International Journal of Software Engineering
and Knowledge Engineering
Vol. 15, No. 6 (2005) 1063–1094

World Scientific
www.worldscientific.com

A DECISION-MAKING SPECIFICATION LANGUAGE FOR VERIFIABLE USER-INTERFACE ADAPTATION LOGIC

ANTHONY SAVIDIS*, MARGHERITA ANTONA*
and CONSTANTINE STEPHANIDIS*,†,‡

*Foundation for Research and Technology — Hellas (FORTH),
Institute of Computer Science, GR-70013, Heraklion, Crete, Greece
†University of Crete, Department of Computer Science, Greece
‡cs@ics.forth.gr

Submitted 25 June 2004
Accepted 6 May 2005

In automatic user interface adaptation, developers pursue the delivery of best-fit user interfaces according to the runtime-supplied profiles of individual end users and usage contexts. Software engineering of automatic interface adaptability entails: (a) storage and processing of user and usage-context profiles; (b) design and implementation of alternative interface components, to optimally support the various user activities and interface operations for different users and usage contexts; and (c) runtime decision-making, to choose on the fly the most appropriate alternative interface components, given the particular user and context profile. In automatic interface adaptation, the decision making process plays a key role in optimal on-the-fly interface assembly, engaging consolidated design wisdom in a computable form. A verifiable language has been designed and implemented which is particularly suited for the specification of adaptation-oriented decision-making logic, while also being easily deployable and usable by interface designers. This paper presents the language, its contextual role in adapted interface delivery and the automatic verification method. The employment of the language in an adaptation-design support tool is discussed, the latter automatically generating language rules by relying upon adaptation rule patterns. Finally, the deployment methodology of the language in supporting dynamic interface assembly is discussed, further generalizing towards dynamic software assembly, by introducing architectural contexts and polymorphic architectural containment.

Keywords: Automatic user interface adaptation; decision-making specification and verification; dynamic software assembly.

1. Introduction

1.1. *Dynamic interface adaptability*

The notion of automatic user interface adaptation reflects the capability of interactive software to adapt during runtime to the individual end-user, as well as to the particular context of use, by delivering a most appropriate interaction experience. The storage location, origin and format of user-oriented information may

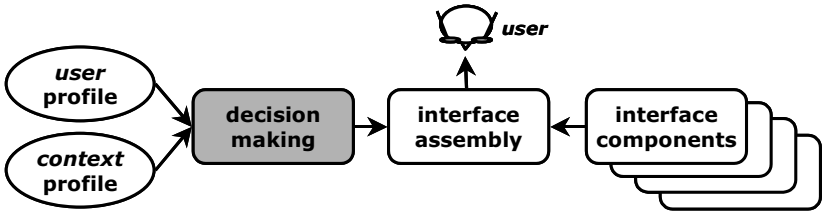

Fig. 1. Outline of the software architecture for interface adaptability, reflecting the critical role of adaptation-oriented decision making.

vary. For example, information may be stored in profiles indexed by unique user identifiers, may be extracted from user-owned cards, may be entered by the user in an initial interaction session, or may be inferred by the system through continuous interaction monitoring and analysis. Additionally, usage-context information, e.g., user location, environment noise, network bandwidth, etc, is normally provided by special-purpose equipment, like sensors, or system-level software. In order to support optimal interface delivery for individual user and usage-context attributes, it is required that for any given user task or group of user activities, the implementations of the alternative best-fit interface components are appropriately encapsulated.

Upon startup and during runtime, the software interface relies on the particular user and context profiles to assemble the eventual interface on the fly, collecting and gluing together the constituent interface components required for the particular end-user and usage-context. This type of best-fit automatic interface delivery, called *interface adaptability*, has been originally introduced in the context of Unified User Interface Development [1]. In this context, runtime adaptation-oriented decision-making is engaged, so as to select the most appropriate interface components for the particular user and context profiles, for each distinct part of the user interface. This logical distinction of the runtime processing steps to accommodate interface adaptability is effectively outlined in the runtime architecture of Fig. 1, illustrating the key principles of the Unified User Interface architecture [1] for automatically adapted interactions.

