input technique to use according to a given look and feel. The obvious advantage over an interactive editor is that the description is look and feel independent, but this can only work if a given look and feel is specified in such a way that it can be described by a set of rules.

The last Garnet tool is *C32*, a browsing tool that uses a spreadsheet metaphor. Any Garnet object can be presented in C32 as a table displaying the names and values of the slots of that object. Whenever the object changes, the table is updated. Conversely, the user can edit the values interactively, with immediate effect on the object. C32 provides a set of facilities to edit formulas in a way similar to conventional spreadsheets: instead

of typing a slot name, one can click on the cell displaying that slot, or even click on the graphical object itself. C32 is a powerful debugging tool in the Garnet environment, in order to understand the web of dependencies defined by the thousands of formulas used throughout the system.

As a conclusion, Garnet probably is the most advanced UIDE to date. Nevertheless there are several problems and weaknesses to this approach. The system is oriented toward the presentation component of the interface, with some facilities for the dialogue component, but no tool is aimed at the definition of the application interface component. This component must be developed by hand, in CommonLisp, and chances are that it is intermixed with the other components.

More generally, one would like to give more abstract descriptions of the interface. Instead of this, the programmer is usually at the level of Lisp or the object system. Indeed, prototypes and constraints provide a powerful way to describe an interface, but higher level models would help even more.

1.5 Ongoing Work and Future Directions

The effort in creating tools to build user interfaces has considerably increased in the last few years. Among the many tools, systems, models and methods that exist, we try in this section to identify a number of current trends and promising directions. We have classified them in two categories. The first category concerns the ability to describe an interface in terms of what it should do rather than how it should do it. The second category concerns the widening of the domain of user interfaces with respect to interaction modes and techniques, and its influence on user interface construction.

1.5.1 From Imperative to Declarative Descriptions

An ever-present goal of computer science is to provide models and tools that make it possible to describe a system by its properties rather than by the way it works. Let us call the former a declarative description and the latter an imperative description. The domain of user interface construction does not escape from this rule. Three main directions are being investigated to that purpose.

The first direction concerns the use of *constraints*. A constraint is a relationship between a set of variables that must hold at all times. Whenever the value of a variable changes, a *constraint solver* must modify the values of other variables in order to satisfy all the constraints. Several techniques can be used for constraint satisfaction: numerical techniques, as used in Juno [67], propagation techniques as used in ThingLab [60], incremental techniques such as the DeltaBlue algorithm [31], or symbolic techniques as in Bertrand [57]. In all cases, the solver must handle the situations where there is no solution to the constraint system and where several solutions are acceptable. The latter can be handled by a notion of distance that is used to choose the solution closest to the current state, in order to achieve the least-astonishment principle: avoid surprising the user with the result of the resolution. The former situation (over-constrained system) can be handled by introducing a hierarchy of constraints [13]; higher priority constraints are satisfied first, while some lower priority constraints can be left unsatisfied.

Constraints have been used for a long time (ThingLab [11] is ten years old), but surprisingly, they have not been adopted as a standard technique for user interface specification. One reason might be the difficulty to implement efficient constraint solvers. Another reason is that the research has focused mainly on constraint solving, and current systems

lack a clean model with well-defined semantics. This is now being investigated, as with the Alien system [22] or the Kaleidoscope programming language [30]. Constraints could then prove to be a very general specification technique, not limited to graphical aspects. For instance, the application interface could be specified with constraints, by replacing active data [43] with constraints. Visual programming environment like Fabrik [49] could also benefit from constraint techniques.

Another direction toward declarative specifications is based on the programming by example paradigm, also called demonstrational interfaces. The principle is to create examples of the desired interface, and have the system infer the whole interface from these examples. Peridot [63] illustrates this approach: the user can create interaction techniques interactively. The system uses a knowledge base to infer the properties of the created objects. For instance, it is possible to build a scrollbar and have the system infer that the thumb must be contained in the scrollbar, and that the thumb can be dragged with the mouse, changing the value of the scrollbar.

