Design representations and development support for user interface adaptation

Constantine Stephanidis, Demosthenes Akoumianakis, and Anthony Savidis

Institute of Computer Science
Foundation for Research and Technology-Hellas

Science and Technology Park of Crete
P.O. Box 1385, GR-71110 Heraklion, Crete, Greece
Tel.: +30-81-391741, Fax : +30-81-391740
E-mail: {cs, as, demosthe}@ics.forth.gr

Abstract

With the advent of Graphical User Interfaces (GUIs) and the advances of input/output technologies, there has been a shift of perspective, from user interface programming tools to environments for designing interaction. This is partly attributed to technological maturity and partly due to the increasing requirement to support a need-driven and user-centered protocol for design, development and implementation of interactive systems. This paper investigates the architectural shortcomings of existing user interface development systems and environments with respect to supporting adaptation of a user interface and discusses methods, techniques and tools that are needed to empower user interface designers. In particular, the paper describes a high level architecture comprising user interface software components that can provide the required design, development and implementation support that is needed to facilitate user interfaces for different user groups with diverse requirements abilities and preferences.

1. INTRODUCTION

The concept of *User Interfaces for All* has been proposed (Stephanidis, 1995; Stephanidis et al., 1995) as a vehicle to efficiently and effectively address the numerous and diverse problems related to the accessibility of interactive applications in different contexts of use. Currently, there are no development tools to practically support the construction of User Interfaces for all. Towards this objective, the notion of *unified user interface development* has been introduced (Stephanidis et al., 1996) with the objective of supporting platform and user independent interface development. A unified user interface development platform requires that only a core of the user interface is developed, while the platform and user-specific interface properties can be automatically handled by special purpose user interface software tools. These tools can automatically manage the platform specific issues and adapt the resulting dialogue to the particular user (group).

In this paper, we are primarily concerned with the latter property of a unified interface development platform, namely the issue of supporting the adaptation of the user interface to the target user (group). The need for adaptation is evident when one considers the diverse needs and requirements of different user groups, the wide availability of alternative and non-conventional input/output technologies, the compelling need for more user friendly products as well as the recent trends towards universal accessibility and greater usability of interactive applications. In this context, adaptation of the user interface has been found to be a quality attribute of a design (Benyon et al., 1993), as well as a pre-requisite for accessibility (Stephanidis et al., 1995; Stephanidis, 1995). As a consequence, future user interface design environments will increasingly be required to support the articulation of plausible adaptations at the semantic, syntactic and lexical levels of interaction in order to allow for different scenarios of use and to support selection of appropriate semantic properties, user task sequencing, interaction styles, input/output devices and interaction techniques, given the intended user groups and the tasks that they should perform with the user interface.

There have been several studies investigating the numerous dimensions of adaptation in interactive software systems (for a review see Dieterich et al, 1993), namely, what constitutes an adaptation constituent, the level and timing of adaptation, the controlling agent, the type of knowledge that is required to arrive at meaningful adaptations, etc. Nevertheless, despite the substantial contributions of these efforts to the study of adaptation, there are still several issues that need attention, if user interface adaptation is to be adequately served by designers and developers of interactive software applications.

Towards this objective, it is necessary to investigate and extract a reference model for user interface architectures in order to understand the range of adaptable and adaptive behavior that is needed and the way in which it can be effectively supported throughout the user interface life cycle (i.e. from the early design phases to implementation and maintenance). In the past there have been several attempts to extract a reference model from concrete user interface architectures in order to classify existing prototypes and to guide the construction of user interface software. The best known architectural abstractions of user interface software include the *Seeheim* (Ten Hagen, 1990), *PAC* (Coutaz, 1991), *ALV*, *MVC* and *Arch/Slinky* models (The UIMS Tool Developers Workshop, 1992). However, they do not consider user interface adaptation aspects. Moreover, in the literature on User Interface Adaptation, architectural abstractions are almost missing. Mostly, particular prototype architectures are reported (see Cote, 1993; Sherman et al., 1993; Zimek, 1991; Anchieri et. al., 1991; Sukaviriya et al., 1993, etc), instead of abstract models. Even those however, limit their scope to address adaptivity which constitutes only one dimension of user interface adaptation in interactive software systems (see next section).

In this paper, we narrow the scope of investigation to consider the type of tools that are needed in order to support the adaptation of user interfaces at the lexical level of interaction to accommodate the diverse abilities of different user groups, including disabled and elderly people. In this context, adaptation is considered in a broad sense, as a technique supporting the specification of lexical elements of interaction in such a way that the resulting user interface is accessible and usable by the intended user group. Our objective is to define user interface software components which provide designers and developers with the required support for articulating adaptation constituents and designing, developing and maintaining the user interface, as opposed to arriving at dedicated and programming-intensive solutions. This objective differs from other approaches and efforts aiming to consider and propose conceptual models for the study of adaptation in human computer interaction (cf. Benyon, 1993). Instead, the normative perspective adopted in this paper is that support for user interface adaptation should be systematically embedded in high level user interface environments and unified development platforms, in order to empower designers and developers to articulate adaptation scenarios and corresponding contexts of use so as to facilitate accessible and more usable interactive systems.

This paper reviews recent efforts towards adaptation of interactive systems and describes a new user interface software architecture which provides a unifying view of adaptation constituents and a high level user interface development platform supporting adaptation. The emphasis is on the software components and tools comprising the platform, their properties and protocol of communication. An implementation of this user interface architecture which supports lexical adaptability of attributes of abstract physical interaction object classes is then presented followed by a discussion of the potential impact of current efforts and plans for future work.

2. OVERVIEW OF RELATED WORK

In the context of the present work, adaptation comprises both adaptability and adaptivity. The term adaptability refers to a technique which is supported by the user interface development platform, as opposed to a programming-intensive characteristic of an interactive application. Consequently, it implies design and development support for the automatic assignment of lexical user interface constituents according to the needs, abilities, requirements and preferences of a target user group. Such a technique can be supported by a dedicated user interface tool which assists the designer of the user interface to select plausible design options and decide on maximally preferred ones so as to ensure the accessibility of the user interface by the target user (group). It is important to note that the above definition for adaptability differs substantially from other definitions commonly encountered in the relevant literature[1], as it implies (i) design and development support for the automatic adaptation of a user interface to the needs, abilities, requirements and preferences of a target user group. Additionally, it implies a range of adaptation constituents which are beyond mere parameterization of the application's functionality. On the other hand, adaptivity refers to a different dimension of adaptation which takes place at run-time (i.e. during an interactive session with an application). This type frequently involves adaptations of the syntactic layer of the user interface (i.e. identification of common patterns of use and simplification by means of macros). Our account of user interface adaptation includes both adaptability and adaptivity as introduced above, unless explicitly stated otherwise.

Research on adaptive and adaptable interfaces in the past has suffered from a lack of supporting tools which allow an interface to be easily created, modified and maintained. Also, adding adaptability and/or adaptivity to user interfaces so far has not been supported comprehensively by any high level user interface system or environment. Some of the early attempts to construct adaptable systems are OBJECTLENS (Lai, et al., 1988), BUTTONS (McLean et al., 1990), Xbuttons (Robertson et al., 1991). All these systems allow the user to modify certain aspects of their interactive behaviour while working with them. More recently, the AURA project (Adaptable User Interfaces for Reusable Applications) of the ESPRIT-II Programme of the Commission of the European Union has investigated thoroughly the issue of adaptability (Koller, 1993) and the underlying architectural abstractions for adaptable systems. AURA's objective was to define tools which would allow the specification of adaptable dialogue behaviour during the early design phases and the coupling of such tools with high

[1]The common definition for an adaptable system refers to a system that provides the user with tools that make it possible to change the systems characteristics (Opperman, 1994). A similar definition for adaptability is found in the work of Fischer (Fischer, 1993).

level user interface development systems (i.e. user interface management systems). Towards this objective, AURA provided implementations of dialogue nets as well as an event-driven dialogue definition language (EDDDL) with pre- and post-conditions. Dialogue nets allow the specification of adaptable dialogue behaviour by means of graphical representation. This graphical representation can be transformed into the notation of EDDDL which is tool-independent and can be used for dialogue specification in user interface management systems (Koller, 1993).

Adaptability has also been the chief objective in the development of the PODIUM system (Sherman et al., 1993). PODIUM's purpose is to show that a User Interface Management System (UIMS) which is supplied with a user interface automatically designed for a large and diverse user community can tailor that user interface automatically for each of many subgroups of a large user community. The only knowledge that the system uses to complete this task is the user characteristics that divide the community into user subgroups and its experience with users who have tailored user interfaces previously ..." (Sherman et al., 1993). According to the authors, the architecture of PODIUM resembles very much the structure of a conventional UIMS. In PODIUM the User Interface Generator is used to design and implement the user interface. The Interaction Handler arbitrates all interaction between the user and PODIUM (which includes allowing the user to custom-tailor the user interface).

In addition to the above research efforts towards adaptability, a number of systems have been developed to investigate the complementary objective of adaptivity. The state of the art in adaptive user interfaces includes OPADE (DeCarolis et al., 1994), AIDA (Cote, 1993), UIDE (Sukaviriya et al., 1993), as well as the results of several projects at national and international levels, such as AID (Browne, 1993), FRIEND21 (Okada, 1994). In addition to the above architectures for adaptable and/or adaptive user interfaces, there have been a few other proposals which, however, are more narrow in scope. Zimek in (Zimek, 1991) described the design of an architecture for adaptable and adaptive UIMS in production. His architecture comprises four functional units, namely, a user modeling component, a task modeling component, a strategy component and a UIMS. One of the interesting aspects in Zimek's architecture is the strategy component. This component is responsible for giving dynamic user support by manipulating the dialogue according to the individual competence and the special problems of the user. An alternative architecture has been described by Arcieri and colleagues in (Arcieri et al., 1991). They propose a functional architecture for a UIMS integrating application-independent user modeling capabilities. Finally, there has been substantial work towards the development of dedicated tools and techniques driving and supporting adaptive behaviour, such as GUMS (Finnin, 1989), UM (Kay, 1995), UMT (Brajnik et al., 1994), BGP-MS (Kobsa et al., 1995), PROTUM (Vergara, 1994), and (Orwant, 1995).

