Interfaces as Specifications

In the MIDAS User Interface Development System

Regina H.B.Cabral (1)
Ivan M. Campos (2)
Donald D. Cowan (3)
Carlos J.P.Lucena (4)

Abstract

This paper describes an evolving User Interface Development System called MIDAS (for Merging Interface Development with Application Specification) which allows interface/systems designers to develop an application-specific user interface interactively, in a prototyping-oriented environment, while refining the specification of the intended application itself. The interface/systems designer receives expert advice on both interface and application software design principles, emerging from MIDAS' knowledge base, and can also animate the intended user dialogue with the interface being designed via an extensive set of visual programming aids. The generated interface can be further customized by the end-user, by flexibly altering the default appearance of the dialogue scenarios. Furthermore, the application-specific end-user interface is also knowledge based. Its domain knowledge covers user modeling and the application domain, in order to adapt itself dynamically to different degrees of user familiarity with the application, from novice to expert. Both the interface code and the programming-in-the-large of the application code are developed within an object-oriented framework. A proposal for a software life cycle model based on the rapid prototyping of user interfaces as a means to refining the specification of the application all the way down to the import-export list and module semantics specification for each and every application module is also presented. The lifecycle model is rule-encoded in MIDAS' knowledge base. The interface/systems designer is guided by the interpretation of those rules. MIDAS aims to provide a testbed for new ideas in human-computer interfaces, knowledge-based support of design activities and life cycle models based on rapid prototyping of user interfaces.

Keywords:

Prototyping, Lifecycle Model, User Interfaces, Specifications, User Interface Management Systems, User Interface Development Systems, Object-Oriented Design and Development, Knowledge Bases, Expert Assistance, User Models, Direct Manipulation Interfaces.

1. Introduction

Making computers easy to use is a major problem of obvious importance. In this paper a user interface is taken as any computer software whose primary function is to provide support and assistance in the use of some other software system, called the application. Human-computer interfaces will continue to share the trend towards both less code writing and more automatic code generation. At the same time, software developers have come to realize that the user interface paradigm is itself a kind of specification notation that expresses the user's intent and desires in terms of images, as opposed to words. The user interface implicitly defines most of the functional requirements, i.e., specifying the user interface often suffices to obtain an

---

1) Federal University of Minas Gerais, Brazil; e-mail: rhbc@lncc.bitnet. Supported by grant from CNPq (RHAE).

2) Federal University of Minas Gerais, Brazil; e-mail: imc@lncc.bitnet. Supported by grants from IBM Brazil and SID Informatica (ESTRA Project).

3) The University of Waterloo, Canada; e-mail: dcowan@watcsg.uwaterloo.ca. Supported by grants from Bell Canada and NSERC.

4) Catholic University of Rio de Janeiro, Brazil; e-mail: cjpl@lncc.bitnet. Supported by grants from CNPq (RHAE) and SID Informatica (ESTRA Project).

almost complete system specification. This is especially true for highly interactive applications.

Highly interactive applications can be classified from the perspective of both the user interface (UI) and the application.

Communication between the UI and the application can proceed in one of two directions, either the UI controls the communication or the application controls the interaction. Communication between the UI and the application could also be in a hybrid form where the UI and the application are in control at different times during the operation.

If menus and dialogue boxes and similar artifacts are the mode of interaction between the UI and the application, then this communication can be viewed in a number of different ways. Basically all modes of interaction could be made available to the user at any time during the operation of the application, that is so-called modeless operation. This "natural" mode of operation is described in more detail in [SCHM86]. An alternative approach would be to restrict the modes of operation. For example, the user could be limited to accessing menus in an hierarchical manner, where a menu in another part of the tree could only be examined by traversing the appropriate portions of the tree and thus also traversing several layers of menu code. The relationship between the application and the UI and so-called modeless operation is shown in Figure 1.

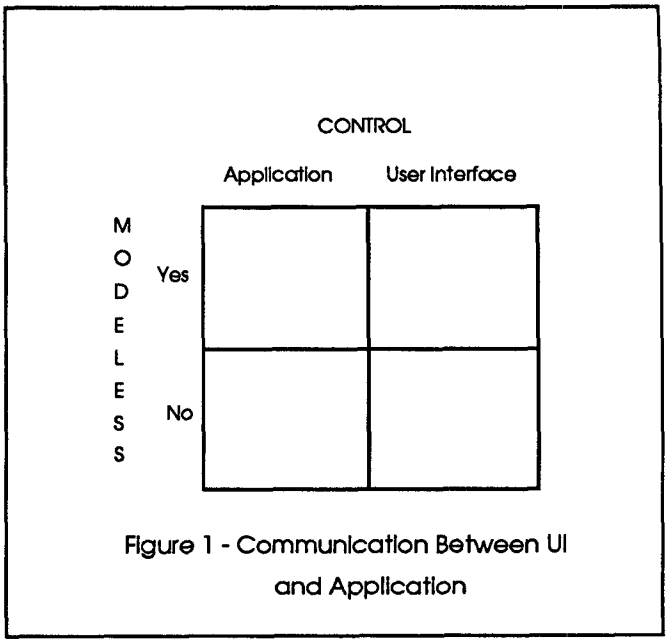

Figure 1 - Communication Between UI and Application

The design of the user interface and application in a highly interactive environment can be viewed in three different ways. These different ways are described in the next three paragraphs.

Some designers have taken the viewpoint that user interfaces are almost independent of the application. Thus the interface can be designed with minimal knowledge of the application, only a specification of the required interactions are needed and then the UI designer can proceed. At the appropriate time the application can be bound to the UI.

Another perspective is provided by the application designer. In this case there is a toolkit available which can be used by the application when a user interaction is required. Unless proper care is taken, the UI becomes tightly bound to the application and a clean separation of application and UI may not be achieved.

A third approach is provided when development of the application and the user interface proceed together. In this case the developer produces the UI but with strong reference to the application. All user interactions are defined in the context of the application and the UI is created. Once the UI is designed the body of the application which does not require direct user interaction can be completed. This approach could have the same difficulty as the one described in the previous paragraph which was application-centred. Suitable use of object-oriented programming may help alleviate this problem.

Requirements analysis and description can be centered around the user interface construction activity, during which requirements are analyzed and described by constructing a prototype. Design, implementation and testing are clustered around the architecture and component prototyping activities, almost merging into one single construction activity. The results of a prototyping-oriented development strategy are further improved if they are supported by adequate tools. One important requirement for such tools is that they support an evolutionary strategy whereby the prototype eventually becomes the constructed system itself. Bischofberger and Keller [BISC89] have proposed a prototyping-oriented software life-cycle model, shown in Figure 2.

A user interface management system (UIMS) is the set of tools which can assist in creating the prototype of a user interface. Within the scope of this paper UIMS are considered as tools that help a programmer create and manage all aspects of user interfaces. UIMS are generally characterized by crisp separation of the application code from the code that implements the user interface, and also by the support for specifying the user interface at a higher level of abstraction than that obtainable with general-purpose programming languages.