Metamouse [61] is similar to Peridot, but applies to the definition of drawing tools. It can infer geometrical relationships between graphical objects. Therefore it is possible to "teach" the system how to align a set of objects along a line, for instance. Eager [25] is another example-based system. It works behind Hypercard [36], watching user actions until it discovers a pattern. At this point, it pops up and proposes to take over the task. While Peridot could be a part of a UIMS, Eager is clearly aimed at the end-user, while Metamouse sits in the middle.

Example-based interface construction is closely related to constraints [65]. These systems can be thought of as a layer on top of a constraint system, trying to recognize which constraints are satisfied, and instantiating these constraints if the user so wishes [53]. Thus, the limits of these systems are the same as any knowledge-based or rule-based system: they will never infer rules (or constraints) that are not in their database. Hence, the construction of the rule database becomes the main problem. In order to have the system "guess the right thing," the rules, and in some cases the inference engine, must be carefully adapted to the domain.

There is no doubt that demonstrational interfaces are going to be an active research area, with some spectacular applications. However, it is not clear whether the main applications will concern user interface construction. As suggested by the examples above, the end user could benefit more from such super-macros facilities.

1.5.2 New Needs, New Models, New Solutions

A decade of UIMSs have been built around the Seeheim model. But the context of user interfaces has changed significantly since the eighties. WIMP (Window-Icon-Menu-Pointing) interfaces are now generalized, because graphical workstations and PCs with mouse and keyboard are everywhere. The principle of direct manipulation has been applied in a large number of applications. The automatization of interface construction already has problems catching up with this evolution; most interactive applications on Unix workstations, Macintoshes and PCs are developed by hand. But the evolution still goes on, with new input devices (dataglove, eye tracking, etc...), new paradigms (computer supported cooperative work, virtual realities), and more computing power (50 Mips workstations today, 100 Mips tomorrow). How can we achieve the challenge of providing tools for user interface construction in such an evolving world ?

The answer lies in our ability to define models that accept a wide range of interactive techniques and application domains. A revision of the Seeheim model has been undertaken by the User Interface Developer's Workshop. The intermediate results of the workshop

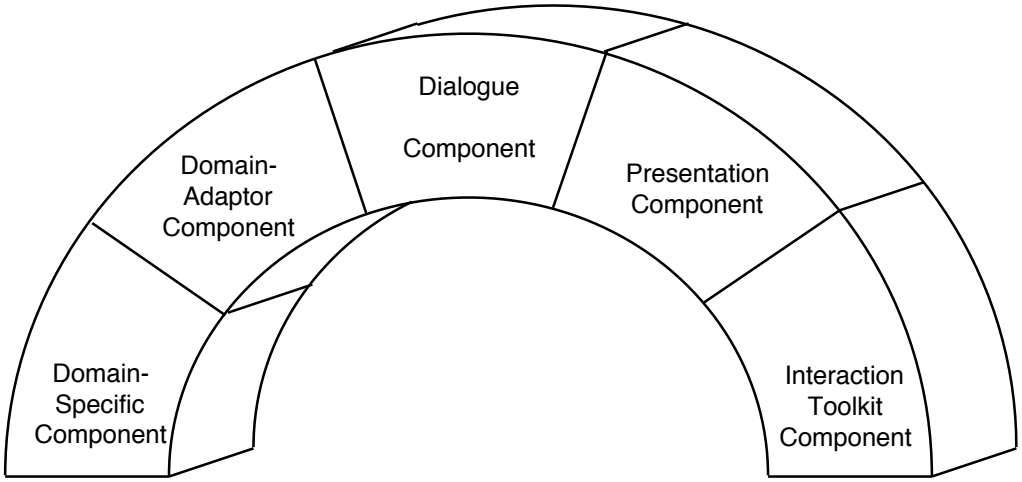

FIGURE 1.9. The Arch model – Seeheim revisited.

were presented at a SIG (Special Interest Group) meeting at the last ACM SIGCHI conference [84]. The workshop has defined a new model, called the Arch model, presented in figure 1.9.