As depicted in Fig. 1, the role of the decision-making sub-system is to effectively drive the interface assembly process by deciding which interface components need to be selectively activated. The interface assembly process has inherent software engineering implications on the software organization model of interface components. More specifically, as for any *task context* (i.e., part of the interface to support a user activity or task) alternative implemented incarnations may need to coexist, conditionally activated during runtime due to decision-making, the need to accommodate interface context *polymorphism* arises. In other words, there is a need to organize interface components around their particular task contexts, enabling task contexts to be supported through multiple, i.e. polymorphic, deliveries. This contrasts with traditional non-adapted interfaces in which all task contexts have singular implementations. The above key software requirements for user interface

design and implementation have been addressed as follows:

- The *Unified User Interface Design Method* [2] reflects the hierarchical discipline of user interface construction, emphasizing the hierarchical organization of task contexts, which may have an arbitrary number of designed alternatives called *design styles*, shortly *styles*. In this framework, the concept of polymorphic task-context hierarchies, shortly polymorphic tasks, has been introduced. Each alternative style is explicitly annotated with its corresponding user and context attributes (i.e., its adaptation design rationale).
- *Dynamic polymorphic containment hierarchies* [3] provide a software engineering method for implementing interface components exhibiting typical hierarchical containment, while enabling the dynamic establishment of containment links among contained and container components. Additionally, all interface components reflect the organizational model of polymorphic task hierarchies, uniquely indexing components according to their particular design-time task-context and style identifiers.

The design and implementation of alternative interface components around hierarchically organized task contexts has been employed in the AVANTI Project (see acknowledgements) for the development of the AVANTI web browser [4], supporting interface adaptability. In Fig. 2, an excerpt from the polymorphic interface component organization of the AVANTI browser is shown, to accommodate implementation of interface adaptability.

1.2. *Key contributions*

As already mentioned, the unified user interface design and development methods have been reported in the literature in the past. Additionally, they have been taught in conference tutorials, see [4, 5], and applied in the course of projects partially funded by the European Commission — (TP1001 — ACCESS[a]; IST-1999-20656 — PALIO[b]; ACTS AC042 — AVANTI[c]; IST-1999-14101 — IS4ALL[d]; IST-2000-25286 — 2WEAR[e]), or by national funding agencies (EPET-II: NAUTILUS[f]). Following the work mentioned above, this paper specifically focuses on effective support for decision-making implementation. In particular, it reports a method which has been purposefully elaborated to be easier for designers to directly assimilate and deploy, in comparison to programming-based approaches using logic-based or imperative-oriented programming languages. Moreover, such a method supports the automatic verification of the adaptation-design rationale,

[a]Development Platform for Unified Access To Enabling Environments, 1995–1997.
[b]Personalized Access to Local Information and Services for Tourists, 2000–2003.
[c]Adaptive and Adaptable Interactions for Multimedia Telecommunications Applications, 1997–2000.
[d]Information Society for All, 2001–2003.
[e]A Runtime for Adaptive and Extensible Wireless Wearables, 2001–2003.
[f]Unified Web Browser for People with Disabilities, 1999–2001.