The search for adequate tools to support a prototyping-oriented development strategy calls for the

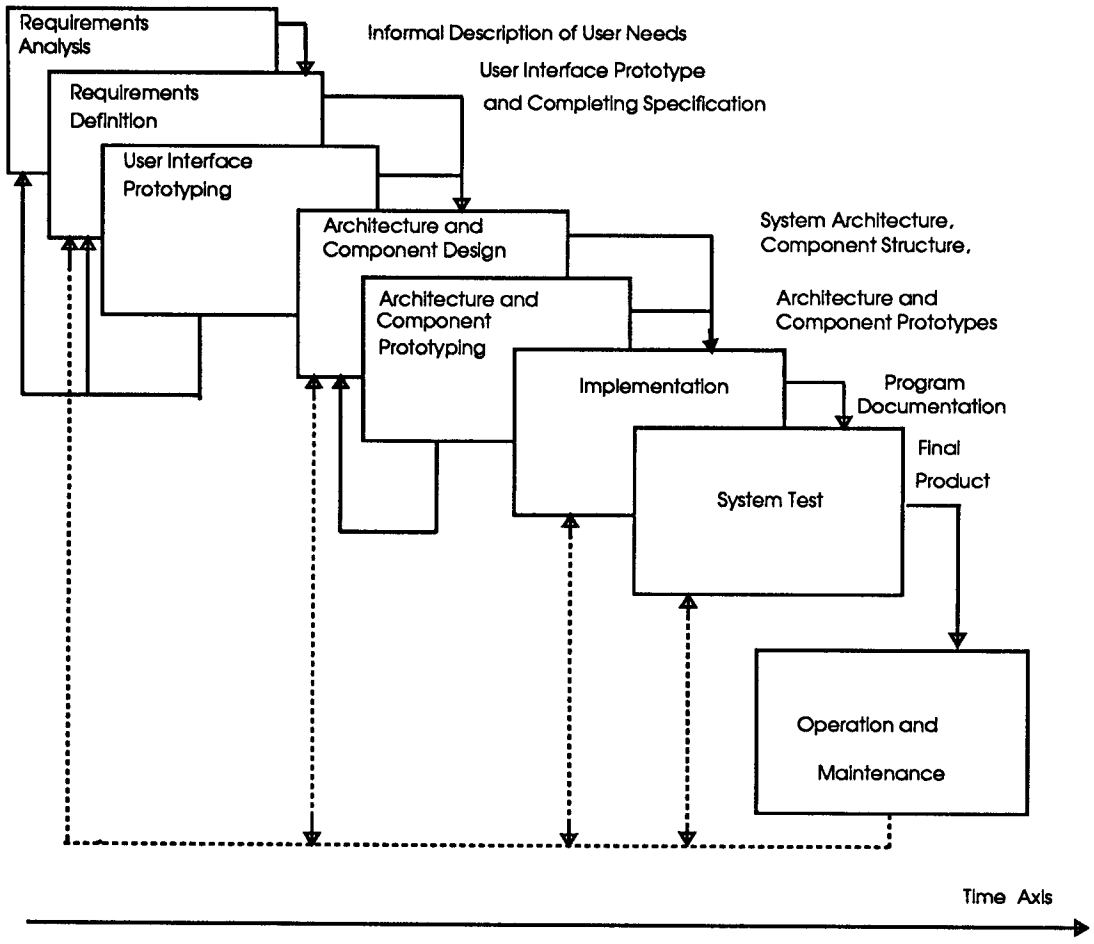

Figure 2 - The Prototyping - Oriented Software Lifecycle

integration of User Interface Management Systems (UIMS) with Computer-Aided Software Engineering (CASE). In this paper we discuss several approaches to UIMS design and implementation, and propose a design strategy that allows the integration of interface design with development of application software. This design strategy is supported by MIDAS as described in Section 4. A UIMS separates interface and application in order to isolate application code and interface specification, and also to allow different interfaces to drive the same application. However, a UIMS does not implement any application code. The main goal of a UIMS is to let interface designers or even end-users design and quickly modify the interface, without requiring extensive programming skills or a deep knowledge about the application.

It seems clear that a conventional UIMS cannot be easily integrated with a prototyping-oriented development strategy because the interface designer must, in this case, be an expert designer of software systems; that is why we are using, for the case of the MIDAS system, the expression User Interface Design System (UIDS), as opposed to UIMS (as done before by Hills [HILL87]). In other words, in contrast with classical UIMS, the UIDS will require the joint participation of the end-user and the system designer in the process of designing the interface (in this case seen also as concomitant requirements specification). In order to contribute further to the attainment of these goals, MIDAS provides knowledge-based assistance on interface design techniques, as well as facilities for end-user fine grain customization of the generated interface. We can now state the requirements that drive the development of the MIDAS environment:

- MIDAS is a User Interface Development System which integrates interface design with prototyping-oriented development of applications.
- MIDAS is a knowledge-based environment which gives support to system designers on both interface

design and application software design principles, having an underlying life cycle model that encourages the simultaneous development of the interface with the precise specification of the application.

- The end products of MIDAS are an application-specific interface which can be further customized by the end-user (by flexibly altering the default appearance of the dialogue scenarios), the binding code between the interface and the application modules (stubs), and the specification of the application software all the way down to the machine processable import-export list and module semantics specification of each application module.

- The application-specific end-user interface is also knowledge based. Its domain knowledge covers user modeling and the application domain, in order to adapt itself dynamically to different degrees of user familiarity with the application, from novice to expert.

- Both the interface code and the application code are developed within an object-oriented framework.

- The MIDAS UIDS architecture attempts to adhere to the ECMA model [SMAR89], which organizes, in a layered model, design principles for object-oriented user interface development (or management) systems.

2. UIDS Architectural Constraints to Support Application Development

2.1. Command dialogues, direct manipulation and modeless interaction

Two key issues had profound impact on the resulting architecture of the MIDAS UIDS: the communication metaphor between user and the application interface, and the degree of freedom the user has to walk about the various communication scenarios that the interface presents, i.e., how "modeless" is the end-user interface.

As to the communication metaphor between end-user and the application interface, the main advantages of direct access to objects, as opposed to indirect or command-oriented dialogues, are the ability to manipulate directly the exhibited objects, and the transparency of the representations of those objects and of the operations upon them. These two advantages have both a psychological and cognitive impact on the user, as they convey a feeling of security and reassurance, thus reducing the mental effort involved in translating the input actions and output representations into operations and objects in the problem domain. A direct manipulation dialogue [SHNE83], [JACO85] must:

- present good metaphors to represent the application world in terms of screen objects and of input actions.

- continuously represent the focussed object.

- allow for fast, incremental and reversible operations, whose impact on the object of interest is immediately visible.

- provide access to the operations on an object.

- presume the results of a manipulation shown on the screen are acceptable as input for subsequent manipulation.

Direct manipulation does not guarantee by itself a well designed dialogue. One must look for further principles and design guidelines, and that brings us to the second key issue. Three relevant aspects [SCHM86] in assessing end-user interface quality are naturalness, consistency inside and outside the application, and the avoidance of modes. Naturalness is seen here as the resulting feeling when the interface

- Does not force you to remember the name of every command.

- Does not allow disastrous actions (like destruction of valuable data) to occur accidentally.

- Does not require you to understand the entire system in order to accomplish tasks.

- Allows you to switch back and forth between several different tasks without forcing you to finish one before beginning the next.

- Provides a variety of ways to input, manipulate and retrieve data.

- Has a spectrum of versions, ranging from beginners to experts.

- Is forgiving about mistakes.

- Allows you to change your mind and undo any number of actions.

Internal consistency requires that all concepts, functions and procedures apply across the parts of an application. For example, text editing should feel the same, whether or not one is editing a field in a spreadsheet or typing a formula for a column. External consistency states

that all concepts, functions and procedures common across applications must be the same. For example, the editing commands to enter source code for a compiler or to type a letter, or to edit the contents of a cell in a spreadsheet should be essentially identical.