The main shortcomings of existing work towards adaptable and adaptive systems is that they do not provide comprehensive support for user interface adaptation in the context of high level development environments and tools. Even the AURA project which comes closest to support adaptation through tools does not account for such details of user-computer interaction. Moreover, the range of adaptation constituents considered do not account for basic design elements such as the choice of input/output devices, the interaction techniques, or other attributes of abstract interaction objects, such as the various types of feedback (i.e. initiation, interim, completion), access policy, navigation policy, topology etc. As a result, these efforts do not provide the support required to address user interface adaptations from the initial design and development phases to implementation and maintenance. In some cases, this limitation is directly attributed to the underlying approach for supporting adaptation, while in other cases it is a question of tools available and the underlying development platform.

3. TOWARDS A SOFTWARE ARCHITECTURE FOR ADAPTABLE AND ADAPTIVE USER INTERFACES

In this section, our emphasis will be on progressively developing the components of a new architecture which can support both adaptability and adaptivity of the user interface. In what follows, it is assumed that the application and user interface implementation are separate concerns, while user interface development is to be supported by a high level user interface tool.

3.1. Architectural abstractions supporting adaptation

It is important to note that both adaptability and adaptivity of user interfaces may be hardcoded, which implies that the user interface code has a pre-set structure. Typically, in such cases, adaptability and adaptivity are both built into the user interface code through rules which are local to the user interface and predetermined. This means that in case that the system's run-time behavior requires enhancements, either in the form of additional adaptability or adaptivity rules, the user interface code would have to be upgraded and recompiled. Of course modifications of the target application may also be required. It follows, therefore, that such architectures lead to *monolithic* systems which are likely to be large (in lines of code) and not easily modifiable.

An alternative approach would be to introduce the adaptability and adaptivity rules as orthogonal to the user interface, but not part of it. This is to say that such rules are not embedded in the user interface code, but they can be collected as supplementary information (in files) which can be consulted by the run-time libraries of the user interface development toolkit (see Figure 1). The architecture described in the diagram of Figure 1 is clearly more flexible in the sense that the rules determining adaptable and adaptive behavior of the user interface are not part of the user interface code. Instead, the user interface development toolkit, in addition to the other functions that it carries out, it also acts as an interpreter of adaptation decisions established either manually or by an external module (see below). This requires that the user interface development toolkit should support an explicit model of the adaptable and adaptive constituents which can be determined by the rules. Such abstractions are always desirable in user interface software and have been integral components of high level user interface development environments.

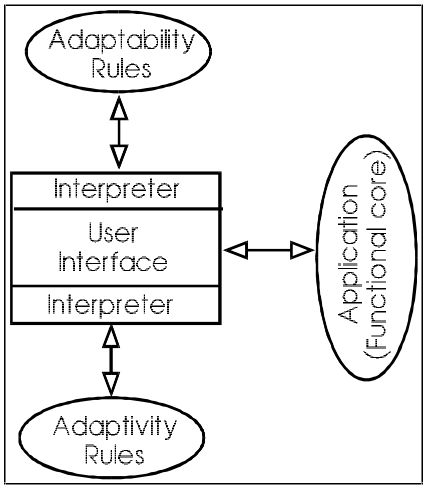


Figure 1: Alternative architecture for supporting adaptation.

The only concern regarding the architectural abstraction depicted in the diagram of Figure 1 is the way in which the adaptability and adaptivity rule sets are produced. Figure 1 assumes that such rules may be hard-coded or editable through templates, but this may lead to ad hoc and non-systematic user interface designs. More specifically, tools can be developed to support the automatic construction of appropriate adaptability and adaptivity rules according to the user's abilities, knowledge, interests as well as any preferences of specific adaptation constituents (i.e. interaction style, dialogue syntax, input/output devices, interaction techniques, etc). This observation leads to an enhanced architectural abstraction which is depicted in the diagram of Figure 2. In this revised architecture for adaptable and adaptive user interfaces, three tools have been introduced, namely, two design assistants that support the automatic generation of adaptability and adaptivity rules respectively, and a user interface development toolkit which is responsible for realizing the adaptable and adaptive user interface on a target platform (i.e. MS-Windows, X-Windowing system, etc).

In the following two sections, the latter architectural abstraction for adaptable and adaptive user interfaces is further elaborated by exemplifying the communication protocols between the user interface development system and the two modules for adaptability and adaptivity. It is assumed that both the user interface development system and the two modules for adaptability and adaptivity seek to adapt abstract physical interaction objects. Such objects encapsulate everything they need (i.e. appearance, placement, behavior and state) in terms of attributes (i.e. `size`, `width`, `topology`, `accessPolicy`, `interactionTechnique`, `inputDevice`, `interimFeedback`, etc). Consequently, both adaptability and adaptivity concern the automatic instantiation of abstract physical interaction objects by means of adapting their attributes.

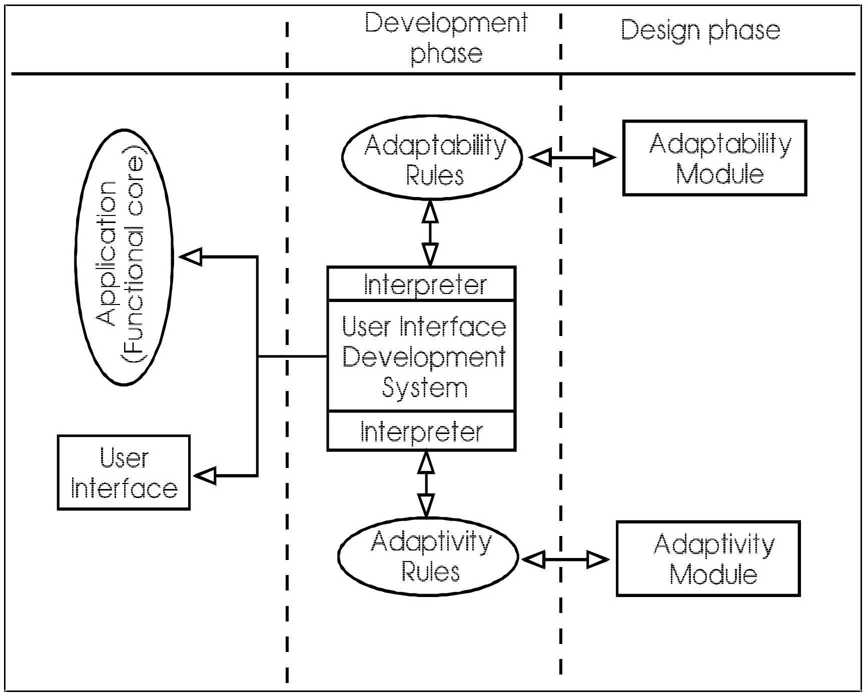


Figure 2: Rroposed architecture for adaptable and adaptive user interfaces

3.2. Communication protocol between the user interface development tool and the adaptability module

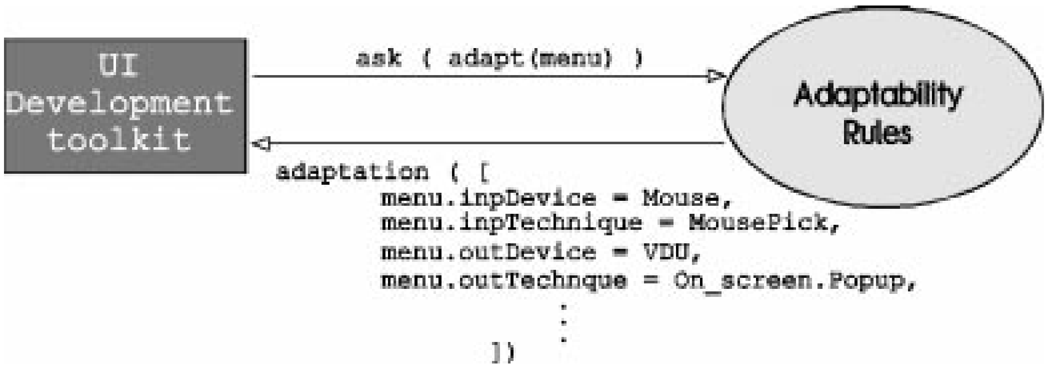
In order to support adaptability, the user interface development system requires knowledge about the adaptable user interface constituents. This knowledge allows the tool to properly instantiate an abstract physical interaction object into a concrete interaction object (e.g. an X Windows system menu). Consequently, the role of the adaptability module is to supply as much as possible of this knowledge. To achieve this, the adaptability module may be executed by the designer of the interface, before the development of the user interface commences, to compile an adaptation design scenario according to the user’s abilities, requirements and preferences. During user interface development, the run-time libraries of a user interface development toolkit may consult the decisions comprising the adaptation scenario, in order to properly realize the adaptable properties of abstract interaction object classes on a target platform. In this respect, the user interface development process is separated from the design phase (i.e. orthogonality), since the user interface development system may be used by the user interface developer after the completion of the task of the adaptability module. As an example, let us consider the development of a simple user interface which involves the construction of a menu. Before the developer implements the interface with the user interface development toolkit, the adaptability module is used to compile a file containing maximally preferred interface adaptability rules. Such rules may follow the format depicted in Table 1. During user interface development, the developer uses the user interface development toolkit to implement the user interface (i.e. select the abstract physical interaction object classes which will be used). The run-time libraries of the UI toolkit utilise the information of Table 1 to adapt accordingly the details of the interaction and implement the user interface on the target platform. In this way, the menu is automatically adapted according to the user's requirements, abilities and preferences. The communication protocol described above is summarised in the diagram of Figure 3.

```
Menu.input_device=keyboard
Menu.inputTechnique=indirectPick2D
Menu.output_device=braille&speechSynthesiser
Menu.outTechnique=tactileTechnique
Menu.on_BrailleLines=2
Menu.on_BrailleCells=80
Menu.interim_feedback=speech
Menu.on_audioVoice=male
Menu.on_audioVolume=4
Menu.on_audioPitch=99
Menu.fontFamily=helvetica
Menu.topology   =horizontal
Menu.access_policy=byKeyboard
```

