MENTOR
An Interactive Design Environment for Automatic User Interface Adaptation

Margherita Antona, Anthony Savidis
and Constantine Stephanidis

ABSTRACT

Unified User Interface design is a method targeted towards the design of user interfaces capable of automatic adaptation behaviour. MENTOR is a tool that has been developed in order to facilitate the process of designing Unified User Interfaces, by providing an appropriate interaction environment and editing facilities for all the phases involved. Automated verification mechanisms for the adaptation logic embedded in design cases are also provided. This Report describes the functionality and interactive facilities of MENTOR, focussing on how the tool supports the underlying method.

Keywords:
Automatic interface adaptation, Unified User Interface design, Unified User Design support tool
# Table of contents

INTRODUCTION ............................................................................................................................ 2
THE UNIFIED USER INTERFACE DEVELOPMENT FRAMEWORK .............................................. 5
THE MENTOR OVERALL ENVIRONMENT .................................................................................. 13
THE DESIGN PARAMETERS EDITOR ....................................................................................... 16
THE STEREOTYPES EDITOR ..................................................................................................... 20
THE POLYMORPHIC TASK HIERARCHY EDITOR ..................................................................... 33
THE PROPERTIES EDITOR ........................................................................................................ 39
THE DESIGN DOCUMENTATION VIEWER ................................................................................ 47
THE MENTOR HELP ................................................................................................................ 49
AUTOMATED VERIFICATION OF ADAPTATION DESIGN IN MENTOR ................................. 50
CONCLUSIONS .......................................................................................................................... 52
REFERENCES ............................................................................................................................. 54
Introduction

MENTOR is an interactive tool for process-oriented support for user interface design that facilitates the design of user- and context-dependent adaptations. Process-oriented support is intended as facilitating the conduct of all phases of an appropriate design method, the encoding of the related results and their direct transformation into user interface specifications. In this respect, process-oriented support share many of the requirements, but is in essence orthogonal, to other forms of design support usually embedded into tools, such as access to relevant design guidelines, design critique, design automation, etc. The underlying rationale in the development of MENTOR is that appropriate process-oriented tool-based support has the potential to contribute to the wider adoption of design practices that take into account user and context diversity in a Universal Access perspective ([Stephanidis et al., 1998], [Stephanidis et al., 1999], [Stephanidis, 2001a]), as well as to facilitate the transition from the design to the implementation phase of user interfaces capable of run-time adaptation behaviour and the possible incremental extension and reuse of design cases.

A suitable design framework for Universal Access is identified in the Unified User Interface design method ([Savidis and Stephanidis, 2004a]), which has been elaborated in recent years as a method supporting the design of user interface capable of automatic adaptation as a technical path towards Universal Access. Such a method does not produce as an output a single design artefact, but a design space populated by different artefacts interrelated through a run-time adaptation logic that captures the interface’s run-time adaptation behaviour. The Unified User Interface design method has been practically applied and validated in the course of large design cases, but its wider adoption and cost-effective application is currently hindered by the lack of process-oriented computer-based tools. As a contribution towards overcoming such a barrier, the MENTOR prototype aims at: (i) facilitating the conduct of Unified User Interface design by providing an appropriate interaction environment and editing facilities for all the phases involved in Unified User Interface design, as well as automated verification mechanisms for the adaptation logic embedded in design cases, and (ii) constituting a preliminary step towards the further investigation of design practices for Universal Access by facilitating the accumulation and reuse of
Unified User Interface design cases. The main identified objectives for such a tool are:

- To provide an appropriate instrument, orthogonal to other forms of design support, for widening and improving the practice of adaptation-based user interface design (from learning to practice).
- To provide practical integrated support for all phases of Unified User Interface Design, by guiding the process and structuring the outcomes of creative design steps through appropriate editing facilities.
- To provide practical support for the transition from design to development of Unified User Interfaces, through the availability of appropriate automated verification and specification generation mechanisms.
- To provide support for the progressive accumulation of Unified User Interface design cases and of the related design experience and knowledge, in particular regarding adaptation, and to offer means for extending and re-using (parts of) past design cases.

This Report describes the functionality and interactive features of MENTOR. The followed approach was that of user-centred design, due to the rather homogeneous target user group, usage context and interactive platform addressed by MENTOR, leading to a very limited need of diversification, which could be easily addressed in the tool without introducing adaptations.

The target user group of MENTOR mainly comprises user interface designers. In order for the tool to contribute to widen the adoption of universal design methods, it was considered reasonable to assume:

- good expertise of target users with (possibly different) task-based design techniques;
- some degree of overall acquaintance with the rationale and fundamentals of the Unified User Interface design method;
- possible lack of initial expertise in the details of the method, particularly concerning the aspects related to defining and encoding different styles and the related adaptation conditions, as well as adaptation relations among styles.

Additionally, the usage context of the MENTOR has been assumed to be the traditional desktop, accessed through standard graphical and direct manipulation
interaction techniques. The current MENTOR prototype has been completely implemented in Microsoft Visual Basic v. 6.0.

The Report is organised as follows. Following a brief description of the Unified User Interface development methodology and of the related Unified User Interface design method, the MENTOR tool’s functionality and interactive facilities are introduced, focussing on how they support the iterative phases and steps of Unified User Interface design.
The Unified User Interface development framework

The Unified User Interface development methodology ([Stephanidis, 2001b], [Savidis and Stephanidis, 2004b]) has been proposed in the recent past as a complete technological solution for supporting Universal Access of interactive applications and services. Unified User Interfaces convey a new perspective into the development of user interfaces, providing a principled and systematic approach towards coping with diversity in the target user requirements, tasks and environments of use ([Stephanidis, 2001a]). The theoretical grounds, upon which Unified User Interface development methodology is based, is provided by the concept of User Interfaces for All, rooted in the idea of applying Universal Design in the field of Human-Computer Interaction (HCI) ([Stephanidis, 2001a]). Subsequently, these principles were extended and adapted to depict a general proposition for HCI design and development, and were intensively tested and validated in the course of real projects ([Stephanidis et al., 2001], [Stephanidis et al., 2004]). Unified User Interfaces provide an engineering methodology supporting automatic adaptation of user interfaces as a technical path towards Universal Access ([Stephanidis, 2001b]).

A Unified User Interface comprises a single (unified) interface specification that exhibits the following properties ([Savidis and Stephanidis, 2004b]):

(i) It embeds representation schemes for user- and usage-context- parameters and accesses user- and usage-context- information resources (e.g., repositories, servers), to extract or update such information.

(ii) It is equipped with alternative implemented dialogue patterns (i.e., implemented dialogue artefacts) appropriately associated to different combinations of values for user- and usage-context- related parameters. The need for such alternative dialogue patterns is identified during the design process, when, given a particular design context, for differing user- and usage-context- attribute values, alternative design artefacts are deemed as necessary to accomplish optimal interaction.

(iii) It embeds design logic and decision making capabilities that support activating, at run-time, the most appropriate dialogue patterns according to particular
instances of user- and usage-context- parameters, and is capable of interaction monitoring to detect changes in parameters.

As a consequence, a unified interface realises:

- User-adapted behaviour (user awareness), i.e., the interface is capable of automatically selecting interaction patterns appropriate to the particular user.
- Usage-context adapted behaviour (usage context awareness), i.e., the interface is capable of automatically selecting interaction patterns appropriate to the particular physical and technological environment.

From a user perspective, a Unified User Interface can be considered as an interface tailored to personal attributes and to the particular context of use, while from the designer perspective it can be seen as an interface design populated with alternative designs, each alternative addressing specific user- and usage-context- parameter values. Finally, in an engineering perspective, a Unified User Interface is a repository of implemented dialogue artefacts, from which the most appropriate according to the specific task context are selected at run-time by means of an adaptation logic supporting decision-making ([Savidis and Stephanidis, 2004b]).