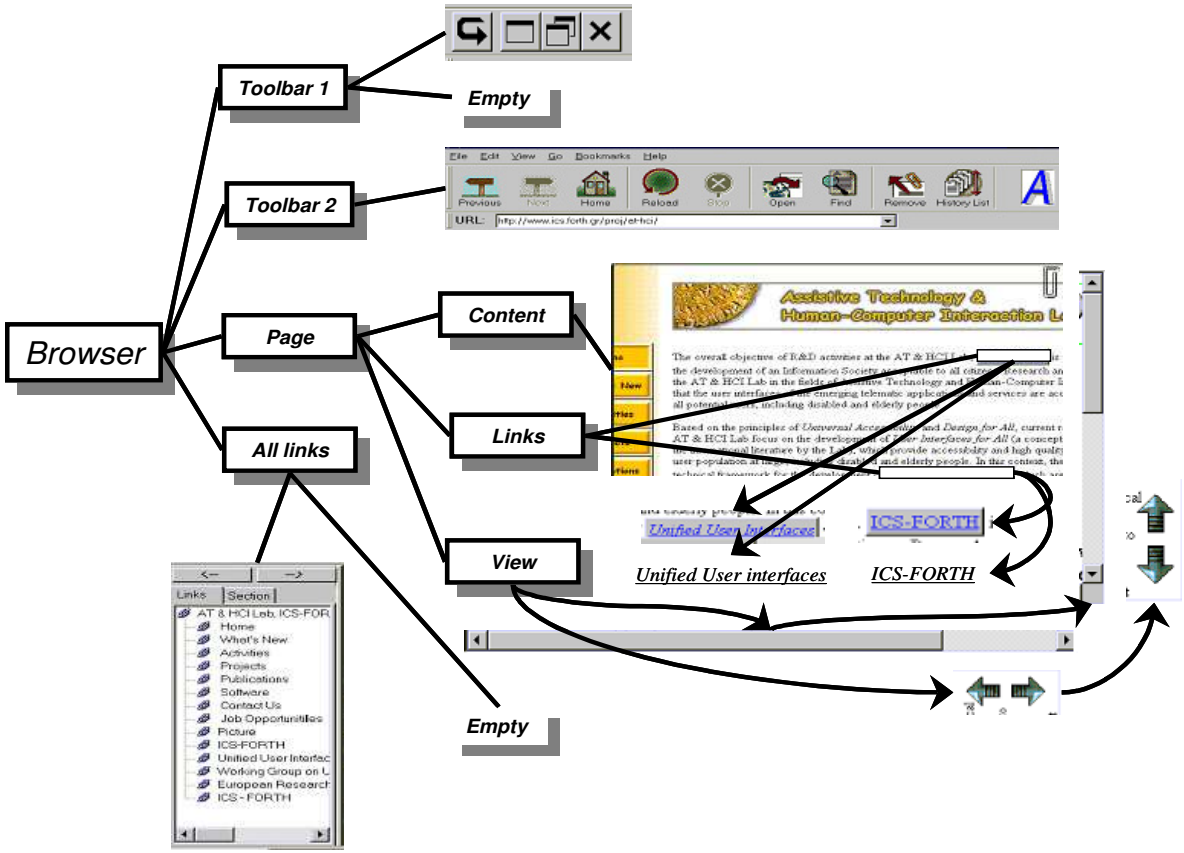

Fig. 2. Parametric polymorphic containment with variant constituent components in the AVANTI browser. The label "Empty" indicates components whose presence may have to be omitted upon dynamic interface delivery for certain user categories.

while also providing computer-assisted design support, through a tool enabling designers to manage the considerable overhead inherent in designing adaptable interfaces with polymorphic task hierarchies. In this context, the Decision Making Specification Language (DMSL) has been designed and implemented, including a method for automatic adaptation-design verification. Additionally, the software engineering deployment approach of the DMSL language is presented, discussing a well-tested architecture to cope with dynamic interface assembly, further generalizing and elaborating towards accomplishing dynamic software adaptability, i.e., software adaptively assembled according to deployment requirements. In this framework, the key contributions are:

- *Decision Making Specification Language* (DMSL)
  - — Supports localized decision blocks for each task context;
  - — Built-in user and context decision-parameters with runtime binding of values;
  - — Can trigger other decision blocks, supporting modular chain evaluations;
  - — References interface component through unique design identifiers;
  - — Supports activation and cancellation commands for interface components;
  - — Provides a method for automatic adaptation-design verification;
  - — Is accompanied by rule-patterns for interface component relationships;
  - — Has been proven in real practice to be understandable and usable by designers [6, 7];
  - — Is supported by an interactive tool for computer-assisted adaptation design.
- *Dynamic software adaptability*, reflecting an architectural generalization of the largely tested and applied interface adaptability software engineering approach as follows:
  - — From interface components to software components;
  - — From physical interface containment to architectural containment;
  - — From user and context parameters to software deployment parameters;
  - — From task contexts to architectural contexts;
  - — From interface-based indexing to architectural-role indexing;
  - — From dynamic interface assembly to dynamic software assembly.