Table 1 : Hypothetical lexical adaptability rules

3.3. Communication protocol between the user interface development tool and the adaptivity module

Having briefly reviewed the communication protocol between the user interface development toolkit and the adaptability module, this section will briefly discuss a possible communication protocol between the user interface development toolkit and the adaptivity module. However, before this protocol is detailed, it is perhaps appropriate to consider the meaning of adaptivity in this context. Let us assume that the user interface developer is to construct a list containing textual items, whose size of the selection set is not fixed, but dynamically determined by the application. It follows, therefore, that the exact look and feel of the list cannot be determined



```
ask ( adapt(menu) )
adaptation ( [
        menu.inpDevice = Mouse,
        menu.inpTechnique = MousePick,
        menu.outDevice = VDU,
        menu.outTechnque = On_screen.Popup,
                     :
                     :
               ])
```

Figure 3 : Communication protocol for lexical user interface adaptability

unless the application specifies the size of the selection set. Moreover, although the adaptability module rulebase may have a clause stating that in case the size of the selection set is larger than 25 items, the list should be contained within a scrolling window, no adaptability rule is deducible because the size of the selection set is not known at design time.

This is a typical case of adaptivity requiring that decisions on the adaptation of attributes of interaction objects be taken at run-time (i.e. while the user interface and the application are running). To handle this issue, the communication protocol introduced previously needs to be slightly revised. More specifically, whereas in the case of adaptability the user interface development toolkit was merely interpreting the adaptability rules before instantiating an interaction object, in this case it should feed the adaptivity module with data (i.e. size of the selection set) so as to enable the latter to fire the appropriate adaptivity rule. This slightly revised communication protocol is depicted in the diagram of Figure 4.

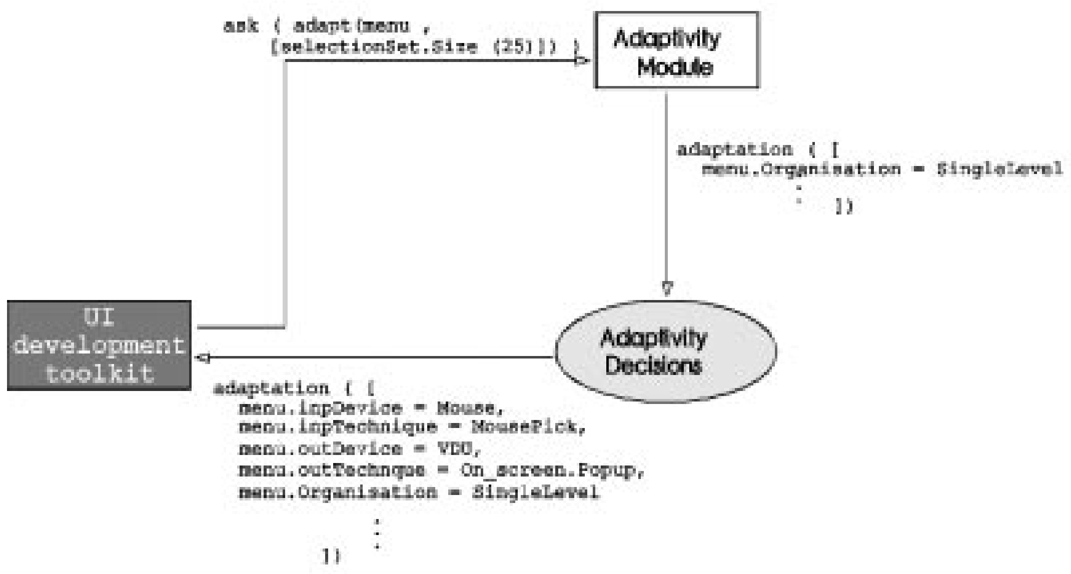

ask (adapt(menu ,
[selectionSet.Size (25)]))
Adaptivity Module
adaptation ([
menu.Organisation = SingleLevel
])
Adaptivity Decisions
UI development toolkit
adaptation ([
menu.inpDevice = Mouse,
menu.inpTechnique = MousePick,
menu.outDevice = VDU,
menu.outTechnque = On_screen.Popup,
menu.Organisation = SingleLevel
])

Figure 4: Communication protocol to support adaptivity

4. IMPLEMENTATION OF A USER INTERFACE DEVELOPMENT PLATFORM TO SUPPORT THE DESIGN AND IMPLEMENTATION OF USER-ADAPTED INTERFACES FOR DISABLED USERS

The above architectural abstraction has been used to design and implement a user interface development platform which supports user adapted interface development for disabled users. The platform comprises a number of tools which implement the properties of the aforementioned architecture. Currently, our developments have focused upon the implementation of tools which determine and apply adaptations at the lexical level of interaction. More specifically, a tool called USE-IT has been implemented which develops a semantics of adaptation at the lexical level and automatically constructs a lexical specification scenario depicting maximally preferred lexical adaptability rules so as to ensure accessibility of the target user interface by the intended user group. In addition, novel user interface development toolkits have been constructed supporting the development of visual and non-visual interaction, while at the same time, providing developers with the required support to interpret and apply the lexical adaptability rules to implement a user-adapted interface.

The reason why our current efforts have been concentrated on providing support for lexical adaptability is due to the fact that this level of adaptability is required to ensure the accessibility of the user interface. In particular, it is claimed that if the development tools can support the level of abstraction required to enable designers and developers to adapt non-trivial attributes of lexical interaction (such as input device, input interaction technique, output device, output interaction techniques, access policy, navigation policy, topology, feedback, etc), then it is possible to develop interfaces which are automatically adapted to the user's needs, requirements, abilities and preferences. To this effect, the tools developed thus far provide support for visual and non-visual interaction in two graphical environments, namely MS-Windows and X-Windowing system by means of integrating toolkits with enhanced or alternative (in the case of non-visual interaction) interaction capabilities.

Another distinction that needs to be made is the emphasis on adaptability as opposed to adaptivity. This is due to the compelling need to consider adaptations during the early design phases as otherwise no accessibility of the user interface by the target user group can be ensured. Consider for instance an adaptive user interface which can adapt certain dialogue characteristics, based on assumptions about the users drawn at run-time (i.e. during an interactive session). Such a facility is not useful in the context of disabled user groups, because it takes no account of the fundamental problem of accessibility. In other words, if no interaction can take place, due to some disability, no assumptions can be drawn and therefore no adaptation can be practically supported. Consequently, adaptation is concerned with both *initiating* and *sustaining* interaction. In this sense, adaptability is a pre-requisite for adaptivity, and needs to be addressed explicitly.

Consequently, the approach to supporting adaptations, which has been followed thus far, can be summarised as follows:

I. User interface adaptation rules are compiled by the USE-IT tool which reasons about adaptation constituents, selects plausible adaptations and decides on maximally preferred ones, through the *unification* of constraints pertaining to the lexical level of interaction (i.e. constraints related to the device availability, the user characteristics and the task that is to be performed with the user interface).

II. User interface adaptation rules are subsequently interpreted by the run-time modules of the underlying high level user interface development toolkit with which the user interface is to be implemented so that adaptations are instantiated onto a target technology platform.

In this manner, adaptation of the lexical layer of interaction is automatically supported during the initial design and development phases of the user interface.

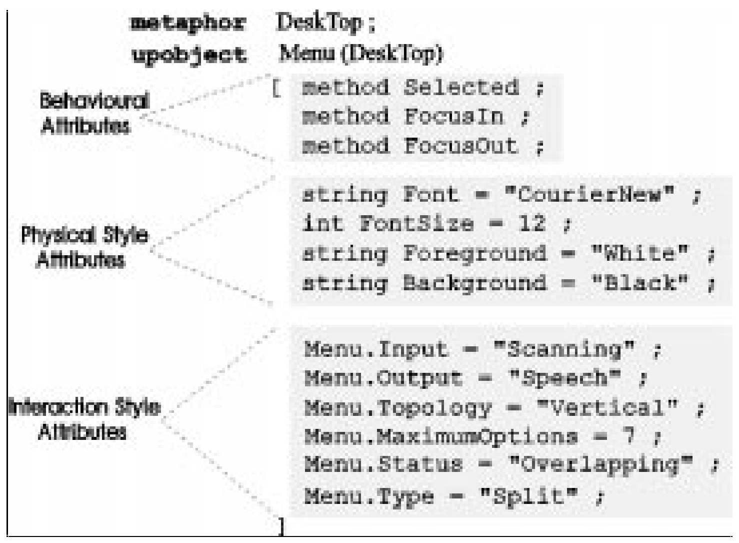
4.1 Adaptation Constituents

In this approach, a*daptable interface constituents are the attributes of abstract interaction object classes*. In the recent literature (Myers, 1990; Bodart et al., 1993), the term abstract interaction object (AIO) has been associated with several properties, briefly summarised as follows: (a) AIOs are application domain independent; (b) they encapsulate all the necessary interaction properties (i.e. appearance, placement, behavior, state, etc) by means of attributes (i.e. size, width, color, and methods such as selection, activation, state change, etc); (c) they preserve a degree of independence from particular windowing systems and environments (i.e.

they are platform independent). In the context of the present work, the term is used in a broader sense to include additional properties such as the following:

- AIOs are adaptable to the end user (i.e. their attributes can be adapted through reasoning);
- AIOs are metaphor independent (e.g. an AIO can be applicable for both the Desktop and Rooms metaphor);
- AIOs may have multiple physical realizations (i.e. they support *polymorphism* at the physical implementation level).