At run-time, the adaptations may be of two types:

(a) adaptations driven from initial user- and context- information known prior to the initiation of interaction, and

(b) adaptations driven by information acquired through interaction monitoring analysis.

The former behaviour is referred to as adaptability (i.e., initial automatic adaptation) reflecting the interface’s capability to automatically tailor itself initially to each individual end-user in a particular context. The latter behaviour is referred to as adaptivity (i.e., continuous automatic adaptation), and characterizes the interface’s capability to cope with the dynamically changing or evolving user and context characteristics. Adaptability is crucial to ensure accessibility, since it is essential to provide, before initiation of interaction, a fully accessible interface instance to each individual end-user ([Stephanidis, 2001b]). Furthermore, 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 / emerging decision parameters that may drive dynamic interface enhancements. This combination of adaptation characteristics and behaviour
make Unified User Interfaces suitable and appropriate for supporting Universal Access ([Stephanidis, 2001b]).

The concept of Unified User Interface is supported by a specifically developed architecture ([Savidis and Stephanidis, 2001]). This architecture consists of independent communicating components, possibly implemented with different software methods and tools that cooperate to perform the types of adaptation mentioned above.

Figure 1 (from [Savidis and Stephanidis, 2001]) depicts the components of the unified interface architecture:

- The Dialogue Patterns Component is responsible for supplying the software implementation of the dialogue artefacts identified in the design process as belonging to a unified interface. Such components may be common across different user- and usage-context- attribute values, in the case no adaptation is needed, or dialogue artefacts that are appropriate for specific attribute values, in case alternative designs for adaptation have been identified.

- The Decision Making Component has the role of deciding at run-time the necessary automatic adaptation actions, and to subsequently communicate such decisions to the Dialogue Patterns Component, which applies them.
• The User Information Server supplies user attribute values either known off-line, without performing interaction monitoring analysis (e.g., motor / sensory abilities, age, nationality, etc., or detected on-line, from real-time interaction-monitoring analysis (e.g., fatigue, loss of orientation, inability to perform the task, interaction preferences, etc.).

• The Context Parameters Server supplies context attribute values (machine and environment) either invariant, i.e., unlikely to change during interaction (e.g., peripheral equipment), or dynamically changing during interaction (e.g., environment noise, failure of particular equipment, etc.). This component does not support device independence, but device awareness, and enables the Decision Making Component to select those interaction patterns, which, apart from fitting the particular end-user attributes, are also appropriate for the type of equipment available to the end-user.

The Unified User Interface development method is not prescriptive regarding how each component is to be implemented ([Savidis and Stephanidis, 2004b]). For example, the User Information Server component may employ alternative ways of representing user-oriented information. A repository of user profiles can serve as a central database of individual user information (i.e., registry). A quite simple but powerful and flexible approach is to represent profiles as a typical list of attributes and related values. More sophisticated user representation and modelling methods can be also employed, including support for stereotypes of particular user categories. In

<table>
<thead>
<tr>
<th>User profile model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
</tr>
<tr>
<td>P₁</td>
</tr>
<tr>
<td>P₂</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Pₙ</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>User profile instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>computer knowledge</td>
</tr>
<tr>
<td>Web knowledge</td>
</tr>
<tr>
<td>ability to use left hand</td>
</tr>
</tbody>
</table>

Figure 2. Example of User Profile Model and Instance
case dynamic user attribute detection is supported, the content may include dynamically collected interaction monitoring information, design information and knowledge processing components. Figure 2 (from [Savidis and Stephanidis, 2004b]) depicts an example of an attribute value based user profile model and instance. Similar considerations hold for the Context Parameters Server.

On the other hand, the Decision Making Component encompasses the logic for deciding the necessary adaptation actions, on the basis of the user- and context-attribute values, received from the User Information Server and the Context Parameters Server, respectively. Such adaptation logic is assumed in the Unified User Interface framework to reflect design decisions, i.e., it is not generated automatically. This implies that the inference mechanisms that drive adaptations at run-time are based on adaptation decisions that have been validated during the design phase of the unified interface. This approach 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 run-time. In practice, adaptations in the Unified User Interfaces framework can be implemented through a rule-based approach, which is considered as practically adequate and well-accepted by interface designers ([Savidis and Stephanidis, 2004b]). An additional consideration is that the outcomes of the decision process need to be represented in a form suitable for being communicated and easily interpreted by the Dialogue Patterns Component. In this respect, two categories of dialogue control actions are considered as providing the expressive power necessary for applying both adaptability and adaptivity ([Savidis and Stephanidis, 2004b]): (i) activation of specific dialogue components; and (ii) cancellation of previously activated dialogue components. Substitution of interaction artefacts is modelled through 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). A recent development in the context of decision – making support for Unified User Interface development, facilitating the embedding of decision-making logic in a way directly editable by designers, is the Decision Making Specification Language (DMSL, [Savidis and Stephanidis, 2004b], [Savidis et al., 2004]). DMSL is organised in “if…then…else” blocks, and each block is associated to a particular dialogue context. The individual end-user and usage-context profiles are represented using the
user. and context. prefix notation, and attribute values are defined as matching quoted strings. The language is equipped with three primitive statements: (a) dialogue, which initiates evaluation for the rule block corresponding to dialogue context value supplied; (b) activate, which triggers the activation of the specified component(s); and (c) cancel, which, similarly to activate, triggers the cancellation of the specified component(s). These rules are compiled in a tabular representation that is executed at run-time. The representation engages simple expression evaluation trees for the conditional expressions.

Figure 3 (from [Savidis and Stephanidis, 2004b]) provides an example of rules in the DMLS language.

<table>
<thead>
<tr>
<th>Dialogue: “toolbar expert”</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (user.&quot;web expertise&quot; = &quot;expert&quot; or user.&quot;web expertise&quot; = “frequent”) then</td>
</tr>
<tr>
<td>[</td>
</tr>
<tr>
<td>if (context.&quot;installation&quot; = “kiosk”)</td>
</tr>
<tr>
<td>dialogue “kiosk toolbar expert”;</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>if (context.&quot;installation&quot; = “desktop”)</td>
</tr>
<tr>
<td>dialogue “desktop toolbar expert”;</td>
</tr>
<tr>
<td>]</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>activate “empty”;</td>
</tr>
<tr>
<td>]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dialogue: “kiosk toolbar expert”</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (user.&quot;visual ability&quot; = “blind”) then</td>
</tr>
<tr>
<td>activate “kiosk toolbar expert hawk”;</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>if (user.&quot;visual ability&quot; = “sighted”) then</td>
</tr>
<tr>
<td>activate “kiosk toolbar expert gui”;</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>activate “user profile error”; // Unexpected user profile attribute value.</td>
</tr>
<tr>
<td>]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dialogue: “desktop toolbar expert”</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (user.&quot;visual ability&quot; = “blind”) then</td>
</tr>
<tr>
<td>activate “desktop toolbar expert hawk”;</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>if (user.&quot;visual ability&quot; = “sighted”) then</td>
</tr>
<tr>
<td>activate “desktop toolbar expert gui”;</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>activate “user profile error”; // Unexpected user profile attribute value.</td>
</tr>
<tr>
<td>]</td>
</tr>
</tbody>
</table>

Figure 3. An example of DMLS rules
The implementation of a Unified User Interface can be either explicitly programmed or produced by compiling an interface specification, and maps abstract interaction elements to concrete / physical resources available in the target toolkits. This is achieved by utilizing specific functionality or tools (e.g., toolkit servers) to connect (or link) with the underlying platform(s) in order to utilize the available interaction resources in a manner that is independent from the platform ([Savidis and Stephanidis, 2004b]).