In [SCHM86] a mode of an interactive computer system is defined as a state of the user interface that lasts for a period of time, is not associated with any particular object, and has no role other than to place an interpretation on operator input. Modes limit the user to performing a certain action at a certain time or placing a special, and possibly unintended, interpretation on the user's action. As much as possible, applications should be modeless.

The Macintosh User Interface Standard [SCHM86] is an example of a modeless event-driven user interface. Instead of a rigid, hierarchically structured set of commands that a user must traverse to reach a desired operation, most commands in the application are available all the time. Such systems are typically designed with a central main event loop, which cycles endlessly waiting for an event to occur. When any one does, it causes the appropriate routine to be executed. On the Macintosh, the lower level system software manages an event queue in which events are posted for later processing. Events of various types are placed in the event queue, packaged with all their appropriate information (the current location of the mouse, state of the keys on the keyboard, for example). The main event loop processes and dispatches these events in a FIFO discipline. The kinds of events placed in the queue include mouse events, keyboard events, window events, etc.

2.2. Relation between the interface and the application semantics

Models of the user interface fall into two broad categories [SIBE86]. The more frequently used contain linguistic models which view the interface as a dialogue between user and computer. The second category, spatial models, includes interactive graphics or direct manipulation models. This distinction is also made by Hutchins [HUTC86], who distinguishes two basic ways humans interact with computers. Before introducing his terminology, we should point out the subtle distinction between dialogue and interface: a dialogue is the observable two-way exchange of symbols and actions between human and computer, whereas an interface is the supporting software and hardware through which this exchange occurs.

Current research in UIMS has two main directions: making it easier to specify user interfaces and increasing the range of user interface styles that are supported. With regard to interface specification, the state-of-the-art also requires a prototyping approach (different from the one discussed in Section 1): user interface specification techniques are so poor that iterative, or trial and error approaches to the design of UIs, are necessary. In this realm, a UI design would first be roughed out and a prototype would be developed. The design of the UI would then be evaluated by examining the performance of users with the prototype. With the knowledge gained from this evaluation, the design would be modified, a new prototype developed, and the evaluation repeated. We see this (re)design-prototype-evaluate cycle as the phase of functional requirements definition in our software life cycle model. Nevertheless, much of UIMS research (e.g. [TANN85]) is predicated on the assumption that the user interface can be separated from the application and the two can be developed separately. It is commonly claimed that UIMS should allow non-programming user interface experts to design and implement user interfaces. In what follows we will discuss the origins of these motivations.

Going back to the classification in [HUTC86], there are two kinds of interactions between humans and computers: the ones based on the conversational model, which correspond to the sequential (or synchronous, deterministic, with mode) dialogue, and the ones based on the world model, which correspond to the asynchronous (or non-deterministic, event-driven, modeless) dialogue.

For sequential dialogue, physical separation at design time of dialogue-related and application software is fairly straightforward. In the asynchronous case, dialogue separation into components can be more difficult to achieve, because the execution of dialogue and application tasks tends to be more closely interleaved, and the two components often share a common data representation of both interface and application objects. Furthermore, in modeless direct manipulation dialogues there is a need for closeness of interface to application semantics, which works against dialogue independence.

Dialogue independence is an approach (hardly achieved by most UIMS) in which design decisions affecting only the human-computer dialogue are isolated from those affecting only application system structure [HART89].

Hill [HILL87] examines three relevant UIMS based on sequential dialogue [JACO85], [WASS82], [OLSE83]. They are classified according to the type of dialogue specification language that is used for the preparation of the interface: recursive transition networks [JACO85], [WASS82] and grammars [OLSE83]. A user interface implementor working with transition networks thinks in terms of states and transitions, while one working with a grammar-based system must think in terms of matching of non-terminals. In transition networks the state is explicit; with grammars, there is no explicit notion of state.

The three examples analyzed in [HILL87] are thereby named external control UIMS. External control UIMS (as the control is external to the application) are designed to control an interactive system and occasionally call application routines to do the work. This type of system can easily support communication from the syntactic to the semantic level through simple parameter passing. Communication (particularly of arbitrary values) in the reverse direction is more difficult to support. The opposite design approach is referred to as internal control. In this case, the application program occasionally calls UIMS modules to collect and perform other tasks. Semantic to syntactic communication is easily supported via parameter passing.

External control UIMS are more popular. However, using the external control model still results in a poor compromise. An alternative model that can efficiently support two-way communication is needed.

Event-based mechanisms are currently the primary underlying control and communication techniques upon which direct manipulation dialogues are constructed. User actions are sensed (usually in a combination of interrupts and polling loops) and communicated to interface software as events. The system can be clearly divided into components, communication among components is done by message passing, and the mechanism becomes quite general by viewing each message within the system as an event. For event-driven dialogues there are some difficult tradeoffs in breaking the system into components. Figure 3 shows a typical configuration for run-time control and communication among components [HART89].

In Figure 3 the dialogue component is subdivided into input dialogue and output dialogue. The input dialogue component knows all the objects in the user interface and in the application, and is sensitive to any events affecting objects as a result of user action. Sequential control requires the top level of control logic to be expressed explicitly by the dialogue developer. A similar top layer of control logic is required to provide application-specific sequencing. The asynchronous control mechanism works because input events get queued in the event queue and eventually handled by the proper objects, and control is yielded to those objects for processing. The dialogue developer is thus afforded great freedom to isolate the behavior of individual objects and actions within complex asynchronous dialogues without concern for the complicated network of control details in the high level part of the structure.

It is the very fact that the direct manipulation interaction style brings the user cognitively closer to the application semantics [HART89] that made us decide for a User Interface Development System based on an object oriented, event-driven direct manipulation UIMS. All the

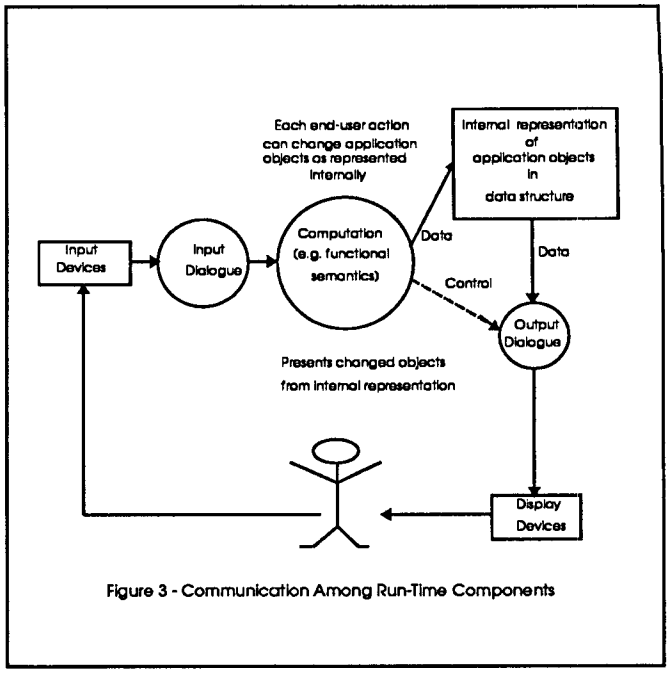

Figure 3 - Communication Among Run-Time Components

freedom that can be exerted in the design of event-driven direct manipulation UIMS calls for a reference model. The reference model allows us to compare our work with existing products and ongoing projects, and to introduce the architectural elements we used in the design of MIDAS.