Abstract Interaction Objects possessing the extended set of properties are referred to as *virtual interaction objects.* Virtual object classes have physical realizations, depending on the target environment, referred to as instances of *virtual objects* (Savidis et al., 1995a). In the present work, adaptable interface constituents are attributes of *physical instances of virtual interaction object* classes. The conceptual object model which is shared by all tools being described in this part of the paper is depicted in the diagram of Figure 6.



```
metaphor   DeskTop;
upobject   Menu (DeskTop)
[ method Selected ;
  method FocusIn ;
  method FocusOut ;

  string Font = "CourierNew" ;
  int FontSize = 12 ;
  string Foreground = "White" ;
  string Background = "Black" ;

  Menu.Input = "Scanning" ;
  Menu.Output = "Speech" ;
  Menu.Topology = "Vertical" ;
  Menu.MaximumOptions = 7 ;
  Menu.Status = "Overlapping" ;
  Menu.Type = "Split" ;
]
```

Figure 6: A conceptual representation of the object model

The following subsections provide a brief account of the development and implementation of user interface tools to support adaptation. In particular, the adaptability module and the toolkits used for interface development and reviewed.

4.2. Deciding on maximally preferred lexical adaptability rules

Adaptation decisions for attributes of AIOs are automatically derived by a tool, called USE-IT. This is accomplished in a sequence of three phases: (i) reasoning about adaptation constituents; (ii) selection of plausible adaptations for each user interface constituent; (ii) decision on maximally preferred adaptations.

The USE-IT tool comprises: (i) a representation of design elements (i.e. models of the user, the task, and the availability of input/output devices); (ii) a representation of adaptation constituents; and (iii) algorithms and an inference engine to reason about, select plausible and decide on maximally preferred adaptations (see Figure 7).

Adaptability decisions are automatically compiled for those attributes whose adaptation is necessary to facilitate accessibility of the interface by the target user group. In this manner, user-specific details (i.e. abilities to operate different input/output devices and/or interaction techniques, access policy for container objects, topology of interaction objects, initiation/interim/completion feedback of interaction objects, etc), are transparent to the interface programmer as this knowledge is embedded within the development platform.

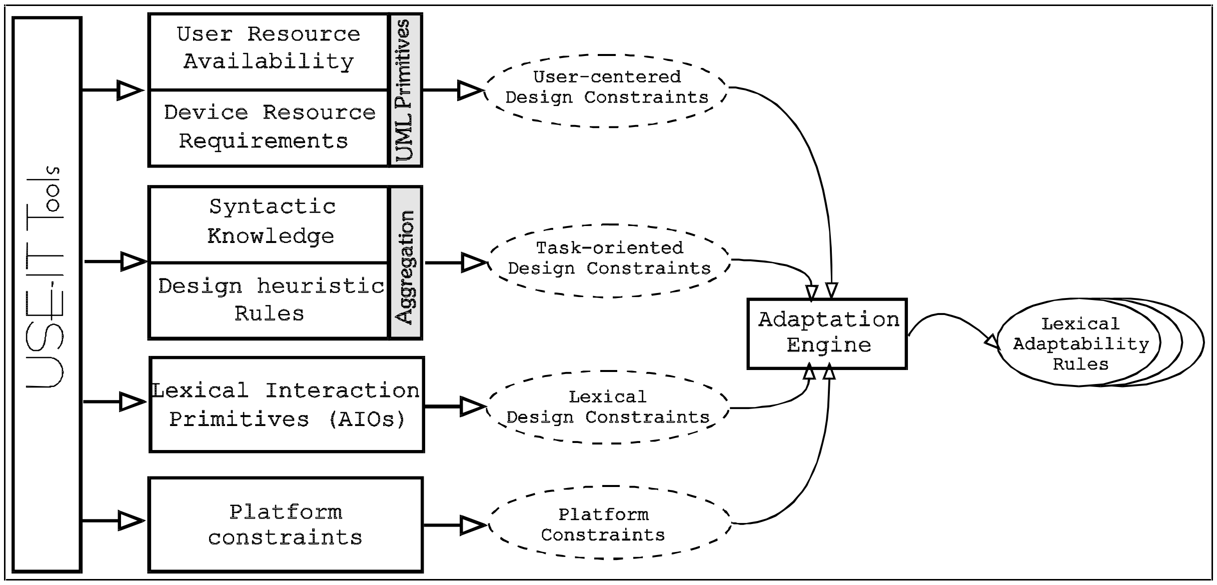
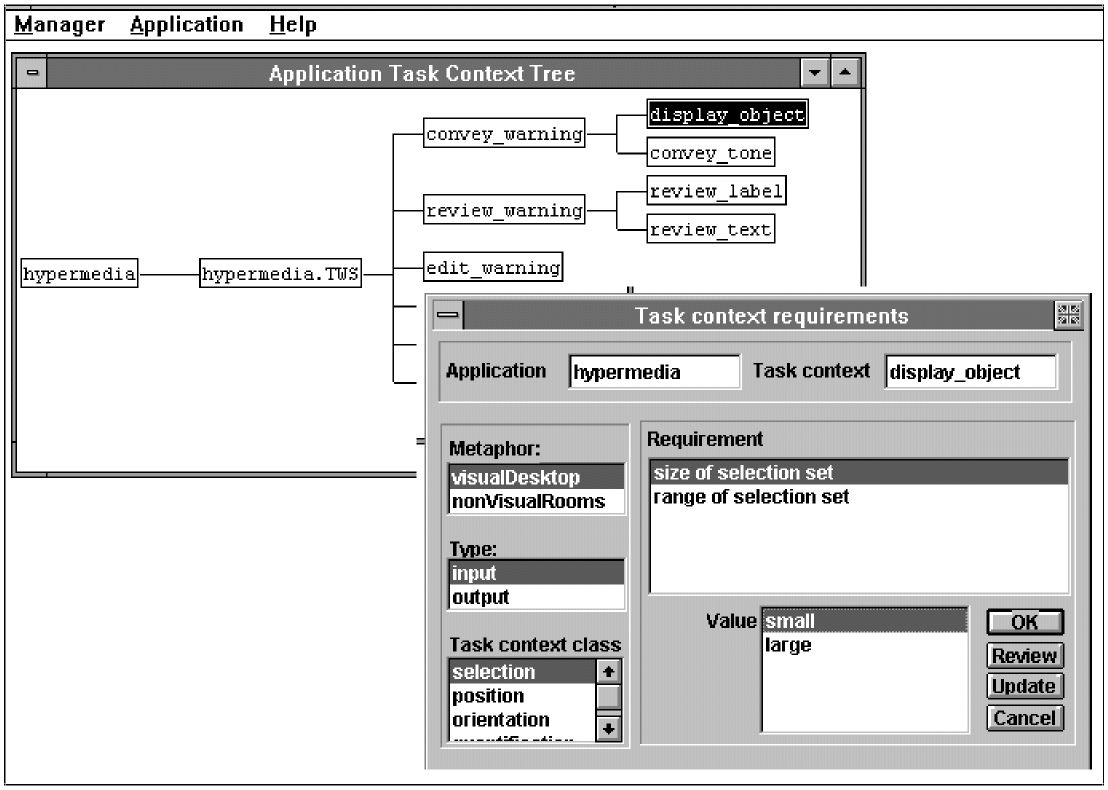

USE-IT Tools
User Resource Availability
Device Resource Requirements
UML Primitives
Syntactic Knowledge
Design heuristic Rules
Aggregation
Lexical Interaction Primitives (AIOs)
Platform constraints
User-centered Design Constraints
Task-oriented Design Constraints
Lexical Design Constraints
Platform Constraints
Adaptation Engine
Lexical Adaptability Rules

Figure 7: Overview of USE-IT's architecture

Lexical adaptability decisions are derived for attributes of abstract interaction objects per *task context* and *interaction metaphor*. In general, interaction metaphors may be either embedded in the User Interface (i.e. menus as interaction objects follow the "restaurant" metaphor) or may characterize the properties of and the attitude of the overall interaction environment (i.e. the desktop metaphor presents the user with an interaction environment based on sheets of papers called windows, folders, etc). In the present work, it is assumed that each development platform (i.e. OSF/Motif, MS-Windows, etc) serves one interaction metaphor (i.e. the visual desktop). Consequently, each of those platforms provides the implementational support that is required for the interactive environment of the metaphor. Different interaction metaphors may be facilitated either through the enhancement of existing development platforms or by developing new ones. An example is CommonKit (Savidis et al., 1995b) which supports non-visual interaction based on the non-visual Rooms interaction metaphor.