2. Related Work

2.1. *Interface adaptation*

Most of the 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, and has been commonly referred to as *adaptation during use*. In adaptive interaction, the collection and processing of interaction monitoring information is performed and analysed, 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 update actions. A well-known adaptive system employing such techniques is MONITOR [8]. Similarly, for the purpose of dynamic detection of user attributes, a monitoring component is employed in the AIDA system [9], while the BGP-MS system [10] implemented a sophisticated user-modelling server.

In such adaptive systems, decision making is characterised by two key properties: (a) it is encapsulated in system implementation, not made editable by user interface designers; and (b) its primary target is the inference of dynamic user characteristics, such as preference to particular information elements or confusion in performing a task, instead of interface component selection as targeted in user interface adaptability. The development of decision kernels for dynamic interface component selection and delivery, upon interaction start-up, has been originally introduced in the definition of unified user interface development, and later applied in the context of large-scale adaptable application developments, such as the AVANTI web browser [4] and the PALIO adaptable multimedia information system [6]. In this context, the DMSL language has been designed to optimise the development process by offering an instrument, directly usable and deployable by designers, particularly suited for adaptation-oriented decision-making specification.

2.2. *Software adaptability*

In the domain of user interface development, re-usable software objects had been employed from the early days of graphical user interfaces (GUIs). The broad deployment of binary-reusable (i.e., compiled) components is carried in user interface software engineering from the early appearance of software libraries of interaction object classes, such as Windows Object Library, Xt/Athena, MAC Toolbox and OSF/Motif. More recently, the development of key component-ware technologies has been carried in out in parallel with advanced GUI libraries, such as COM (known as former ActiveX) for MFC and Java Beans for JFC. Although the notion of a software component technically denotes the packaging of software in the form of a binary-level reusable object over a component-ware technology, we will use the term component to generally refer to any independent part of a software system that plays a distinct functional role. In this context, interface components range from typical interaction object classes (e.g., menus, windows, buttons, check boxes, etc.), to more composite interactive artifacts serving specific roles (e.g., file menu, title bar, settings' dialogue box, print dialogue box, URL bar, etc.). Currently, primary emphasis is put on flexible component deployment during development time, rather than on automatic software assembly from constituent interface components. The need for software adaptability has been identified in [11], mainly emphasizing static software properties such as extensibility, flexibility and performance tunability, without negotiating the automatic and dynamic software assembly. Similarly, in [12], adaptability is also considered a key static property of software components, which can be pursued through aspectual decomposition, i.e., by employing aspect-oriented programming methods.

3. Decision-Making Logic Implementation

3.1. *Activation and cancellation decisions for interface components*

The outcome of a decision-making process is a sequence of *activation* and *cancellation* commands of named interface components, which are to be appropriately applied in the interface assembly process. The necessity of a component coordination command-set in implementing adaptation has been identified very early in [13], while the capability to manage dynamic interface component selection with just two fundamental commands has been introduced in the context of unified user interface development [1, 3]. In this context, the functional role of those commands in dynamic interface assembly is defined below:

- *Activation* implies the necessity to deliver the corresponding component to the end-user. Effectively, delivery may imply instantiation (i.e., instance creation) of the respective component class, in a way dependent on the implementation form of the component (i.e., for OOP classes, dynamic instantiation suffices, for component-ware technologies replication and object reference extraction is required).
- *Cancellation* implies that a previously activated component needs to be removed on the fly from the interface delivered to the end-user. In this case, cancellation is typically performed by destruction of the corresponding instance.