2.3. A reference model of object-oriented UIMSs

Terminology for object oriented interface development features and tools has not stabilized yet. For example, the term tool is used to refer to anything from a complete interface development environment to a library routine for a single small interface feature.

Hayes and Bara [HAYE89] have recently classified the features of what they called Graphical User Interfaces (GUIs). Recognizing that there are some hybrids, they show that most GUIs consist of three major components: a windowing system, an imaging model and an application program interface (API).

The windowing system is a set of programming tools and commands for building the windows, menus and dialogue boxes that appear on the screen. The imaging model defines how fonts and graphics are actually created on the screen. The API is a set of programming language function calls that allow the interface designer to specify which windows, menus, scroll bars, icons, etc. will appear on the screen. On top of these three elements some systems also have tools for creating interfaces and developing integrated applications. In [HAYE89] the authors compare twelve different systems supplied by different vendors or organizations interested in standards. Although the paper

allows a comparison of different systems, the proposed classification does not manage to provide any insight into design principles to be followed by GUI designers. In fact, the different systems are structured in distinct ways, what makes, for instance, the windowing system level of their classification correspond to a different level in the model presented in the sequel. Also, an object-oriented UIMS is not explicitly defined in the paper.

In their User Interface Assessment Report [SMAR89], the ECMA TC33 discussed the desirability of incorporating into PCTE [BOUD89] a common user interface for the tools to be developed for the platform. For that purpose they proposed a User Interface Reference Model, shown in Figure 4, to facilitate the discussion about features and tools that constitute a UIMS. From our point of view this model constitutes more than a mere classification of levels and mechanisms of a UIMS. In fact, with the minor adaptations we have proposed, they achieve a precise definition of the concept of a UIMS and have also proposed a design approach to such systems (for instance, one that allows interfacing under certain conditions, with existing GUIs). The model described in Section 4 utilizes the basic approach taken by ECMA as described.

In the sequel we describe the various layers of the model shown in Figure 4. We have adapted the definitions to fit our interpretation of the document and our point of view about UIMS design.

- Data Stream Encoding: similar to the X protocol in X Windows [SCHE86]. It is the layer which is connected to the X server. It includes the implementation features of the screen and input drivers.

- Base Window System Interface: offers a set of primitives to manipulte windows and graphics. The X library of the X Windows system is at this level.

- Toolkit Intrinsics: provides the support for the definition of user interface object types. It defines the implementation paradigm for the implementation of the toolkit layer. It can be, for instance, a C language binding with object oriented flavor, generic facilities for class manipulation.

- Toolkit: provides a set of user interface object types which have a defined behavior. It is structured in the form of a library of classes containing a number of resources (attributes) and operations (methods and procedures). This level provides the binding with the application. Both Motif [OSFM90] and Presentation Manager [PETZ89] have this level clearly defined.

- Presentation: this level provides the means for the organization of the instances of the toolkit object types needed by an application. This level has an associated structural model, or metaphor, of the human-computer interface (forms, desktops, etc.) that serve as frameworks for understanding the elements of human-computer interfaces and for guiding the development of the dialogue. The User interface Language of Motif is at this level.

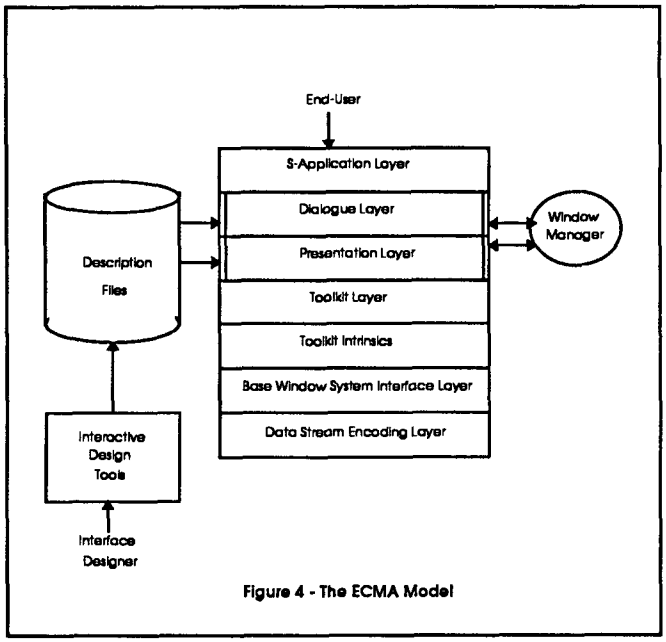

Figure 4 - The ECMA Model

- Dialogue: its purpose is to handle the dialogue, i.e., the synchronization of the different possible operations available to the user that should be done in an application independent way.

- S-Application: the Structured Application Layer includes the application code organized in such a way as to make it possible to communicate with the user interface through appropriate mechanisms.

A UIMS interacts at the Dialogue and Presentation layers in Figure 4. In other words, the interface designer uses a set of interactive design tools grouped around dialogue and presentation notations to define a specific interface to an end-user.

3. Relation to Other Object-Oriented UIMSs and Prototype Development Systems

At this point we are ready to compare the MIDAS approach with the ones taken by existing object oriented UIMSs and prototyping environments. Surveying all the literature in this area is well beyond the scope of this paper. An excellent survey has been recently published by Hartson and

Hix [HART89], which covers a number of those systems, including some that use knowledge-based approaches.

In the area of UIMS our work has been considerably influenced by MacApp [SCHM86], InterViews [LINT89] and GWUIMS [SIBE86]. Although MacApp and Interviews are often called toolkits, as opposed to UIMS, we take the comparison as valid because they simplify the creation of both the controlling elements of interfaces (buttons, menus, etc.) and of the data to be manipulated. They both overcome problems ususally associated with "classical" UIMS, in the sense that they avoid relying on an interpreted specification language, and are adequate for the design and construction of event-driven interfaces.

GWUIMS [SIBE86] was designed on the basis that a strict logical separation between the lexical, syntactic and semantic levels of the user interface (we will later refer to this approach as the "language view of programs") is not possible. The authors hold that it is not possible to build systems which handle semantic errors and at the same time give intelligent feedback if a strict separation between the lexical/syntactic domain and the semantic domain of the application is maintained. In addition to the user interface itself, GWUIMS consists of representaion objects (R-objects), interaction objects (I-objects) and application objects (A-objects). For example, a representation object may only exchange messages with interaction objects and objects within the user interface. This design restriction on message paths was used in GWUIMS as a means of enforcing a logical separation between different linguistic levels within the UIMS. This approach, further improved in [SZEK88], gives rise to the notion of structured application that first appeared in this paper in connection with the ECMA model. The structuring of the objects is captured in MIDAS in the form of a prototyping-based life cycle which drives the system.

Three recent doctoral theses, by Hill [HILL87], Szekely [SZEK88] and Myers [MYER89] have also influenced our work in the MIDAS system. Hills' work stresses that the ease of use of a UIMS is irrelevant if the UIMS cannot support the types of interfaces that are desired. He concentrates on making major extensions to the range of UI styles that can be supported, on the assumption that better interfaces will be different interfaces. His major contributions are the solutions given to the problems of specifying concurrent dialogues, and supporting communication and synchronization among the various components of a UIMS. This support for communication and synchronization influenced the design of MIDAS and, in particular, we borrowed the expression User Interface Development System, which better characterizes his and our work, while contrasting with the concept of a UIMS.

