Univ Access Inf Soc (2004) 3: 165–193
DOI 10.1007/s10209-004-0096-8

LONG PAPER

Anthony Savidis · Constantine Stephanidis

Unified user interface development: the software engineering of universally accessible interactions

Published online: 4 May 2004

Abstract In the information society, the notion of "computing-platform" encompasses, apart from traditional desktop computers, a wide range of devices, such as public-use terminals, phones, TVs, car consoles, and a variety of home appliances. Today, such computing platforms are mainly delivered with embedded operating systems (such as Windows CE, Embedded/ Personal Java, and Psion Symbian), while their operational capabilities and supplied services are controlled through software. The broad use of such computing platforms in everyday life puts virtually anyone in the position of using interactive software applications in order to carry out a variety of tasks in a variety of contexts of use. Therefore, traditional development processes, targeted towards the elusive "average case", become clearly inappropriate for the purposes of addressing the new demands for user- and usage-context diversity and for ensuring accessible and high-quality interactions. This paper will introduce the concept of unified user interfaces, which constitutes our theoretical platform for universally accessible interactions, characterized by the capability to self-adapt at run-time, according to the requirements of the individual user and the particular context of use. Then, the unified user interface development process for constructing unified user interfaces will be described, elaborating on the interactive-software engineering strategy to accomplish the run-time self-adaptation behaviour.

Keywords Development processes · Software engineering · Unified user interfaces · User-adapted interfaces · User interface architectures

A. Savidis · C. Stephanidis (✉)
Foundation for Research and Technology - Hellas,
Institute of Computer Science, Science and Technology Park of Crete, GR-71110 Heraklion, Crete, Greece
E-mail: cs@ics.forth.gr
Tel.: +30-2810-391741
Fax: +30-2810-391740

1 Introduction

Today, software products support interactive behaviours that are biased towards the "typical", or "average" able-bodied user, familiar with the notion of the "desktop" and the typical input and output peripherals of the personal computer. This has been the result of software developers' assumptions regarding the target user groups, the technological means at their disposal and the type of tasks supported by computers. Thus, the focus has been on "knowledgeable" workers, capable and willing to use technology in the work environment, to experience productivity gains and performance improvements.

Though the information society is still in its infancy, its progressive evolution has already invalidated (at least some of) the assumptions in the above scenario. The fusion between information technologies, telecommunications and consumer electronics has introduced radical changes to traditional markets and complemented the business demand with a strong residential component. At the same time, the type and context of use of interactive applications is radically changing, due to the increasing availability of novel interaction technologies (e.g., personal digital assistants (PDAs), kiosks, cellular phones and other network-attachable equipment) which progressively enable nomadic access to information.

The above paradigm shift poses several challenges: users are no longer only the traditional able-bodied, skilled and computer-literate professionals; product developers can no longer *know* who their target users will be; information is no longer relevant only to the business environment; and artefacts are no longer bound to the technological specifications of a pre-defined interaction platform. In this context, users are potentially *all* citizens of an emerging information society who demand customized solutions to obtain timely access to virtually any application, from anywhere, at any time.

This paper will introduce the concept of unified user interfaces and point out some of the distinctive properties that render it an effective approach towards universal access within the information society. Subsequently, the unified user interface development approach will be described as an approach conveying a new perspective on the development of user interfaces; it provides a principled and systematic approach towards coping with diversity in the target users groups, tasks and environments of use. In other words, unified user interface development provides a pathway towards accommodating the interaction requirements of the broadest possible end-user population and contexts of use.

The notion of unified user interfaces originated from research efforts aiming to address the issues of accessibility and interaction quality for people with disabilities [1]. The primary intention was to articulate some of the key principles of *design for all* in a manner that would be applicable and useful to the conduct of Human-Computer Interaction (HCI). Subsequently, these principles have been extended, appropriately adapted, compared with existing techniques, intensively tested and validated in the course of several projects, and formulated in an engineering code of practice that depicts a concrete proposition for interface development in the light of universal access.

2 The concept of unified user interfaces

A unified user interface is the interaction-specific software of applications or services which is capable of self-adapting to the individual end user requirements and contexts of use. Such an adaptation may reflect varying patterns of interactive behaviour, at the physical, syntactic or semantic level of interaction, to accommodate specific user- and context-oriented parameters.

Practically speaking, from the end-user point of view, a unified user interface is actually an interface that can automatically adapt to the individual user attributes (e.g., requirements, abilities, and preferences), as well as to the particular characteristics of the usage-context (e.g., computing platform, peripheral devices, interaction technology and surrounding environment). Therefore, a unified user interface realises the combination of:

– *User-adapted behaviour*, i.e., the automatic delivery of the most appropriate user interface for the particular end-user (*user awareness*);
– *Usage-context adapted behaviour*, i.e., the automatic delivery of the most appropriate user interface for the particular situation of use (*usage context awareness*).

Hence, the characterisation "unified" does not have any particular behavioural connotation, at least as seen from an end-user perspective. Instead, the notion of "unification" reflects the specific software engineering strategy needed to accomplish this behaviour, emphasising the proposed development-oriented perspective. More specifically, in order to realise this form of adapted behaviour, a unified user interface reflects the following fundamental development properties:

– It encapsulates alternative dialogue patterns (i.e., implemented dialogue artefacts), for various dialogue design contexts (i.e., a sub-task, a primitive user action, a visualisation), appropriately associated to the different values of user- and usage-context related attributes. The need for such alternative dialogue patterns is dictated by the design process: given any particular design context, for different user- and usage-context attribute values, alternative design artefacts are needed to accomplish optimal interaction;
– It encapsulates representation schemes for user- and usage-context parameters, internally utilising user- and usage-context information resources (e.g., repositories, servers), to extract or to update user- and usage-context information.
– It encapsulates the necessary design knowledge and decision making capability for activating, during run-time, the most appropriate dialogue patterns (i.e., interactive software components), according to particular instances of user- and usage-context attributes.

The property of unified user interfaces to encapsulate alternative, mutually exclusive, design artefacts, which can be purposefully designed and implemented for a particular design context, constitutes one of the main contributions of this research work within the interface software engineering arena. As it will be discussed in subsequent sections, this notion of adaptation has not been supported until now. Previous work in adaptive interfaces has put emphasis primarily on adaptive interface updates, driven from continuous interaction monitoring and analysis, rather than on optimal design instantiation before interaction begins.

A second contribution concerns the advanced technical properties of unified user interfaces, associated with the encapsulation of interactive behaviours targeted to the various interaction technologies and platforms. This requires, from the engineering point of view: (a) the organisation of alternative dialogue components on the basis of their corresponding dialogue design context (i.e., all dialogue alternatives of a particular sub-task are placed around the same run-time control unit); (b) the use of different interface toolkit libraries (e.g., Java for graphical user interfaces (GUIs) and HAWK [33] for non-visual interaction), and (c) embedding alternative dialogue control policies, since the different toolkits may require largely different methods for interaction management (e.g., interaction object control, display facilities, direct access to I/O devices). Those issues have not been explicitly addressed by previous work on adaptive systems; none of the known systems has been targeted in supporting universally accessible interactions, since the various known demonstrators and their particular architectures were clearly focused on the GUI interface domain.

A third contribution concerns the specific architectural proposition that is part of the unified user interface development approach. This proposition presents a code of practice that provides an engineering "blueprint" for addressing the relevant software engineering issues: distribution of functional roles, component-based organisation, and support for extension and evolution. In this context, the specific engineering issues addressed by the proposed code of practice are as follows:

- How are implemented dialogue components organised, where do they reside and which are the algorithmic control requirements in order to accomplish run-time adaptation?
- How should interaction monitoring be organised, and what are the algorithmic requirements for the control of monitoring during interaction?
- Which are the steps involved in employing existing implemented interactive software, and how can it be extended or embedded in the context of a unified user interface implementation?
- Where is user-oriented information stored and in what form; in what format is user-oriented information communicated and between which components?
- What components are required to dynamically identify information regarding the end user?
- During run-time, a unified user interface decides which dialogue components should comprise the delivered interface, given a particular end user and usage context; where is the implementation of such adaptation-oriented decision-making logic encapsulated and in what representation or form?
- How are adaptation decisions communicated and applied and what are the implications for the organisation of implemented dialogue components?
- What are the communication requirements between the various components of the architecture to perform adaptation?
- How is dynamic interface update performed, based on dynamically detected user attributes?
- Which of the components of the architecture are reusable?
- How are the components of the architecture affected if the unified user interface is extended to address additional user- and usage-context parameter values?
- Which implementation practice and tool is better suited for each component of the architecture?

In this paper, the various aspects of the architecture will be discussed on the basis of a specific unified interactive application that is built following this code of practice, namely the AVANTI browser [38]. The target user audience for the AVANTI browser[1] included: able-bodied, motor-impaired and blind users, with different computer literacy expertise, in different contexts of use (office, home, public terminals at stations/airports, PDAs, etc). The interface implementation of the AVANTI browser reached 80 KLOCs (thousands of lines of code), and served as the main testbed for validating the unified user interface development approach. The discussion will be based on specifically chosen adaptation scenarios, currently not implemented by existing Web browsers, elaborating on the way they have been architecturally structured and addressing lower-level engineering strategies.

[1] The AVANTI browser has been developed in the context of the AVANTI project.

3 Related work

Previous efforts in development methods, some of which were explicitly targeted towards universally accessible interactions, fall under four general categories: (a) research work related to user interface tools, concerning prevalent architectural practices for interactive systems; (b) developments targeted to enabling the user interface to dynamically adapt to end users by inferring particular user attribute values during interaction (e.g., preferences, application expertise); (c) special purpose tools and architectures developed to enable the interface to cater for alternative modalities and interaction technologies; and (d) alternative access systems, concerning developments that have been targeted to "manual" or "semi-automatic" accessibility-oriented adaptations of originally non-accessible systems or applications.

3.1 User interface software architectures

Following the introduction of GUIs, early work in user interface software architectures had focused on window managers, event mechanisms, notification-based architectures and toolkits of interaction objects. Those architectural models were quickly adopted by mainstream tools, thus becoming directly encapsulated within the prevailing user interface software and technology (UIST). Today, all available user interface development tools support object hierarchies, event distribution mechanisms, and callbacks at the basic implementation model. In addition to these early attempts in identifying architectural components of user interface software, there have been other architectural models, with a somewhat different focus, which, however, did not gain as much acceptance in the commercial arena as was originally expected. The *Seeheim* Model [16], and its successor, the *Arch* Model [43], have been mainly defined with the aim to preserve the so called "principle of separation" between the interactive and the non-interactive code of computer-based applications. These models became popular as a result of the early research work on User Interface Management Systems (UIMS) [26], while in the domain of universal access systems they do not provide concrete architectural and engineering guidelines other than those related to the notion of separation.

Apart from these two architectural models, mainly referring to the *inter-layer* organization aspects of interactive applications, there have been two other more implementation-oriented models with an object-oriented flavour: the Model View Controller (MVC) model [14, 24] and the Presentation-Abstraction-Control (PAC) model [10]. Those models focus on *intra-layer* software organization policies, by providing logical schemes for structuring the implementation code. All four models, though typically referred to as architectural frameworks, are today considered as meta-models, following the introduction of the term for UIMS models in [43]. They represent abstract families of architectures, since they do not meet the fundamental requirements of a concrete software architecture, as defined by [19]: "an architecture should provide a structure, as well as the interfaces between components, by defining the exact patterns by which information is passed back and forth through these interfaces". Additionally, the key flavour in those approaches is the separation between semantic content and representation form, by proposing engineering patterns enabling a clear-cut separation, so that flexible visualizations and multiple-views can be easily accomplished.

3.2 Adaptive interaction

Most of the existing work regarding system-driven, user-oriented adaptation concerns the capability of an interactive system to dynamically (i.e., during interaction) detect certain user properties, and accordingly decide various interface changes. This notion of adaptation falls in the *adaptivity* category, i.e., adaptation performed after the initiation of interaction, based on interaction monitoring information; this has been commonly referred to as *adaptation during use*. Although adaptivity is considered a good approach to meeting diverse user needs, there is always the risk of confusing the user with dynamic interface updates. Work on adaptive interaction has addressed various key technical issues concerning the derivation of appropriate constructs for the embodiment of adaptation capabilities and facilities in the user interface; most of those technical issues, if relevant to unified user interfaces, will be appropriately discussed in subsequent sections of this paper.

Overall, in the domain of adaptive systems, there have been mainly two types of research work: (a) theoretic work on architectures; and (b) concrete work regarding specific systems and components demonstrating adaptive behaviour. On the one hand, theoretic work addressed high-level issues, rather than specific engineering challenges, such as the ones identified earlier in this paper. Those architectures convey useful information for the general system structure and potential control flow, mainly helping to understand the system as a whole. However, since they are mainly conceptual models, they cannot be deployed to derive the detailed software organisation and the regulations for run-time coordination, as it is needed in the software implementation process.

On the other hand, developments in adaptive systems have been mainly focused on particular parts or aspects of the adaptation process, such as user modeling or decision-making, and less on usage-context adaptation or interface component organisation. Additionally, the inter-linking of the various components, together with the corresponding communication policies employed, primarily reflected the specific needs of the particular systems developed, rather than a generalised approach facilitating re-use. Finally, while most known adaptive systems addressed the issue of dynamic interface adaptation, there has been no particular attention to the notion of individualised dynamic interface assembly before initiation of interaction. This implies that the interface can be structured on-the-fly by dialogue components maximally fitting end-user and usage-context attributes.

This notion of optimal interface delivery before interaction is initiated constitutes the most important issue in universally accessible interactions; unless the initial interface delivered to the user is directly accessible, there is no point to apply adaptivity methods, since the latter assumes that interaction is already taking place.

3.2.1 Which architecture for adaptivity?

Although concrete software architectures for adaptive user interfaces have not been clearly defined, there exist various proposals as to what should be incorporated in a computable form into an adaptive interactive system. In [11], these are characterised as "categories of computable artefacts", and are summarised as being "the types of models that are required within a structural model of adaptive interactive software". An appropriate reference to [18] is also made, regarding structural interface models:

> Structural descriptive models of the human-computer interface...serve as frameworks for understanding the elements of interfaces and for guiding the dialogue developer in their construction.

However, developers require concrete software architectures for structuring and engineering interactive systems, and software systems in general. From this point if view, the information provided in [11] does not fulfil the requirements of an interface structural model, as defined in [18], nor of a software architecture, as defined in [19] and [25]. This fact supports the initial argument that a concrete, generic architectural framework for adaptive interfaces, and automatically adapted interactions as a broader category, is not yet available. This argument will be further elaborated on, as various aspects of existing work on adaptive interfaces are incrementally analysed in the subsequent sections.

3.2.2 User models versus user modeling frameworks

In all types of systems aiming to support user adaptivity in performing certain tasks, both embedded user models and user-task models have played a significant role. In [23], an important distinction is made between the user modeling component, encompassing methods to represent user-oriented information, and the particular user models as such, representing an instance of the knowledge framework for a particular user (i.e., an individual user model), or user group (i.e., a stereotype model). However, this distinction explicitly associates the user model with the modeling framework, thus necessarily establishing a dependency between the adaptation-targeted decision-making software (which would need to process user models) and the overall user-modeling component. This remark reveals the potential architectural hazard of rendering an adaptive system "monolithic": since the user model is linked directly with the modeling component, and decision-making is associated with user models, it may be deemed necessary or appropriate that all such knowledge categories be physically located together.