The Unified User Interface development paradigm has been carefully elaborated in such a way as to be general enough, so as not to exclude particular design and implementation practices, while, at the same time, it offers sufficient details to drive the engineering process. As any new development paradigm, Unified User Interface development requires some initial investment to be effectively adopted, assimilated and applied. However, if the constructed software products are intended to be used by user populations with diverse requirements, operated in different usage contexts, it is argued that the gains will outweigh the overhead of additional resources that need to be invested ([Stephanidis, 2001a], [Savidis and Stephanidis, 2004b]). A particularly important aspect of developing unified user interfaces concerns their design.

The Unified Interface design method is a hybrid process-oriented design method enabling the organization of diversity-based design decisions around a single hierarchical structure, purposefully developed to support the design of Unified User Interfaces ([Savidis et al, 2001], [Savidis and Stephanidis, 2004a]). It proposes a specific design process to cater for the management of an evolving design space, in which alternative design artefacts can be associated to variations of the design problem parameters.

The method addresses the following main objectives:

1. Enabling the collection and organisation of all design alternatives required for adaptation into a single unified user interface design space which can be produced during a single design phase.

2. Documenting, for each alternative design artefact, a design rationale, in a form that facilitates implementation, including the run-time relationships with the rest of the artefacts within the same design context, as well as the specific associated problem parameters values.
3. Supporting design evolution, by enabling the effective extension of different design contexts, by addressing new (combination of) user- and usage-context-attribute values.

The process of designing Unified User Interfaces does not lead to a single design outcome, but to a structured design space. It collects and appropriately represents alternative designs, along with the conditions under which each design should be instantiated at run-time (i.e., an adaptation-oriented design rationale).

The Unified User Interface design method encompasses a variety of techniques such as task analysis, abstract design, design polymorphism and design rationale ([Savidis and Stephanidis, 2004a]).

The following sections of this Report describe in parallel the phases and steps involved in Unified User Interface design, and the facilities offered by the MENTOR tool in order to support the method’s conduct and practice.
The MENTOR overall environment

The key elements of the Unified User Interface design process are ([Savidis and Stephanidis, 2004a]):

- Hierarchical design discipline building upon the notion of task analysis, empowered by the introduction of task-level polymorphism.
- Iterative design process model, emphasising abstract task analysis with incremental polymorphic physical specialisation.
- Formalisation of run-time relationships among the alternative design artefacts associated to the same design context.
- Documentation recording the consolidated design rationale of each alternative design artefact.

The overall MENTOR environment is constituted by an MS Windows application with a Multiple Document Interface (MDI) style. MENTOR is based on four separate but interrelated editing facilities, namely:

1. The Design Parameters Editor, which supports the encoding of declarations of design parameters attributes and related value spaces.
2. The Stereotypes Editor, which supports the creation of stereotypes of adaptation conditions (i.e., macros of instantiated combinations of design parameters).
3. The Polymorphic Task Hierarchy editor, which supports the structured process of polymorphic task hierarchy decomposition.
4. The Properties Editor, which supports the specifications of properties of the design artefacts (nodes in the hierarchy).

Additionally, a Design Documentation Viewer is provided for inspecting and delivering the results of a design case. A help facility is also included.

The MENTOR design reflects the need of using these facilities in an iterative and interrelated fashion, and of providing a ‘contextual’ view and composition of a design case. Therefore, the main four editing environments are based on an in-place-editing interactive style, avoiding as much as possible the need to engage the designer in ‘deep’ modal dialogues to perform design tasks, loosing site of the overall picture.
Available functions are supported through icon toolbars and menus. Various design case encoding strategies are freely allowed, with the unique condition that for defining stereotypes and style conditions, some design parameters need to be already declared, as the latter constitute the ‘vocabulary’ out of which stereotypes and conditions are composed. So, for example, a designer may begin a design case by specifying a set of design parameters, then proceed to define stereotypes, and finally use the stereotypes in the condition properties of styles in the polymorphic task hierarchy. Alternatively, it is possible to start a polymorphic task decomposition and gradually add design parameters (and stereotypes) as the need arises. All types of design information can be incrementally added or modified at any time, with the unique condition that design parameters attributes and values can not be deleted or changed if they are in use in some stereotype or style condition.

Figure 4. The overall MENTOR design environment

Figure 4 depicts the overall MENTOR interactive environment. The Design Parameters Editor is visible in the left upper part of the Figure, while the Stereotypes Editor appears in the right upper part. In the lower part of the figure, the Polymorphic Task Hierarchy Editor is displayed, with the Properties Editor overlapping on its
space. All editors can be closed and resized as needed. The Design Parameters Editor, the Stereotypes Editor and the Polymorphic Task Hierarchy Editor can also be minimised and maximised.

The overall MENTOR environment provides standard file management functionality (i.e., opening and saving design cases, saving and printing design documentation), as well as string searching facilities in all components of a design case.

The following sections will describe in details each of the tool’s interactive components.
The Design Parameters Editor

An essential part of Unified User Interface design is the conceptual categorisation of diversity aspects in all relevant dimensions (users, context of use, access terminal / platform), and the identification of the target design parameters for each design case ([Savidis et al, 2001]). There is no predefined or fixed set of attribute categories or values, which are chosen as part of the design process (e.g., by interface designers, or Human Factors experts). Values do not need be finite sets. The broader the set of attributes and related values, the higher the potential for interface adaptation (i.e., for alternative designs). For instance, commercial systems realizing a single design for an “average” user have no differentiation capability at all. The Unified User Interface design method does not provide pre-defined models of relevant parameters, based on the consideration that the degree to which a comprehensive model of design parameters can be achieved based on the current knowledge and wisdom on designing for diversity can not be determined, and, besides, not all parameters in such a model would be relevant for different design cases. For example, according to the targeted final user groups, different sets of design guidelines may need to be taken into account during design. Therefore, designers should be free to use case-specific design parameters, to experiment with different sets of them, and to create their own (partial) taxonomies according to the type of design cases they address and the target user population. Accumulated experience in designing for diversity is expected to progressively lead to the identification of more commonly valid classifications of design parameters. For example, human abilities relevant for determining alternative choices in a design case are likely to affect similar design cases. Some examples of attribute classes that designers may consider in Unified User Interface design are general computer-use expertise, domain-specific knowledge, role in an organizational context, motor abilities, sensory abilities, mental abilities, etc.

Since, as already mentioned, the Unified User Interface design method does not pose any restrictions as to the attribute categories considered relevant, or the target value domains of such attributes, it needs to provide a suitable framework in which the definition of user- and usage-context- attributes constitutes an explicit part of the
design process. It is the responsibility of interface designers to choose appropriate attributes and corresponding value ranges, as well as to define appropriate design alternatives when necessary. For simplicity, designers may choose to elicit only those attributes from which differentiated design decisions are likely to emerge. The construction of context and platform attributes may follow the same representation approach as users’ characteristics. Examples of potential context attributes are acoustic noise and light sources, while examples of potential relevant platform attributes are processor speed, memory, secondary storage, peripheral equipment, resolution, screen physical size and graphics capabilities.

The Design Parameters Editor of MENTOR supports the encoding of flat (non hierarchical) declarations (signature) of design parameters attributes and related value spaces. These constitute the “vocabulary” for defining stereotypes and polymorphic style conditions, i.e., provide the “adaptation space” of the unified user interface under design. Therefore, the Design Parameters Editor has been designed in such a way as to allow easy editing of parameters while preserving the signature consistency. Three types of design parameters values are allowed, according to the specifications of the DMSL language (see previous section): (i) enumerated, i.e., values belong to a list of (more than two) strings specified by the designer; (ii) Boolean, i.e., values True or False; and (iii) integer, which are specified by supplying the minimum and maximum bounds of the integer range allowed as a value. Value ranges define the space of legal values for a given attribute.

Figure 5 depicts the Design Parameters Editor, which displays parameters attributes, type and value ranges into an intuitive three column table.