The components of the model can be described as follows:

- *The Interaction Toolkit Component* implements the physical interaction with the user. It corresponds to a user interface toolkit such as OSF/Motif or OpenLook.
- *The Domain-Specific Component* implements the domain functionality, ie. what is called the "application" in the Seeheim model.
- *The Dialogue Component* is the keystone of the Arch model. It provides task-level sequencing, and mapping between domain-specific formalisms and UI-specific formalisms.
- *The Domain-Adaptor Component* implements domain related tasks required for human interaction, not present in the domain component.
- *The Presentation Component* provides a set of toolkit independent objects to the dialogue component.

This model is more realistic than the Seeheim model because it integrates the two components that always exist in an interactive application: the interaction toolkit component and the domain-specific component. But its main feature lies in the branching facility, illustrated in figure 1.10. The branching facility makes it possible to integrate several toolkit and domain specific components, thus providing extensibility.

Unfortunately, the branching facility has disappeared in more recent works of the User Interface Developer's Workshop [85]. In this article, the Arch model is presented as an instance of the *Slinky* metamodel. This metamodel features the five components that we have presented above, but does not impose a rigid allocation of functions among these components. The name, "Slinky", was chosen to emphasize this flexibility, as in the popular Slinky^{TM} toy. By changing the balance of functionality between the components, the Slinky metamodel produces a family of Arch models. The Slinky metamodel is different from other models, including Seeheim, PAC, MVC, in that it is evaluative rather than prescriptive. An evaluative model makes it possible to compare existing systems and leads to a better understanding of the architectures that are actually used. On the other hand, the purpose of evaluation does not justify the branching facility that was present in the first Arch model, because such branching is not representative of existing applications.

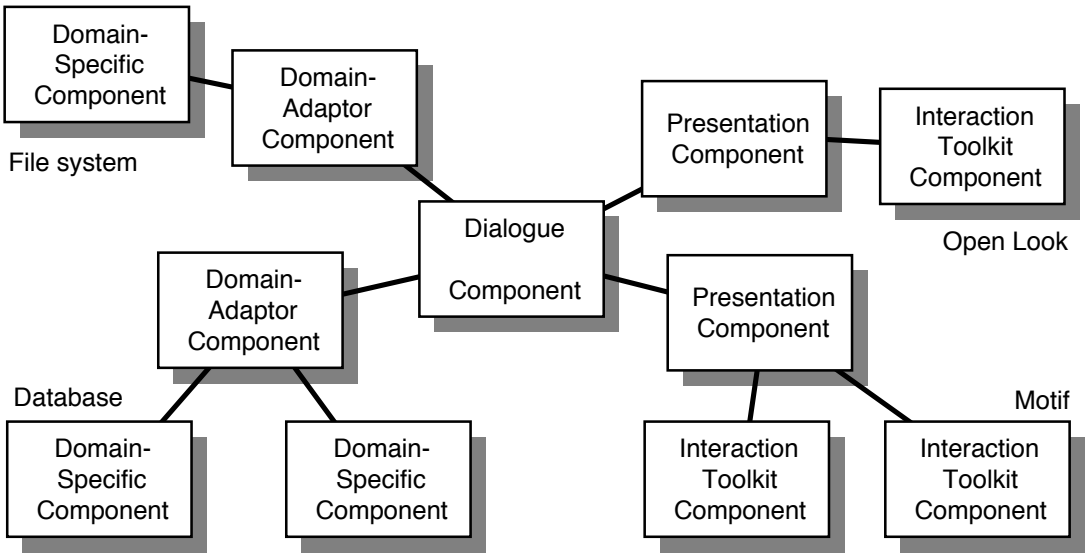

FIGURE 1.10. Branching facilities in the Arch model.

Nevertheless, the Arch model provides a useful framework for classifying ongoing research in the domain of user interface construction. A first important direction concerns the growing interest in specifying the domain-adaptor component of the interface. We have already seen that Serpent provides some support for specifying the application interface in terms of an active database. The Higgens UIMS [47] uses attribute techniques, which have long been used in compiler construction, in order to describe the semantics of the application. In particular, incremental evaluation techniques are available. The attributed graphs used by Higgens can be thought of as a generic dialogue-adaptor component. The classification of Coutaz [24] is also relevant to the domain-adaptor component as it gives some insights into the way data are exchanged between application and interface.