More recent work has reflected the technical benefits of physically splitting the user-modeling component from the adaptive inference core, putting the responsibility of decision making directly to applications, while enabling remote sharing of user modeling servers by multiple clients [22].

3.2.3 Alternative dialogue patterns and the need for abstraction

The need for explicit design as well as the run-time availability of design alternatives has been already identified in the context of interface adaptation [5]. In view of the need for managing alternative interaction patterns, the importance of abstractions has been identified, starting from the observation that design alternatives constructed with an adaptation perspective are likely to exhibit some common dialogue structures. In [7] it is pointed out that "flexible abstractions for executable dialogue specifications" are a "necessary condition for the success of adaptable human-computer interfaces". This argument implies that an important element in the success of adaptive systems is the provision of implemented mechanisms of abstraction in interactive software, allowing the flexible run-time manipulation of dialogue patterns.

3.2.4 Dynamic user attribute detection

The most common utilization of internal dialogue representation has involved the collection and processing of interaction monitoring information. Such information, gathered at runtime, is analysed internally (through different types of knowledge processing) to derive certain user attribute values (not known prior to the initiation of interaction), which may drive appropriate interface adaptivity actions. A well-known adaptive system employing such techniques is MONITOR [3]. Similarly, for the purpose of dynamic detection of user attributes, a monitoring component, in conjunction with a UIMS, is employed in the AIDA system [9]. An important technical implication in this context is that dialogue modeling must be combined with user models. Thus, as discussed earlier, it becomes inherently associated with the user-modeling component, as well as with adaptation-targeted decision-making software. Effectively, this "biases" the overall adaptive system architecture towards a monolithic structure, turning the development of adaptive interface systems into a rather complicated software engineering process. It is argued that such an engineering tactic, placing all those components together within a monolithic system implementation, is a less than optimal architectural option. Moreover, considering that this approach has been adopted to address the issue of dynamic user attribute detection, it will be shown that a more flexible, distributed, and re-use enabled engineering structure can be adopted to effectively pursue the same goal.

This argument is also supported by the fact that in most available interactive applications, internal executable dialogue models exist only in the form of programmed software modules. Higher-order executable dialogue models (which would reduce the need for low-level programming), as those previously mentioned, have been supported only by research-oriented UIMS tools. Conversely, the outcome of interface development environments, like Visual Basic, is, at present, in a form more closely related to the implementation world, rendering the extraction of any design-oriented context difficult or impossible. Hence, on the one hand, dynamic user attribute detection will necessarily have to engage dialogue-related information, while on the other hand, it is unlikely that such required design information is practically extractable from the interaction control implementation.

3.2.5 System-initiated actions to perform adaptivity

The final step in a run-time adaptation process is the execution of the necessary interface updates at the software level. In this context, four categories of system-initiated actions to be performed at the dialogue control level have been distinguished [8], for the execution of adaptation decisions: (i) *enabling* (i.e., the activation or deactivation of dialogue components); (ii) *switching* (i.e., selecting one from various alternative pre-configured components); (iii) *re-configuring* (i.e., modifying dialogue by using pre-defined components); and (iv) *editing* (i.e., no restrictions on the type of interface updates). The preceding categorization represents more a theoretical perspective, rather than reflecting an interface engineering one. Furthermore, the term "component" denotes mainly visual interface structures, rather than referring to implemented sub-dialogues, including physical structure and/or interaction control.

In this sense, it is argued that it suffices to define only two action classes, applicable on interface components: (a) *activate* components; and (ii) *cancel* activated components (i.e., deactivate). As it will be discussed, these two actions directly map to the implementation domain (i.e., activation means "instantiation" of software objects, while cancellation means "destruction"), thus considerably downsizing the problem of modeling adaptation actions.

3.2.6 Structuring dialogue implementation for adaptivity

The notion of *interface component* refers to implemented sub-dialogues provided by means of pre-packaged, directly deployable, software entities [37]. Such entities increasingly become the basic building blocks in a component-based software assembly process, highly resembling the hardware design and manufacturing process. The need for configurable dialogue components has been identified in [8], as a general capability of interactive software to visualize some important implementation parameters, through which the flexible fine-tuning of interactive behaviours may be performed at runtime.

However, the analysis in [8] is based on a theoretical ground, and mainly identifies requirements, without proposing specific approaches to achieving this type of desirable functional behaviour. For instance, the proposed distinction among "scalar", "structured" and "higher-order" objects does not map to any interface engineering practice. Moreover, the definition of adaptation policies as "changes" at different levels neither provides any concrete architectural model, nor reveals any useful implementation patterns. The results of such theoretical studies are good for understanding the various dynamics involved in adaptive interaction; however, they do not provide added-value information for engineering adaptive interaction.

It can be concluded from the above that the incorporation of adaptation capabilities into interactive software is far from trivial and cannot be attained through the existing, traditional approaches to software architecture. Therefore, there is a genuine requirement for the definition of a new software architecture that can accommodate the adaptation-oriented requirements of unified user interfaces. Such an architectural framework is described in the next section.

3.3 Multi-modal interfaces and abstract interaction objects

Work on multi-modal interaction has emphasized the identification of models, architectures and tools that enable the interface to dynamically cater for the presence of multiple I/O modalities in enabling the user to perform the same task. The relationship between multi-modal interfaces and universally accessible interactions has been identified quite early, in particular with the introduction of meta-widgets [4], as a method to separate the logical aspects of interaction objects from the physical form. This would enable an interface to employ, during interaction, alternative I/O modalities, so that accessibility at the lexical level of interaction can be enabled. Similar concepts, emphasising the need for abstract objects, have been expressed in [20], where a possible architectural profile of a toolkit supporting abstract objects is proposed. The notion of virtual objects [29], supporting open and extensible policies for mapping to physical I/O modalities with fine-grain programming support, has been proposed together with full tool support, in the context of the HOMER UIMS [31]; this UIMS introduced the concept of dual interfaces, in which a single interface is built by utilising, within the same software implementation, multiple interaction technologies—Windows, Xt/Xaw and Comonkit [30], taking advantage of control sharing, abstraction and toolkit integration mechanisms.

Although the work on multi-modal interfaces has been a serious advancement towards universally accessible interactions, it has been primarily targeted to the lexical level of interaction and the issue of accessibility. When it comes to user- and usage-context oriented adaptation, those approaches do not offer any particular architectural and engineering strategy. For instance, there are clearly no answers to critical questions such as where the alternative implemented dialogue components should reside, at which point dynamic activation/deactivation is controlled, where decision making is performed, where user-oriented information resides and in what form, where interface design information is stored and in what form, etc.

3.4 Alternative access systems

The typical technical approach of alternative access systems is to "intervene" at the level of the particular interactive application environment (e.g., MS Windows, or the X Windowing system) in which inaccessible applications are deployed, and produce appropriate software and hardware technology so as to make that environment alternatively accessible (i.e*, environment level adaptation*). In theory, potentially all applications running through that interactive environment can be made accessible. The typical architecture of those systems is indicated in Fig. 1. Such systems usually rely on well-documented and operationally reliable software infrastructures, supporting the effective and efficient extraction of dialogue primitives during user-computer interaction. Such dynamically extracted dialogue primitives are stored in a dynamically maintained so-called off-screen model (OSM), that is used to reproduce the dialogue primitives at runtime, to alternative input/output forms, directly supporting user access, by employing alternative accessible interaction techniques and devices (e.g., scanning keyboards, cursor routing,

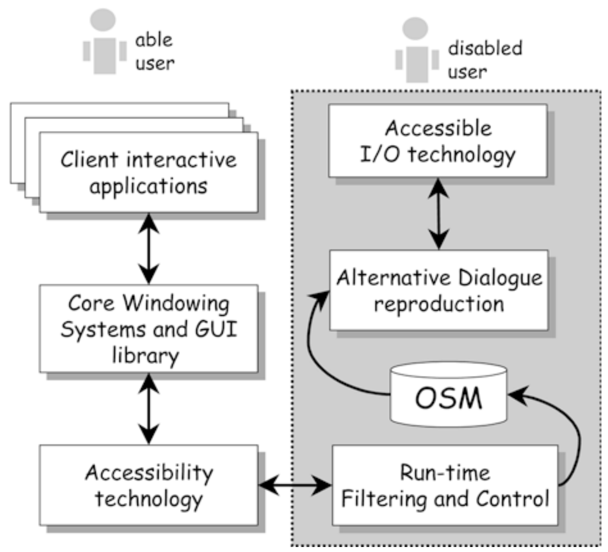

able user
disabled user
Client interactive applications
Core Windowing Systems and GUI library
Accessibility technology
Accessible I/O technology
Alternative Dialogue reproduction
OSM
Run-time Filtering and Control

Fig. 1 The typical architecture of alternative access systems

mouse emulators, etc.). Examples of such software infrastructures are the Active Accessibility Technology by Microsoft, the Java Accessibility Technology by JavaSoft, and the GNOME Accessibility Framework by the GNOME Foundation.

The main disadvantage of such automatic interface reproduction methods is the limited interface quality accomplished. This is due to the fact that such an automatic reproduction is merely based on the heuristic transformation of the inaccessible interface design. However, to achieve a high quality of interaction, it is necessary to drive explicit interface design-, implementation- and evaluation- processes, for the particular applications, specifically addressing the needs of the target user population (with disability). Effectively, less than optimal interaction can thus be accomplished, while in some cases there are dialogue artefacts that cannot be either filtered or reproduced (e.g., direct manipulation rubber-banding dialogues, multi-media interfaces, visualisations, etc.). For instance, in the context of non-visual interfaces, the need for non-visual user interfaces to be more than automatically generated adaptations of visual dialogues has been identified in [29]. Previous work in the domain of alternative access systems included early research systems produced by the GUIB Project [17], auditory GUIs [35], and the Mercator Project [27], as well as commercial systems like ASAW (Automatic Screen Access for Windows), by MicroTalk, and IN CUBE Enhanced Access (relying upon speech input and other commercial screen readers), by Command Corp, Inc.

3.5 Model-based interfaces

Model-based interface development tools are, in principle, a very promising category of interface tools for universally accessible interactions, since they potentially incorporate computable user- and design- models, even though in most previous model-based tools such as UIDE [12] and Humanoid [42], the emphasis was on application and dialogue modeling rather than user- and usage-context modeling. Model-based tools, such as, for example, Mobi-D [28], have tried to incorporate methods for mapping user models to dialogue models, by enabling developers to describe user- and application-model instances, while letting the tool (based on heuristics mapping rules) produce a working interface version. However, the capabilities for universally accessible dialogue design and implementation are rather limited: on the one hand, the capability for designing and implementing new artefacts is restricted in comparison to commercial graphic construction tools (such as Visual Basic), while, on the other hand, all available methods are bound to GUIs. From the run-time point of view, model-based tools do not incorporate the capability to produce and assemble alternative interface versions ‘‘on-the-fly’’, based on model instances supplied as an input. In principle, model-based tools provide the means for specifying the logic of model mapping; i.e., how from one model domain (such as a user-model), one can map to another model domain (such as the dialogue model).

However, the role of user modeling as a run-time decision factor that can directly affect the dialogue style selection, even for the same dialogue model context (a run-time variation of the dialogue-model, i.e., polymorphism), is clearly the most important missing ingredient in model-based approaches. Furthermore, from the software-interface engineering point of view, model-based tools do not propose a general-purpose development strategy (one that can be widely applied with commonly used programming languages). Instead, they appear as stand-alone non-interoperable research tools. As a consequence, the main disadvantage of model-based, tools, from a software engineering viewpoint, is their monolithic nature. It is argued that the emphasis of future work should be shifted towards model-based development allowing the open employment of diverse tools. Each such tool should serve a distinctive purpose in the interface development process, and it should interoperate on the basis of a common architectural structure, specific development roles, and well-defined protocols for information exchange.

4 Unified user interface development

4.1 The strategy

A unified user interface consists of run-time components, each with a distinctive role in performing at runtime an interface assembly process, by selecting the most appropriate dialogue patterns from the available implemented design space (i.e., the organised collection of all dialogue artefacts produced during the design phase). A unified user interface does not constitute a single software system, but becomes a distributed architecture consisting of independent inter-communicating components, possibly implemented with different software methods/tools and residing at different physical locations. These components cooperate together to perform adaptation according to the individual end-user attributes and the particular usage context. At run-time, the overall adapted interface behaviour is realised by two complementary classes of system initiated actions:

a. adaptations driven from initial user- and context-information, acquired without performing interaction monitoring analysis (i.e., what is "known" before starting observing the user or the usage-context); and
b. adaptations decided on the basis of information inferred or extracted by performing interaction monitoring analysis (i.e., what is "learnt" by observing the user or the usage-context).

The former behaviour is referred to as *adaptability* (i.e., initial automatic adaptation, performed before initiation of interaction) reflecting the capability of the interface to proactively and automatically tailor itself to the attributes of each individual end user. The latter behaviour is referred to as *adaptivity* (i.e., continuous automatic adaptation), and characterizes the capability of the interface to cope with the dynamically changing/evolving characteristics of users and usage contexts. Adaptability is crucial to ensure accessibility, since it is essential to provide, before the initiation of interaction, a fully accessible interface instance to each individual end user. Adaptivity can be applied only on accessible running interface instances (i.e., ones with which the user is capable of performing interaction), since interaction monitoring is required for the identification of changing or emerging decision parameters that may drive dynamic interface enhancements.

The complementary roles of adaptability and adaptivity are depicted in Fig. 2, while the key differences among these two adaptation methods are illustrated in Table 1. This fundamental distinction is made due to the different run-time control requirements between those two key classes of adaptation behaviours, requiring different software engineering policies.

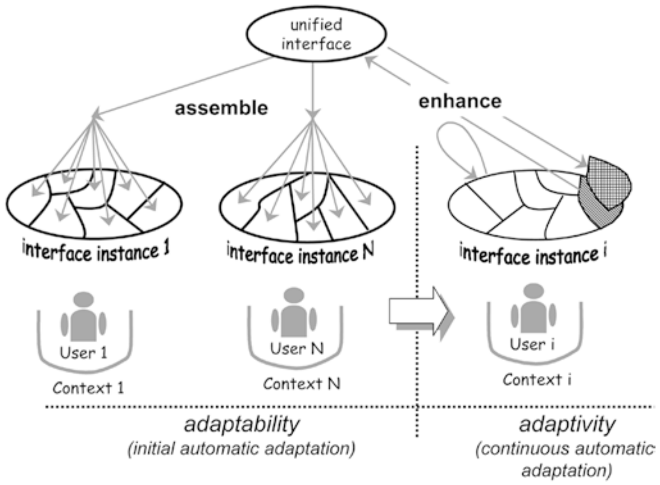


Fig. 2 The complementary roles of adaptability and adaptivity

4.2 The unified user interface software architecture

In this section, the detailed run-time architecture for unified user interfaces will be discussed, in compliance with the definitions of architecture provided by the Object Management Group (OMG) [25], and [19], according to which an architecture should supply an organisation of components, a description of functional roles, detailed communication protocols or appropriate application programming interfaces (APIs), and key component implementation issues. Following the presentation of the architecture within the next section, concrete examples from the AVANTI browser will be illustrated, by describing: (a) how their implementation is split into the different architectural components; (b) what type of implementation approach has been taken in each different component; and (c) how the various components communicate (i.e, control flow) to accomplish the appropriate adaptation behaviour.

Firstly, an outline of the adopted architectural components will provide information regarding: (a) the functional role; (b) the run-time behaviour; (c) the encapsulated context; and (d) the implementation method. The components of the unified user interface architecture are (see Fig. 3):

– The Dialogue Patterns Component (DPC);
– The Decision Making Component (DMC);
– The User Information Server (UIS);
– The Context Parameters Server (CPS).