Parameters can belong to two different domains, namely the user domain, referring to parameters representing user characteristics, and the context domain, referring to parameters representing characteristics of the context of use and of the interactive platform(s) of the unified user interface under design. The domain can be switched by using the related option buttons.

Designers can add new design parameters by using the first button on the icon toolbar. A new attribute is automatically added with a default name and type (Boolean), to be subsequently modified as needed. Attributes and value ranges are editable by double clicking on the related cell of the table. Standard copy and paste facilities are allowed in the editing text boxes. As shown in Figure 5, the type of each attribute is selected through a combo box appearing upon double clicking on the central column of the
table. Enumerated values are separated by commas, while minimum and maximum bounds in integer values ranges are separated by a dash. When new attributes are added, placeholder values are displayed in the corresponding value range cell showing the declaration syntax. The value range True and False for Boolean attributes is automatically set upon selecting the Boolean type for an attribute, and is not editable. It is not possible to assign the same attribute to different parameters, while the same values can be assigned to different parameters.

![Design Parameters Editor](image)

**Figure 5. The MENTOR Design Parameters Editor**

Deletion of a selected design parameter is performed by using the third button on the Design Parameters Editor. A parameter can be deleted if it not used in any stereotype or style condition. The tool also offers the possibility of importing design parameters from other design cases, in order to support reusability of previous designs. The related dialogue, depicted in Figure 6, is activated by using the second button on the Design Parameters Editor icon toolbar.
Designers can choose a design case from which they wish to import design parameters into the current project, and then select the attributes to be imported. Consistency checking between the parameters already declared in the current design case and the parameters selected for importing is automatically performed by the system according to the following policy:

- If the attribute name of a parameter selected for importing is not identical to the attribute name of any of the parameters already declared in the current design case, then the parameter is imported.

- If a parameter has an attribute name identical to the name of a parameters already declared in the current design case, but a non identical type or values range, then the user is requested to confirm the overwriting of the currently defined parameter.

- If a parameter has identical name, type, and value range with any of the already existing parameters in the current design case, it is not imported.
The Stereotypes Editor

One of the fundamental aspects of Unified User Interface Design is the support for associating alternative designs to different combinations of user and context design parameters, and producing an adaptation logic determining the adaptation behaviour of the interface at run-time. In order to support such a process, MENTOR provides an editor for the creation of stereotypes of adaptation conditions (i.e., macros of instantiated combinations of design parameters, thus facilitating the preparation of conditions ready to use in polymorphic task decomposition. Semantically, stereotypes are partial descriptions of users or contexts in the space defined by the current design parameters, and refer to sets of users or contexts for which they hold true at run-time. Each stereotype is, therefore, constituted by a name (e.g., blind user) and an associated condition (e.g., User.vision == False). Conditions are formulated in a very simple expression language, the condition fragment of DMSL. More specifically, atomic conditions in DMSL have the form <Attribute> <Operator> <Value>, where:

- <Attribute> is constituted by a prefix specifying the parameter domain (i.e., User. for user-related parameters, and Context. for context-related parameters) followed by any of the design parameters attributes defined in the current design case, (e.g., User.vision).

- <Operator> represents an equality (==), inequality (!=) or comparison (>,-,<=) relation between the attribute and its assigned value. Only equality and inequality operators can be used with enumerated and Boolean attributes.

- <Value> corresponds to an instance of the values declared as appropriate for the specific attribute. With enumerated attributes, <Value> indicates one of the allowed enumerated values as declared in the Design Parameters Editor, e.g., User.Expertise == Expert. With Boolean attributes, <Value> is either True or False, e.g., User.Vision == True. Finally, with integer attributes, <Value> can be a constant integer within the bounds specified in the Design Parameters Editor for the specific attribute, e.g., User.Age >= 18, or, alternatively, an arithmetic expression, which may contain variables referring to the value of another attribute.
(either instantiated in the same stereotype or not), e.g., \( User.Age == 2003 - User.YearOfBirth \).

Atomic conditions can be negated by using the negation operator \( NOT \), e.g., \( NOT User.Expertise == Expert \). Finally, atomic conditions can be combined into complex conditions through the \( AND \) and \( OR \) operators, e.g., \( User.Vision == True AND User.Expertise == Expert, User.Age > 18 OR User.Expertise == Expert \). Ordinary parentheses may be used to establish operators’ precedence.

A condition is a partial description of users or contexts in the space defined by the current design parameters, and semantically refers to a sets of users or contexts for which it holds true (at run-time).

Figure 7 depicts the standard appearance of the Stereotypes Editor.

Figure 7. The Stereotypes Editor

User and context stereotypes can be defined by selecting the related option button. On the left part of the editor, the list of currently declared stereotypes is displayed. In the upper right, the condition currently associated to the selected stereotype is shown, and can be directly edited. An informal textual description of the currently selected stereotype can also be provided in the right lower part.

Alternative ways of introducing stereotypes are supported. A new stereotype can be directly created (first button on the Editor’s toolbar). In this case, the new attribute is added to the current list with a default name, and empty condition and description. Stereotypes names are unique in a design case.

Similarly to design parameters, stereotypes can also be imported into the current design case from a pre-existing one (second button on the Editor’s toolbar). The related dialogue is as depicted in Figure 8. The consistency policy for stereotypes importing is as follows:
o A stereotype can be imported in the current project only if the attributes included in its associated condition are declared in the current project, and the value assigned to each attribute falls into the range of legal attributes for that attribute in the current project. For example, a stereotype with a condition `User.Vision == True` can not be imported into a design case where either the attribute `Vision` is not defined, or the type of such an attribute is not Boolean.

o A ‘legal’ stereotype whose name and condition are not identical to the name and condition of any of the stereotypes included in the current project is directly imported

o If the name or condition of a ‘legal’ stereotype is identical to the name / condition of one of the stereotypes already defined, then the user is requested to confirm the overwriting of the existing stereotype with the imported one.

Figure 8. The stereotype import dialogue

Finally, two or more stereotypes can be merged into a new stereotype whose condition is composed by the conditions of the merged stereotypes through the *AND* or the *OR* operator (third button on the Editor’s toolbar). For example, the stereotypes *Expert User*, with the condition `User.Expertise == Expert`, and *Sighted User*, with the condition `User.Vision == True`, are *AND*-merged into the stereotype *Sighted Expert User* with the condition `User.Expertise == Expert AND User.Vision == True`. To perform merging, the user needs to tick the stereotypes to be merged in the Stereotypes Editor list and than select the desired merging operator (see Figure 9).
The conditions and descriptions resulting from merging are automatically filled in by the system (see Figure 10).

![Figure 9. Merging Stereotypes](image)

![Figure 10. The results of stereotypes merging](image)

Easy and syntactically correct editing of stereotypes conditions is supported through the provision of a Condition Editor which is activated (and deactivated) by the fourth button on the toolbar of the Stereotypes Editor. Figure 11 depicts the Stereotypes Editor appearance when the Condition Editor is activated. As shown, the Condition Editor is displayed-in-place on the right side of the Stereotypes Editor, and, during its activation, the free-text editing facilities for stereotypes are not available. Instead, the Condition Editor provides the possibility of entering conditions by selection of
elements and use of buttons, and has been designed in order to facilitate beginner users of MENTOR to acquire familiarity with the expression language for adaptation conditions.

Table 1 reports the details of use of the Condition Editor. The Condition Editor practically guides designers to composing syntactically correct conditions by contextually and progressively activating interactive elements on the interface according to the condition composition phases (atomic vs. complex) and the selected condition(s) on the main text area. Additionally, it completely automates text input, through the use of combo boxes and buttons, so as to avoid the use of undefined attributes or values, as well as typing errors. However, the Condition Editor as such can not ensure the semantic consistency of the composed conditions. Additionally, when the Condition Editor is not activated and the users encode conditions using free text input, all types of errors (lexical, syntactic and semantic) are likely to occur. Therefore, a checking module for adaptation conditions has been included in MENTOR, capable of verifying their lexical and syntactic correctness, as well as semantic consistency. In the Stereotypes Editor, the checking routines can be called, both in the free text and Condition Editor mode, by using the fifth button on the Editor’s toolbar. The condition appearing in the main text area of the Stereotypes Editor is then checked.