3.2. *Outline of the DMSL language*

The decision-making logic is defined in independent decision blocks, each uniquely associated to a particular task context; at most one block per distinct task context may be supplied. The decision-making process is performed in independent sequential *decision sessions*, and each session is initiated by a request of the interface assembly module for execution of a particular *initial decision block*. In such a decision session, the evaluation of an arbitrary decision block may be performed, while the session completes once the computation exits from the initial decision block.

The outcome of a decision session is a sequence of activation and cancellation commands, all of which are directly associated to the task context of the initial decision block. Those commands are posted back to the interface assembly module as the product of the performed decision-making session. In Fig. 3, an example decision block is shown, being an excerpt of the implementation of the decision logic AVANTI browser (see also Fig. 2), for selecting the best alternative interface components for the "link" task context. The interface design relating to this adaptation decision logic is provided in Fig. 4.

The primary decision parameters are end-user and the usage-context profiles, defined as two built-in objects, i.e., `user` and `context`, whose attributes are syntactically accessible in the form of named attributes. The binding of attribute names to attribute values is always performed at run-

```
taskcontext link  [
       evaluate linktargeting;
       evaluate linkselection;
       evaluate loadconfirmation;
]
taskcontext linktargeting [
       if (user.abilities.pointing == accurate) then
              activate "manual pointing";
       else
              activate "gravity pointing";
]
taskcontext linkselection [
       if (user.webknowledge in {good, normal}) then
              activate "underlined text";
       else
              activate "push button";
]
taskcontext loadconfirmation [
       if (user.webknowledge in {low, none} or context.net==low) then
              activate "confirm dialogue";
       else
              activate "empty";
]
```

Fig. 3. An example of a simple decision block to select the most appropriate delivery of web links for the individual end-user; note that names in italics are not language keywords but are treated as string constants, i.e., **`user.webknowledge`** is syntactic sugar for `user."webknowledge"`.

time. The encapsulation of composite attributes in user and context profiles is easily allowed due to the syntactic flexibility of attributes reference. For instance, `user.abilities.vision` and `user.abilities.hearing` are syntactic sugar for `user.″abilities.vision″` and `user.″abilities.hearing″`, where `″abilities.vision″` and `″abilities.hearing″` are two distinct independent ordinal attributes of the user built-in object. Consequently, even though all attributes in the DMSL language are semantically scalar, the flexibility of attribute names allows syntactical simulation of aggregate structures. Additionally in Fig. 3, the chain evaluation of other decision blocks through the evaluate command is illustrated. The latter can be employed when the adaptation decisions for a particular task context require decision-making for particular sub-task contexts.

Upon startup, the interface assembly module causes the execution of decision sessions for all polymorphic task contexts in a hierarchical manner (see Fig. 5), so that the required alternative interface components, given the particular end-user and usage-context, are effectively *marked* for interface delivery. Subsequently, the assembly process is performed, hierarchically instantiating and gluing together all marked interface components with the interface components of unimorphic task contexts.

As appears in the right part of Fig. 5, it is possible that more than a single alternative style can be selected for a particular polymorphic task context. This is dependent on the particular design rationale of the alternatives styles, while DMSL