The Peridot system [MYER89] allows the interface designer to design and implement asynchronous interfaces in a direct manipulation manner. The designer does no programming whatsoever in the conventional sense, since all commands and actions are given visually. The designer draws the screen display that the end-user will see, and then performs actions just as the end-user would. The results are immediately visible and executable on the screen, and can be easily edited. The designer gives examples (hence the term programming-by-example) of typical values for parameters and actions, and Peridot automatically infers the general case. Since we do not envisage "automatic programming" in MIDAS (visual programming + programming-by-example), and since the expected MIDAS user is a systems designer following a prototyping oriented life cycle, we see major differences between our work and Myers' [MYER89]. Nevertheless, we envisage the possibility of generating end-user interfaces via MIDAS which would be user-customizable in a style similar to Peridot's.

The work by Szekely [SZEK88] is based on the language view of programs introduced by Foley [FOLE82], Moran [MORA81] and Newman [NEWM79]. In this view, the language to communicate with a program has four levels, called conceptual, semantic, syntactic and lexical, respectively. The conceptual level describes the tasks the user is able accomplish by using the program. The semantic level describes these operations and the objects they operate on. The syntactic and lexical levels describe how the user accesses the objects and operations using the input and output devices. Szekely [SZEK88] has introduced the notion of "communication concepts" that can explain the behavior of the interface features of programs. They capture the distinctions in semantics that are relevant to the construction of graphical user interfaces. Each class of communication concept captures a different semantic distinction. Szekely also shows how to decompose interactive programs into a functionality component and a user interface component that communicate via program abstractions rather than via user interface abstractions. The program abstractions identified by the author support a variety of graphical user interfaces providing extensive semantic feedback, whose implementation traditionally required violating the separation between functionality and user interface. The author produces a catalog of implementation techniques and an object-oriented UIMS based on them. The language view of programs is an integral part of the prototyping-oriented life cycle model that drives the MIDAS environment.

The above comparisons take UIMSs as references, with the granted exception of Hills' UIDS. In all cases, the described systems aim at synthesizing a UI. In MIDAS, we are interested in synthesizing a complete system, taking a UI-derived specification as a starting point. Recent literature shows examples of other systems sharing the same goals as MIDAS. Ongoing developments like TOPOS [BISC89] and OSU [LEWI89] are two good examples.

Both systems, like MIDAS, are prototyping-oriented software development environments. TOPOS includes tools for requirements analysis and definition, for user interface construction and for system testing, dealing with components at up to three different levels of completion at the same time:

- components which consist of nothing more than a defined interface and a description of the functionality they should provide.
- components which are currently prototyped to make sure they are feasible as planned.
- components that were parts of existing applications and can be reused.

OSU [LEWI89] is a program for writing other programs by a combination of user interface design with sequencing of the user interface interactions and program generation techniques. In object-oriented programming terminology, the goal is to produce objects and message-passing configurations automatically that implement a specific application. The authors claim that OSU will be able to prototype systems that can generate a wide spectrum of applications by combining a number of domain-specific tools.

As mentioned earlier, MIDAS owes many of its design principles to the technologies developed in the area of UIMS, being itself a UIDS that shares a set of goals with TOPOS and OSU. Nevertheless, there are a number of differences between MIDAS and these last two. For the moment we will stress one of them: MIDAS is driven by a life cycle model that considers the joint development of user interface and application. This life cycle model uses concepts from the language view of programs and is stored in MIDAS' knowledge base.

4. The MIDAS Life Cycle Model

MIDAS is an environment whereby a set of tools support complete requirements definition of the user application by developing its interface. This prototyping-oriented life cycle model is shown in Figure 5. It fits into the general approach of expressing the software process through a series of small prototyping subprocesses first introduced in [BOEH86] with his spiral model of software development and enhancement.

The main results of the development of the user interface, shown inside the dotted rectangle in Figure 5, are a working user interface, the architecture of the application program, an already "bound" application component prototype. As in [BISC89], our approach integrates interface development with application development. The rationale behind it is that the user interface implicitly defines most of the functional requirements, exception handling, and the majority of non-functional requirements. For highly interactive applications, it is intuitive that by specifying the user interface one often obtains an almost complete systems specification, the complete development of a user interface is the basis for the definition of application requirements in the MIDAS environment.

The first five phases of the proposed life cycle are highly integrated. From an informal description of the user needs, embodied in the requirements analysis, the interface designer starts to identify the application objects. This is a step in which his knowledge of the class library of the application world is of utmost importance, because the more he knows about existing classes the more code may be reused. In other words, identifying objects in the application world will always include fitting them into the application world class hierarchy (second phase). In order to further characterize the objects, the inteface/systems designer (hereafter referred to as interface designer, to emphasize the role of interfaces as specifications) identifies (or designs) the functionality of the objects in the application. This is tantamount to spelling out the messages understood by objects, their class and instance variables, and any globals that the application might need. This is a very critical point because the functionality of most application objects is reflected at the user interface level (the one being designed) as commands, presented to the end-user in the chosen user interface display metaphor.

Even though user interfaces vary greatly, they are composed of just a few basic information display metaphors, as shown in Table 1 [FISH88].

Just as there are several ways to display a piece of information on a computer screen, so there are many ways to interact with, or command, the application program. Some current UIMSs make use of more than one command interaction metaphor. The more commonly emphasized are command line interpreter, menus, and function/control keys. One can also verify that there is a strong trend towards direct manipulation as a convenient and "gestalt prone" metaphor.

It is therefore in the third phase that application objects are further characterized by being assigned functionality (materialized as messages that the object understands), and part of that functionality surfaces at the user interface level either as direct manipulation of objects in the screen or as selection of an item in a pop-up menu, and so forth.

Phase four represents the choice of command interaction style. The interface (system) designer lays down the "scenes" and sets the script of the dialogue in his visual programming interaction with MIDAS.