Table 2 presents some examples of system feedback upon checking of correct conditions and conditions containing different types of errors.
Table 1. Use of the MENTOR Condition Editor

Atomic conditions are composed using the combo boxes at the bottom of the Condition Editor, which are progressively activated and contextually filled from left to right.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Operator</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expertise</td>
<td>==</td>
<td>Expert</td>
</tr>
<tr>
<td>Vision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MouseUse</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Attribute combo box contains a list of the design parameters declared in the current design case.

The Operator combo box contains the list of available operators for the selected attribute.

The Value combo box contains the list of available values for the selected attribute.

When the selected attribute is of type integer, the value combo box is substituted by a text box.

When an atomic condition is ready, i.e. attribute, operator and value have been selected, the up-arrow button in condition editor middle area is activated, and the condition can be moved to main text area of the editor.
Table 1. Use of the MENTOR Condition Editor

Upon pressing the up-arrow button, the condition is moved to the main text area of the editor, and the combo boxes at the bottom return at the initial state to allow composing new atomic conditions.

When the user clicks at any point or starts selecting in the main text area, the entire condition in the corresponding text line(s) is automatically selected. Pressing the down arrow button blocks the main text area and resets the bottom combo boxes for reediting the selected condition. After the condition is modified, it can be moved again to the main text area to substitute the originally selected condition.

Atomic conditions need to be syntactically connected through **AND** or **OR** operators. This is performed using the AND and OR buttons in the middle area of the editor. These two buttons work as follows:
- They appear un-pressed if the selected condition does not contain the related operator at the end.
- They appear pressed if the selected condition contains the related operator at the end.
- Upon pressing an un-pressed button, the related operator is either added at the end of the selected condition or substitutes the other operator if present.
- Upon depressing a pressed button, the related operator is removed from the selected condition.
### Table 1. Use of the MENTOR Condition Editor

Conditions may be negated by using the NOT button in the middle area of the editor. The NOT button appears un-pressed when the selected condition is not negated, and pressed when it is negated. Pressing the NOT button when un-pressed will result in adding a NOT operator at the beginning of the selected condition. If the selected line is negated the NOT button when pressed will result in removing the NOT operator from the selected condition.

Parentheses can be used to establish operators' precedence in complex conditions, by using the ( ) button in the middle area of the editor. To set parentheses, two or more atomic conditions must be selected. When two or more lines are selected, the ( ) button appears activated and un-pressed if no parenthesis is included in the selected lines. Pressing the button in this state will result in adding a left parenthesis at the beginning of the first selected line (following a NOT operator if present) and a right parenthesis at the end of the last selected line (preceding and AND or OR operator if present). The ( ) button appears activated and pressed when the selected lines contain an equal number of left and right parentheses. Pressing the button in this state removes parentheses in the selected lines. Finally, the ( ) button appears pressed but inactivated when the selected lines do not contain an equal number of left and right parentheses.
### Table 1. Use of the MENTOR Condition Editor

Pressing the red X button in the middle area of editor deletes the selected line(s). If the line(s) contain(s) any operator, the operator is also deleted. If the selected line(s) contain(s) an equal number of left and right parentheses, the parentheses are also deleted. If the selected line(s) do not contain an equal number of left and right parentheses, the parentheses in the selected text are deleted and their corresponding left or right parentheses outside the selected text are searched and deleted as well.

### Table 2. Examples of condition checking feedback

<table>
<thead>
<tr>
<th>Correct condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Correct Condition Diagram]</td>
</tr>
</tbody>
</table>

![Condition Checking Feedback Diagram]
<table>
<thead>
<tr>
<th>Error: Condition contains an unknown item</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Error: Condition contains an unknown item" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Error: Condition contains a syntactic error</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image2" alt="Error: Condition contains a syntactic error" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Error: Condition contains an attribute with an inappropriate value</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image3" alt="Error: Condition contains an attribute with an inappropriate value" /></td>
</tr>
</tbody>
</table>
Table 2. Examples of condition checking feedback

| Error: Condition contains an integer attribute with a value out of the specified range |
|---|---|
| Error: Condition is not consistent because of inconsistent values for the same attribute |
| Error: Condition is not consistent due to different operators for the same attribute and value |
Table 2. Examples of condition checking feedback

<table>
<thead>
<tr>
<th>Error</th>
<th>Check Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition is not consistent due to inconsistent operator and value combination for the same attribute</td>
<td><img src="image1.png" alt="Image" /></td>
</tr>
<tr>
<td>Condition calculates to an integer value incompatible with subsequent comparison operator and value</td>
<td><img src="image2.png" alt="Image" /></td>
</tr>
</tbody>
</table>
Table 2. Examples of condition checking feedback

<table>
<thead>
<tr>
<th>Warning: Condition contains integer variable (warning)</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Screenshot of condition checking feedback window]</td>
</tr>
</tbody>
</table>

Warning: Condition contains integer variable (warning)
The basic representation adopted in Unified User Interface design is called polymorphic task hierarchy ([Savidis et al, 2001]), and combines: (a) hierarchical task analysis; (b) design polymorphism, i.e., the possibility of assigning alternative decompositions to the same (sub)task if required based on (combinations of) design parameters; and (c) user-task oriented operators.

Figure 12 depicts an example of polymorphic task hierarchy, illustrating how two alternative dialogue styles for a “delete file” task may be designed. Alternative decomposition “styles” are depicted in the upper part of the figure, and an exemplary polymorphic decomposition, which includes physical design annotation, appears in the lower part.

The hierarchical decomposition adopts the original properties of hierarchical task analysis, enriched with the capability to differentiate and represent design alternatives for the same task, mapping to varying design parameters through polymorphism. Task operators are based on the powerful Communicating Sequential Processes (CSP) language for describing the behaviour of reactive systems ([Hoare, 1978]), and enable the expression of dialogue control flow formulae for task accomplishment. Basic task operators include sequencing, parallelism, logical exclusion and repetition. Additional (or alternative) operators may be used as necessary.

In a polymorphic task hierarchy, the root represents design abstractions, while leaf nodes represent concrete interaction components. Polymorphic decomposition leads from abstract design pattern to a concrete artefact. Three categories of design artefacts may be subject to polymorphism on the basis of user- and usage-context- parameter values, and namely:

- User tasks, relating to what the user has to do; user tasks are the centre of the polymorphic task decomposition process.

- System tasks, representing what the system has to do, or how it responds to particular user actions (e.g., feedback); in the polymorphic task decomposition process, they are treated in the same manner as user tasks.
Physical designs, which concern the interface components on which user actions are to be performed; physical interface structure may also be subject to polymorphism.

User tasks, and in certain cases, system tasks, are not necessarily related to physical interaction, but may represent abstraction on either user- or system- actions. System tasks and user tasks may be freely combined within task “formulas”, defining how sequences of user-initiated actions and system-driven actions interrelate. The physical design, providing the interaction context, is associated with a particular user task, and provides the physical dialogue pattern associated to a task-structure definition. Hence, it plays the role of annotating the task hierarchy with physical design information.
Each alternative polymorphic decomposition is called a decomposition style, or simply a style, and is attributed a unique name. Alternative task sub-hierarchies are attached to their respective styles. Polymorphism constitutes a technique for potentially increasing the number of alternative interface instances represented by a typical hierarchical task model. However, the Unified User Interface design method does not require the designer to follow the polymorphic task decomposition all the way down the user-task hierarchy, until primitive actions are met. A non-polymorphic task can be specialized at any level, following any design method chosen by the interface designer.