Table 1 Key differences between adaptability and adaptivity in the context of unified user interfaces

Adaptability	Adaptivity
1. User- and usage-context attributes are considered known prior to interaction.	**1.** User- / usage-context attributes are dynamically inferred / detected.
2. "Assembles" an appropriate initial interface instance for a particular end-user and usage-context.	**2.** Enhances the initial interface instance already "assembled" for a particular end-user and usage-context.
3. Works before interaction is initiated.	**3.** Works after interaction is initiated.
4. Provides a user-accessible interface.	**4.** Requires a user-accessible interface

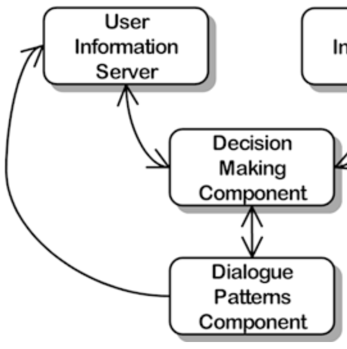

User Information Server
Context Information Server
Decision Making Component
Dialogue Patterns Component

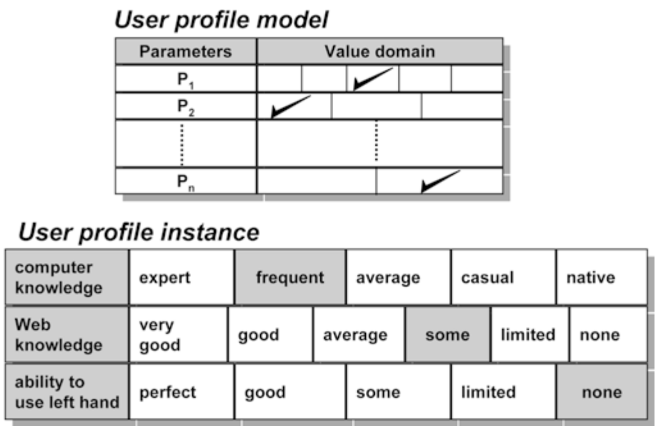
Fig. 3 The components of the unified user interface architecture

4.2.1 The UIS

4.2.1.1 The functional role The functional role supplies user attribute values: (i) known off-line, without performing interaction monitoring analysis (e.g., motor/sensory abilities, age, nationality, etc.); and (ii) detected on-line, from real-time interaction-monitoring analysis (e.g., fatigue, loss of orientation, inability to perform the task, interaction preferences, etc.)

4.2.1.2 The run-time behaviour Run-time behavior plays a two-fold role: (i) it constitutes a server that maintains and provides information regarding individual user profiles; and (ii) encompasses user representation schemes, knowledge processing components and design information, to dynamically detect user properties or characteristics.

4.2.1.3 The encapsulated content This component may need to employ alternative ways of representing user-oriented information., a repository of user profiles serves as a central database of individual user information (i.e., the registry). In Fig. 4, the notion of a profile structure and a profile instance, reflecting a typical list of typed attributes is shown; this model, though quite simple, is proved in real practice to be very powerful and flexible (can be stored in a database, thus turning the profile manager to a remotely accessed database). Additionally, more sophisticated user representation and modeling methods can be also employed, including support for stereotypes of particular user categories. In case dynamic user attribute detection is to be supported, the content may include dynamically collected interaction monitoring information, design information and knowledge processing components, as it is discussed in the implementation techniques. Systems such as BGP-MS [21], PROTUM [44], or USE-IT [2] encompass similar techniques for such intelligent processing.


User profile model

Parameters	Value domain
P_1	✓
P_2	✓
⋮	⋮
P_n	✓

User profile instance

computer knowledge	expert	▨ frequent	average	casual	native	
Web knowledge	very good	good	average	▨ some	limited	none
ability to use left hand	perfect	good	some	limited	▨ none	

Fig. 4 The notion of a profile structure and a profile instance

4.2.1.4 Implementation From a knowledge representation point of view, static or pre-existing user knowledge may be encoded in any appropriate form, depending on the type of information the user information server should feed to the decision making process. Moreover, additional knowledge-based components may be employed for processing retrieved user profiles, drawing assumptions about the user, or updating the original user profiles. In Fig. 5, the internal architecture of the UIS employed in the AVANTI browser is presented. It should be noted that the first version of the AVANTI browser produced in the context of the AVANTI Project employed BGP-MS [22] for the role of the UIS. The profile manager has been implemented as a database of profiles. The two other sub-systems, (i.e., the monitoring manager, the modeling and the inference) are needed only in case dynamic user attribute detection is required.

The interaction monitoring history has been implemented as a time-stamped list of monitoring events (the structure of monitoring events is described in the analysis of communication semantics) annotated with simple dialogue design context information (i.e., just the subtask name). In the user models, all the types of dynamically detected user attributes have been identified (e.g., inability to perform a task, loss of orientation—those were actually the two dynamically detectable attributes required by the design in the AVANTI browser). Each such attribute is associated with its corresponding behavioural action patterns. In the specific case, the representation of the behavioural patterns has been implemented together with the pattern-matching component, by means of state automata. For instance, one heuristic pattern to detect loss of orientation has been defined as "*the user moves the cursor inside the Web-page display area, without selecting a link, for more than N seconds*". The state automaton starts recording mouse moves in the page area, increasing appropriately a weight variable and a probability value, based on incoming monitored mouse moves, while finally triggering detection when no intermediate activity is successfully performed by the user. This worked fine from an implementation point of view. However, all such heuristic assumptions had to be extensively verified with real users so as to assert the relationship between the observable user behaviour and the particular inferred user attributes. This is a common issue in all

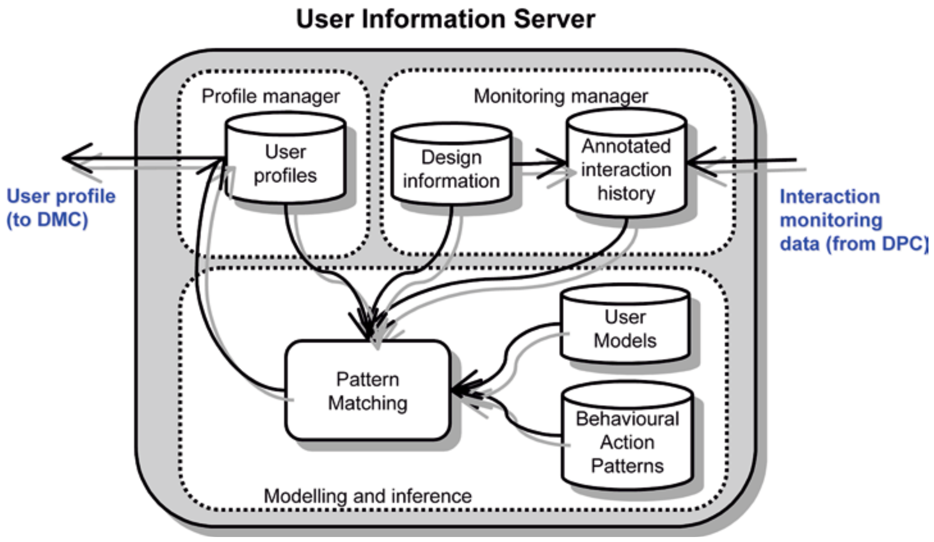


Fig. 5 The internal architecture of the UIS employed in the AVANTI browser

adaptive systems that employ heuristics for detecting user attributes at runtime, practically meaning that the validity of the "assumptions inferred" is dependent on the appropriateness of the specific user action patterns chosen.

4.2.2 The CPS

4.2.2.1 The functional role The purpose of this component is to supply context attribute values (machine and environment) of two types: (i) (potentially) invariant, meaning unlikely to change during interaction, e.g., peripheral equipment; and (ii) variant, dynamically changing during interaction (e.g., due to environment noise, or the failure of particular equipment, etc.). This component is not intended to support device independence, but to provide *device awareness*. Its purpose is to enable the DMC to select those interaction patterns, which, apart from fitting the particular end-user attributes, are also appropriate for the type of equipment available on the end-user machine.

4.2.2.2 The run-time behaviour The usage-context attribute values are communicated to the DMC before the initiation of interaction. Additionally, during interaction, some dynamically changing usage-context parameters may also be fed to the DMC for decisions regarding *adaptive* behaviour. For instance, let us assume that the initial decision for selecting feedback leads to the use of audio effects. Then, the dynamic detection of an increase in environmental noise may result in a run-time decision to switch to visual feedback (the underlying assumption being that such a decision does not conflict with other constraints).

4.2.2.3 The encapsulated content This component encompasses a listing of the various invariant properties and equipment of the target machine (e.g., hand-held binary switches, a speech synthesiser for English, a high resolution display (mode 16bits, 1024×768), a noisy environment, etc.). In this context, the more information regarding the characteristics of the target environment and machine is encapsulated, especially concerning I/O devices, the better adaptation can be achieved (information initially appearing redundant is likely to be used in future adaptation-oriented extensions).

4.2.2.4 The implementation The registry of environment properties and available equipment can be implemented easily as a profile manager in the form of a database. Such information will be communicated to the DMC as attribute/value pairs. However, if usage-context information is to be dynamically collected, such as environment noise, or reduction of network bandwidth, the installation of proper hardware sensors or software monitors becomes mandatory.

4.2.3 The DMC

4.2.3.1 The functional role The role of this component is to decide, at runtime, the necessary adaptability and adaptivity actions, and to subsequently communicate those to the DPC (the latter being responsible for applying adaptation-oriented decisions).

4.2.3.2 The run-time behaviour To decide adaptation, this component performs a kind of rule-based knowledge processing, so as to match end-user- and usage-context attribute values to the corresponding dialogue artefacts, for all the various dialogue contexts.

4.2.3.3 The encapsulated content This module encompasses the logic for deciding the necessary adaptation actions, on the basis of the user- and context- attribute values, received from the UIS and the CPS, respectively.

Such attribute values will be supplied to the DMC, prior to the initiation of interaction within different dialogue contexts (i.e., initial values, resulting in initial interface adaptation), as well as during interaction (i.e., changes in particular values, or detection of new values, resulting in dynamic interface adaptations).

In the proposed approach, the encapsulated adaptation logic should reflect pre-defined decisions during the design stage. In other words, the inference mechanisms employ well-defined decision patterns that have been validated during the design phase of the various alternative dialogue artefacts. In practice, this approach leads to a rule-based implementation, in which embedded knowledge reflects adaptation rules that have been already constructed and documented as part of the design stage. This decision-making policy is motivated by the assumption that if a human designer cannot decide upon adaptation for a dialogue context, given a particular end-user and usage-context, then a valid adaptation decision can not be taken by a knowledge-based system at runtime. Later in this paper, while discussing some implementation details of the AVANTI browser, specific excerpts from the rule base of the decision engine will be discussed.

4.2.3.4 Implementation The first remark regarding the implementation of decision making concerns the apparent "awareness" regarding: (a) the various alternative dialogue artefacts (how they are named—e.g., *virtual keyboard*, for which dialogue context they have been designed—e.g., *http address text field*); (b) user- and usage-context attribute names, and their respective value domains (e.g., attribute *"age"*, being *integer* in *range 5...110*).

The second issue concerns the input to the decision process, being individual user- and usage-context attribute values. Those are received at run-time from both the UIS and the CIS, either *by request* (i.e., the DMC takes the initiative to request the end-user and usage-context profile at start-up to draw adaptability decisions), or *by notification* (i.e., when the UIS draws assumptions regarding dynamic user attributes, or when the CPS identifies dynamic context attributes).

The third issue concerns the format and structure of knowledge representation. In all developments that we have carried out, it has been proven that a rule-based logic implementation is practically adequate. Moreover, all interface designers engaged in the design process emphasised that this type of knowledge representation approach is far closer to their own way of rule-based thinking in deciding adaptation. This remark has led to excluding, at a very early stage, other possible approaches, such as heuristic pattern matching, weighting factor matrices, or probabilistic decision networks.

The final issue concerned the representation of the outcomes of the decision process in a form suitable for being communicated and easily interpreted by the DPC. In this context, it has been practically proven that two categories of dialogue control actions suffice to communicate adaptation decisions: (i) *activation* of specific dialogue components; and (ii) *cancellation* of previously activated dialogue components.

These two categories of adaptation actions provide the expressive power necessary for communicating the dialogue component manipulation requirements that realise both adaptability and adaptivity (see Table 2). (In Table 3, the decision-making logic is defined). Substitution is modelled by a message containing a series of *cancellation* actions (i.e., the dialogue components to be substituted), followed by the necessary number of *activation* actions (i.e., which dialogue components to activate in place of the cancelled components). Therefore, the transmission of those commands in a single message (i.e., *cancellation* actions followed by *activation* actions) is to be used for implementing a substitution action. The need to send in one message packaged information regarding the cancelled component, together with the components which take its place, emerges when the implemented interface requires knowledge of all (or some) of the newly created components during interaction. For instance, if the new components include a container (e.g., a window object) with various embedded objects, and if upon the creation of the container information on the number and type of the particular contained objects is needed, it is necessary to ensure that all the relevant information (i.e., all engaged components) is received as a single message. It should be noted that, since each *activation/cancellation* command always carries its target UI component identification (see Table 4), it is possible to engage in substitution requests components that are not necessarily part of the same physical dialogue artefact. Also, the decision to apply substitution is the responsibility of the DMC.

Table 2 The user interface component manipulation requirements (left), for adaptability and adaptivity, and their expression via cancellation/activation adaptation actions

	Adaptability	**Adaptivity**
***Initial selection* of UI components**	• Via ***activation*** actions.	• *Not for adaptivity.*
Dynamic selection of UI components	• *Not for adaptability.*	• Via ***activation*** actions.
Dynamic cancellation of UI components	• *Not for adaptability.*	• Via ***cancellation*** actions.
Dynamic substitution of UI components	• *Not for adaptability.*	• Via a message containing ***cancellation*** actions, followed by the necessary number of ***activation*** actions.

Table 3 Excerpts from the decision making logic specification, reflecting an "if...then...else" rule format. The rules are organised according to the particular dialogue context (e.g. user task/display view/feedback). At runtime, individual user attributes are made available through user. prefix, while usage-context attributes through the context. prefix; the decision on activation/cancellation of the appropriate dialogue component(-s) is defined by activate or cancel rules, while re-evaluation with a new dialogue context is triggered by a dialogue statement. All decisions made within an evaluation round are posted as a single message

Decision making logic in DMSL – an excerpt

```
dialogue : "toolbar expert" [
        if ( user."web expertise" = "expert" or user."web expertise" = "frequent" ) then
        [
                if (context."installation" = "kiosk")
                        dialogue "kiosk toolbar expert";
                else
                if (context."installation" = "desktop")
                        dialogue "desktop toolbar expert";
        ]
        else
                activate "empty";
]
```

```
dialogue : "kiosk toolbar expert" [
        if ( user."visual ability" = "blind" ) then
                activate "kiosk toolbar expert hawk";
        else
        if ( user."visual ability" = "sighted" ) then
                activate "kiosk toolbar expert gui";
        else
                activate "user profile error";      // Unexpected user profile attribute
                value.
]
```

```
dialogue : "desktop toolbar expert" [
        if ( user."visual ability" = "blind" ) then
                activate "desktop toolbar expert hawk";
        else
        if ( user."visual ability" = "sighted" ) then
                activate "desktop toolbar expert gui";
        else
                activate "user profile error";              // Unexpected user profile
                attribute value.
]
```

One issue regarding the expressive power of activation and cancellation decisions categories concerns the way dynamic interface updates (i.e., changing style or appearance, without closing or opening interface objects) can be effectively addressed. The answer to this question is related to the specific connotation attributed to the notion of a dialogue component. A dialogue component may not only implement physical dialogue context, such as a window and embedded objects, but may concern the activation of dialogue control policies, or be realised as a particular sequence of interface manipulation actions. In this sense, the interface updates are to be collected in an appropriate dialogue implementation component (e.g., a program function, an object class, a library module) to be subsequently activated (i.e., called) when a corresponding activation message is received. This is the specific approach taken in the AVANTI browser, which, from a software engineering point of view, enabled a better organisation of the implementation modules around common design roles.