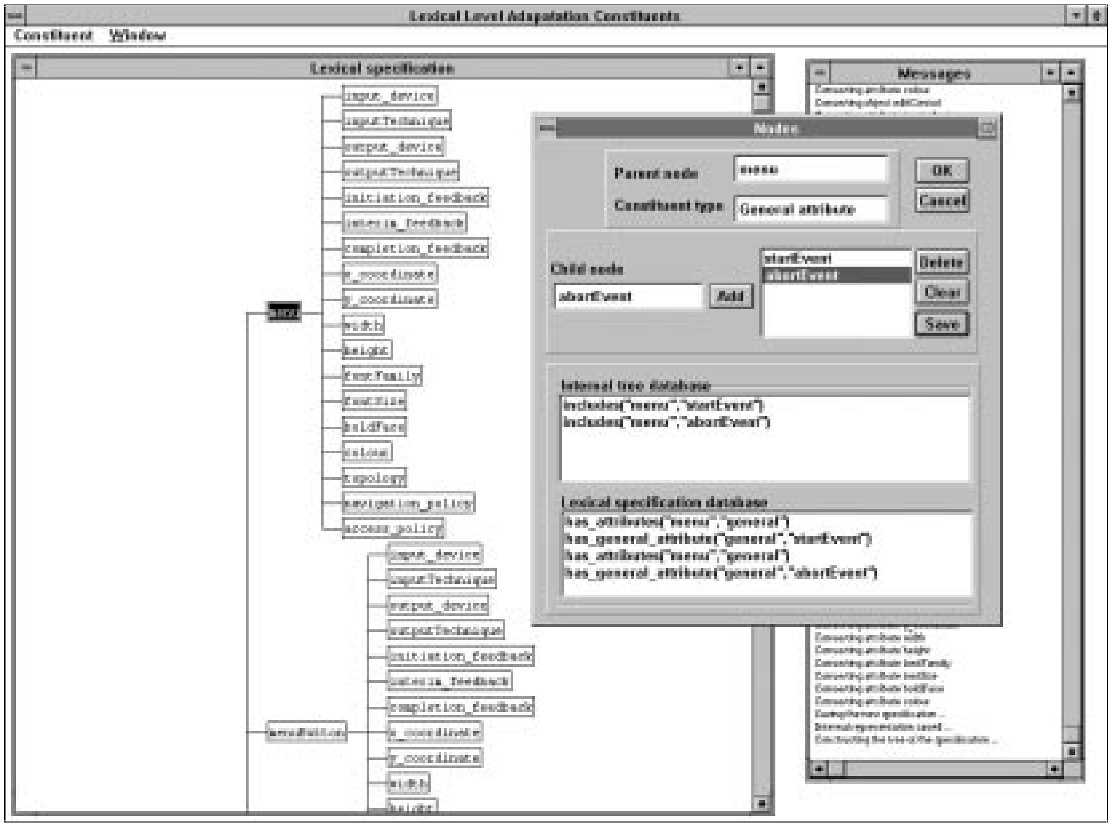
The second construct which determines the derivation of lexical adaptability rules by the USE-IT tool is the application specific *task context*. Task contexts facilitate the derivation of lexical adaptability rules based on syntactic knowledge. In other words, different rules may be inferred for the same object depending on the particular dialogue state. Consequently, a task context characterizes a "dialogue state" and can be conceived as the context of a given task in which the user is engaged at a particular time. Dialogue states are indicators of what the user interface or the user is doing at any point in time. It is important to note that dialogue states are application-specific and their purpose is two-fold. First they render the user interface adaptation process task-aware. Secondly, they provide the means for reasoning towards lexical adaptability rules based on syntactic level user interface knowledge. In other words they allow for different adaptations of the same object depending on the dialogue state. Thus, a menu item may be selected through voice input when the user wishes to review a text file, while when selecting to update the text, the corresponding menu item may be activated through pointing in 2D. In such a case, the adaptable attributes are the input device and the input technique used to interact with the menu item.

Task context attributes are characterization criteria which identify the application oriented interaction requirements in the current state of the dialogue. Each task context is described by means of its associated application requirements. Towards this, a small set of characterization criteria have been used to provide the required information (i.e. whether the task requires 2D positioning, size of the selection set in case the user has to do selection, range of a value that has to be entered, etc). The designer does not need to instantiate all criteria for a task context, but only those which are relevant. These properties are stored in a task schema, which is a collection of attribute-value pairs associated to the particular application specific task context, and then, they are interpreted to depict task-related constraints (i.e. the task requires a relative device, the selection set is large, etc). It is important to note that the USE-IT tool provides the designer with facilities which allow the elicitation of design representations based on the definition of task context hierarchies (see upper window of Figure 8), instantiation of each node in the hierarchy (see lower window of Figure 8) and revisions of the lexical specification layer and the user interface adaptation constituents.



Application: hypermedia
Task context: display_object
Metaphor: visualDesktop
Type: input
Task context class: selection
Requirement: size of selection set
Value: small

Figure 8: Building and compiling task context schemas



Parent node: menu
Constituent type: General attribute
Child node: abortEvent

startEvent
abortEvent

Internal tree database

```
included("menu","startEvent")
included("menu","abortEvent")
```

Lexical specification database

```
has_attributes("menu","general")
has_general_attribute("general","startEvent")
has_attributes("menu","general")
has_general_attribute("general","abortEvent")
```

```
Converting attribute colour
Converting object editControl
Converting attribute width
Converting attribute height
Converting attribute fontFamily
Converting attribute fontSize
Converting attribute boldFace
Converting attribute colour
Saving the new specification ...
Internal representation saved ...
Constructing the tree of the specification ...
```

Figure 9: Maintaining the lexical level specification

The USE-IT tool implements a semantic network to represent constructs of the lexical layer (see Figure 9). The designer may select to edit an existing lexical specification or create a new one from scratch depending on the requirements of the target application and the user group. In this manner, the designer can disregard several interaction objects, or attributes of interaction objects or specific values of the domain of an attribute which are not suitable or desirable for the particular user group. For instance, cognitive impaired users have difficulties with certain interactive behaviors (i.e. menus, pop-up windows, etc). Consequently, the designer may select to remove these elements from the lexical specification, thus causing the adaptation engine of USE-IT to disregard certain interaction object classes or some of their attributes.

Each abstract interaction object class is related to attributes via semantic relationships. These relationships specify the type of each attribute, as this is important for the adaptation process. Thus, an attribute may be classified either as a general attribute, or an appearance attribute, or a behavior attribute. General attributes are common to all interaction object classes. The general attributes supported in the current version of USE-IT are `inputDevice`, `inputTechnique`, `outputDevice`, `outputTechnique`, `initiationFeddback`, `interimFeedback`, `completionFeddback`. In a future version, it is planed to introduce two additional general attributes, namely `startEvent`, `abortEvent`. During the adaptation process, the semantic network description is used to compile a frame representation of the abstract interaction object class being adapted. Attributes of an object frame are adapted by a recursive call to an algorithm which considers general attributes first, then appearance attributes and finally behavior attributes. An adaptability decision may be established either as a result of a default rule, a preference or a search towards a maximally preferred option.

Defaults and preferences are typically used to adapt simple appearance attributes such as borderSize, fontFamily, typeFace, etc. General and behavior attributes are adapted through reasoning towards a maximally preferred option. Typically, the adaptation of such attributes depends on three types of design constraints, namely:

- device availability;
- user-oriented design constraints which are compiled from the user model;
- task-oriented design constraints which are derived from the task context schema and depict application task requirements.

Device availability is built into the system through a tool which allows the construction of a domain-specific device model (Akoumianakis et al., 1995). More specifically, the designer selects a particular device description (containing all available devices) or builds a new description from scratch. The primitives used to build device models are summarised in Table 3.

`<Device>::-`	`Dname,[HumanControlAct],[ContactSite],[PerformanceParameters], Qualityattribute.`
`<HumanControlAct>::-`	`Movement of one hand | Movement of both hands |Directed eye Gaze | Head and neck movement |Movement of lower limbs |Vocalisations.`
`<ContactSite>::-`	`finger tips of hand |fist|left upper part of head| right upper part of head|top of head| ControlExtender`
`<ControlExtender>::-`	`Hand held pointer|Hand held pen|mouthStick|headStick.`
`<PhysicalAction>::-`	`MotorAttribute| VisualAttribute|HearingAttribute|TactileAttribute| CommunicationAttribute|LearningAttribute.`
`<MotorAttribute>::-`	`Constant|Term.`
`<VisualAttribute>::-`	`Constant|Term.`
`<HearingAttribute>::-`	`Constant|Term.`
`<TactileAttribute>::-`	`Constant|Term.`
`<Com/tionttribute>::-`	`Constant|Term.`
`<LearningAttribute>::-`	`Constant|Term.`
`<QualityAttribute>::-`	`Term.`

Table 3 : A model-theoretic view of input devices

Thus, each device is modelled in terms of pragmatic attributes depicting device operation requirements. A typical device description is shown in the diagram of Figure 10.