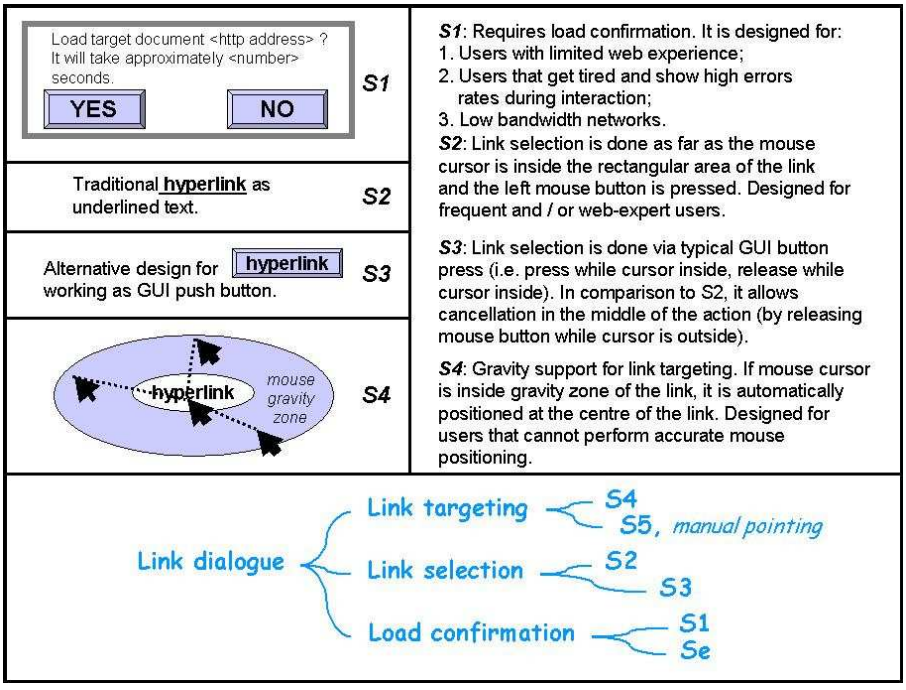

Fig. 4. The link selection task context, with its various sub-task contexts, and the associated design logic, which is encapsulated within the decision blocks of Fig. 3. Se is used to indicate an "empty" component (i.e., no load confirmation dialogue supported). S5 is the typical manual direct pointing of links using the mouse.

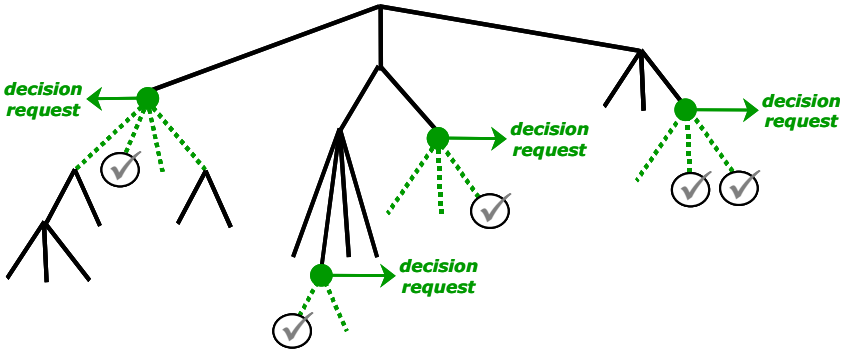

Fig. 5. Illustrating the hierarchical posting of decision requests, causing decision sessions for each polymorphic task context (shown with decomposition alternatives as dashed lines), and marking of selected alternative styles (i.e., interface components), after each decision session completes.

does not restrict decision blocks to output only a single activation command. Additionally, as it will be explained next, the relationships among the alternative styles of a polymorphic task context are completely formalized in the DMSL language,

associated with well-defined rule patterns for implementing the decision block of polymorphic task contexts. This serves two key objectives: (a) guiding designers in the organization and implementation of decision blocks; and (b) allowing developers to implement interactive design instruments that automate the generation of decision-blocks from the design relationships of alternative components. In Sec. 3.4, the *Mentor* interactive adaptation-design tool will be described [7] which exploits this feature.