Humanoid [82] also investigates the application interface, by using templates that map application data to presentation objects. Hence it lies more in the center of the Arch model, with an implicit dialogue component, a presentation component and a generic domain-adaptor component. This approach also appears in other systems. For instance, Hypercard [36] provides an environment to develop applications based on the model of stacks of cards. WISh [5] provides an iconic model based on a client-server architecture that makes it possible to build iconic interfaces that integrate several applications. Unidraw [86] is a framework for building domain-specific graphical editors. TGE [52] is a generic editor for applications that use graphs or trees to represent their data. Although these systems address very different domains, they all provide similar services, ie. a generic interface for applications in the form of a domain-adaptor component, and they implement the other components of the Arch model. If they could share the same dialogue component, then it would be possible to combine them according to the branching facility of the Arch model.

1.5.3 New Dimensions of Interaction

A common characteristic of the systems we have presented so far is that they deal only with graphical interfaces that use a display and a pointing device, and that interact with a single user. Other dimensions of user interaction deserve more attention, because they represent important directions for the future of user interfaces. Let us present some of them.

Animated interfaces bring animation to the user interface. Animations are already used for the purpose of interaction: for instance, when an icon is double-clicked on the Macintosh Finder, a rectangle grows from that icon to the window being opened. However, with the exception of Animus [26] and Whizz [20], tools for defining animated interfaces have not been investigated very much. Most animation systems only address visualization, for instance algorithm animation [14], whereas animation could be used efficiently as a medium by itself. Robertson et al. [74] assess that an animated display is essential to the understanding of the system response by the user. Adding audio output to interfaces, which to some extent can be considered a problem similar to animation, is known to be effective [33]. Again, as yet no tool exists to include auditory feedback in a user interface.

Multi-user interfaces [56], also known as groupware and CSCW (Computer-Supported Cooperative Work), are systems that allow a group of users to work on a common task in a shared environment. The field of CSCW is becoming a domain of active research, but very few tools exist to develop such systems (see for instance Liza [34]). This domain poses new and unexpected problems to user interface construction: if a document is to be edited by several users at the same time, each user must be aware of what the other users are doing. This raises the issue of feedback of other users actions, which is very different from feedback of the user's own actions. As an example, the Grove editor [28] uses a number of interesting ideas in this domain: text turns from blue to black as it gets aged, and "clouds" indicate the areas of the document where other users are working. Unfortunately, generalizing such techniques and making them available in a UIMS remains to be seen.

Multimodal interfaces have the ability to manage several input channels at the same time. Among others, the Sassafras UIMS [45] can handle multimodal input in the case where the devices are independent: for instance a mouse controls the brush of a painting tool while a slider controls the width of the brush. But composing the input of several input devices in order to specify one application command is by far more complex. For instance, saying the sentence *"put that there"* while the user specifies the object and the location with mouse clicks requires an input and dialogue model which is far beyond the capabilities of today's UIMS models. The difficulty lies in the semantic information which is necessary to interpret the input data, and the real-time feedback that is required. In the example above, the first mouse click must be mapped to an object while the second mouse click specifies a location.

Such problems are even more complex in virtual worlds environments, where a user can potentially use gesture input, eye-tracking, voice recognition, etc. It is encouraging to see that UIMSs are being developed for such systems. The UIMS presented in [58] uses a rule-based kernel, and separate processes to handle low-level input (the system currently uses 7 workstations and 2 PCs). It features several logical levels so that the mapping between input devices and application commands can be redefined transparently, even at run-time.

1.6 Conclusion: The Future of UIMSs

The goal of creating a UIMS that supports the whole life-cycle of user interface development is still ahead. The engineering of user interfaces is at its very beginning; integrating state-of-the-art techniques for software engineering into human-computer interface engineering is one of the challenges of future UIMSs. Most efforts are currently focused on the implementation and maintenance phases, but specification and test are going to be of particular importance. It is worth noting that these problems are not specific to user interface construction, but apply to software development in general.