4.2.4 The DPC

4.2.4.1 The functional role This component is responsible for supplying the software implementation of all the dialogue artefacts that have been identified in the design process. Such implemented components may vary from dialogue artefacts that are common across different user- and usage-context attribute values (i.e., no adaptation needed), to dialogue artefacts that will map to individual attribute values (i.e., alternative designs have been necessitated for adapted interaction). Additionally, as it has been previously mentioned, apart from implementing physical context, various components may implement dialogue-sequencing control, perform interface manipulation actions, maintain shared dialogue state logic, or apply interaction monitoring.

4.2.4.2 The run-time behaviour The DPC should be capable of applying at run-time, *activation* or *cancellation* decisions originated from the DMC. Additionally, interaction monitoring components may need to be dynamically installed/uninstalled on particular physical dialogue components. This behaviour will serve the run-time interaction monitoring control requests from the UIS, so as to provide continuous interaction monitoring information back to the UIS for further intelligent processing.

4.2.4.3 The encapsulated content The DPC either embeds the software implementation of the various dialogue components, or is aware of where those

Table 4 Communicated messages between the UIS and the DMC

UIS→DMC	
Message class	Exporting an end-user profile.
Content structure	Sequence of <*Attribute, Value*> pairs.
When communicated	When requested by the Decision Making Component.
Example	`{ <"domain knowledge", "limited">, <"visual ability", "sighted">, <"age", "35">, <"motor ability", "fine">, ...}`
Message class	Dynamic detection of user parameters.
Content structure	*Parameter*, *Dialogue Context*.
When communicated	Each time an inference is made by the User Information Server.
Example	`{ "user confused", "Link Selection Task" }`
DMC→UIS	
Message class	Requesting user profile.
Content structure	Message contains just request header (i.e. content is empty).
When communicated	Before decision making is initiated in the Decision Making Component.
Example	`{}`
Message class	Requesting explicitly the value of a user attribute.
Content structure	*Attribute*.
When communicated	As needed by the Decision Making Component.
Examples	`{ "age" }, { "domain knowledge" }, { "Web knowledge" }`

components physically reside, by employing dynamic query, retrieval and activation methods. The former is the typical method that can be used if the software implementation of the components is provided locally, by means of software modules, libraries or resident installed components. Usually, most of the implementation is to be carried out in a single programming language. The latter approach reflects the scenario in which distinct components are implemented on top of component-ware technologies, usually residing in local/remote component repositories (also called registries or directories), enabling re-use with dynamic deployment.

In the development of the AVANTI browser, a combination of these two approaches has been employed, by implementing most of the common dialogue components into a single language (actually in C++, by employing all the necessary toolkit libraries), while implementing some of the alternative dialogue artefacts as independent Active X components that were located and employed on-the-fly. The experience from the software development of the AVANTI-browser has proved that: (a) the single language paradigm makes it far easier to perform quick implementation and testing of interface components; (b) the component-based approach largely promotes the binary format re-use of implemented dialogue components, while offering far better support for dynamic interface assembly, which is the central engineering concept of unified user interfaces (this issue will be elaborated upon in the discussion section of the paper).

4.2.4.4 The implementation The micro-architecture of the DPC internally employed in the AVANTI-browser, as outlined in Fig. 6, emphasises internal organisation to enable extensibility and evolution by adding new

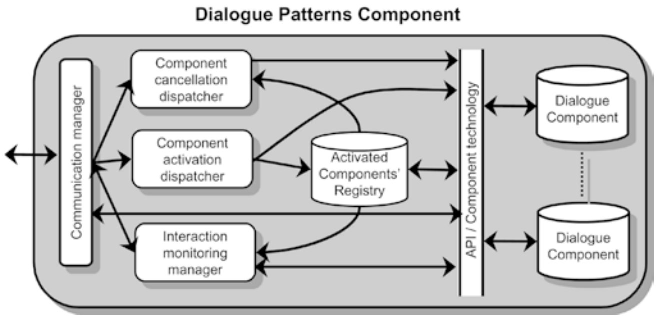

Dialogue Patterns Component
Communication manager
Component cancellation dispatcher
Component activation dispatcher
Interaction monitoring manager
Activated Components' Registry
API / Component technology
Dialogue Component
Dialogue Component

Fig. 6 The micro-architecture of the DPC internally employed in the AVANTI browser

dialogue components. Additionally, it reflects the key role of the DPC in applying adaptation decisions. The internal components are:

– The *activation dispatcher*, which "locates" the source of implementation of a component (or simply uses its API, if it is a locally used library), to activate it. In this sense, activation may imply a typical instantiation in OOP terms, calling of particular service functions, or activating a remotely located object. After a component is activated, if cancellation is to be applied to this component, it is further registered in a local registry of activated components. In this registry, the indexing parameters used are the particular dialogue context (e.g., a sub-task, for instance "http address field"), and the artefact design descriptor (i.e., a unique descriptive name provided during the design phase– for instance, "virtual keyboard"). For some categories of components, cancellation may not be defined during the design process, meaning there is no reason to register those at run-time for possible future cancellation (e.g., components with a temporal nature, that only perform some interface update activities).
– The *cancellation dispatcher*, which locates a component based on its indexing parameters and "calls" for cancellation. This may imply a typical destruction in OOP terms, calling internally particular service functions that may typically perform the unobtrusive removal of the physical view of the cancelled component, or the release of a remote object instance. After cancellation is performed, the component instance is removed from the local registry.
– The *monitoring manager,* which plays a two-fold role: (a) it applies monitoring control requests originated from the UIS by, firstly, locating the corresponding dialogue components, and, secondly, requesting the installation (or uninstall) of the particular monitoring policy (this requires implementation additions in dialogue components, for performing interaction monitoring and for activating or deactivating the interaction monitoring behaviour); and (b) it receives interaction monitoring notifications from dialogue components and posts those to the UIS.
– The *communication manager,* which is responsible for dispatching incoming communication (activation, cancellation and monitoring control) and posting outgoing communication (monitoring data, and initial adaptation requests). One might observe that there is also an explicit link between the dialogue components and the communication manager. This reflects the initiation of interaction in which the dialogue control logic (residing within dialogue components) requests iteratively the application of decision-making (from the DMC). Such requests will need to be posted for all cases involving dialogue component alternatives for which adapted selection has to be appropriately performed.

The *dialogue components,* in which the real implementation of physical dialogues, dialogue control logic, and an interaction monitoring method, are typically encompassed. In practice, it is hard to accomplish isolated implementation of the dialogue artefacts as independent "black boxes", that can be combined and assembled on-the-fly by independent controlling software. In most designs, it is common that physical dialogue artefacts are contained inside other physical artefacts. In this case, if there are alternative versions of the embedded artefacts, it turns out that to make containers fully orthogonal and independent with respect to the contained, one has to support intensive parameterisation and pay a "heavier" implementation overhead. However, the gains are that the implementation of contained artefacts can be independently re-used across different applications, while in the more "monolithic" approach, re-use requires deployment of the container code (and recursively, of its container too, if it is contained as well). A later section will discuss this issue in further details.

4.2.5 Adaptability and adaptivity cycles

The completion of an adaptation cycle, being either adaptability or adaptivity, is realized in a number of distributed processing stages performed by the various components of the unified architecture. During these stages, the components communicate with each other, requesting or delivering specific pieces of information. Figure 7 outlines the processing steps for performing both the initial adaptability cycle (to be executed only once), as well as the two types of adaptivity cycles (i.e., one starting from "dynamic context attribute values", and another starting from "interaction monitoring control"). Local actions indicated within components (in each of the four columns) are either outgoing messages, shown in bold typeface, or necessary internal processing, illustrated via shaded rectangles.

4.3 Inter-component communication semantics

This section presents the communication protocol among the various components in a form emphasising the rules that govern the exchange of information

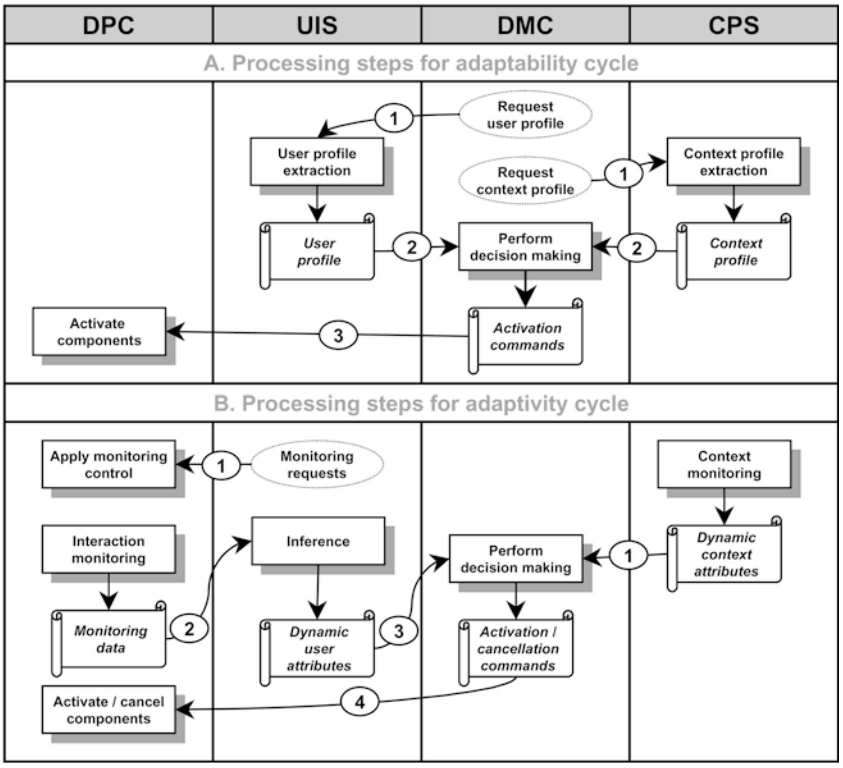


Fig. 7 The processing steps for performing both the initial adaptability cycle, as well as the two types of adaptivity cycles

among various communicating parties, as opposed to a strict message syntax description . Hence, the primary focus will be on the semantics of communication, regarding: (a) the type of information communicated; (b) the content it conveys; and (c) the usefulness of the communicated information at the recipient component side.

In the unified software architecture, there are four distinct bi-directional communication channels (outlined in Fig. 8), each engaging a pair of communicating components. For instance, one such channel concerns the communication between the UIS and the DMC. Each channel practically defines two protocol categories, one for each direction of the communication link, e.g., UIS➔DMC (i.e., the type of messages sent from UIS to DMC) and DMC➔UIS (i.e., the type of messages sent from DMC to UIS). The description of the protocols for each of the four communication channels follows.

4.3.1 Communication between the UIS and the DMC

In this communication channel, there are two communication rounds: (a) prior to the initiation of interaction, where the DMC requests the user profile from the UIS, and the latter replies directly with the corresponding profile as a list of attribute/value pairs; and (b) after the initiation of the interaction, each time the UIS detects some dynamic user attribute values (on the basis of interaction monitoring), it communicates those values immediately to the DMC. In Table 4, the syntax of the messages communicated between the UIS and the DMC is defined and simple examples are provided.

4.3.2 Communication between the UIS and the DPC

The communication among these two components aims to enable the UIS to collect interaction monitoring information, as well as to control the type of monitoring to be performed. The UIS may request monitoring at three different levels: (a) the *task* (i.e., when *initiated* or *completed*); (b) the *method* for interaction objects (i.e., which logical object action has been accomplished—like "pressing" a "button" object); and (c) the *input event* (i.e., a specific device event—like "mouse move" or "key press").

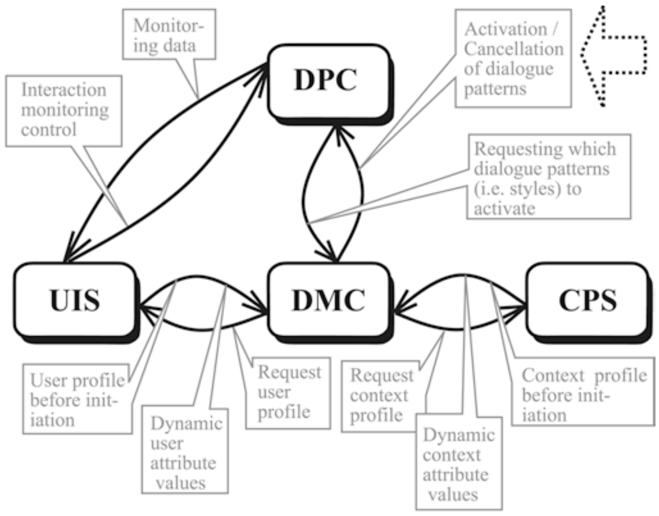

Monitor-ing data
Interaction monitoring control
DPC
Activation / Cancellation of dialogue patterns
Requesting which dialogue patterns (i.e. styles) to activate
UIS
DMC
CPS
User profile before init-iation
Dynamic user attribute values
Request user profile
Request context profile
Dynamic context attribute values
Context profile before init-iation

Fig. 8 Four distinct bi-directional communication channels

In response to monitoring control messages, the DPC will have to: (a) activate or cancel the appropriate interaction monitoring software modules; and (b) continuously export monitoring data, according to the monitoring levels requested, back to the User Information Server (initially, no monitoring modules are activated by the DPC). In Table 5, the syntax of messages communicated between the UIS and the DPC is defined, and simple examples are provided.

4.3.3 Communication between the DMC and the DPC

As it has been mentioned, inside the DPC, alternative implemented dialogue artefacts are associated to their respective dialogue context (e.g., their sub-tasks). As part of the design process, dialogue artefacts have been assigned an indicative name, unique across the rest of alternatives for the same dialogue context. Each such implemented dialogue artefacts, associated to particular dialogue contexts, will be referred to as a *style*. Styles of a given dialogue context can thus be identified by reference to their designated name.

At start-up, before initiation of interaction, the initial interface assembly process proceeds as follows: the top-level dialogue component requests for its associated user-task the names of the *styles* (i.e., implemented dialogue components) which have to be activated, so as to realise the adaptability behaviour. Recursively, each container dialogue component repeats the same process for every embedded component. For each received request, the DMC triggers the adaptability cycle and responds with proper activation messages. Additionally, after the initiation of interaction, the DMC may communicate dynamic style activation/cancellation messages to the DPC, as a consequence of the dynamic detection of user- or usage-context attribute values (originated from the UIS). Table 6 defines the syntax of messages

Table 5 Communicated messages between the UIS and the DPC

UIS→DPC	
Message class	Controlling the level of monitoring.
Content structure	*Status*, *Level*. *Status* = **on** or **off**. *Level* = *TaskLevel* or *EventLevel* or *MethodLevel*. *EventLevel* = **event**, *ObjectName*, *EventCategory*. *TaskLevel* = **task**, *TaskName*. *MethodLevel* = **method**, *ObjectName*, *MethodCategory*.
When communicated	As needed by the User Information Server inference mechanisms.