In MENTOR, The Polymorphic Task Hierarchy Editor allows designers to perform polymorphic task decomposition and encode the results in a hierarchy. Figure 13 depicts an overview of the Editor.

![Figure 13. The Polymorphic Task Hierarchy Editor](image)

The Editor is based on standard hierarchy representation facilities and does not use any specific visualisation techniques. To facilitate navigation, hierarchy nodes can be opened or closed, and artefacts can be selected using the combo box at the top of the editor, which provides the entire path of each artefact.

Different categories of design artefacts are distinguished through icons. The Editor’s main function is to guide the decomposition process by contextualising the available decomposition actions according to the category of design artefact selected in the
hierarchy. Decomposition actions can be performed either through the available toolbar, which can be docked or appear as a floating toolbox, or through pop-up menus.

The contextualisation of decomposition actions follows the specification of the polymorphic task decomposition steps. Table 3 displays, for each category of design artefact, the related icon, the types of admissible decomposition actions (i.e., the possible categories of children in the hierarchy), as well as the “decomposition buttons” available on the Editor’s toolbar (or toolbox) when an artefact belonging to the category is selected in the hierarchy.

Upon pressing one of the buttons (or the corresponding item in the pop-up menu), a new artefact of the chosen category is inserted as a children of the currently selected artefact (see Figure 14), with a default name that can be modified using the Properties Editor (see next section). Artefacts’ names are unique in a design case.

![Figure 14. The process of decomposing artefacts](image)

Figure 14. The process of decomposing artefacts
Table 3. Categories of design artefacts and related icons and admissible decompositions

<table>
<thead>
<tr>
<th>Icon</th>
<th>Design Artefact Category</th>
<th>Admissible decompositions</th>
<th>Toolbar appearance</th>
</tr>
</thead>
</table>
| ![Unimorphic User Task](image1) | Unimorphic User Task | • Unimorphic user task  
• Unimorphic system task  
• Polymorphic user task  
• Polymorphic system task  
• Unimorphic physical design or  
Polymorphic physical design (only one) | ![Design Toolbox](image2) |
| ![Unimorphic System Task](image3) | Unimorphic System Task | • Unimorphic user task  
• Unimorphic system task  
• Polymorphic user task  
• Polymorphic system task  
• Unimorphic physical design or  
Polymorphic physical design (only one) | ![Design Toolbox](image2) |
| ![Unimorphic physical design](image4) | Unimorphic physical design | • Unimorphic physical design  
• Polymorphic physical design | ![Design Toolbox](image2) |
| ![Polymorphic user task](image5) | Polymorphic user task | • Unimorphic user style  
• Polymorphic user style | ![Design Toolbox](image2) |
| ![Polymorphic system task](image6) | Polymorphic system task | • Unimorphic system style  
• Polymorphic system style | ![Design Toolbox](image2) |
| ![Polymorphic physical design](image7) | Polymorphic physical design | • Unimorphic physical design style  
• Polymorphic physical design style | ![Design Toolbox](image2) |
| ![Unimorphic User Style](image8) | Unimorphic User Style | • Unimorphic user task  
• Unimorphic system task  
• Polymorphic user task  
• Polymorphic system task  
• Unimorphic physical design or  
Polymorphic physical design (only one) | ![Design Toolbox](image2) |
| ![Unimorphic System Style](image9) | Unimorphic System Style | • Unimorphic user task  
• Unimorphic system task  
• Polymorphic user task  
• Polymorphic system task  
• Unimorphic physical design or  
Polymorphic physical design (only one) | ![Design Toolbox](image2) |
| ![Unimorphic physical design Style](image10) | Unimorphic physical design Style | • Unimorphic physical design  
• Polymorphic physical design | ![Design Toolbox](image2) |
| ![Polymorphic User Style](image11) | Polymorphic User Style | • Unimorphic user style  
• Polymorphic user style | ![Design Toolbox](image2) |
| ![Polymorphic System Style](image12) | Polymorphic System Style | • Unimorphic system style  
• Polymorphic system style | ![Design Toolbox](image2) |
| ![Polymorphic physical design style](image13) | Polymorphic physical design style | • Unimorphic physical design style  
• Polymorphic physical design style | ![Design Toolbox](image2) |
In the Polymorphic Task Hierarchy Editor, artefacts can also be deleted, copied, cut, and pasted. When an artefact is deleted, its children are also deleted, in order to preserve the correctness of the resulting hierarchy (since the children of the deleted task may not be appropriate children of the deleted task father’s node). Artefacts can only be pasted as children of nodes that admit the category of the copied/cut artefact as decomposition. Pasting results in producing a copy of the copied/cut artefact at the selected point in the hierarchy, including all its properties, with the exception of properties that are related to the position of the artefact in the hierarchy (i.e., inherited properties, see next section), which are updated according to the point in which the artefact is pasted. For example, it is not possible to paste a style as a child of a unimorphic artefact, etc.
The Properties Editor

Part of the polymorphic task decomposition process consists in assigning specific properties to the artefacts in the polymorphic hierarchy. Different categories of artefacts involve different properties, some of which are particularly important for the purposes of the overall adaptation design of the resulting interface. All artefacts need to be assigned a unique name, and can have an (optional) textual description. Unimorphic tasks and styles can also encode temporal relations between subtasks, in a textual form. Physical designs can be associated to images representing the designs in any format (sketches, prototypes, etc). Styles need to be assigned adaptation conditions. Finally, polymorphic artefacts need to encode design relations between children styles. Table 4 summarises the properties available for each category of design artefact in MENTOR. In the design of the overall MENTOR environment, it was considered particularly important for artefacts properties to be editable ‘in context’, while also working on the polymorphic task hierarchy. Therefore, the interrelationships between the Polymorphic Task Hierarchy Editor and the Properties Editor were carefully considered. The Properties Editor appears as a floating window in the overall tool environment (see Figure 4), therefore ensuring that the Polymorphic Task Hierarchy Editor is not hidden while the user is editing properties, and always displays the data related to the currently selected artefact in the Polymorphic Task Hierarchy Editor. Properties are shown in a simple table-like form with can be directly edited in place. For those properties which need to be set using selection rather than text entry, special in-place mini-editors have been designed, to be displayed within the editing space of the specific property in the Properties Editor.
<table>
<thead>
<tr>
<th>Design Artefact Category</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unimorphic User Task</td>
<td>• Name</td>
</tr>
<tr>
<td></td>
<td>• Description</td>
</tr>
<tr>
<td></td>
<td>• Subtask Relations</td>
</tr>
<tr>
<td>Unimorphic System Task</td>
<td>• Name</td>
</tr>
<tr>
<td></td>
<td>• Description</td>
</tr>
<tr>
<td></td>
<td>• Subtask Relations</td>
</tr>
<tr>
<td>Unimorphic physical design</td>
<td>• Name</td>
</tr>
<tr>
<td></td>
<td>• Description</td>
</tr>
<tr>
<td></td>
<td>• Image(s)</td>
</tr>
<tr>
<td>Polymorphic user task</td>
<td>• Name</td>
</tr>
<tr>
<td></td>
<td>• Description</td>
</tr>
<tr>
<td></td>
<td>• Design Relations</td>
</tr>
<tr>
<td>Polymorphic system task</td>
<td>• Name</td>
</tr>
<tr>
<td></td>
<td>• Description</td>
</tr>
<tr>
<td></td>
<td>• Design Relations</td>
</tr>
<tr>
<td>Polymorphic physical design</td>
<td>• Name</td>
</tr>
<tr>
<td></td>
<td>• Description</td>
</tr>
<tr>
<td></td>
<td>• Design Relations</td>
</tr>
<tr>
<td>Unimorphic User Style</td>
<td>• Name</td>
</tr>
<tr>
<td></td>
<td>• Description</td>
</tr>
<tr>
<td></td>
<td>• Subtask Relations</td>
</tr>
<tr>
<td></td>
<td>• Condition</td>
</tr>
<tr>
<td>Unimorphic System Style</td>
<td>• Name</td>
</tr>
<tr>
<td></td>
<td>• Description</td>
</tr>
<tr>
<td></td>
<td>• Subtask Relations</td>
</tr>
<tr>
<td></td>
<td>• Condition</td>
</tr>
<tr>
<td>Unimorphic physical design Style</td>
<td>• Name</td>
</tr>
<tr>
<td></td>
<td>• Description</td>
</tr>
<tr>
<td></td>
<td>• Image(s)</td>
</tr>
<tr>
<td></td>
<td>• Condition</td>
</tr>
<tr>
<td>Polymorphic User Style</td>
<td>• Name</td>
</tr>
<tr>
<td></td>
<td>• Description</td>
</tr>
<tr>
<td></td>
<td>• Design Relations</td>
</tr>
<tr>
<td></td>
<td>• Condition</td>
</tr>
<tr>
<td>Polymorphic System Style</td>
<td>• Name</td>
</tr>
<tr>
<td></td>
<td>• Description</td>
</tr>
<tr>
<td></td>
<td>• Design Relations</td>
</tr>
<tr>
<td></td>
<td>• Condition</td>
</tr>
<tr>
<td>Polymorphic physical design style</td>
<td>• Name</td>
</tr>
<tr>
<td></td>
<td>• Description</td>
</tr>
<tr>
<td></td>
<td>• Design Relations</td>
</tr>
<tr>
<td></td>
<td>• Condition</td>
</tr>
</tbody>
</table>
Figure 15 depicts the appearance of the Properties Editor when a unimorphic task artefact is selected in the Polymorphic Task Hierarchy Editor. The three properties are specified textually in place. Changing the artefact’s name in the Properties Editor also updates the name in all places where it appears in the Polymorphic Task Hierarchy Editor. No special provision is made for editing or checking subtasks temporal relations, since the types of operators and temporal relations notation can vary from case to case.