3.3. *Relationships among alternative styles and associated rule patterns*

The emergence of alternative styles of polymorphic task contexts during adaptation design aims primarily to address the varying user and usage-context attribute values. For instance, as appears in the example of Figs. 3 and 4, the degree of the end-user web expertise leads to alternative styles for interactively supporting link selection. However, although this remark may lead to an initial assumption that all styles are mutually exclusive, there are additional design relationships among alternative styles, as demonstrated in the context of unified user interface design [2]. Those relationships are:

- *Exclusion or incompatibility* is applied if the various alternative styles are deemed to be usable only within the scope of their associated user and usage-context attribute values, because from the usability point of view it is inappropriate to concurrently instantiate both styles.
- *Compatibility* is applicable among alternative styles for which the concurrent presence during interaction allows the user to perform certain actions in alternative ways, without introducing usability problems.
- *Augmentation* aims to enhance the interaction with another particular style that is found to be valid, but not sufficient to facilitate the effective accomplishment of the supported user task. For instance, if during interaction it is detected that the user is unable to perform a certain task, task-sensitive guidance through a separate, but compatible, style could be delivered. In other words, the augmentation relationship is assigned to two styles when one can be used to enhance the interaction while the other is active (see Fig. 6).
- *Substitution*, exhibiting a very strong link with adaptivity techniques, is applied in cases where, during interaction, it is decided that some styles need to be substituted by others. For instance, the ordering, arrangement or availability of certain operations may change (see Fig. 6) on the basis of interaction monitoring and extraction of information regarding frequency of use and repeating usage patterns. In this case, some styles would need to be cancelled, while others would need to be activated.

In the DMSL language those relationships are not injected as a part of the semantics, but, as an alternative, concrete rule patterns are delivered, effectively

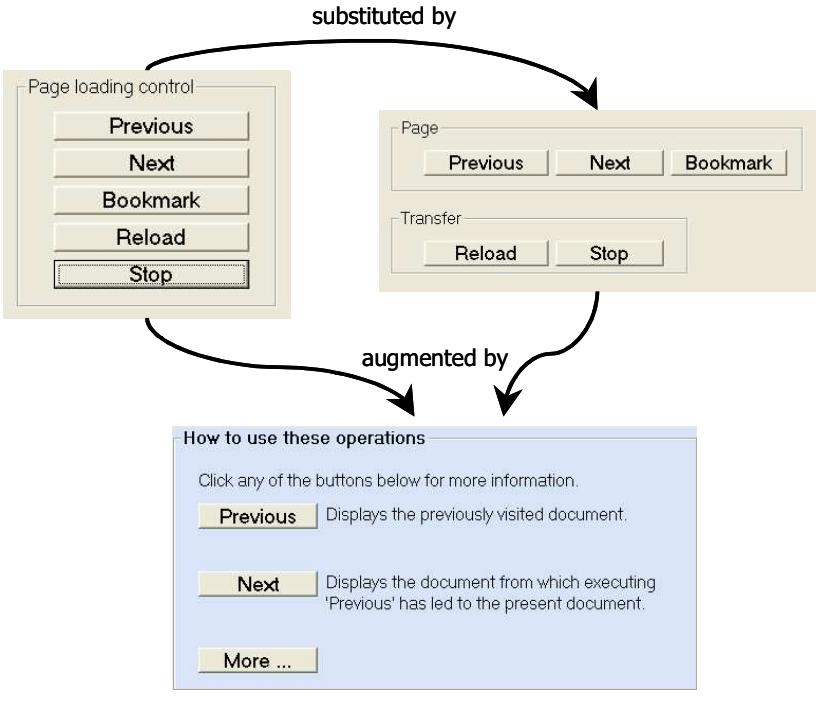

Fig. 6. Alternative styles and their design relationships for the "page loading control" task context of the AVANTI browser — from [2].