Examples	`{ on, task, "link selection" }`
	`{ off, event, "HTMLPageWindow", "KeyPress" }`
	`{ on, method, "BWDPageButton", "Pressed" }`
	`{ on, method, "HTMLPageScrollbar", "Scrolled" }`

DPC→UIS	
Message class	Monitoring data.
Content structure	*TaskBased* or *EventBased* or *MethodBased*. *TaskBased* = **task**, *TaskName*, *TaskAction*. *TaskAction* = **initiated** or **completed**. *EventBased* = **event**, *ObjectName*, *EventCategory*, *EventData*. *EventData* = Sequence of <*Parameter*, *Value*> pairs. *MethodBased* = **method**, *ObjectName*, *MethodCategory*.
When communicated	When the corresponding user actions are performed.

Examples	`{ task, "link selection", initiated }`
	`{ task, "link selection", completed }`
	`{ event, "PageLoadingControlToolbar", "KeyPress", <"key", "a"> }`
	`{ event, "BWDPageButton", "MouseButtonPress", <"button", "2"> }`
	`{ method, "StopLoadingButton", "Pressed" }`
	`{ method, "ReloadButton", "Pressed" }`

Table 6 Communicated messages between the DMC and the DPC

DPC→DMC	
Message class	Requesting active styles for a particular dialogue context
Content structure	*DialogueContext*
When communicated	Upon start-up, to initiate the necessary dialogue patterns.
Examples	`{ "link selection" }, { "page loading control" }` `{ "page display control" }, { "toolbar expert" }`
DMC→DPC	
Message class	Posting decisions regarding component activation / cancellation.
Content structure	*Decision =(DecisionType, ComponentSignature)+* *DecisionType =* **activation** or **cancellation**. *ComponentSignature = TaskName, ComponentName*.
When communicated	• At the end of the adaptability cycle, prior to initiation of interaction. • At the end of each adaptivity cycle, after initiation of interaction.
Examples	`{activation, "link selection", "direct"}` `{activation, "stop loading", "by confirmation"}` `{cancellation, "link selection", "direct"}` `{activation, "link selection", "by confirmation"}` `{activation, "confirm_ok_font", "times28"}` `{activation "confirm_ok_color", "rgb{0,255,0}"}` `{activation, "confirm_cancel_color", "rgb{255,0,0}" }`

communicated between the DMC and the DPC, and shows simple examples.

4.3.4 Communication between the DMC and the CPS

The communication between these two components is very simple. The DMC requests the various context parameter values (i.e., the usage-context profile), and the CPS responds accordingly. During interaction, dynamic updates on certain context property values are to be communicated to the DMC for further processing (i.e., possibly new inferences will be made). Table 7 defines the classes of messages communicated between the DMC and the CPS, and shows simple examples.

5 Development highlights

In this section, selected scenarios from the development of the AVANTI browser are discussed, focusing on the technical challenges related to different components of the implemented architecture. As it will be shown, in some scenarios, the implementation of the dialogue components has been comparatively the most demanding task, while for others scenarios, the logic for dynamic user attribute detection required particular attention. The AVANTI browser has been subject to intensive usability evaluation with end users, through which initial design scenarios have been updated, leading to the final validated unified user interface design. The emphasis of the evaluation process has been on verifying the added value of the adaptation-specific artefacts. The overall AVANTI system design, implementation and evaluation, in the form of a case study for unified user interface development, is discussed in detail in [38].

In Fig. 9, three instances of the AVANTI-browser interface are depicted, demonstrating adaptation based on the characteristics of the user and the usage context. Specifically, Fig. 9a presents a simplified instance of the browser interface intended for use by a user ''unfamiliar with Web browsing''. Note the ''basic'' web-access user interface with which the user is presented, as well as the fact that links are presented as buttons, increasing their affordance (at least in terms of functionality) for users familiar with windowing applications in general. The second instance, Fig. 9b, presents an interface instance to be used in the context of a ''kiosk installation''. Note that the typical windowing paradigm is abandoned and replaced by a simpler, content-oriented interface (even scrollbars, for example, have been replaced by ''scroll buttons'', which perform the same function as scrollbars, but have different presentation and behavioural attributes—e.g., they ''hide'' themselves if they are not selectable). Furthermore, ''usage-context sensitive''

Table 7 . Communicated messages between the DMC and the CPS

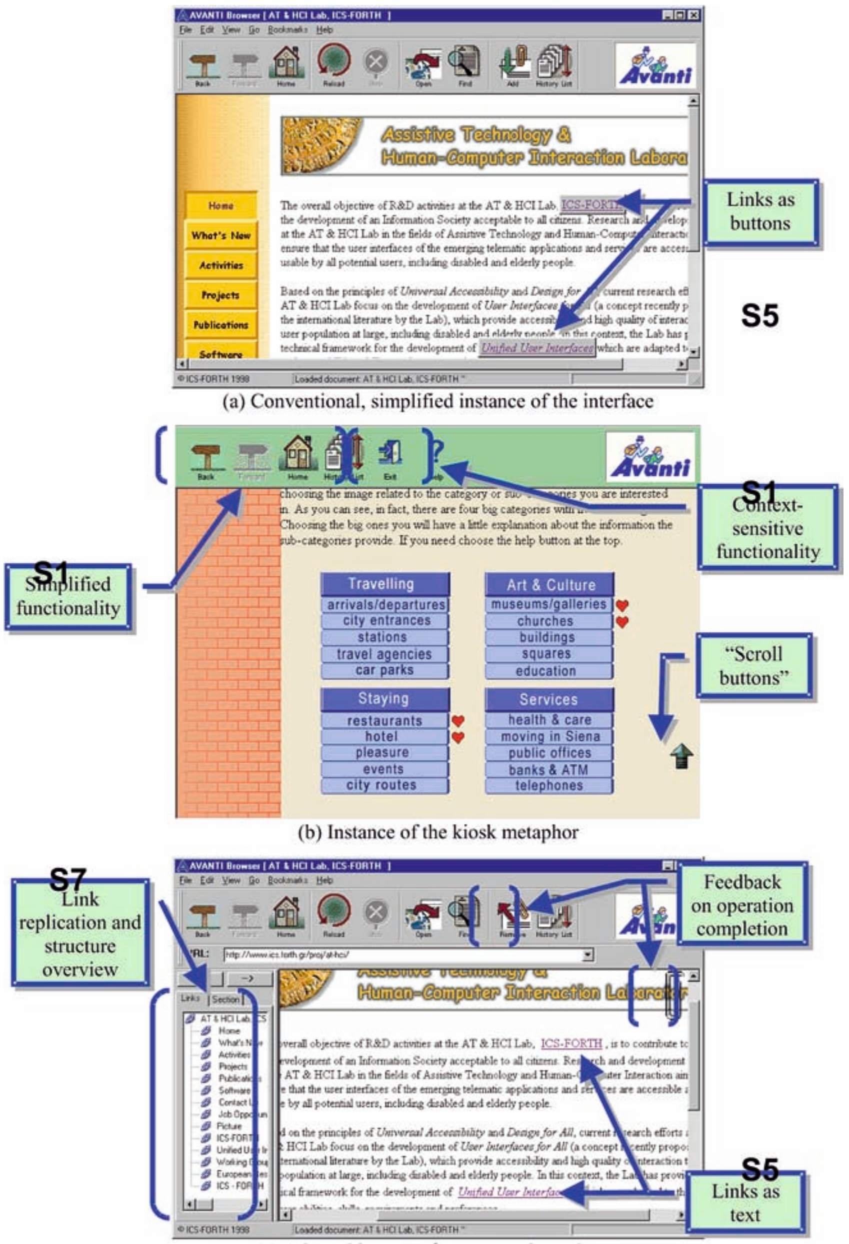
DMC→CPS	
Message class	Requesting context parameter values.
Content structure	This message contains only the message header (i.e. empty content).
When communicated	Prior to initiation of interaction, beginning of adaptability cycle.
Examples	`{ }`
CPS→DMC	
Message class	Responding with a list of usage-context parameter values.
Content structure	Sequence of <*Attribute, Value*> pairs.
When communicated	When requested by the Decision Making Component.
Examples	`{ <"screen resolution", "640, 480">, <"mouse available", "yes">, <"terminal position", "120 cm">, <"software volume ctrl", "yes">, <"speech synthesis", "yes"> }`
Message class	Dynamic usage-context attribute values.
Content structure	*Attribute*, *Value*.
When communicated	Each time a usage-context attribute value is dynamically modified.
Examples	`{ "environment noise", "65 dB" }` `{ "user in front of terminal", "no" }` `{ "user in front of terminal", "yes" }`

buttons in the interface's "toolbar" are supported (i.e., adding the "exit" button in "kiosk" usage context, since the typical "File" menu providing the "exit" option in desktop mode is not available in "kiosk" mode). Finally, in the third instance, Fig. 9c, the interface has been adapted for an "experienced Web user". Note the additional functionality that is available to the user (e.g., a pane where the user can access an overview of the document itself, or of the links contained therein, an edit field for entering the URLs of local or remote HTML documents).

The scenario of Fig. 9 introduced challenges mainly for the organisation of the dialogue components. More specifically, in Fig. 10, the organisation structure of the dialogue components is outlined. The management of alternative components has been handled in a way directly reflecting the design logic, through appropriate OOP abstraction mechanisms (the AVANTI browser has been implemented in C++). For instance (see Fig. 10), the toolbar has been implemented as a container component separating its contained constituents that could be subject to dynamic presence (e.g., the "expert commands" component that is to be conditionally activated, has been purposely defined as a separate component, to enable independent activation control). Additionally, the toolbar itself has been defined as an abstract container, meaning it can be physically realised with alternative container styles (e.g. a managed window, a toolbar, a simple rectangular area, an auditory container). Following a similar approach, the various contained constituents (like the "expert commands"), when supporting alternative realisations (e.g., "kiosk" or "desktop" versions, each further supplied with "GUI" or "non-visual" versions), were also structured as abstract classes, enabling all the various alternative versions to be implemented as concrete instantiations of the abstract class.

Following such an implementation approach, container components manipulate constituents through abstract object instances, that, for each contained constituent abstract class A are constructed during runtime as follows: assume *A* has alternative implementations *A1,...,An*; then, an *Ai* instance will be created following the adaptation decision received from the DMC. This distinction between the abstract component API and its internal implementation has been proved to be a very simple, still powerful, method for handling the component organization needed in the AVANTI browser.

In Fig. 11, the activation of dialogue components that provide switch-based scanning [34] for motor-impaired user access is depicted. The implementation of those dialogue techniques did not follow the component abstraction pattern as described before. From the initial design stages, it became evident that the scanning dialogue patterns were needed for all GUI dialogue components, and thereafter, for every constituent GUI object of those components. Hence, from a design point of view, the diversification originally emerges at the level



(a) Conventional, simplified instance of the interface

(b) Instance of the kiosk metaphor

(c) Adapted instance for an experienced user

Fig. 9. Three instances of the AVANTI browser interface

of interaction objects, rather than at higher-level dialogue components. Based on this remark, the pattern has been applied at a lower, but more generic level, as indicated in the lower-part of Fig. 10. The GUI object library has been supplied with two alternative implementations: (a) the original Windows intact implementation; and (b) an augmented version, supporting switch-based dialogue control (such as hierarchical scanning, virtual keyboard, and automatic window toolbars, as shown in Fig. 11).

The previous abstraction technique worked well for embedding scanning in GUI dialogue objects. Then, we tried to re-apply the same technique for the HAWK non-visual toolkit, a software library specifically designed and implemented to enable the development of non-visual interactive applications [33]. It turned out that further abstraction by encompassing the augmented GUI toolkit and the HAWK toolkit, within a single abstract toolkit, could not work in practice for a demanding application, such as a Web browser. The reason was twofold: (i) the programmers of the GUI dialogues required fine-grain control on all visual attributes and methods, thus putting a GUI bias on the abstract toolkit; and (b) the programmers of the non-visual dialogues required similarly fine-grain control on all non-visual interaction techniques, inherently putting a "HAWK-specific" bias on the abstract toolkit. As a result, it was decided that abstraction should remain at the level of dialogue components, following the core of the unified user interface development approach, rather

Fig. 10 The organisation structure of the dialogue components

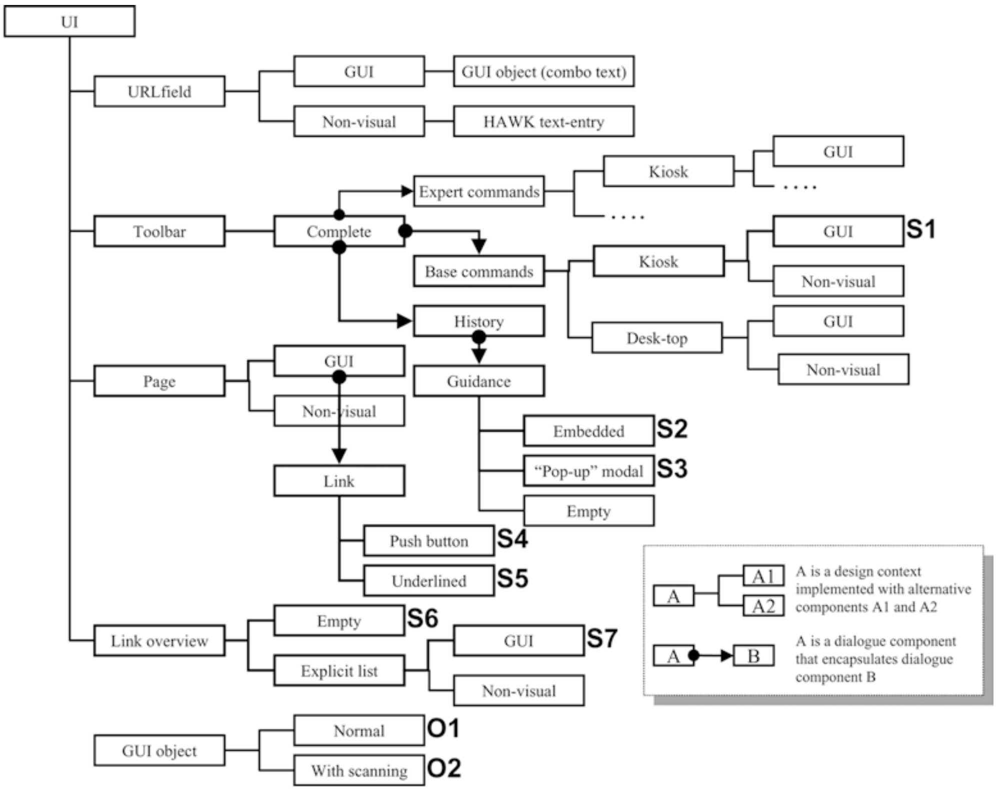
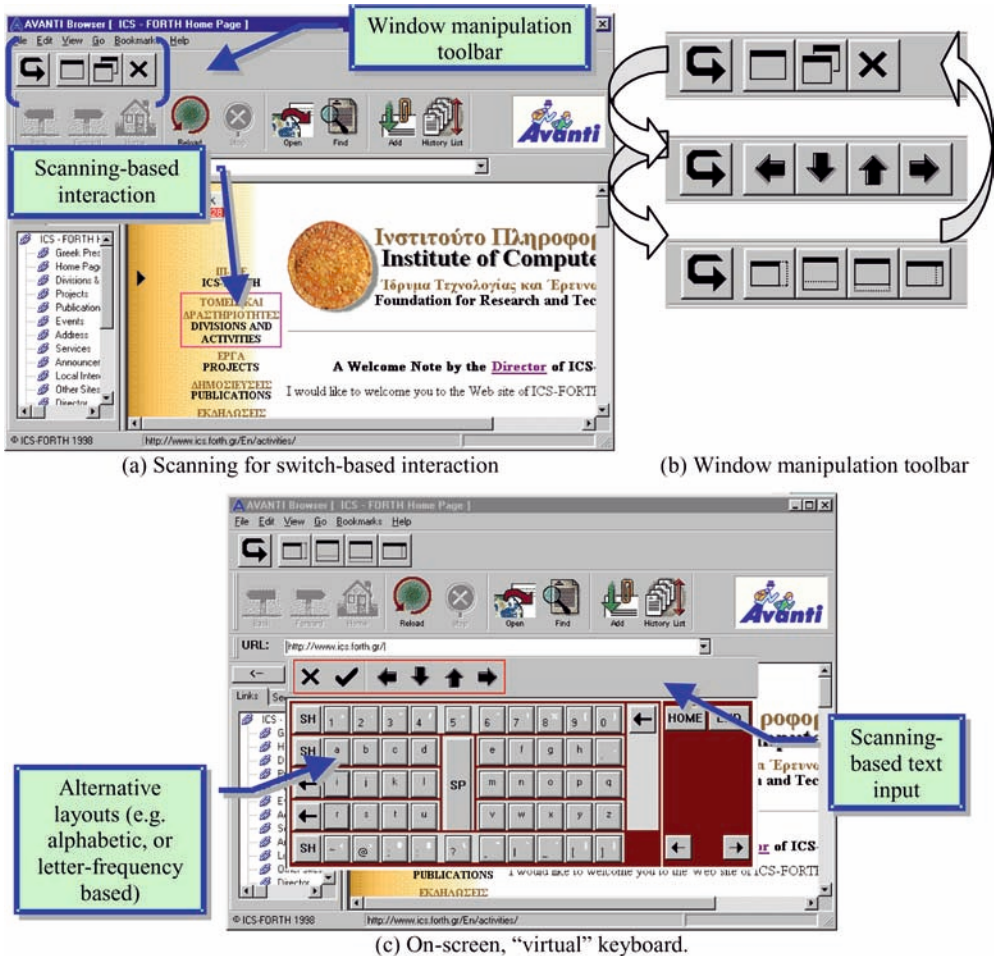