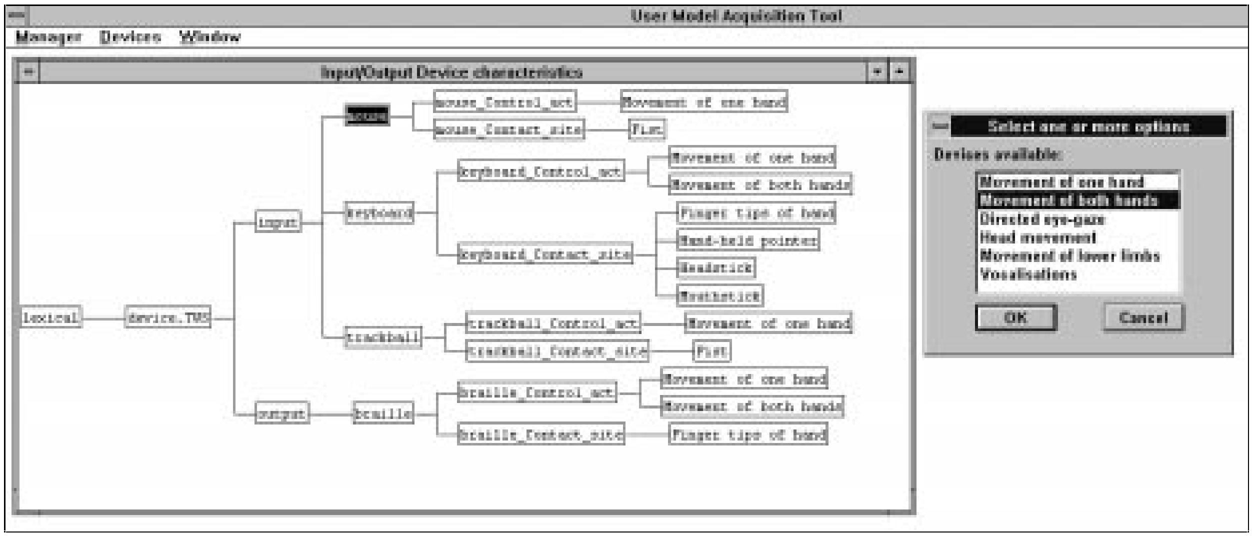
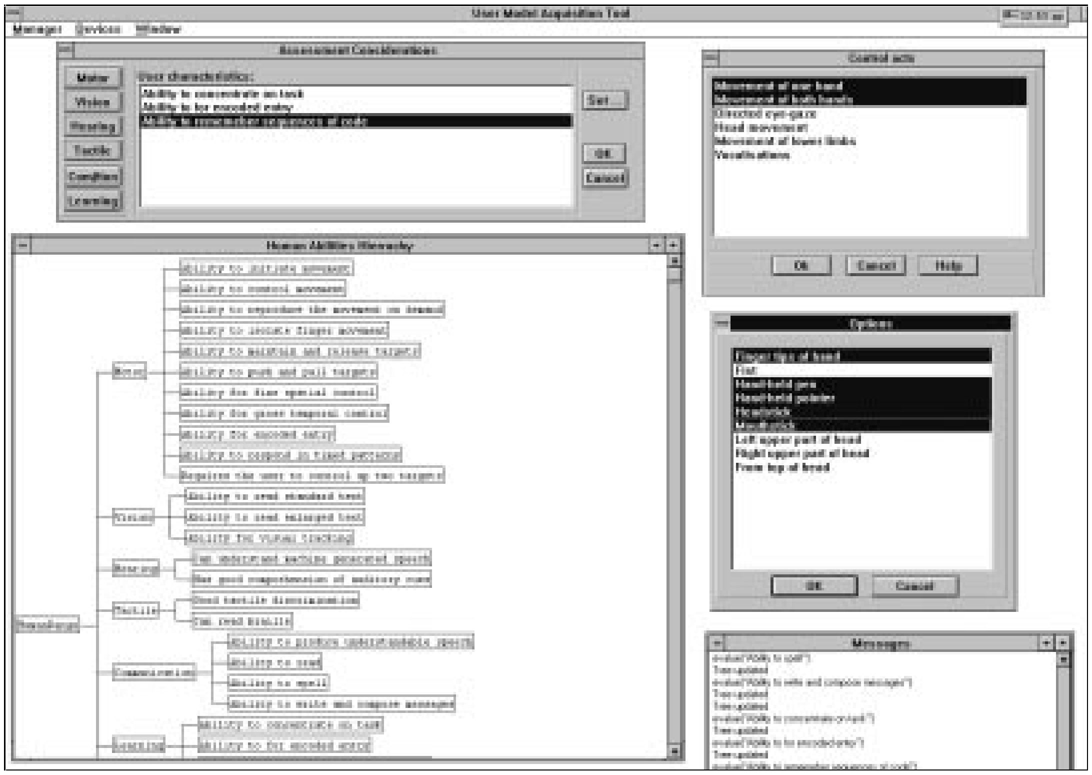


Figure 10: Building device model (Allocating control acts)



User characteristics:
Ability to concentrate on task
Ability to for encoded entry
Ability to remember sequences of code

Ability to initiate movement
Ability to control movement
Ability to reproduce the movement on demand
Ability to isolate finger movement
Ability to maintain and release targets
Ability to push and pull targets
Ability for fine spatial control
Ability for gross temporal control
Ability for encoded entry
Ability to respond in timed patterns
Requires the user to control up two targets
Motor
Ability to read standard text
Ability to read enlarged text
Ability for visual tracking
Vision
Can understand machine generated speech
Has good comprehension of auditory cues
Hearing
Good tactile discrimination
Can read Braille
Tactile
Ability to produce understandable speech
Ability to read
Ability to spell
Ability to write and compose messages
Communication
Ability to concentrate on task
Ability to for encoded entry
Learning

Movement of one hand
Movement of both hands
Directed eye-gaze
Head movement
Movement of lower limbs
Vocalisations

Finger tips of hand
Fist
Hand-held pen
Hand-held pointer
Headstick
Mouthstick
Left upper part of head
Right upper part of head
Front top of head

evalue("Ability to spell")
Tree updated
evalue("Ability to write and compose messages")
Tree updated
Tree updated
evalue("Ability to concentrate on task")
Tree updated
evalue("Ability to for encoded entry")
Tree updated
evalue("Ability to remember sequences of code")

Figure 11: Building a user model using UMAT

A user model is constructed interactively by declaring the abilities possessed by a particular user. There are six classes of abilities, namely motor, visual, hearing, tactile, communication and learning abilities. For each one of those classes the designer can allocate a range of specific ability parameters as required. For example, the specific parameters of motor abilities may contain the identification of the user's reliable control acts, contact sites, but also possession of functional capabilities such as ability to push and pull, ability to perform the control act on demand, etc. During an interactive session, the designer describes a prospective user by instantiating each one of the ability classes, and subsequently, as many of their parameters as required. Figure 11 indicates a typical session with UMAT which allows the designer to build the user's description. The tree depicted in the lower-left hand side of the diagram of Figure 11 represents in a hierarchical form the contents of the current user model.

It is important to mention that the underlying representation of ability classes and parameters forms a network which can be developed from scratch by the designer, according to the requirements of a particular scenario of use. This means that UMAT does not operate upon pre-defined ability classes and corresponding parameters. Instead, it allows the designer to build a desirable description of characteristic abilities which influence the current design scenario. This was necessitated by mainly two reasons. The first relates to the broad range of user characteristics that are usually needed to describe users, and which are not always known or can not be predicted in advance. Consequently, the designer should be allowed to modify and sometimes totally redefine the contents of the knowledge base and the inferencing facilities that have been used in a particular context. The second and most important reason accounts for the fact that existing assessment manuals suggest clusters of context-independent abilities. Thus, they would recommend a scanning device, if the user possessed the abilities of gross temporal control, visual tracking skills, and control movements and contact sites that allow the operation of a switch. However, the switch may be perfectly appropriate in a totally different scenario of use. For instance, a switch may be preferable for a young and computer-illiterate child using an educational software application requiring a small number of selection targets. In this case, the rule which should trigger the use of the switch is a totally different one than in the previous case. Consequently, it becomes apparent that the domain UMAT intends to capture and model is not subject to a single interpretation. This, in turn, necessitates the modifiability of the underlying representation language (i.e. both the constants of the language and the inferencing facilities).

User centered-design constraints are declared by a three-argument predicate:

```
constraint(user,Constituent, assignment)
```

Such constraints are derived automatically by interpreting the contents of the selected device model against the current user model. The interpreter is a routine which translates the disjunctive semantics of a device model into a set of rules and subsequently runs these rules against the current user model. Disjunctions in the device model are due to the fact that a device may be operated with more than one control act and for each control act more than one contact sites may be used. This gives rise to a disjunctive problem description which is translated into a conjunctive formulation by means of compiling rules. Such rules are stored in a file which is subsequently run against the user model. During this process, the translator considers the indifference classes in an ascending order starting with the first indifference class. Thus, at the end of the process, UMAT is able to select the input/output devices that are maximally preferred and can be operated by the user. Accordingly, it derives any additional information regarding other lexical attributes which are dependent upon the selection of an input/output device.

On the other hand, task-oriented constraints are derived from the task context schema descriptions which contain syntactic knowledge about the various dialogue states. A task context schema is a representation language which is used to consolidate application-specific task requirements, in terms of application-specific task context requirements and a set of initial preference and/or indifference expressions. The designer can interactively specify the application requirements of a dialogue state in terms of general characterizations (e.g.

```
/*------------------------ Task context aggregation policy ----------------------------*/
 policy(`Link selection`,
      speed_of_cursor_movement(true),
      continuous,discrete   ).
/*-------------------- End of task context aggregation policy --------------------------*/
/*-----Known Preferences in criterion speed_of_cursor_movement(true) ---------*/
1: preference(`Link selection`,inputDevice,
      speed_of_cursor_movement(true),
      keyboard,mouse ).
2: indifferent(`Link selection`, inputDevice,
      speed_of_cursor_movement(true),
      mouse,   trackball ).
3: preference(`Link selection`, inputDevice,
      speed_of_cursor_movement(true),
      trackball,data_tablet ).
4: indifferent(`Link selection`, inputDevice,
      speed_of_cursor_movement(true),
      data_tablet,   joystick ).
5: indifferent(`Link selection`, inputDevice,
      speed_of_cursor_movement(true),
      joystick,lightpen ).
```

Figure 12: Task context schema

selection, text entry), aggregation criteria (e.g. non-visual interaction) and intentions. Such a design representation is subsequently run against preference constraints (Akoumianakis et al., 1995) which allow the derivation of missing information and the ranking of competing alternatives into indifference classes. A typical task context schema is depicted in the diagram of Figure 12.

4.3. Towards adaptability decisions

To facilitate adaptation decisions based on the three sets of design constraints identified above, a data structure has been developed, which serves the purpose of consolidating the semantics of adaptation of a particular attribute into a formal representation which allows USE-IT to decide on the maximally preferred option. This data structure is referred to as the *adaptability model tree* of an adaptation constituent. An adaptability model tree is attribute specific and, once compiled, it encapsulates all plausible adaptability decisions for a particular attribute of an abstract interaction object class. To demonstrate the details of this data structure, as well as the semantics that it can accommodate, let us consider a hypothetical scenario. Let us assume that the attribute to be adapted is inputDevice and that the user and task oriented constraints are as follows:

$$U_{constraints} = \{ \mathit{keyboard, data\_tablet, joystick} \}$$
$$T_{constraints} = \{\mathit{mouse, trackball , keyboard, data\_tablet} \}$$

Let us further assume that the device availability constraints are:

$$DA_{constraints} = \{\mathit{mouse, trackball , keyboard, data\_tablet, joystick, lightpen} \}$$

Given the above sets of constraints the adaptability model tree for attribute inputDevice is depicted in the diagram of Figure 13.