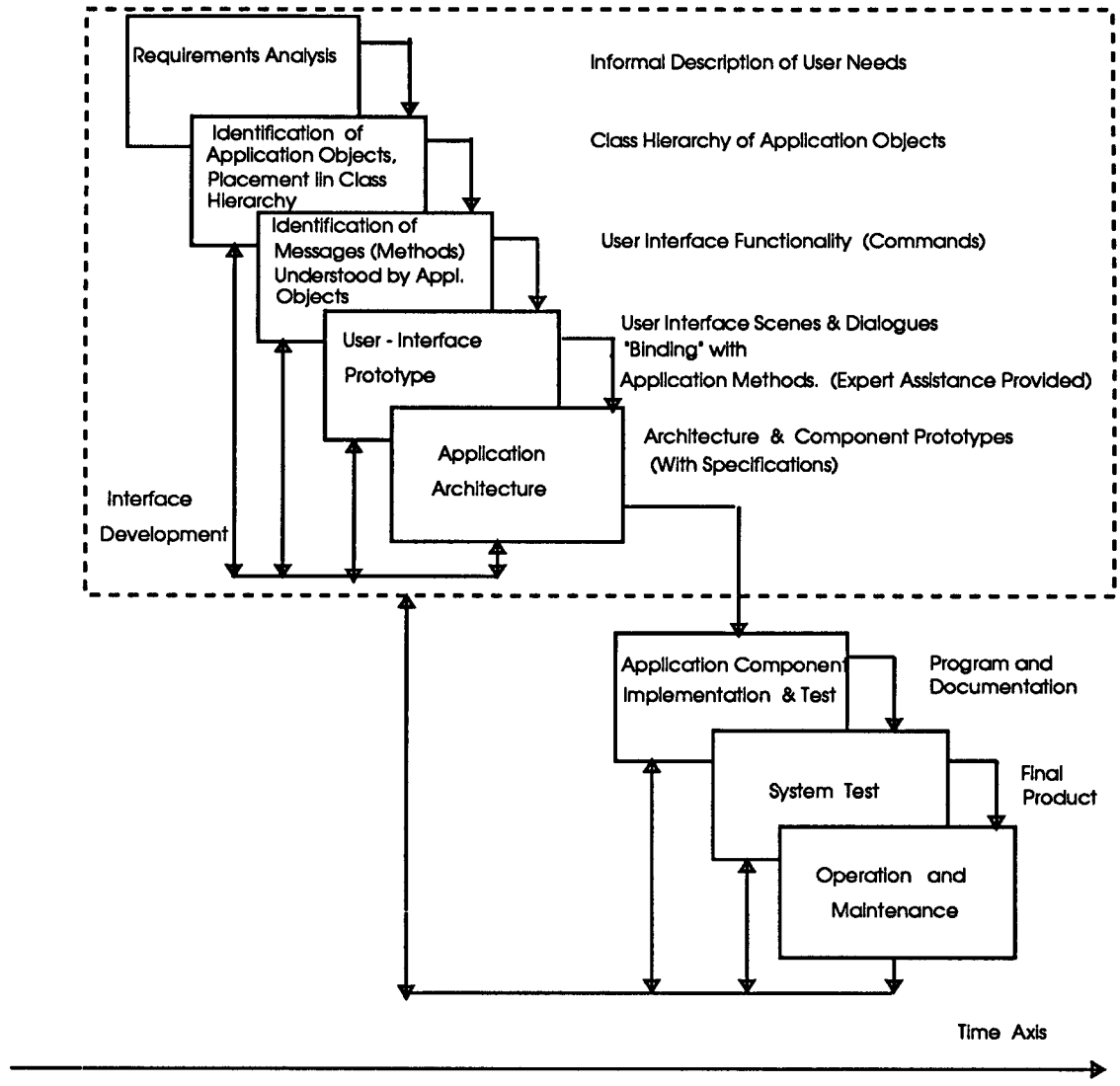

Figure 5 - MIDAS Prototyping - Oriented Lifecycle Model

MIDAS produces modeless interfaces, according to an end-user dialogue front-end template. Modelessness is achieved because this template includes event-driven code, i.e., the end-user interface is built by adding application-specific code to the existing expandable template. One can visualize this mechanism in the pseudo-code shown in Table 2.

Notice that this body of code is fixed and (although not induced by the syntax adopted here) implemented as object oriented software. Interrupt handlers enqueue all events, packed with all necessary information to process them. An asynchronous loop dequeues them and (as if in) a CASE statement, each different class of event gets the appropriate processing. This structure allows for non-deterministic dialogues to be designed, since the inner workings of the end-user dialogue template does not impose any particular sequencing of events on any dialogue. MacApp [SCHM86], for instance, also has a built-in event-driven loop. In order to enforce the look and feel and everything else that defines the Macintosh User Interface Standard, this loop has more sequencing (and is more complex in nature) "hardwired" within it. MIDAS "microprograms" these constraints on an otherwise free, nondeterministic, loop by means of rules expressing interface design techniques in the knowledge base.

The link between the user interface code and the application code happens in the above pseudocode at each tag of the CASE statement. MIDAS' library of user

DISPLAY METAPHOR	APPLICATIONS
Tables	Financial spreadsheets
	Relational databases
	Reminder/appointment schedulers
	Disk directory listings
Forms	Database records
	Individual account information
	Order entry systems
	Configuration/setup applications
Text Objects	Word processors
(words, paragraphs)	Page composition systems
	On-line news retrieval systems
	On-line help facilities
Graphical Objects	Computer-aided design (CAD)
(boxes, circles)	Graphics editing programs
	Page composition systems
	Project management/scheduling tools
	Process control graphics
	Business graphics display

Table 1 - User Interface Display Metaphors

```
BEGIN
   ON event DO enqueue (queued_event);
   REPEAT
   WHILE NOT empty (queue) DO
      BEGIN
         dequeue (queued_event);
         CASE queued_event_class OF
            A: process_A_event (queued_event);
            B: process_B_event (queued_event);
            C: process_C_event (queued_event);
            ......
            ......
            ......
         END;
      END;
      UNTIL user_quits;
END;
```

Table 2- Pseudo-code for handling events

interface building classes includes command objects and view objects (in MacApp terminology) that do everything related to the mechanics of interface dialogues (like sensing mouse clicks, exhibiting menus and capturing user options, dragging images, etc.). It is left to the interface designer the task of providing the semantics of the application by implementing the behavior of all application objects. It is the very nature of object-oriented programming, in particular inheritance and polymorphism, that makes it possible to have such a layered architecture in which application code is seamlessly incorporated into the interface code which acts as "main program glue".

Not every piece of application code is interface-related, although it may now be clear enough how intertwined are the choice of objects in the application world, the resulting functionality of the interface, and the structure of the application code. There is still plenty of space for architectural creativity at the designer's disposal. The exchange of messages among objects within the application is totally free territory. For those application objects that receive messages originated from an interface interaction, MIDAS has all information necessary to characterize that object's (class) interface, and a "do-nothing" stub can immediately be filled in for simulation purposes. The internal structure of purely application code is left to the systems designer in this very phase.

MIDAS provides an executable specification notation [HOFF88] that allows the systems designer to complete the characterization of all application objects and simulate the behavior of the final product. Since the interface (system) designer has available the specifications of the already existing (in the class library) application objects, MIDAS provides methods for configuration management and program validation [ALEN90]. It is worth noticing that, in a prototyping-oriented environment, programming in the large [DERE75] is isomorphic to configuration management.

All phases outside the dotted rectangle in Figure 4 follow current standard software engineering techniques. The lifecycle model presented in this section is rule-encoded in MIDAS' knowledge base. The interface designer is guided by the interpretation of those rules. In other words, the rules embody a software process, and MIDAS can be seen as a process-driven user interface development system.

5. Architecture

The architecture of MIDAS is shown in Figure 6. MIDAS interacts with the interface designer via a meta interface, in a direct-manipulation mode, thus instantiating all objects at the Presentation level (the "scenes" the user will see) and also the thread of control between these scenes (the "script" for the interaction), at the Dialogue level.

The design database contains the class library of interface objects, encompassing event handlers, command objects, window managers, view presenters, etc. In other words, these are the classes that actually carry out the functionality of the event-handling loop in the end-user dialogue front-end template. These classes are the actual toolkit that an interface designer would use in an environment without direct manipulation facilities in a meta interface, expressing himself by writing actual source code.

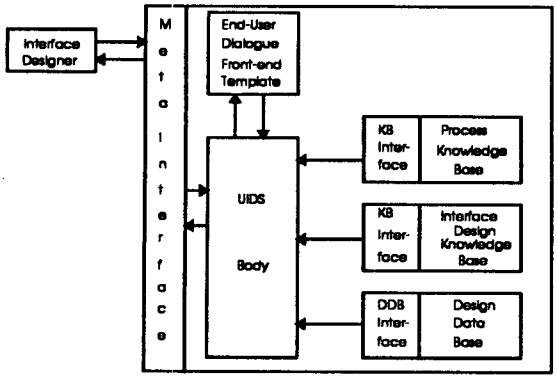

Figure 6 - The Architecture of MIDAS

Most of the "look" component of the "look and feel" of user interfaces is contained there. Whenever a novel family of user interface objects becomes noteworthy, systems programmers can produce the object classes templates that instantiate these new features on user interface screens and include them in the DDB repository. The interface designer browses the contents of the DDB via the meta interface and the UIDS body.