Fig. 11 The activation of dialogue components that provide switch-based scanning for motor-impaired user access

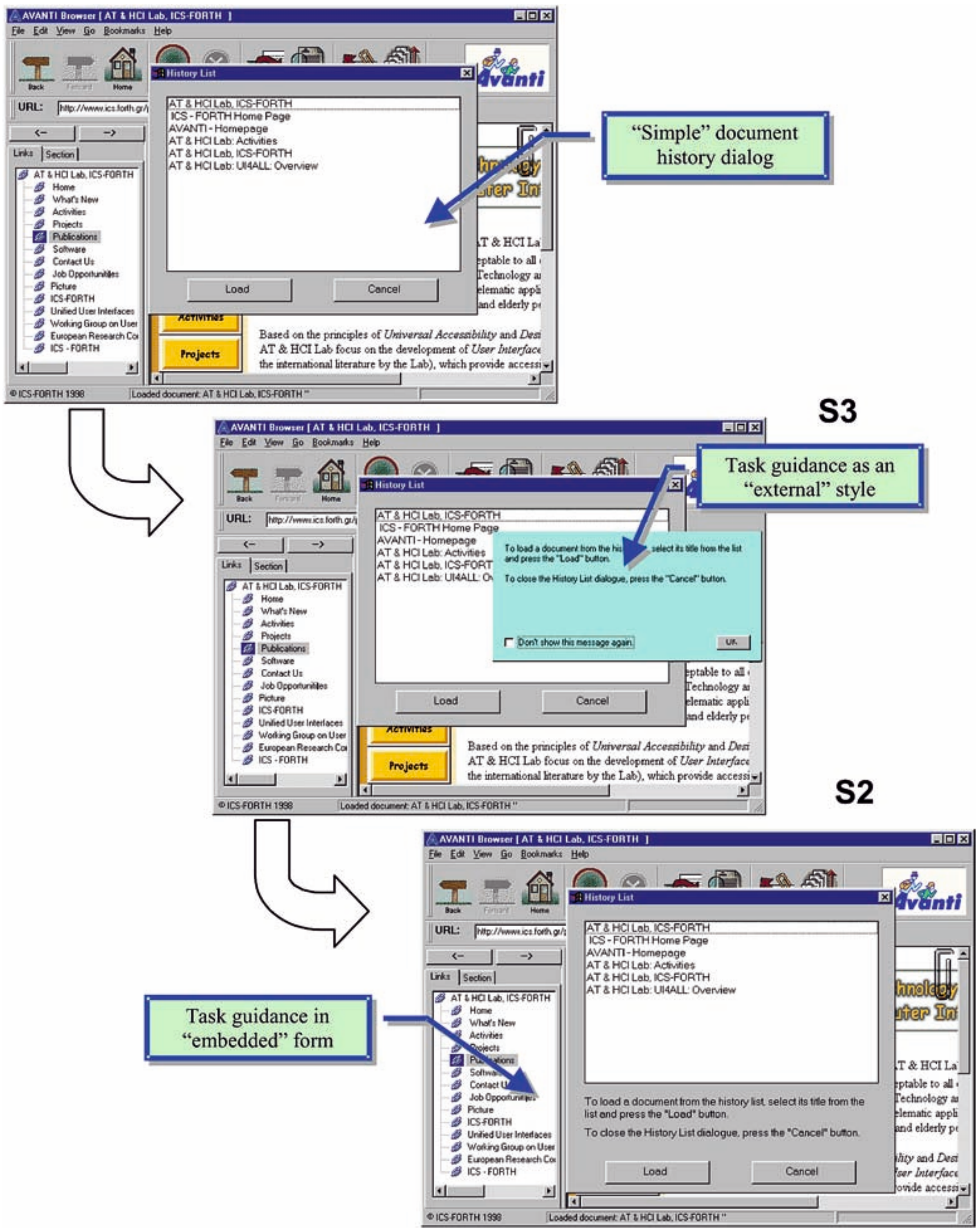

Window manipulation toolbar
Scanning-based interaction
(a) Scanning for switch-based interaction
(b) Window manipulation toolbar
Alternative layouts (e.g. alphabetic, or letter-frequency based)
Scanning-based text input
(c) On-screen, "virtual" keyboard.

than at the level of lower-level dialogue primitives like interaction objects, as it had been applied in previously reported work, for example in dual interface development [29]. The strategies for unification that can be employed at the level of interaction objects, namely integration, augmentation, expansion and abstraction, when developing for diverse users and platforms, are described in detail in [32].

In Fig. 12, the activation of alternative components for history-task guidance is shown. In this case, the component organisation scheme previously described has been employed. However, the most challenging issue in this scenario concerned the dynamic identification of the need to provide guidance, by detecting a potential confusion or inability to perform the task triggering the provision of ‘‘pop-up’’ guidance. This required particular focus in the implementation of the UIS, with heuristic algorithms based on state automata for the recognition of particular user situations (like confusion), while it inherently necessitated interaction-monitoring software to be integrated within all the dialogue components.

5.1 The DMSL

In Table 3, the decision-making logic is defined in decision making specification language (DMSL), which is a compact language that has been specifically designed


AVANTI Browser [AT & HCI Lab, ICS-FORTH]
History List
AT & HCI Lab, ICS-FORTH
ICS - FORTH Home Page
AVANTI - Homepage
AT & HCI Lab: Activities
AT & HCI Lab, ICS-FORTH
AT & HCI Lab: UI4ALL: Overview
Load
Cancel
"Simple" document history dialog
S3
Task guidance as an "external" style
Don't show this message again
OK
S2
Task guidance in "embedded" form

To load a document from the history list, select its title from the list and press the "Load" button.

To close the History List dialogue, press the "Cancel" button.

Fig. 12 The activation of alternative components for history-task guidance

and implemented for this purpose. The characteristics of this approach are as follows:

- The decision logic is organised in "if...then...else" blocks, and each block is associated to a particular dialogue context. During run-time, the DMC is always asked (by the DPC) to carry out decision making for specific dialogue contexts; hence, this approach enables the localisation and splitting of the decision making logic, significantly boosting run-time performance (the corresponding "logic" blocks are identified through a hash-table).
- The individual end-user and usage-context profiles are made syntactically available through the ***user.*** and ***context.*** prefixes, while attribute values are defined as matching quoted strings (there are also built-in functions for string manipulation, and numeric context extraction including range checks). The use of quoted strings is necessary to enable arbitrary run-time interpreted attribute values to be defined.
- There are only three primitive statements: (a) ***dialogue***, which initiates evaluation for the rule block corresponding to dialogue context value supplied (in Table 3, the statement *dialogue "kiosk toolbar expert"* initiates, after the particular block is evaluated, the evaluation of the "kiosk toolbar expert" block); (b) ***activate***, which adds the specified string value to the set of activation decisions to be posted (when the overall evaluation phase is completed) to the DPC. It should be noted that the activation decision with value "user profile error" is to be interpreted by the DPC to designate, for example, an incomplete user profile (i.e., it is up to the interface developer to handle such a case—the decision making logic simply posts a decision indicating the type of the error); and (c) ***cancel***, which, similarly to ***activate***, adds the specified string value to the list of cancellation decisions to be posted to the DPC.
- The rules are compiled in a tabular representation that is executed at runtime. This representation engages simple expression evaluation trees for the conditional expressions, leading finally to the execution of the basic ***activate***, ***cancel*** and ***dialogue*** statements. Additionally, the execution engine performs cycle elimination by simply recording the dialogue contexts in each evaluation sequence, and disabling a dialogue context to be evaluated twice.
- The representation as "*if...then...else*" clauses limits the chances for more sophisticated decision making. Actually, the initial implementation of the decision logic in the AVANTI browser was hard-coded in C + +, while it quickly became evident that this logic reflected simple rule patterns. DMSL is a more recent development in the context of decision-making support for unified user interface development, supporting embedding of decision-making logic in a way directly editable by designers. This also enables different designers to contribute to different dialogue contexts, due to the localization ability of DMSL decision logic according to dialogue contexts.

The employment of string encoding for the identification of the dialogue artefacts to be posted to the DPC, via activation or cancellation messages, is a purposeful decision, because it enables additional information to be embedded within the message content, and to be subsequently interpreted dynamically by the corresponding dialogue components of the DPC. For instance, assume that display size and screen resolution of the terminal is to be used as a parameter to dynamically select alternative ways of layout organisation for particular dialogue boxes, e.g., for "file management"—in short "fm". In this case, from an interface design point of view, there are two possible policies: (a) to enumerate alternative designs (e.g., "small fm", "medium fm", "large fm"), corresponding to discrete values of the display size (e.g., "small", "medium", "large"); and (b) to require the dialogue box implementation to analogically compute layout based on display size (e.g., width and height) information.

While the former case requires three alternative artefacts with three distinct identifiers, the latter requires a single artefact, which, however, necessitates the display size information. To address such cases, the DPC artefact indexing policy may universally employ string identifiers with the internal syntactic form *"<string id>: <extra parameters>"*. Hence, in the current example, the string content would actually have the form "fm: <*width, height*>", with run-time examples such as "fm:1024,768" or "fm:200,100". In this case, the artefact programmer would need to be aware of this formal structure, and would be responsible for parsing and extracting the run-time parameters, and subsequently using those parameters as an input to the dynamic layout calculation algorithm. The technique of employing string representations for messages that can encompass dynamically interpretable request identification and parameters is employed in dynamically extensible servers. The latter may evolve with time and provide new types of services to clients, without however altering the basic remote access protocol.

6 Discussion

In this section, development situations in which unified user interface development can encounter difficulties are identified and discussed. Subsequently, the particular software engineering focus of unified user interface development on the organisation and re-use of dialogue components based on their design role is explored in relation to existing adaptive interface development proposals. The particular run-time behaviour of unified user interfaces in virtually assembling an appropriate interface instance out of a pool of dialogue components, given a particular end-user and usage-context, is considered in view of existing methods for individualised interaction.

Finally, some lower-level barriers practically experienced in implementing the various components are reported.

6.1 Where unified user interface development may not be appropriate

Unified user interface development aims to address interface engineering issues emerging: (a) when variations of the end-user attributes impose different interaction policies "from the beginning", i.e., when the initial interface is provided to the user, or "in the middle", i.e., when interaction is already taking place; and (b) similarly, when variations on the usage-context impose interaction-policy differentiation requirements. However, not all such situations can be effectively tackled through unified user interface development.

For instance, if for a specific user category the interface design should reflect severe differentiations regarding the overall task structure, as well as the physical design, practically limiting the possibilities for abstraction and unification, then it may be more cost-effective to provide a customised interface design and implementation. Such scenarios have emerged in particular in the context of efforts targeted to providing applications for users with cognitive impairments and learning difficulties, or for instance when designing applications for children [15]. During the early stages of the design process, it quickly became evident that the interface design itself was very much customised, regarding both the dialogue elements (i.e., instead of using typical windowing objects, cartoon-based representations had to be employed), and the overall dialogue structure (simple modal dialogues, with intensive friendly help and increased error tolerance). Moreover, all such designed dialogue components were only appropriate in the context of the specific design, practically excluding the chances for further re-use or combination with other dialogues not designed for this specific user group. In this case, the embedding of such customised designs into a unified-development process may introduce additional implementation complexity without particular interaction-quality benefits. As a consequence, if the overall design is so customised and "closed" that there are practically no possibilities to re-use or combine dialogue components for other user- or usage-context attribute values, then the unified user interface development approach should be avoided.

A similar situation also emerged in the early stages of developing applications for specific computing platforms, when no particular emphasis was given in addressing user diversity. For instance, the initial development of applications for mobile phones engaged radically different design elements with respect to those typically occurring in desktop applications. Additionally, early phones merely provided very primitive interaction facilities that largely restricted the chances of producing embedded applications that could address the needs of different user groups. In such cases, the target platform is considered to be virtually invariant, implying that the design stage assumes a specific I/O profile. At the same time, software vendors, when faced with such restricted interface development tools, are naturally not interested in investing to support user-adapted behaviour. Consequently, if user- and usage-context diversity is not a priority to be addressed, there is no need to apply unified user interface development. Moreover, until recently, the structure of dialogues in typical mobile-phone applications has reflected a menu-based style with modal dialogues. Additionally, the corresponding software libraries employed for interface development have been strictly platform-dependent, enabling no further re-use or combination with other dialogue components beyond their specific platform.

However, following the recent developments in the context of applications for mobile phones, more sophisticated development tools have become available, offering facilities that are approaching those for typical PDAs (e.g., Java 2.0 Micro Edition/J2ME by JavaSoft, and Binary Runtime Environment for the Wireless/BREW, by QualComm). Additionally, the advanced facilities for digital audio playback, voice recognition, and speech synthesis, together with the enhanced graphics display elements (GUI controls, high resolution, and a run-time configurable colour palette) offer the necessary ingredients to develop applications for diverse user attributes, including those people with disabilities, elderly people, children, etc., in the form of unified interactive applications.