From this figure, it follows that the total number of branches in an adaptability model tree equals the number of constraint sets. In other words, each branch in the tree corresponds to a constraint set. The intersection of the three branches defines the minimal model tree which satisfies all design constraints. Thus, for the situation described in the diagram of Figure 13, the minimal model tree is defined by the set:

$$MIN_{model} = \{ ((\mathit{input\_device(keyboard), input\_device(data\_tablet)}),$$
$$\mathit{input\_device(keyboard), input\_device(data\_tablet)})\}$$

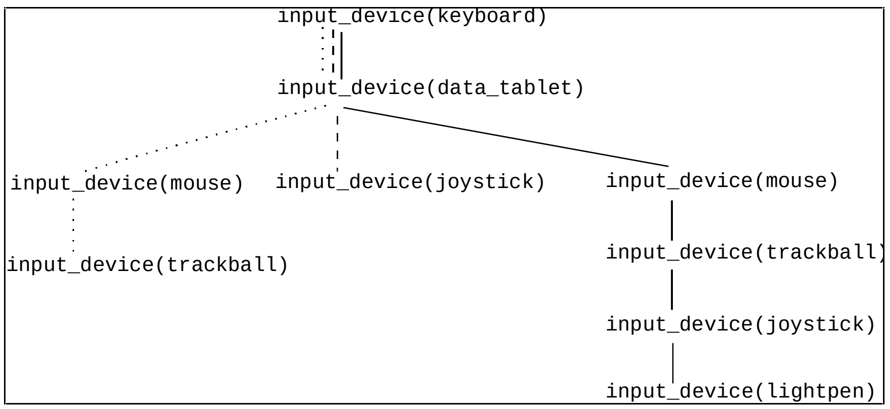


Figure 13: Adaptability model tree for attribute inputDevice

Any one of the elements of this set could be a plausible adaptation for the attribute input_device. However, for the purposes of the present work, USE-IT decides in favor of the solution which preserves maximal multi-modality. Thus, the maximally preferred option is defined by the expression:

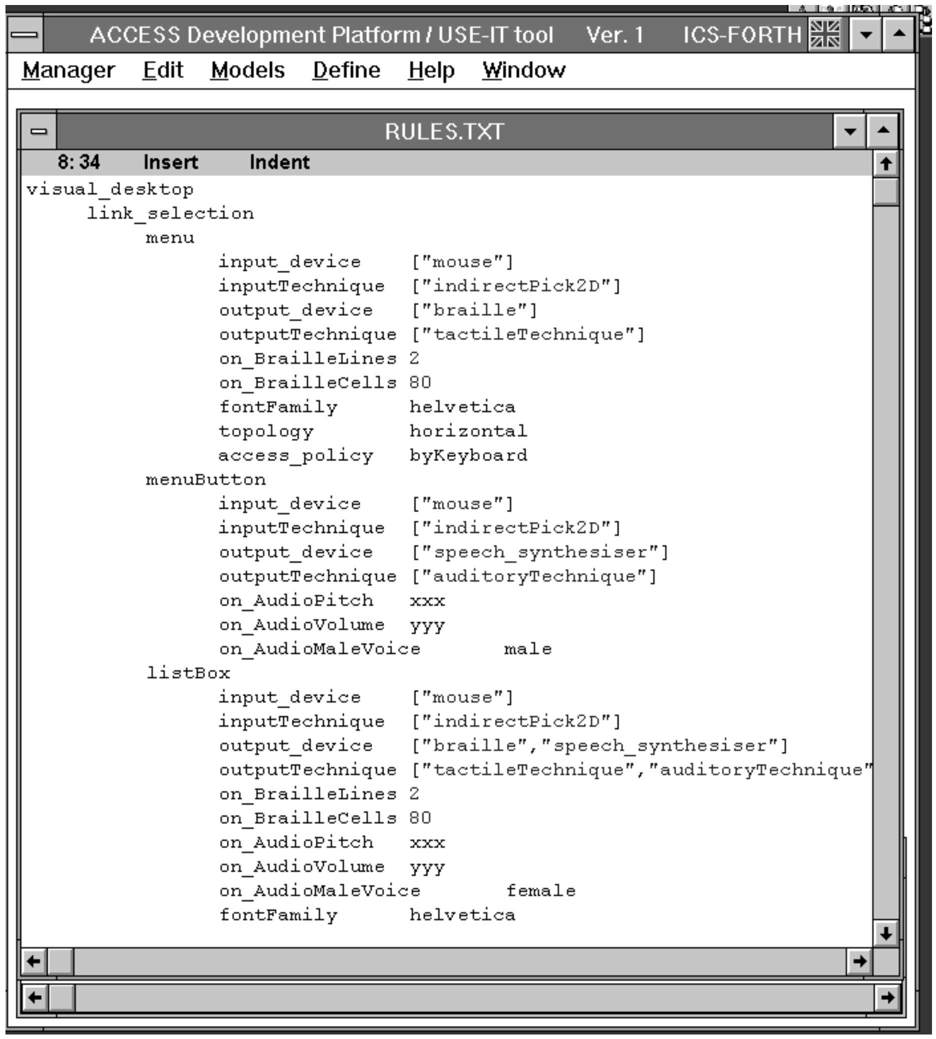
```
input_device(keyboard) ∧ input_device(data_tablet)
```

Consequently, the adaptability decision which is compiled for this attribute is as follows:

`Metaphor.taskContext.Object.`*input_device* = `[keyboard, data-tablet]`

The above procedure is applied for the adaptation of all attributes for which there is no default or preference expression in the corresponding knowledge bases.

Sample output of the USE-IT tool is depicted in the diagram of Figure 14, where some lexical adaptability decisions are listed for abstract interaction object. Currently, USE-IT adapts all abstract interaction object classes of an interaction metaphor assigned by the designer, for each task context of a particular user interface. Additionally, the file depicted in Figure 14 may contain decisions for more than one interaction metaphor if this desired.


ACCESS Development Platform / USE-IT tool Ver. 1 ICS-FORTH
Manager Edit Models Define Help Window
RULES.TXT
8: 34 Insert Indent

```
visual_desktop
     link_selection
          menu
                input_device    ["mouse"]
                inputTechnique  ["indirectPick2D"]
                output_device   ["braille"]
                outputTechnique ["tactileTechnique"]
                on_BrailleLines 2
                on_BrailleCells 80
                fontFamily      helvetica
                topology        horizontal
                access_policy   byKeyboard
          menuButton
                input_device    ["mouse"]
                inputTechnique  ["indirectPick2D"]
                output_device   ["speech_synthesiser"]
                outputTechnique ["auditoryTechnique"]
                on_AudioPitch   xxx
                on_AudioVolume  yyy
                on_AudioMaleVoice       male
          listBox
                input_device    ["mouse"]
                inputTechnique  ["indirectPick2D"]
                output_device   ["braille","speech_synthesiser"]
                outputTechnique ["tactileTechnique","auditoryTechnique"
                on_BrailleLines 2
                on_BrailleCells 80
                on_AudioPitch   xxx
                on_AudioVolume  yyy
                on_AudioMaleVoice       female
                fontFamily      helvetica
```

Figure 14: Output of USE-IT depicting a sample of lexical adaptability rules derived automatically for the task context link_selection.

4.4. Applying adaptability decisions through an interface development framework

Until now, design support has been merely faced as the generation of general high-level design suggestions (such as, for example, hypothetical design guidelines, like use of forms with explicit confirmation for field values, which could be targeted to specific or non-specific application domains). The problem with such approaches for practically supporting the interface design process can be summarized as follows:

I. Guidelines are too specialized and their applicability is limited to particular contexts.
II. Guidelines have been too general and lose power and value when applied for specific contexts.
III. The relevant topic had small or no relevance with the target application domain.
IV. The initial interaction objectives of guidelines were different, even contradictory with the target domain and consequently the guidelines were not applicable.
V. The topic and objectives had strong relevance, however, applying the guidelines directly could be problematic, while an "adaptation" of the original guidelines would be much less than practically trivial.
VI. Different design guidelines for the same topics and objectives give incompatible instructions.
VII. Large number of guidelines.
VIII. Guidelines are not structured in a way reflecting a comprehensive design process.

The previous problems limit considerably the practical integration of such generated design suggestions during the design process. Moreover, the automatic applicability of such design decisions through an interface development system cannot be realized. Currently, there is no support for explicitly incorporating design decisions into the interface development process by means of automatic interpretation and realization of decisions within the resulting interface implementation.

The USE-IT tool has over-passed this difficulty by extending design suggestions to more concrete interface design scenarios. The USE-IT tool has the ability to generate different rules concerning attributes of interaction object classes according to the particular interaction contexts. These decisions can be interpreted and applied automatically during user-computer interaction by interactive applications, which are built through specific interface development systems (see Figure 15). Consequently, such interactive applications practically implement the design decisions generated by the USE-IT tool. This behavior is achieved by the proper synergy of the USE-IT tool with the interface development framework which has to understand and apply decisions provided by the USE-IT tool. This strategy of implementationally separating systems with different roles during the development process has many advantages:

- *Modification independence*, since modifications at one system do not affect the other. In our approach, the communication between the systems is reduced to the file which is produced by the USE-IT tool and read by the interface implementation, while the protocol is mainly the syntax of that file.
- *Implementation independence*, since different programming languages can be employed for different systems. For instance, the USE-IT tool has been developed via the Prolog language, while the interface implementation is provided in the C++ language.

- Design role resolution, since the USE-IT tool is targeted to interface designers and human-factors specialists, while the interface development systems concern interface implementation experts. It should be noted that existing interface development environments usually require that the designer also deals with implementation notations.
- Knowledge reusability, since design decisions for a specific domain can be directly re-used for interactive applications within the same domain. This is possible since the design decisions can be easily transferred to the new interface implementation (the interface implementation will automatically apply the rules).