The specification of (some parts of) an interface has been addressed by different models, some of which we have described. Unfortunately, the models that are well-suited are generally not formally defined. For instance, most event-response languages have no formal semantics. As a consequence, interfaces cannot be proved, they cannot be tested automatically, and they cannot be reused easily. This will not be overcome until formal models and techniques are defined and widely used.

In addition to traditional software engineering issues, the engineering of user interfaces raises human factors issues that are hard to integrate into a design methodology [17]. For instance, most systems fail to represent the user's task model. This has a tremendous impact on the quality of user interfaces, such as the lack of undoing facilities, context-sensitive help, macros, etc. More and more, the user's model is taken into account when specifying an interface, through techniques like rapid prototyping, scenarios, etc. But the representation and use of this model in the interface itself is far more difficult, because it is not directly reflected in the user's input, which is the only perception of the user by the system.

User interface engineering is subject to a strong "technology pull"; this is not likely to change in the next few years. A positive consequence is the overall effort dedicated to the development of interfaces and the development of tools for building interfaces. A more questionable consequence is the current standardization process; a user interface standard could jeopardize the possibility to create a new generation of UIMSs, in a similar way as GKS made it difficult to create direct manipulation interfaces and multi-threaded dialogues.

We expect future UIMSs to consist of a set of specific tools that address the different aspects of user interface construction, rather than big systems that cover the whole development process. These tools will use different techniques, ranging from traditional programming using libraries to interactive construction and declarative specification. To be effective, these tools will need to be open and interoperable so that they can be integrated into a user interface development environment.

Acknowledgements:

I want to thank Len Bass and Joëlle Coutaz, and the other members of the IFIP WG2.7 working group, for clarifying a number of concepts and ideas. Stéphane Chatty and Solange Karsenty provided useful comments and fruitful discussions during the preparation of this article. I am also grateful to Chris Weikart and Heather Sacco for proofreading the article.

1.7 References

[1] Apple Computer Inc. *Inside Macintosh*. Addison-Wesley, 1985.

[2] G. Avrahami, K. P. Brooks, and M. H. Brown. A two-view approach to constructing user interfaces. In *Proc. ACM SIGGRAPH*, pages 137–146, July 1989.

[3] L. Bass and J. Coutaz. *Developing Software for the User Interface*. The SEI Series in Software Engineering. Addison-Wesley, 1991.

[4] L. Bass, E. Hardy, R. Little, and R. Seacord. Incremental development of user interfaces. In G. Cockton, editor, *Engineering for Human-Computer Interaction*, pages 155–173. North-Holland, 1990.

[5] M. Beaudouin-Lafon. User interface support for the integration of software tools: an iconic model of interaction. In *Proc. ACM Symposium on Software Development Environments (SIGSOFT)*, pages 187–196, Nov. 1988.

[6] M. Beaudouin-Lafon and S. Karsenty. A framework for man-machine interface design. In *Proc. European Unix Users Group Conference (EUUG)*, pages 1–10, Sept. 1987.

[7] G. Berry, P. Couronné, and G. Gonthier. The Esterel synchronous programming language and its mathematical semantics. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency, LNCS 197*, pages 389–448. Springer-Verlag, 1985.

[8] B. Betts et al. Goals and objectives for user interface software. *ACM Computer Graphics*, 21(2):73–78, Apr. 1987.

[9] B. Boehm. A spiral approach to software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[10] P. R. Bono, J. L. Encarnação, F. R. A. Hopgood, and P. J. W. ten Hagen. GKS – the first graphics standard. *IEEE Computer Graphics and Applications*, July 82.

[11] A. Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, Oct. 1981.

[12] A. Borning and R. Duisberg. Constraint-based tools for building user interfaces. *ACM Transactions on Graphics*, 5(4):345–374, Oct. 1986.

[13] A. Borning, R. Duisberg, B. Freeman-Benson, A. Kramer, and M. Woolf. Constraint hierarchies. In *Proc. ACM OOPSLA*, pages 48–60, Oct. 1987.