6.2 An emphasis on design-oriented dialogue component re-use

Generally, practical support for re-use is reflected by two key factors: (a) the ability to locate software components that match certain design criteria; and (b) the facility to deploy located software components. As a result, when development methods emphasise either better matching, or easier deployment, re-use is further promoted.

In unified user interface development, the organisation of dialogue components according to their associated dialogue design context is a fundamental technical property (see Fig. 10 outlining software component relationships), while the decision logic, which practically documents the design role of each component, engages naming conventions that are directly employed in the target implementation for run-time indexing purposes (see the excerpt from Table 3). Such design-oriented organisation of dialogue implementation units facilitates straightforward location within the overall system implementation, through design parameters.

When for particular container components all the various contained components are "unimorphic", i.e., they are supplied with a single invariant version, container or contained component parameterisation is not applied, thus making containers implementationally dependent on the contained components. This decision practically eliminates the chances for potential re-use of either container or contained component categories. However, in unified user interface development, the implementation of containers has to be independent of the corresponding contained components, by employing abstraction patterns, since the variability and the extensibility of contained components has to be supported. The latter approach makes containers and contained components largely parameterised, enabling easier deployment and re-use.

Overall, even though the unified user interface development approach does not introduce any new techniques for software re-usability, and has not been designed with a primary emphasis on re-usability as such, it proposes a development discipline in which re-usability is effectively reflected and promoted. In this context, it is important to highlight the software engineering maturity of this specific development code of practice, which supports component extensibility, re-usability, and orthogonality, thus facilitating dynamic run-time interface assembly.

6.3 The concept of the dynamic interface assembly

The concept of the dynamic interface assembly reflects the key run-time mechanisms to support adaptability in unified user interfaces. Previous work in adaptive interaction, involving techniques such as the detection of user attributes, adaptive prompting and localised lexical-level modifications (e.g., re-arranging menu options, or adding/removing operation buttons). The issue of making the interface fit from the beginning to individual users has been addressed in the past mainly as a *configuration problem*, requiring interface developers to supply configuration editors so that end users could fit the interface to their particular preferences. However, such methods are limited to fine-tuning some lexical-level aspects of the interface (e.g., tool-bars, menus), while they always require explicit user intervention, i.e., there is no automation. In this context, the notion of adaptability as realised in unified user interfaces offers new possibilities for automatically-adapted interactions, while the architecture and run-time mechanisms to accomplish dynamic interface assembly constitute a unique software engineering perspective.

Some similarities with dynamic interface assembly can be found in typical Web-based applications delivering dynamic content. The software engineering methods employed in such cases are based on the construction of application templates (technologies such as Active Server Pages by Microsoft—ASP or Java Server Pages—JSP by JavaSoft, are usually employed), with embedded queries for dynamic information retrieval, delivering to the user a Web page assembled on-the-fly. In this case, there are no alternative embedded components, just content to be dynamically retrieved, while the Web page assembly technique is mandatory when HTML-based Web pages are to be delivered to the end-user (in HTML, each time the content changes, a different HTML page has to be written). However, in case a full-fledged embedded component is developed (e.g., as an ActiveX object or Java Applet), no run-time assembly is required, since the embedded application internally manages content extraction and display, as a common desktop information retrieval application.

The implementation of unified user interfaces is organised in hierarchically structured software templates, in which the key placeholders are parameterised container components. This hierarchical organisation, as it has been reflected in the development excerpts, mirrors the fundamentally hierarchical constructional nature of interfaces. The ability to diversify and support alternatives in this hierarchy is due to containment parameterisation, while the adapted assembly process is realised by selective activation, engaging remote decision making on the basis of end-user and usage-context information.

In Fig. 13, the concept of parametric container hierarchies is illustrated. Container classes expose their containment capabilities and the type of supported contained objects by defining abstract interfaces (i.e., abstract OOP classes) for all the contained component classes. These interfaces, defined by container class developers, constitute the programming contract between the container and the contained classes. In this manner, alternative derived contained-component classes may be instantiated at runtime as constituent elements of a container. Following the definition of the polymorphic factor PL, which provides a practical metric of the number of possible alternative run-time configurations of a component, the PL of the top level application component gives the number of the possible alternative dynamically assembled interface instances (see also Fig. 2).

From a programming point of view, in the AVANTI browser, the activation control of dialogue components for run-time assembly has been mainly realised through typical library function calls. Such function calls engage object instances corresponding to dialogue components, without employing any component-ware technology. Hence, this run-time assembly behaviour has been accomplished without the need of locating, fetching, and combining components together. Nevertheless, efforts have been devoted to applying and testing the latter approach in real practice, by employing a component-ware technology (DCOM/ActiveX) for a limited number of dialogue components. This required a more labour-intensive implementation approach (from a C++ point of view, while isolated testing of components with Visual Basic was far easier), for packaging dialogues to make them component-enabled, as well as for further activating and using them at runtime. However, there are some evident advantages:

- Dialogue components need not be carried altogether, but can be dynamically loaded, thus promoting a thin DPC implementation;
- In effect, the core logic of the DPC, apart from dialogue components, can be also packaged as a com-

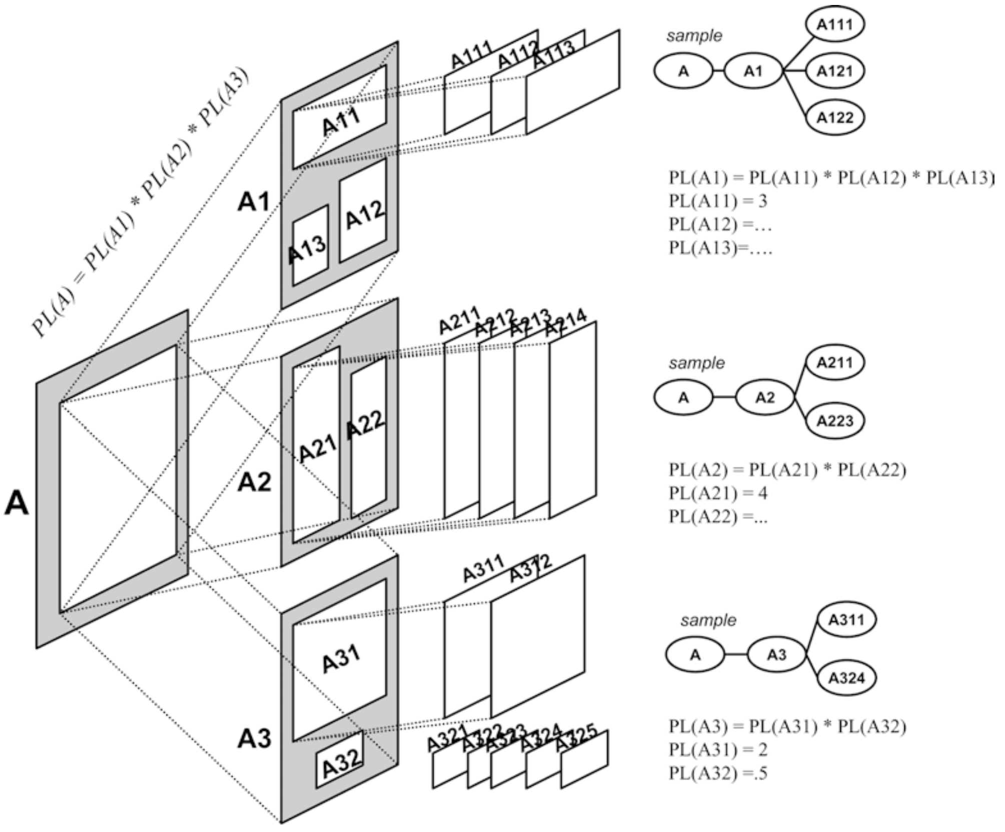


Fig. 13 The concept of parametric container hierarchies

ponent itself, making it reusable across different applications;

– Automatic updates and extensions of components are directly supported, enabling new versions, or even new dialogue components (addressing more user- and usage-context attribute values), to be centrally installed in appropriate component repositories.

6.4 An emphasis on extensibility, maintenance, and code sharing

It is well known in software engineering that "the more we share, the less we have to change". This principle reflects the ability to globally apply changes in common software artefacts "in one shot", assuming they are reused as they are by different components of a software application. On the opposite side, through replication and customisation of similar but slightly different software structures, one has to pay the overhead of manually updating all distinct occurrences of the replicated software structure. The latter approach is known to introduce "entropy increase" in software development, while requiring the repetition of update schemes in similar modules, residing in different source code bases. The theoretical solution to the replication syndrome is the mechanism of abstraction inherent in OOP languages, emphasising sharing of responsibilities, with standardised object collaborations. In unified user interface development, those principles are directly reflected in the organisational structure of dialogue components, promoting code sharing for the invariant interface components, with parametric containment of the variant design alternatives. Such an approach has been practically proved to lead to more easily extensible and maintainable systems, reducing the effort necessary to introduce variations of functionality or to globally modify common interface artefacts.

6.5 Additional implementation issues

6.5.1 Tool limitations in open parametric containment

The concept discussed above is practically constrained by limitations on the physical containment regulations imposed by container interaction objects in different toolkits. More specifically, concrete contained components are bound to be developed using the same interaction object library as the particular container instance component. Two possible development policies have been tested for this scenario:

– Trying to mix containers from different toolkits through the same programming language. This option did not work with GUI toolkits at all, starting even from the compile/link phases. Different toolkits simply define common super-classes, posing either compile conflicts (e.g., "replicate definitions" has been the most common error) or link conflicts (libraries cannot be combined together).

– Trying to employ component technologies, taking advantage of inter-operability among different component technologies. In particular, the following implementations have been tested: Java containers/contained classes as Java beans, and MFC containers/contained classes as ActiveX components. Overall, this has proved to be a quite demanding task, while it did not work in both directions: (a) the use of ActiveX containers containing Java Beans components works through the use of the Java Beans Bridge for Active X, the latter being part of Java plug-ins; the "bridge" enables packaging of Java Beans in the form of Active X components (the "bridge" at runtime looks to the system like an ActiveX control, while at the same time giving to Java Beans a Java environment); and (b) the use of JavaBeans containers with embedded Active X components required the manual re-implementation of ActiveX contained components in the form of Java Beans, through the use of the Java Beans Migration Assistant for Active X; therefore, excluding chances for run-time inter-operability.

6.5.2 The exposure of a generic interaction monitoring API

Following the micro-architecture of the DPC, the management of monitoring requests as well as the posting of dynamically collected monitoring data to the UIS is centralised to a single embedded component. This does not merely reflect a logical organization, but it is the implementation approach that has been employed in the context of the AVANTI browser. To achieve such a centralised monitoring control, all dialogue components need to expose (i.e., implement) one common programming interface (i.e., an abstract class), for installing/un-installing monitoring functionality (mainly event handlers). This particular API (see Fig. 14 for an outline of the dialogue component implementation components), enabled the registration of appropriate handlers by the monitoring manager, to be called by respective dialogue components each time interaction events subject to monitoring are detected. The monitoring manager would then package such events by means of monitoring data messages, to be subsequently posted to the UIS.

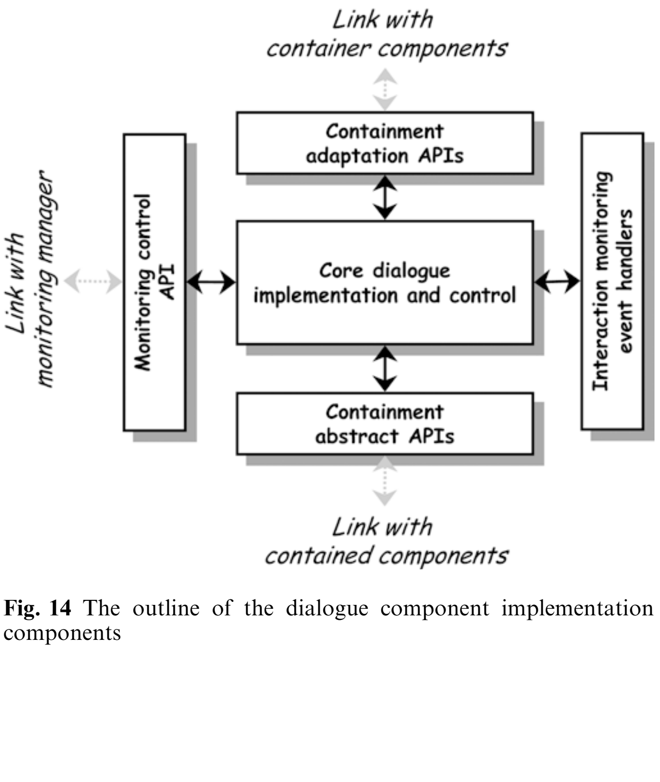


Fig. 14 The outline of the dialogue component implementation components

It should be noted that, to make the abstract monitoring API as generic as possible, it was decided to employ typical string representations for event categories and event data, making each component responsible for dynamic interpretation and encoding. Thus, we avoided the need for updating the monitoring manager each time new event classes or event data types emerged. Moreover, this approach enabled us to package the core functionality of the DPC as a single invariant re-usable component, clearly separated from the implementation of the dialogue components.

7 Conclusions

Equal participation of all citizens in the information society is a recognised socio-technical imperative that presents software vendors with a variety of challenges. Eliminating or reducing the digital divide necessitates the delivery of a wide range of software applications and services, for a variety of computing platforms, accessible and usable by diverse target user groups, including people with disabilities and elderly people. Software firms will be encouraged to work more actively towards this objective if a cost-effective engagement strategy can be clearly formulated. The required technical knowledge to build user- and usage-context adapted interfaces may include user modeling, task design, action patterns, cognitive psychology, rule-based systems, network communication and protocols, multi-platform interfaces, component repositories, and core software engineering. Moreover, the development of monolithic systems, encapsulating the necessary computable artefacts from those domains of required expertise, is not a viable development strategy. Additionally, software developers prefer incremental engagement strategies, allowing a stepwise entrance to new potential markets, by delivering successive generations of products encompassing layers of novel characteristics. Similarly, the development of software applications supporting universal access, i.e., software applications accessible by anyone, anywhere, and at any time, requires a concrete strategy supporting evolutionary development, software re-use, incremental design, and modular construction.

The unified user interface development discussed in this paper claims to offer a software engineering proposition that consolidates process-oriented wisdom for constructing universally accessible interactions. Evolution, incremental development and software reuse are some of the fundamental features of unified user interface development. These are reflected in the ability to progressively extend a unified user interface, by incrementally encapsulating computable content in the different parts of the architecture, to cater for additional users and usage contexts, by designing and implementing more dialogue artefacts, and by embedding new rules for the decision-making logic. Such characteristics are particularly important and relevant to the claimed feasibility and viability of the proposed software engineering process and directly facilitate the practical accomplishment of universally accessible interactions.

The concept of unified user interfaces reflects a new software engineering paradigm that addresses effectively the need for interactions automatically adapted to the individual end-user requirements and the particular context of use. Following this technical approach, interactive software applications encompass the capability to appropriately deliver "on-the-fly" an adapted interface instance, performing appropriate run-time processing that engages:

- The utilisation of user- and usage-context oriented information (e.g., profiles), as well as the ability to detect dynamically user- and usage-context attributes during interaction;
- The management of appropriate alternative implemented dialogue components, realising alternative ways for physical-level interaction;
- Adaptation-oriented decision making that facilitates: (a) the selection, before initiation of interaction, of the most appropriate dialogue components comprising the delivered interface, given any particular dialogue context, for the particular end-user and usage-context profiles (i.e., adaptability); and (b) the implementation of appropriate changes in the initially delivered interface instance, according to dynamically detected user- and usage-context attributes (i.e., adaptivity);
- Run-time component co-ordination and control, to dynamically assemble or alter the target interface; this user interface is composed "on-the-fly" from the set of dynamically selected constituent dialogue components.