The process knowledge base contains rules to support the interface designer in following MIDAS' software lifecycle model. In other words, these rules are a formal expression of this lifecycle model. The UIDS body manages the communication between the process knowledge base and the meta interface, thus implementing the behavior of a software process expert.

The software process is encoded as a collection of rules, each rule having a precondition and an action. As in [KAFE87], whenever the condition is true, the corresponding action may be executed. Each activity in the knowledge base corresponds to a tool that actually performs it, taking into account a number of parameters for execution. One such parameter tells, as an example, whether or not the activity can be fired automatically, without the intervention of the user. Classical forward and backward chaining interpretation strategies are applied to the rules. In forward chaining, if the precondition of an activity is satisfied, it may be triggered. Backward chaining is applied when the user invokes an activity whose precondition is not yet satisfied. Given that the execution of an activity gives rise to a postcondition, the expert advisor always performs backwards chaining in order to find activities it can perform that contribute to the satisfaction of that precondition. In case the precondition cannot be satisfied after all attempts made by the expert advisor, the user is informed of this situation, with appropriate context information. For example, if the code for an application module is missing, there is nothing MIDAS can do, except request the user to fill in the code or a stub thereof. In essence, the expert advisor behaves opportunistically, that is, it keeps updating postconditions and preconditions, and whenever an activity can be triggered, it commands the machinery to do so. A wise choice of automatically triggered activities is essential, in order to disable those that might get performed at undesired moments, getting in the way of the interface designer.

The interface design knowledge base [GUAR89] contains rules embodying knowledge of a different sort, namely that of interface design techniques. Expert advice on interaction techniques, on choice of screen objects that better convey the desired functionality, window tiling, superimposing colors, etc., are the kinds of functions carried out by this architecture module, also with the intervention of the UIDS body. Most of the "feel" in the "look and feel" of any particular user interface generated with the UIDS will be the result of the interface designer's interaction with the interface design knowledge base.

The UIDS body is the environment controller. As it interactively helps build the end-user interface, it makes use of a major piece of data (used as code in the end-user interface), depicted in Figure 6 as "end-user dialogue front-end template". This piece of code becomes part of the generated end-user interface, as explained below.

The end-user interface architecture shown in Figure 7 is the piece of software that interacts with the user and has all the functional "bindings" with the application software. The dialogue front-end becomes the actual interface between the application user and the application binding code (which in turn links the dialogue front-end to the code that implements the semantics of the application).

The actual connection between the dialogue front-end and the application code is done by supplying methods that implement (override) assumed behavior at high levels of the class hierarchy, and by supplying new classes of objects (for instance, those that do not directly interact with the end-user interface objects) that produce transformations in the "back-end" application objects. This diversity of ways to attach application code to the dialogue front-end is represented in Figure 7 as application-binding code architectural module, and the pure application code is shown as the architectural modules within the dotted rectangle.

Szekely's communication concepts [SZEK88], presenters and recognizers were designed to fulfill exactly this role. Some of MacApp classes, such as TCommand [SCHM86], instantiate command objects that exchange messages with those in the application, to bring about the changes needed to reflect menu, mouse and keyboard commands. An illustrative example of how this layered

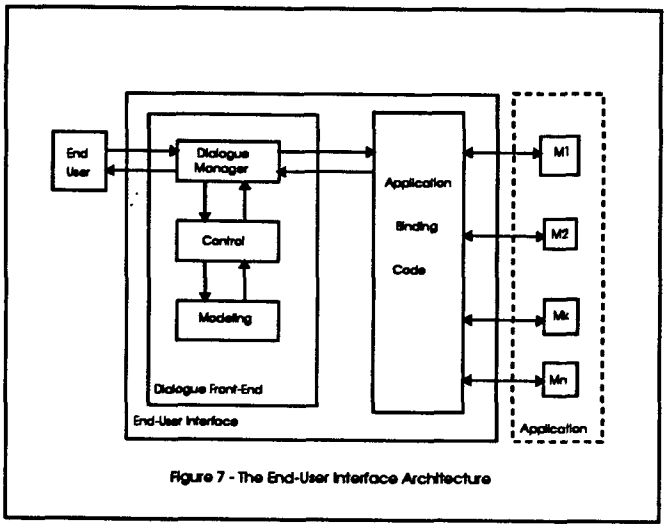

Figure 7 - The End-User Interface Architecture

object-oriented code merges with the application code is the "undo" command. The Macintosh interface standard strongly suggests that every application object be able to undo its previous transformation. The main body of MacApp's event-driven loop, upon receiving an "undo" command from some end-user action, sends it to a command object which, in its turn, sends it to the actual application object. Again, although the message selector is always the same ("undo"), each receiving object (be it a command object or an application object) will react differently, due to polymorphism and the natural overriding of message selectors provided by class hierarchy.

The dialogue front-end is composed of a dialogue manager module, a control module and a modeling module. The latter builds representation models about the user, which are used to adapt the interface dynamically to his needs. The models are subject to constant modification, resulting from interaction feedback. The resulting interface is reached through an interactive and iterative process, as shown in Figure 8.

A user stereotype [RICH83] is initially arbitrated (drawn from the end-user template), and the interface is designed according to it. Upon using this interface, the user gathers his mental model about the interface and the application. On the other hand, the dialogue manager, in a stepwise manner, observes and analyzes the user's behavior, and builds a model of the user's mental model. The latter is used to further adapt the interface, thus allowing the user to (again) update his own mental model, and so forth.

Deciding about how and when to adapt to the user are critical, for they are directly related to the user's confidence in the system [MUIR87], which is sinonymous with the confidence the user acquires about the predictability of the interface's behavior. Therefore it is of utmost importance

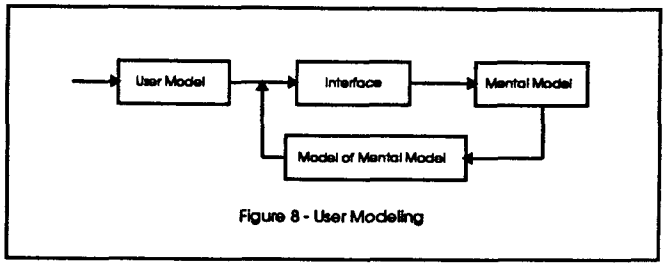

Figure 8 - User Modeling

that adaptation to the user be done in a logically perceivable way. For that, either the system informs the user about the alterations that are about to happen, or adaptation is done only in those functions that do not jeopardize the user's confidence in the system.

The dialogue manager module is responsible for the actual exchange of symbols (in the broad sense) between the end-user and the application program. In this module one encapsulates the various dialogue types available at the interface and the knowledge-gathering code that allows the inteface to know more about the user as the dialogue flows.

The control module manages the other two modules. Its main task is to ensure that the dialogue flows in a natural and cooperative way. To that end, it incorporates an inference mechanism that allows the dynamic adaptation of the dialogue to the user, depending upon the context of the interaction. Besides that, the control module needs filtering mechanisms that allow the gathering of information relevant to the modeling module, i.e., the control module is the means through which the interface is able to make decisions based on the knowledge sources that it can access.