The interface development systems which has been implemented supports powerful methods for abstraction of interaction objects and interaction techniques. This has been an important feature for practically supporting the design decisions generated by the USE-IT tool which relies upon a sophisticated model of the lexical layer of interaction that is not supported by existing toolkits of interaction objects. It should be noted that with more sophisticated and well structured models of the lexical layer, it is possible to accomplish high quality of adaptability. Consequently, it is critical to have interface development systems which practically provide better organization and decomposition of lexical interaction elements. We have utilized the object abstraction methods of the interface development framework so as to match the structural aspects of the lexical level of interaction as they are realized from the USE-IT tool point of view.

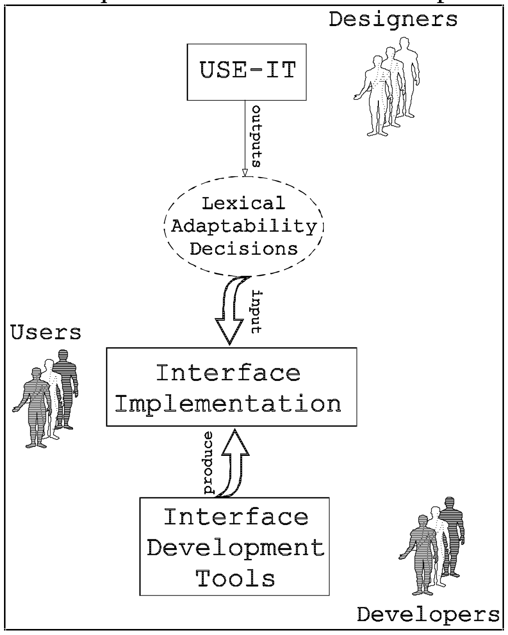


Figure 15: Synergy of USE-IT with interface development tools for automatic application of lexical design decisions

5. SUMMARY AND CONCLUSIONS

In this paper, an attempt has been made to review some of the existing efforts described in the relevant literature, identify their limitations and consider alternative architectural abstractions that may be used to support the design, development and maintenance of adaptation in user interfaces. In this context, adaptation was considered in a broad sense encompassing both adaptability and adaptivity. It was shown how adaptable and adaptive constituents can be integrated in a high level user interface development platform that provides the required design and development support. In addition, the paper described specific developments, undertaken in the context of the ACCESS (TP1001) project of the TIDE Programme of the Commission of the European Union (DG XIII), towards the definition, design and implementation of tools supporting the design and implementation of user-adapted interfaces. More specifically, a user interface design assistant was described which automatically derives adaptation decisions at the lexical level of interaction. These decisions relate to the adaptation of lexical attributes of abstract interaction objects required to ensure the accessibility of the user interface by different user groups, including disabled people. In addition, the structure and capabilities of novel interface development toolkits has been presented. These toolkits are capable of interpreting the adaptation decisions of USE-IT to provide the required development and implementation support of user-adapted interfaces for two interaction metaphors supporting visual and non-visual interaction, respectively.

ACKNOWLEDGMENTS

The present work is carried out in the context of the ACCESS project (TP1001) funded by the TIDE Programme of the Commission of the European Union. Partners in this consortium are: CNR-IROE (Prime Contractor), Italy; ICS-FORTH, Greece; University of Athens, Greece; RNIB, U.K.; SELECO, Italy; MA Systems Ltd., U.K.; Hereward College, U.K.; National Research and Development Centre for Welfare and Health, Finland; VTT, Finland; PIKO Systems, Finland; University of Hertfordshire, U.K.

REFERENCES

Ancieri, F., Dell'Ommo, P., Nardelli, E., Vocca, P (1991): *A user Modeling System*, in Human Aspects in Computing - Design and Use of Interactive Systems and work with terminals, Bullinger (Editor), pp. 440-447, Elservier.

Akoumianakis, D., Stephanidis C. (1995): *Developing domain-specific device representations to facilitate user interface design for disabled people*, submitted for publication.

Benyon, D., Murray, D. (1993): *Adaptive Systems: from intelligent tutoring to autonomous agents*, Knowledge-Based Systems, 6(4), pp.197-219

Bodard, F., Hennebert, A-M., Leheureux, J-M, Provot, I., Vanderdonckt, J. (1994): *A model-based Approach to Presentation: A Continuum from Task Analysis to Prototype*, in Proceedings of Eurographics Workshop on Design, Specification and Verification of Interactive Systems, pp. 25-39.

Brajnik, G., Tasso, C. (1994): *A Shell for Developing Non-Monotonic User Modelling Systems*, International Journal of Human Computer-Studies, 40, pp. 31-62.

Browne, P., D. (1993): *Experiences from the AID project*, in M. Schneider-Hufschmidt, T. Kuhme and U. Mallinowski (Eds.), Adaptive User Interfaces, pp: 69-78, Amsterdam: Elsevier Science Publishers B.V, North-Holland.

Cote-Munoz, A., H. (1993): AIDA: *An Adaptive System for Interactive Drafting and CAD Applications*, in M. Schneider-Hufschmidt, T. Kuhme and U. Mallinowski (Eds.), Adaptive User Interfaces, Amsterdam: Elsevier Science Publishers B.V, North-Holland, pp. 225-240.

Coutaz, J. (1990): *Architecture models for interactive software: Failures and trends*, In Engineering for Human-Computer Interaction, G. Cocton (Ed), North-Holland, pp: 473-490.

Dieterich, H., Malinowski, U., Kuhme, T., Schneider-Hufschmidt, M.(1993): *State of the Art in Adaptive User Interfaces. Adaptive User Interfaces: Principles and Practice*, M. Schneider-Hufschmidt, T Kuhme and U. Malinowski (Eds.), Adaptive User Interfaces, Amsterdam: Elsevier Science Publishers B.V, North-Holland, pp. 13-48.

De Carolis, B., de Rises, F. (1994): *Modelling Adaptive Interaction of OPADE by petri Nets*, SIGCHI, Vol. 26, No. 2, pp. 48-52.

Finin, T. (1989): GUMS: *A General User Modelling Shell*, in User Models in Dialogue Systems, A. Kobsa, W. Wahlster (editors), pp. 411-430.

Kay, J. (1995): *The um toolkit for reusable, long-term user models*, User Modelling and User-adapted Interaction, 4(3).

Kobsa, A., Pohl, W. (1995): *The user modelling shell system BGP-MS*, in User Modelling and User-adapted interaction 4(2), pp. 59-106.

Koller, F. (1993): *A demonstrator based investigation of adaptability*, in M. Schneider-Hufschmidt, T. Kuhme and U. Mallinowski (Eds.), Adaptive User Interfaces, pp. 183-196, Amsterdam: Elsevier Science Publishers B.V, North-Holland.

Lai, K., Malone, T. (19988): *Object Lens: A Spreedsheet for Cooperative Work*, Proc. of the Conference on CSCW, ACM, New York, pp.115-124.

MacLean, A., Carter, K., Lovstrand, L., Moran, T, (1990): *User-Tailorable Systems: Pressing the Issues with Buttons*, CHI’90, ACM, New York, pp. 175-182.

Myers, A., B. (1990): *A new Model for Handling Input*, ACM Transactions on Information Systems, 8(3), pp. 289-320.

Okada (1994): *Adaptation by task intention identification*, in FRIEND 21 Conf. Proc., Japan, 1995.

Orwant, L., J. (1995): *Heterogeneous Learning in the Doppelganger User Modelling System*, in User Modelling and User Adapted Interaction , 4(2), pp: 107-130.

Robertson, G., Henderson, D., Card, S. (1991): *Buttons as First Class Objects on an XDesktop*, UIST '91, ACM, New Yoark, pp. 35-44.

Savidis, A., Stephanidis, C. (1995a): *Developing Dual User Interfaces for Integrating Blind and Sighted Users : The HOMER UIMS"*, in Proceedings of CHI'95 Conference on Human Factors in Computing Systems, pp:106-113, ACM Press.

Savidis, A., Stephanidis, C. (1995b): *Developing Non-Visual Interaction on the basis of the Rooms metaphor*, in Companion of CHI'95 Conference on Human Factors in Computing Systems, pp. 146-147, ACM Press.

Sherman, H., E., Shortliffe, H., E. (1993): *A User-Adaptable Interface to predict Users' Needs*, in M. Schneider-Hufschmidt, T. Kuhme and U. Mallinowski (Eds.), Adaptive User Interfaces, Amsterdam: Elsevier Science Publishers B.V, North-Holland, pp. 285-315

Stephanidis, C., Savidis, A., Akoumianakis, D. (1995): *Towards user interfaces for all*, Conference Proceddings of 2nd TIDE Congress, pp. 167-170.

Stephanidis, C. (1995): *Towards User Interfaces for All: Some Critical Issues*, in Proceedings of HCI International ‘95 Conference on Human Computer Interaction, pp. 137-143, Elsevier.

Stephanidis, C., Savidis, A., Akoumianakis, D. (1996): *Development tools towards User Interfaces for All*, to appear in Internation Journal og Human-Computer Interaction.

Sukaviriya, P., Foley, J (1993): *Supporting Adaptive Interfaces in a knowledge-based user Interface Environment*, In W. D. Gray, W. E. Hefley, and D. Murray (Eds.), Proceedings of the 1993 International Workshop on Intelligent User Interfaces (pp. 107-114), Orlando, FL. New York: ACM Press.

Ten Hagen, P.J.W. (1990): *Critique of the Seeheim model*. In User Interface Management and Design, Duce, D., A., Gomes, M., R., Hopgood, F., R., A., and Lee, J., R. (Eds), Eurographics Seminars, Springer-Verlag.

The UIMS Developers Workshop (1992): *A Metamodel for the run-time architecture of an interactive system*, SIGCHI Bulletin 24, 1.

Vergara, H. (1994): *PROTUM - A Prolog based Tool for User Modelling*, Bericht Nr. 55/94 (WIS-Memo 10), University of Konstanz, Germany.

Zimek (1991): *Design of an adaptable/adaptive UIMS in production*, in Human Aspects in Computing - Design and Use of Interactive Systems and work with terminals, Bullinger (Editor), pp. 748-752, Elservier.