The unified user interface development strategy provides a distributed software-architecture with well-defined functional roles (i.e., which component does what), inter-communication semantics (i.e., which component requests what and from whom), control-flow (i.e., when to do what), and internal decomposition (i.e., how the implementation of each component is internally structured). One of the unique features of this development paradigm is the emphasis on dynamic interface assembly for adapted interface delivery, reflecting a software engineering practice with: repository-oriented component organisation, parametric containers with abstract containment APIs, and common interaction-monitoring control with abstract APIs. Although the method itself is not intended to be intensively prescriptive from the low-level implementation point of view, specific successful practices that have been technically validated in field work regarding decision making and dynamic user-attribute detection, have also been discussed, focusing on micro-architecture details and internal functional decomposition.

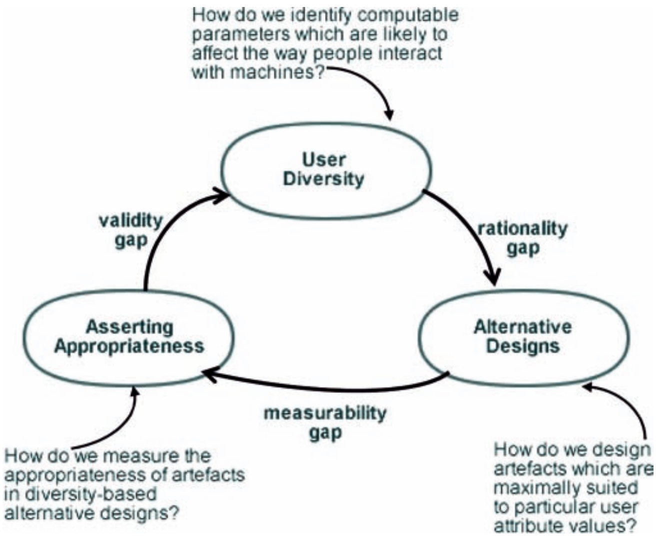

How do we identify computable parameters which are likely to affect the way people interact with machines?
User Diversity
validity gap
rationality gap
Asserting Appropriateness
Alternative Designs
measurability gap
How do we measure the appropriateness of artefacts in diversity-based alternative designs?
How do we design artefacts which are maximally suited to particular user attribute values?

Fig. 15 Further research and development work for unified user interfaces

The unified user interface development was presented in several tutorials [37, 39, 40, 41], and was the development approach adopted in several research projects partially funded by the European Commission[2] - (TP1001 - ACCESS; IST-1999–20656 – PALIO; ACTS AC042 – AVANTI), or national government funding agencies (e.g., EPET-II NAUTILUS).

In this context, this development method has been systematically deployed and tested in practical situations where universal access for computer-based interactive applications and services was the predominant issue. It introduces the fundamental notion of adapted interface delivery "before initiation of interaction", and addresses the technical challenges of coping with the inherent run-time dynamic interface assembly process.

The proposed approach establishes one possible technical route towards constructing universally accessible interactions: it enables incremental development and facilitates the expansion and upgrade of dialogue components as an ongoing process, entailing the continuous engagement and consideration of new design parameters, and new parameter values. It is anticipated that future research work may reveal alternative approaches or methods. At the same time, further research and development work for unified user interfaces is required to address some existing challenges, mainly related to design issues (see Fig. 15).

[2] See the Acknowledgements section

Following Fig. 15, one top-level issue concerns the way that specific varying user attributes affecting interaction are to be identified. In other words, there is a need to *identify diversity* in those human characteristics that are likely to dictate alternative dialogue means.

Subsequently, even if a set of those attributes is identified, it is still unclear how to conduct a design process to produce the necessary alternative dialogue artefacts for the different values of those attributes. Hence, it is necessary to *design for diversity*, relying upon appropriate design rationale clearly relating diverse attribute values with specific properties of the target dialogue artefacts. Currently, there is only limited knowledge about how to perform effectively this transition from alternative user-attribute values to alternative design artefacts, and it can be characterised as a *rationalisation gap*.

Finally, the issue of how to structure appropriately alternative patterns for diverse user attribute values should be addressed in a way that the resulting designs are indeed efficient, effective and satisfactory for their intended users and usage-contexts. Such a process requires appropriate evaluation methods, and the capability to measure the appropriateness of designed artefacts. At present, this is still a "missing link", characterised as the *measurability gap*. Unless we are able to assert the appropriateness of the alternative dialogue artefacts designed for diverse user attributes, we cannot validate the overall dynamically delivered interface. The inability to formulate and conduct such an evaluation process creates a *validity gap*.

Work currently underway, as well as future work, is expected to address these issues in an attempt to bridge the identified gaps.

Acknowledgements The authors wish to acknowledge with appreciation the guidance of the managing editor and the contribution of the anonymous reviewers in significantly improving the quality of the manuscript during the peer review process. The unified interface development method has been originally defined in the context of the ACCESS TP1001 (the development platform for unified ACCESS to enabling environments) project, partially funded by the TIDE Program of the European Commission, and lasted 36 months (from January 1st, 1994 to December 31st, 1996). The partners of the ACCESS consortium are: CNR-IROE (Italy) - Prime contractor; ICS-FORTH (Greece); University of Hertforshire (UK); University of Athens (Greece); NAWH (Finland); VTT (Finland); Hereward College (UK); RNIB (United Kingdom); Seleco (Italy); MA Systems & Control (UK); PIKOMED (Finland). The first large-scale application of the unified interface development method has been carried out in the context of the AVANTI AC042 (Adaptable and Adaptive Interaction in Multimedia Telecommunications Applications) project, partially funded by the ACTS Program of the European Commission, and lasted 36 months (from September 1st, 1995 to August 31st, 1998). The partners of the AVANTI consortium are: ALCATEL Italia, Siette division (Italy) - Prime Contractor; IROE-CNR (Italy); ICS-FORTH (Greece); GMD (Germany), VTT (Finland); University of Siena (Italy), MA Systems and Control (UK); ECG (Italy); MATHEMA (Italy); University of Linz (Austria); EUROGICIEL (France); TELECOM (Italy); TECO (Italy); ADR Study (Italy). The PALIO "Personalised Access to Local Information and Services for Tourists" project (IST-1999–20656) is partly funded by the Information Society Technologies Program of the European Commission—DG Information Society. The partners in the PALIO consortium are: ASSIOMA S.p.A. (Italy)—Prime Contractor; CNR-IROE (Italy); Comune di Firenze (Italy); FORTH-ICS (Greece); GMD (Germany); Telecom Italia Mobile S.p.A. (Italy); University of Sienna (Italy); Comune di Siena (Italy); MA Systems and Control Ltd (UK); FORTHnet (Greece). The NAUTILUS "Unified Web Browser for People with Disabilities" project is funded by EPET-II Programme (Operational Programme for Research & Technology of the General Secretariat for Research and Technology, Hellenic Ministry of Development). The partners in the NAUTILUS Consortium are: ICS-FORTH (Greece), TRD International SA (Greece) and NCDP (Greece).

References

1. ACCESS Project (1996) The ACCESS project—development platform for unified access to enabling environments. RNIB Press, London
2. Akoumianakis D, Savidis A, Stephanidis C (1996) An expert user interface design assistant for deriving maximally preferred lexical adaptability rules. In: Proceedings of the 3rd World Congress on Expert Systems, Seoul, Korea, 5–9 February 1996
3. Benyon D (1984) MONITOR: a self-adaptive user-interface. In: Proceedings of the IFIP Conference on Human-Computer Interaction: INTERACT '84 (vol. 1), Elsevier, Amsterdam
4. Blattner MM, Glinert JA, Ormsby GR (1992) Metawidgets: towards a theory of multimodal interface design. In: Proceedings of COMPSAC '92, IEEE Computer Society Press, New York
5. Browne D, Norman M, Adhami E (1990) Methods for building adaptive systems. In: Browne D, Totterdell M, Norman M (eds) Adaptive user interfaces, Academic Press, London
6. Browne D, Totterdell M, Norman M (eds) (1990) Conclusions. Adaptive user interfaces, Academic Press, London
7. Cockton G (1987) Some critical remarks on abstractions for adaptable dialogue managers. In: Proceedings of the 3rd Conference of the British Computer Society, People & Computers III, HCI Specialist Group, University of Exeter, Cambridge University Press, Cambridge, UK
8. Cockton G (1993) Spaces and distances—software architecture and abstraction and their relation to adaptation. In: Schneider-Hufschmidt M, Kühme T, Malinowski U (eds) Adaptive user interfaces—principles and practice, Elsevier, Amsterdam
9. Cote Muñoz J (1993) AIDA—an adaptive system for interactive drafting and CAD applications. In: Schneider-Hufschmidt M, Kühme T, Malinowski U (eds) Adaptive user interfaces—principles and practice, Elsevier, Amsterdam
10. Coutaz J (1990) Architecture models for interactive software: failures and trends. In: Cockton G (ed) Engineering for human-computer interaction, Elsevier, Amsterdam
11. Dieterich H, Malinowski U, Kühme T, Schneider-Hufschmidt M (1993) State of the art in adaptive user interfaces. In: Schneider-Hufschmidt M, Kühme T, Malinowski U (eds) Adaptive user interfaces—principles and practice, Elsevier, Amsterdam
12. Foley J, Kim W, Kovacevic S, Murray K (1991) GUIDE—an intelligent user interface design environment. In: Sullivan J, Tyler S (eds) Architectures for intelligent interfaces: elements and prototypes, Addison-Wesley, Reading, MA
13. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns, elements of reusable object-oriented software. Addison-Wesley, Reading, MA
14. Goldberg A (1984) Smalltalk-80: the interactive programming environment. Addison-Wesley, Reading, MA
15. Grammenos D, Stephanidis C (2002) Interaction design of a collaborative application for children. In: Bekker MM, Markopoulos P, Kersten-Tsikalkina M (eds) Proceedings of the International Workshop Interaction Design and Children, Eindhoven, The Netherlands, 28–29 August, 2002

16. Green M (1985) Report on dialogue specification tools. In: Pfaff G (ed) User interface management systems, Springer, Berlin Heidelberg New York
17. GUIB Project (1995) Textual and graphical user interfaces for blind people. The GUIB PROJECT—Public Final Report, RNIB Press, UK
18. Hartson R, Hix D (1989) Human-computer interface development: concepts and systems for its management. ACM Comp Surv 21(1):241–247
19. Jacobson I, Griss M, Johnson P (1997) Making the reuse business work. IEEE Comp 10:36–42
20. Kawai S, Aida H, Saito T (1996) Designing interface toolkit with dynamic selectable modality. In: Proceedings of the Second Annual ACM Conference on Assistive Technologies (ASSETS '96), Vancouver, Canada, 11–12 August 1996
21. Kobsa A (1990) Modeling the user's conceptual knowledge in BGP-MS, a user modeling shell system. Comp Intellig 6:193–208
22. Kobsa A, Pohl W (1995) The user modeling shell system BGP-MS. User Model User Adapt Inter 4(2):59–106
23. Kobsa A, Wahlster W (eds) (1989) User models in dialog systems. Springer, Berlin Heidelberg New York
24. Krasner GE, Pope ST (1988) A description of the model view controller paradigm in the Smalltalk-80 system. J Obj-Orient Prog 1(3):26–49
25. Mowbray TJ, Zahavi R (1995) The essential CORBA: systems integration using distributed objects. Wiley, New York
26. Myers B (1995) User interfaces software tools. ACM Trans Hum-Comp Inter 12(1):64–103
27. Mynatt E, Weber G (1994) Nonvisual presentation of graphical user interfaces: contrasting two approaches. In: Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '94), ACM Press, New York
28. Puerta AR (1997) A model-based interface development environment. IEEE Soft 14(4):41–47
29. Savidis A, Stephanidis C (1995a) Developing dual interfaces for integrating blind and sighted users: the HOMER UIMS. In: Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '95), Denver, Colorado, 7–11 May 1995
30. Savidis A, Stephanidis C (1995b) Building non-visual interaction through the development of the rooms metaphor. In: Companion Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '95), Denver, Colorado, 7–11 May 1995
31. Savidis A, Stephanidis C (1998) The HOMER UIMS for dual user interface development: fusing visual and non-visual interactions. Int J Interact Comp 11(2):173–209
32. Savidis A, Stephanidis C (2001) Development requirements for implementing unified user interfaces. In: Stephanidis C (ed) User interfaces for all—concepts, methods, and tools, Lawrence Erlbaum, Mahwah, NJ
33. Savidis A, Stergiou A, Stephanidis C (1997a) Generic containers for metaphor fusion in non-visual interaction: the HAWK interface toolkit. In: Proceedings of the 6th International Conference on Man-Machine Interaction Intelligent Systems in Business (INTERFACES '97), Montpellier, France, 28–30 May 1997
34. Savidis A, Vernardos G, Stephanidis C (1997b) Embedding scanning techniques accessible to motor-impaired users in the WINDOWS object library. In: Salvendy G, Smith MJ, Koubek RJ (eds) Design of computing systems: cognitive considerations. In: Proceedings of the 7th International Conference on Human-Computer Interaction (HCI International '97), San Francisco, CA, 24–29 August 1997
35. Schwerdtfeger RS (1991) Making the GUI talk. BYTE 16(12):118–128
36. Short K (1997) Component based development and object modeling. Texas Instruments Software, version 1.0
37. Stephanidis C, Akoumianakis D, Paramythis A (1999) Coping with diversity in HCI: techniques for adaptable and adaptive interaction. Tutorial No. 11 of the 8th International Conference on Human-Computer Interaction (HCI International '99), Munich, Germany, 22–26 August 1999
38. Stephanidis C, Paramythis A, Sfyrakis M, Savidis A (2001) A case study in unified user interface development: The AVANTI Web browser. In: Stephanidis C (ed) User interfaces for all—concepts, methods, and tools, Lawrence Erlbaum, Mahwah, NJ
39. Stephanidis C, Savidis A, Akoumianakis D (1997) Unified user interface development: tools for constructing accessible and usable user interfaces. Tutorial No.13 of the 7th International Conference on Human-Computer Interaction (HCI International '97), San Francisco, CA, 24–29 August 1997
40. Stephanidis C, Savidis A, Akoumianakis D (2001a) Engineering universal access: unified user interfaces. In: Tutorial of the 1st Universal Access in Human-Computer Interaction Conference (UAHCI 2001), jointly with the 9th International Conference on Human-Computer Interaction (HCI International 2001), New Orleans, LA, 5–10 August 2001
41. Stephanidis C, Savidis A, Akoumianakis D (2001b) Universally accessible UIs: the unified user interface development. Tutorial of the ACM Conference on Human Factors in Computing Systems (CHI 2001), Seattle, Washington, 31 March–5 April 2001
42. Szekely P, Luo P, Neches R (1992) Facilitating the exploration of interface design alternatives: the HUMANOID model of interface design. In: Proceedings of the the ACM Conference on Human Factors in Computing Systems (CHI 1992) ACM Press, New York
43. UIMS Developers Workshop (1992) A meta-model for the runtime architecture of an interactive system. SIGCHI Bullet 24(1):32–37
44. Vergara H (1994) PROTUM—a Prolog based tool for user modeling. Bericht Nr. 55/94 (WIS-Memo 10), University of Konstanz, Germany