6. Conclusions

In this paper we have described and justified the design strategy adopted in the MIDAS User Interface Development System. MIDAS is actually an umbrella project, which links together various ongoing research and development efforts carried out under the supervision of the authors.

As it stands, MIDAS relies on an interface and development framework based on objects, which parallels ET++ [WEIN89]. The design and implementation work in the areas of user modeling and expert assistance in interface design can be found in [CABR90] and [GUAR89]. The specification of modules and its use to implement a deductive configuration management system is described in [ALEN90].

A large number of existing software systems have user interfaces which are driven by menus and dialogues. Another aspect of the MIDAS work is determining a design discipline which will allow these existing systems to be

moved to different user interfaces. Much of the work is based on object-oriented techniques and language specification of interface interactions [DURA90]. The development of visual programming concepts and supporting tools is at a fairly early stage.

The key new ideas communicated in this paper are the notions of a prototype-oriented lifecycle model for object oriented UIDS, and its associated architecture. In particular, the notion of an object oriented process-driven UIDS has no parallel in the literature. The implementation process being adopted follows the proposed lifecycle model, i.e., the next results we want to achieve are the UIDS' interface generation via a sequence of prototypes together with a complete, formal version of the lifecycle model.

Acknowledgements

Cabral, Campos and Lucena want to thank the Computer Systems Group of the University of Waterloo for the enjoyable working atmosphere provided during their stay from January to March 1990.

7. References

[ALEN90] Alencar, P.; Lucena, C.; "A Logical Approach for Evolving Software Systems", to appear

[BISC89] Bischofberger, W.; Keller, R.; "Enhancing the Software Life Cycle by Prototyping", Structured Programming, 1:47-59, 1989

[BOEH86] Boehm, B.; "A Spiral Model of Software Development and Enhancement", ACM Software Engineering Notes, Vol.11, No.4, August 1986

[BOUD89] Boudier, G. et al; "An Overview of PCTE and PCTE+", SIGPLAN Notices, Vol.24, No.2, 1989

[CABR90] Cabral, R.H.B.; "Intelligent Interfaces: Conceptualization, Taxonomy and Support Architectures", (in Portuguese) to appear

[DERE75] DeRemer, F.; Kron, H.H.; "Programming-in-the-Large versus Programming-in-the-Small", IEEE Transactions on Software Engineering, V. SE-2, No.2, June 1976

[DURA90] Durance, Carl M.; "An Approach to Application Software Mobility Across User Interface Toolkits". M. Math. Thesis, Computer Science Department, University of Waterloo, Canada, 1990.

[FISH88] Fisher, A.S.; "CASE: Using Software Development Tools", John Wiley & Sons, Inc., 1988

[FOLE82] Foley, J.; van Dam, A.; Fundamentals of Interactive Computer Graphics, Addison-Wesley, 1982

[GUAR89] Guaranys, P.; Lucena, C.; "PUC: A Knowledge Based Environment for Planned User Communication", Proceedings of the Computer Software and Applications Conference (IEEE, COMPSAC 89), 1989

[HART89] Hartson, H.R.; Hix, D.; "Human-Computer Interface Development: Concepts and Systems for its Management", ACM Computing Surveys, Vol.21, No.1, 1989

[HAYE89] Hayes, F.; Baran, N.; "A Guide to GUIs", BYTE, July 1989

[HILL87] Hill, R.D.; "Supporting Concurrency, Communication and Synchronization in Human-Computer Interaction", University of Toronto, Computer System Research Institute, Technical Report CSRI-197, 1987

[HOFF88] Hoffman, D.; "Practical Interface Specification", Software - Practice and Experience, Vol.19(2), February 1989

[HUTC86] Hutchins et al; "Direct Manipulation Interfaces in User Centered System Design", in D.A. Norman and S.W. Drapers, Eds. Lawrence Erlbaum Assoc., Hillsdale N.J., 1986

[JACO85] Jacob, R.J.K.; "An Executable Specification Technique for Describing Human-Computer Interaction" in Hartson, H.R., Ed., Advances in Human-Computer Interaction, Vol.1, Ablex Publishing Corp., Norwood, N.J., 1985

[KAFE87] Kaiser, G.E.; Feiler, P.H.; "Intelligent Assistance without Artificial Intelligence", Proceedings of Thirty-Second IEEE Computer Society International

Conference (COMPCON), San Francisco, CA, February 1987

[LEWI89] Lewis, T.G.; Handloser III, F.; Bose, S.; Yang, S.; "Prototypes from Standard User Interface Management Systems", IEEE Computer, May 1989

[LINT89] Linton, M.A. et al; "Composing User Interfaces with Interviews", IEEE Computer, February 1989

[MORA81] Moran, T.; "The Command Language Grammar: a Representation for the User Interface of Interactive Computer Systems", International Journal of Man-Machine Studies, 15:3-50, 1981

[MUIR87] Muir, B.M.; "Trust Between Humans and Machines, and the Design of Decision Aids", International Journal of Man-Machine Studies, 27, 1987, 619-629

[MYER89] Myers, B.A.; Creating User Interfaces by Demonstration. University of Toronto, Computer Systems Research Institute, Technical Report CSRI-196, May 1987

[NEWM79] Newman, W.; Sproull, R.; Principles of Interactive Computer Graphics. McGraw-Hill, 1979

[OLSE83] Olsen, D.R.Jr.; Dempsey, E.P.; "Syntax Directed Graphical Interaction", SIGPLAN Notices 18.6, 1983

[OSFM90] "OSF/MOTIF Programmer's Guide", Revision 1.0, Prentice Hall, 1990

[PETZ89] Petzold, C.; "Programming The OS/2 Presentation Manager", Microsoft Press, Redmond, Washington, USA, 1989

[RICH83] Rich, E.; "Users Are Individuals: Individualizing User Models", International Journal of Man-Machine Studies, 18, 1983, 199-214

[SCHE86] Scheifler, R.W.; Gettys, J.; "The X Window System", ACM Transactions on Graphics, V.5, No.2, April 1986, 79-109

[SCHM86] Schmucker, K.J.; Object-Oriented Programming for the Macintosh, Productivity Products International, Inc., 1986

[SHNE83] Shneiderman, B.; "Direct Manipulation: a Step Beyond Programming Languages", IEEE Computer, 16, August 1983, 57-69

[SIBE86] Sibert, J.L.; Hurley, W.D. and Bleser, T.W.; "An Object-Oriented User Interface Management System", Computer Graphics: SIGGRAPH'86 Conference Proceedings. Vol. 20, No. 4, August 1986

[SMAR89] Smart, J. et al; TC33 Technical Assessment Ad-Hoc Group: User Interfaces; "User Interface Technical Assessment Report", ECMA, March 1989

[SZEK88] Szekely, P.; Separating the User Interface from the Functionality of Application Programs. PhD thesis, Carnegie-Mellon University, January 1988

[TANN85] Tanner, P.P.; Buxton, W.A.S.; "Some Issues in Future User Interface Management Systems Development" in Pfaff, G., Ed., IFIP WG 5.2 Workshop on User Interface Management, 1985

[WASS82] Wasserman, A.I.; Shewmake, D.T.; "Rapid Prototyping of Interactive Information Systems", ACM Software Engineering Notes 7.5, 1982

[WEIN89] Weinand, A. et al; "Design and Implementation of ET++, a Seamless Object-Oriented Application Framework", Structured Programming 10/2, 1989