![Polymorphic Hierarchy Editor](image)

Figure 15. The Properties Editor appearance for Unimorphic tasks

Figure 16 depicts the in-place mini-editor for attaching images to unimorphic physical design artefacts, which is activated by editing the corresponding cell in the properties table. Images will need to be separately created (e.g., drawings produced in drawing environments, scanned paper designs, snapshots of prototypes, etc). Any image format can be used. Through a standard dialogue the user can chose an image file to be attached to the currently selected physical design artefact in the task hierarchy. It is also possible to attach more than one image. Images can also be removed from the list of current attachments and displayed underneath the list area. Full size view of attached images is also possible in a separate viewer (see Figure 17).
Style conditions entered as properties of styles are obviously expressed in the same expression language used for the definition of stereotypes. The mini-editor for style conditions in the Property Editor allows three different ways of entering the related data:

- By free text typing in the mini-editor main text area.
- By selecting (multiple) stereotypes. When the mini-editor is the stereotype selection mode, the list of available (i.e., previously defined) stereotypes is displayed at the bottom of the main text area (see left-hand side of Figure 18). The user can then tick the stereotypes to be used. Upon clicking the up-arrow button on the mini-editor toolbar, the condition(s) of the selected stereotype(s) are copied into the main text area of the mini-editor, and can if necessary be further edited and modified.
- By using the MENTOR Condition Editor (see right-hand side of Figure 18), which is displayed in the mini-editor. The characteristics and use of the Condition Editor have been described in Table 1.
The condition mini-editor also supports two forms of verification on styles conditions. The first refers to the consistency of each condition separately, and is equivalent to the verification facilities described for stereotypes conditions (see Table 2). The second type of verification, on the other hand, concerns hierarchical relations among styles in the polymorphic task hierarchy. A style in the task hierarchy in fact (implicitly) inherits the conditions on its ancestor styles. Each polymorphic decomposition applies within the design context (i.e., specified conditions) of a higher-level polymorphic decomposition in the same hierarchy branch, if present. In Figure 19, an example is depicted illustrating a style that inherits the condition of an ancestor style in the same branch of the hierarchy. The inherited condition is automatically filled in by the Property Editor in the condition cell of the related style.

Figure 18. Editing style conditions in the Property Editor
Figure 19. Inheritance of style conditions in polymorphic task hierarchies

For the resulting adaptation logic to be correct, the inherited and the new condition of a style must be compatible (i.e., not inconsistent). Therefore, the hierarchical verification mechanism of MENTOR checks that such a compatibility constraint is not violated. Feedback on hierarchical consistency of styles is provided in a manner analogous to other types of checking (i.e., through a small dialogue window). Clearly, the checking of each of the conditions involved in isolation needs to be automatically performed before checking their hierarchical consistency, since in case the separate conditions are not syntactically correct or inconsistent, the hierarchical checking can not proceed.

Another important aspect of Unified user Interface design towards determining the run-time adaptation behaviour of a user interface is the assignment of adaptation relations between different styles of a polymorphic artefact. Four fundamental relationships among alternative styles (concerning the same polymorphic artefact) have been identified in Unified User Interface design [Savidis and Stephanidis, 2004a], reflecting the way in which artefacts may be employed during interaction for an individual user in a particular context. These are:

- **Incompatibility**: Only one style is available to the user during interaction.
- **Compatibility**: Any, some, or all styles may be available to the user during interaction
- **Substitution**: A style S1 substitutes a style S2. When S1 is available during interaction, and at some point S2 should be also made available, S1 must be closed down
- **Augmentation**: A style S1 augments a style S2. If S2 is available during interaction, S1 may also become available.
In MENTOR, adaptation relations among styles are formulated as properties of polymorphic artefacts. Figure 20 depicts a view of such properties editing task.

**Figure 20. Editing Design relations among styles of polymorphic artefacts in the Properties Editor**

Binary design relations are constructed by sequentially selecting, using the combo boxes underneath the main text area in the mini-editor:

- the first relation argument (i.e., one of the styles defined as children of the currently selected polymorphic artefact)
- the type of the relation (choosing among the four available types in the Unified User Interface design method, i.e., incompatibility, compatibility, augmentation and substitution)
- the second relation argument (another style among the ones defined as children of the currently selected polymorphic artefact).

When a relation is completed, it can be copied into the main text area of the mini-editor by using the up-arrow button on the toolbar. As many relations as necessary can be composed.

MENTOR also supports verifying that the conditions on the two styles selected as arguments of a design relation are compatible with the type of the relation. This facility is introduced in order to ensure that the resulting run-time adaptation logic will be semantically sound on the basis of the declared design parameters, and will not contain ambiguities resulting in problems when applying adaptations (e.g., the Decision-Making components of the design interface encounters a situation where two styles are declared as incompatible but the conditions on these styles can hold on
the same user / context at the same time). For design relations to be checked, their individual arguments (conditions) must have been first checked for consistency. All mini-editors are closed when the user clicks elsewhere in the application’s interface and the table view of the Properties Editor is re-displayed.
The Design Documentation Viewer

One of the main objectives of MENTOR is to automate the creation of design case documentation based on the designs encoded in each project. Design documentation is considered particularly important in the provision of tool-based support for Unified Design, as the method aims at structuring the outcomes of adaptation design in a form suitable for direct implementation. In the design of MENTOR it was decided therefore to support the collection of all available information in a design case into a textual design documentation report to be subsequently used for several purposes, such as design reviewing and evaluation, interface documentation, and, most importantly, implementation of the adaptation logic. The design report can be produced and displayed, upon the user’s request, at any time during design. It has the following content:

1. Title: the design project’s name
2. Design parameters: this section lists all defined designed parameters and related values
3. Stereotypes: the list of defined stereotypes
4. Polymorphic Task Hierarchy: a textual representation of the current state of the polymorphic task hierarchy, with each node numbered for easy reference to its properties in the subsequent section.
5. Design artefact properties: A numbered list of all artefacts in the hierarchy with their attached properties. Physical design images are also inserted in the text.
6. Design logic: a list of DMSL rules automatically produced by the tool on the basis of the current styles conditions and design relations in polymorphic artefacts of the design case. DMSL rules as produced by MENTOR can be directly embedded in the decision-making component of the designed Unified Interface.
Figure 21. The MENTOR design report viewer

Figure 21 depicts the MENTOR Design Documentation Viewer, displaying a design report for the current case. In the lower part of the figure it is possible to see the derived DMSL rules. In the Documentation Viewer, the text can not be edited. To perform modifications, the user needs to change the data in the related editor(s), and regenerate the design report. Design reports can be saved (independently from design cases) in RTF and MS Word formats. They can also be directly printed from MENTOR.
The MENTOR Help

MENTOR is equipped with standard help facilities, aimed at both introducing users-designers to the use of the tool and facilitating the acquisition of familiarity with the Unified User Interface design method as embodied in the tool. The MENTOR help topics index is shown in Figure 22.

![Design Tool Help](image)

**Figure 22. The MENTOR help index**
Automated verification of adaptation design in MENTOR

The effectiveness and appropriateness of the designed interface adaptation behaviour will (partly) depend on the characteristics of the adaptation logic. On the other hand, adaptation design in Unified User Interfaces can be considered as less familiar to designers, who need to be assisted in acquiring experience with adaptation logic and practicing it systematically. Therefore, appropriate means are required so as to facilitate designers in “debugging” their adaptation design and make sure that the delivered adaptation logic can reliably be embedded directly into the unified user interface development cycle. MENTOR provides automated facilities supporting adaptation design, based on the rationale that: (i) at least some of the possible problems in the adaptation logic can be proactively prevented or identified at design time, and (ii) that the correct transmission of design specification to the implementation phase can be ensured by the automatic generation of such specifications. Related facilities in MENTOR include the automatic verification of the designed adaptation logic and the automatic generation of the design report, and are meant to both support designers in efficiently conducting non creative design tasks, and ensure a smooth design – implementation transition phase. The alternative to such an approach would be the detection of these errors during the implementation, testing or the empirical evaluation phases of the designed interfaces, implying a considerably more extensive effort and feedback cycle. For example, without design time checking, an unnoticed syntactic error in a style condition would be propagated to the implementation phase of the decision-making module, where it would be detected in compiling DMSL rules. However, since the DMSL language does not necessarily enforce satisfiability checking of its rules, it could be the case that an unsatisfiable (contradictory) is detected only at the testing or the evaluation phases of a prototype unified interface, where identifying the source of the problem (e.g., a style is never activated in the interface because its condition is unsatisfiable) can potentially be much more complex. Although no data are currently available on the frequency of occurrence of design “errors” in the adaptation logic, nor a complete picture exists of
the possible impact of such errors on the adaptation behaviour of the final interface, it is believed that avoiding the occurrence of some of the problems in the first place is a wise strategy. Therefore, the possibility is offered to designers to verify their adaptation design step-by-step at any point during the design process, and to automatically forward design specifications to the implementation phase. The above constitute a form of formal verification not addressed in previous design support research efforts, i.e., the verification of the correctness of the adaptations embedded into the designed unified interface.

Table 5 summarises the verification facilities in MENTOR, based on the hierarchical properties of Polymorphic Task decomposition and on the logical properties of the language used for adaptations formulation (i.e., the DMSL language and in particular its condition fragment), as well as on the consideration of possible mismatches in the design-implementation transition.

Table 5. Verification of adaptation behaviour correctness

<table>
<thead>
<tr>
<th>Failure</th>
<th>Requirement</th>
<th>Adopted solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lexically and/or syntactically incorrect rules</td>
<td>Ensuring rules lexical and syntactic correctness</td>
<td>Lexical and syntactic checker on DMSL condition expressions (using the GOLD Parser [GOLD Parser Documentation])</td>
</tr>
<tr>
<td>Adaptation rules are not applicable</td>
<td>Ensuring logical satisfiability of adaptation rules</td>
<td>Formal verification of DMSL condition expressions satisfiability</td>
</tr>
<tr>
<td>Adaptation rules are called in a wrong interaction context</td>
<td>Ensuring hierarchical consistency and incremental specification of style conditions</td>
<td>Formal verification of hierarchical consistency and subsumption of DMSL condition expressions in a polymorphic task hierarchy</td>
</tr>
<tr>
<td>Adaptation rules are ambiguous</td>
<td>Ensuring contextually unambiguous adaptation conditions for designed styles</td>
<td>Formal verification of properties of style conditions according to the type of selected design relation.</td>
</tr>
<tr>
<td>Mismatches between profiles in the user and context server and the adaptation logic in the decision making module</td>
<td>Correct forwarding of design parameters and adaptation logic from the design to the implementation phase</td>
<td>Automatic generation of design report including DMSL rules</td>
</tr>
</tbody>
</table>

A formal account of the satisfiability verification mechanisms adopted in MENTOR is reported in [Savidis et al., 2004].
Conclusions

This Report has described the interactive environment of MENTOR, a tool for the process-based support of the Unified User Interface design method, focusing in particular on the available functionality and facilities.

MENTOR is intended to constitute a first step towards the provision of process-oriented tool-based support for Unified User Interface design, as it offers facilities to perform all the steps involved in the Unified User Interface design method, and to encode the design outcomes through specialised editing facilities. The functionality offered by the tool reflects and supports the basic aspects and phases of the method’s conduct. The tool is intended to support designers’ creativity by avoiding the automation of the creative steps of Unified User Interface design, and namely the case-dependent definition of design parameters and the polymorphic task decomposition from abstract tasks to physical designs, while automating less creative aspects, such as the automated generation of design specifications including all outcomes of a design case. A correct and appropriate transition from adaptation design to adaptation implementation is also targeted through verification facilities for checking the satisfiability, hierarchical discipline and method’s compliance of the design logic embedded in polymorphic task hierarchies, as well as its transformation into run-time adaptation rules. The formal and informal parts are seamlessly integrated into the interaction environment, and a very simple expression language (a fragment of DMSL) for specifying adaptation conditions.

The tool’s graphical environment aims at an interactive simplification of the intrinsically increased complexity of the Unified User Interface design method with respect to current design practices. Alternative options for facilitating the method’s practice by less expert and more expert designers are also offered (e.g., the Condition Editor vs. free-text specification of style conditions).

Extensibility of design cases is ensured by simply reflecting the intrinsic characteristics of Unified User Interface design method, which allows incrementality in design parameter specification and in style specification to cater for newly introduced design parameters. Reusability of design cases is also partially addressed
through import facilities for design parameters and stereotypes and string base search facilities throughout design cases.

Since the tool is not related to any particular interface development toolkit, and does not presuppose any modelling of the physical level of interaction, it poses no constraints on the interfaces and on the types of adaptation which can be designed by using it. However, the tool allows linking images to physical design artefacts, so that physical designs are directly embedded into the polymorphic task hierarchy and reported as design specifications.

MENTOR is intended to constitute a useful contribution towards addressing the challenges related to design support under a Universal Access perspective, and in particular towards:

- widening and enhancing the practice of design for Universal Access through the provision of a tool that supports the process-oriented conduct of an appropriate design method, and facilitates its adoption;
- improving cost-effectiveness and quality of designing for diversity, by computationally supporting the extension and reuse of design cases;
- offering the concrete opportunity of conducting large scale design cases, by providing a potential repository of design knowledge for further investigation and refinement of the practice of designing for diversity.
References