[14] M. H. Brown. *Algorithm Animation*. PhD thesis, Brown University, 1987.

[15] W. Buxton. Lexical and pragmatic considerations of input structure. *ACM Computer Graphics*, pages 31–37, Jan. 1983.

[16] W. Buxton, M. R. Lamb, D. Sherman, and K. C. Smith. Towards a comprehensive user interface management system. *ACM Computer Graphics - Proc. SIGGRAPH*, 17:35–42, July 1983.

[17] S. K. Card, T. P. Moran, and A. Newell. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, 1983.

[18] L. Cardelli. Building user interfaces by direct manipulation. In *Proc. ACM Symposium on User Interface Software and Technology (UIST)*, pages 151–166, Oct. 1989.

[19] L. Cardelli and R. Pike. Squeak: a language for communicating with mice. *ACM Computer Graphics - Proc. SIGGRAPH*, 19(3):199–204, 1985.

[20] S. Chatty. Integrating animation with user interfaces. In J. Larson and C. Unger, editors, *Engineering for Human-Computer Interaction*. North-Holland, 1992. Proc. IFIP WG2.7 Working Conference, Ellivuori, Finland, August 1992.

[21] D. Clément and J. Incerpi. Specifying the behavior of graphical objects using Esterel. Research Report 836, INRIA, France, Apr. 1988.

[22] E. Cournarie and M. Beaudouin-Lafon. Alien: a prototype-based constraint system. In *Second Eurographics Workshop on Object Oriented Graphics*, pages 93–114, June 1991.

[23] J. Coutaz. PAC, an implementation model for dialog design. In *Proc. Interact'87*, pages 431–436, Sept. 1987.

[24] J. Coutaz and S. Balbo. Applications: A dimension space for user interface management systems. In *Proc. ACM SIGCHI*, pages 27–32, May 1991.

[25] A. Cypher. Programming repetitive tasks by example. In *Proc. ACM SIGCHI*, pages 33–39, May 1991.

[26] R. A. Duisberg. Animated graphical interfaces using temporal constraints. In *Proc. ACM SIGCHI*, pages 131–136, Apr. 1986.

[27] E. A. Edmonds. Adaptive man-computer interfaces. In M. J. Coombs and J. L. Alty, editors, *Computing Skills and the User Interface*. Academic-Press, 1981.

[28] C. Ellis, S. Gibbs, and G. Rein. Groupware, some issues and experiences. *Comm. ACM*, 34(1):38–58, Jan. 1991.

[29] M. A. Flecchia and R. D. Bergeron. Specifying complex dialogues in ALGEA. In *Proc. ACM CHI+GI*, pages 229–234, 1987.

[30] B. N. Freeman-Benson. Kaleidoscope : Mixing constraints, objects and imperative programming. In *Proc. ECOOP-OOPSLA*, pages 77–87, 1990.

[31] B. N. Freeman-Benson, J. Maloney, and A. Borning. An incremental constraint solver. *Comm. ACM*, 33(1):54–63, Jan. 1990.

[32] The Garnet compendium: Collected papers, 1989–1990. Technical Report CMU-CS-90-154, School of Computer Science, Carnegie Mellon University, Nov. 1990. Edited by Brad A. Myers.

[33] W. W. Gaver, R. B. Smith, and T. O'Shea. Effective sounds in complex systems: the Arkola simulation. In *Proc. ACM SIGCHI*, pages 85–90, May 1991.

[34] S. J. Gibbs. LIZA: An extensible groupware toolkit. In *Proc. ACM SIGCHI*, pages 29–35, May 1989.

[35] A. Goldberg. *Smalltalk-80 – The Interactive Programming Environment*. Addison-Wesley, 1984.

[36] D. Goodman. *The Complete Hypercard Handbook*. Bantam Books, 1987.

[37] M. Green. Report on dialogue specification tools. In G. E. Pfaff, editor, *User Interface Management Systems*, Eurographics Seminars, pages 9–20. Springer-Verlag, 1985.

[38] M. Green. The University of Alberta UIMS. *ACM Computer Graphics - Proc. SIGGRAPH*, 19(3):205–213, 1985.

[39] M. Green. A survey of three dialogue models. *ACM Trans. on Graphics*, 5(3):244–275, July 1986.

[40] R. A. Guedj et al., editors. *Methodology of Interaction*. North-Holland, 1980.

[41] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[42] H. R. Hartson and D. Hix. Human-computer interface development: Concepts and systems for its management. *ACM Computing Surveys*, 21:5–92, 1989.

[43] T. R. Henry and S. E. Hudson. Using active data in a UIMS. In *Proc. ACM Symposium on User Interface Software and Technology (UIST)*, pages 167–178, Oct. 1989.

[44] R. D. Hill. Supporting concurrency, communication and synchronization in human-computer interaction – the Sassafras UIMS. *ACM Trans. on Graphics*, 5(3):179–210, July 1986.

[45] R. D. Hill. Event-response systems: A technique for specifying multi-threaded dialogues. In *Proc. ACM CHI+GI*, pages 241–248, 1987.

[46] F. R. R. Hopgood, D. A. Duce, J. R. Gallop, and D. C. Sutcliffe. *Introduction to the Graphical Kernel System, GKS*. Number 19 in Apic Studies in Data Processing. Academic-Press, 1983.

[47] S. E. Hudson and R. King. Semantic feedback in the Higgens UIMS. *IEEE Trans. on Software Engineering*, 14(8):1188–1206, Aug. 1988.

[48] J.-M. Hullot. SOS Interface, un générateur d'interfaces homme-machine. In AFCET Informatique, editor, *Actes des Journées Langages Orientés Objet*, pages 69–78, 1986.

[49] D. Ingalls, S. Wallace, Y.-Y. Chow, F. Ludolph, and K. Doyle. Fabrik: A visual programming environment. In *Proc. ACM OOPSLA*, pages 176–190, Sept. 1988.

[50] R. J. Jacob. Using formal specifications in the design of a human-computer interface. *Comm. ACM*, 26(4):259–264, Apr. 1983.

[51] S. C. Johnson. Language development tools on the Unix system. *IEEE Computer*, 13(8), Aug. 1980.

[52] A. Karrer and W. Scacchi. Requirements for an extensible object-oriented tree/graph editor. In *Proc. ACM Symposium on User Interface Software and Technology (UIST)*, pages 84–91, Oct. 1990.

[53] S. Karsenty, J. A. Landay, and C. Weikart. Inferring graphical constraints with Rockit. Technical Report #17, Digital Equipment Corporation, Paris Research Laboratory, Mar. 1992.

[54] D. J. Kasik. A user interface management system. *ACM Computer Graphics*, 16(3):99–105, July 1982.

[55] M. F. Kleyn and I. Chatravarty. EDGE – a graph based tool for specifying interaction. In *Proc. ACM Symposium on User Interface Software and Technology (UIST)*, pages 1–14, Oct. 1989.

[56] K. L. Kraemer and J. C. King. Computer-based systems for cooperative work and group decision making. *ACM Computing Surveys*, 20(2):115–145, June 1988.

[57] W. Leler. *Constraint Programming Languages*. Addison-Wesley, 1987.

[58] J. B. Lewis, L. Koved, and D. T. Ling. Dialogue structures for virtual worlds. In *Proc. ACM SIGCHI*, pages 131–136, May 1991.

[59] M. A. Linton, J. M. Vlissides, and P. R. Calder. Composing user interfaces with InterViews. *IEEE Computer*, pages 8–22, Feb. 1989.

[60] J. H. Maloney, A. Borning, and B. N. Freeman-Benson. Constraint technology for user-interface construction in ThingLab II. In *Proc. ACM OOPSLA*, pages 381–388, Oct. 1989.

[61] D. L. Maulsby, I. H. Witten, and K. A. Kittlitz. Metamouse: Specifying graphical procedures by example. In *Proc. ACM SIGGRAPH*, pages 127–136, July 1989.

[62] J. McCormack and P. Asente. An overview of the X Toolkit. In *Proc. ACM Symposium on User Interface Software and Technology (UIST)*, pages 46–55, Oct. 1989.

[63] B. A. Myers. Creating interaction techniques by demonstration. *IEEE Computer Graphics and Applications*, 1987 1987.

[64] B. A. Myers. User-interface tools: Introduction and survey. *IEEE Software*, pages 15–23, Jan. 1989.

[65] B. A. Myers. Creating user interfaces using programming by example, visual programming and constraints. *ACM Trans. on Programming Languages and Systems*, 12(2):143–177, 1990.

[66] B. A. Myers et al. Garnet, comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer*, pages 71–85, Nov. 1990.

[67] G. Nelson. Juno, a constraint-based graphics system. *ACM Computer Graphics - Proc. SIGGRAPH*, 19(3):235–243, July 1985.

[68] NSL. *XFM2 User's Manual*. N.S.L., 57–59 rue Lhomond, 75 005 Paris, France, 1990.

[69] D. R. Olsen, Jr. SYNGRAPH: a graphical user interface generator. *ACM Computer Graphics - Proc. SIGGRAPH*, 17(3):43–50, July 1983.

[70] D. R. Olsen, Jr. MIKE: the menu interaction kontrol environment. *ACM Trans. on Graphics*, 5(4):318–344, Oct. 1986.

[71] Open Software Foundation, Cambridge. *OSF/Motif Programmer's Reference Manual*, 1989.

[72] G. E. Pfaff, editor. *User Interface Management Systems*. Eurographics Seminars. Springer-Verlag, 1985.

[73] M. Prime. User interface management systems – a current product review. *Computer Graphics Forum*, 9, 1990.

[74] G. G. Robertson, J. D. Mackinlay, and S. K. Card. Cone trees: Animated 3D visualizations of hierarchical information. In *Proc. ACM SIGCHI*, pages 189–194, May 1991.

[75] R. W. Scheifler and J. Gettys. The X Window System. *ACM Trans. on Graphics*, 5(2):79–109, Apr. 1986.

[76] Serpent overview. Technical Report CMU/SEI-89-UG-2, Carnegie Mellon University, Software Engineering Institute, Aug. 1989.

[77] K. J. Shmucker. MacApp: An application framework. *Byte Magazine*, pages 189–192, Aug. 1986.

[78] B. Shneiderman. Direct manipulation: a step beyond programming languages. *IEEE Computer*, pages 57–69, Aug. 1983.

[79] SmetherBarnes. *SmetherBarnes Prototyper User's Manual*. P.O. Box 639, Portland, Oregon, 87.

[80] D. C. Smith, C. Irby, R. Kimball, B. Verplank, and E. Harslem. Designing the Star user interface. *Byte Magazine*, 7(5), Apr. 1984.

[81] Sun. *OLIT Specifications*. Sun Microsystems, Mountain View, 1990.

[82] P. A. Szekely. Template-based mapping of application data to interactive displays. In *Proc. ACM Symposium on User Interface Software and Technology (UIST)*, pages 1–9, Oct. 1990.

[83] J. J. Thomas et al. Graphical interaction technique (GIIT) – workshop summary. *ACM Computer Graphics*, pages 5–30, Jan. 1983.

[84] User Interface Developer's Workshop. The Arch model: Seeheim revisited. Presented at ACM SIGCHI, Apr. 1991.

[85] User Interface Developer's Workshop. A metamodel for the runtime architecture of an interactive system. *ACM SIGCHI Bulletin*, 24(1), Jan. 1992.

[86] J. M. Vlissides and M. A. Linton. Unidraw: A framework for building domain-specific graphical editors. Technical Report CSL-TR-89-380, Computer Systems Laboratory, Stanford University, July 1989.

[87] A. I. Wasserman. Extending state transition diagrams for the specification of human-computer interaction. *IEEE Trans. on Software Engineering*, SE-11:699–713, Aug. 1985.

[88] P. D. Wellner. Statemaster: A UIMS based on Statecharts for prototyping and target application. In *Proc. ACM SIGCHI*, pages 177–182, May 1989.