Exclusion(S_1, S_2)	**Compatibility(S_1, S_2)**

if (S_1.*cond*) then
activate S_1;
else
if (S_2. *cond*) then
activate S_2;
	if (S_1.*cond*) then
	activate S_1;
	if (S_2. *cond*) then
	activate S_2;

Substitution(S_1 by S_2)	**Augmentation(S_1 by S_2)**

if (S_2. *cond and* isactive(S_1)) then [
cancel S_1;
activate S_2;
]
else
if (S_1.*cond*)
activate S_1;
	if (S_1.*cond*)
	if (not isactive(S_1)) then
	activate S_1;
	else
	if (S_2. *cond*) then
	activate S_2;

Fig. 7. The decision rule patterns associated to the relationships among alternative styles; the style condition is the boolean expression engaging the user and context attribute values for which the style is designed.

mapping those relationships to implementation skeletons of decision blocks. This gives adaptation designers the freedom not to necessarily adopt those particular design relationships, in case, for instance, they do not choose to employ unified design as the adaptation-design approach. In Fig. 7, the DMSL decision-rule patterns are provided, for the previously described style relationships.

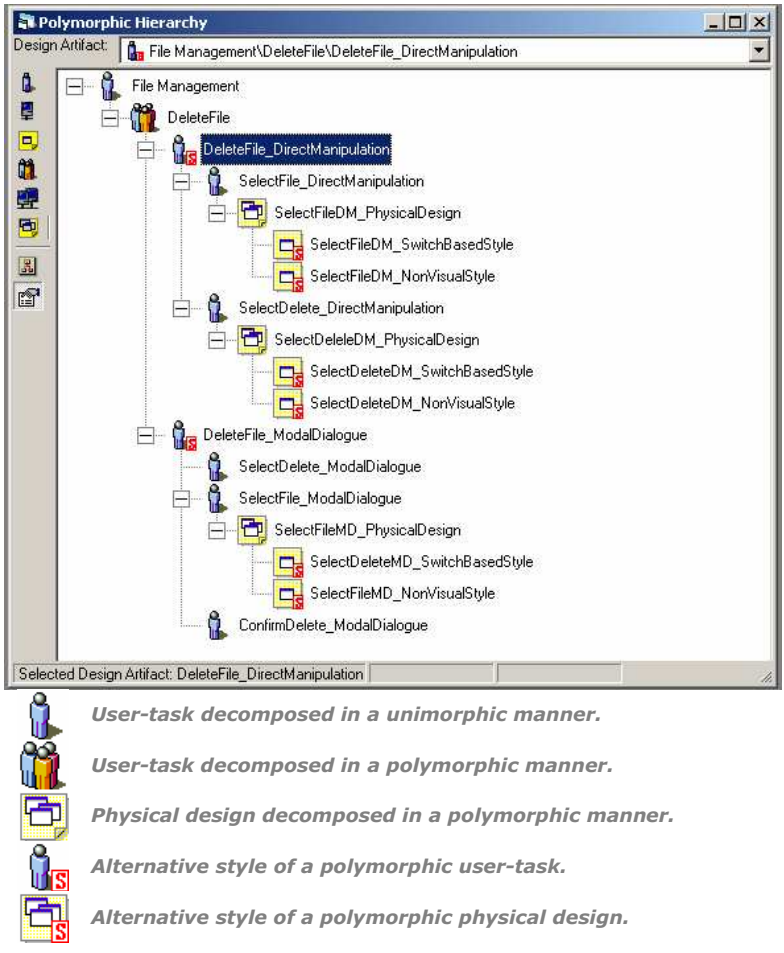

Fig. 8. A snapshot of adaptation-design in the Mentor tool, showing the structure of a polymorphic task hierarchy for the File Manager task context and its relevant sub-task contexts.

3.4. *The Mentor tool supporting interactive adaptation design*

As it has been previously mentioned, the design of adaptation entails the definition of polymorphic realizations of task contexts through alternative designed styles that are associated together by means of exclusion, compatibility, augmentation or substitution relationships. To provide tool-based assistance for such a demanding design process, it is necessary to enable the effective interactive manipulation of the overall design space of task polymorphic contexts and their alternative styles.

In the Mentor tool for interactive adaptation design, the polymorphic hierarchical decomposition of task contexts, either related to user tasks, system tasks (e.g., feedback) or physical design (e.g., graphical design), is genuinely supported. As depicted in Fig. 8, the task context *FileManagement*, is decomposed into

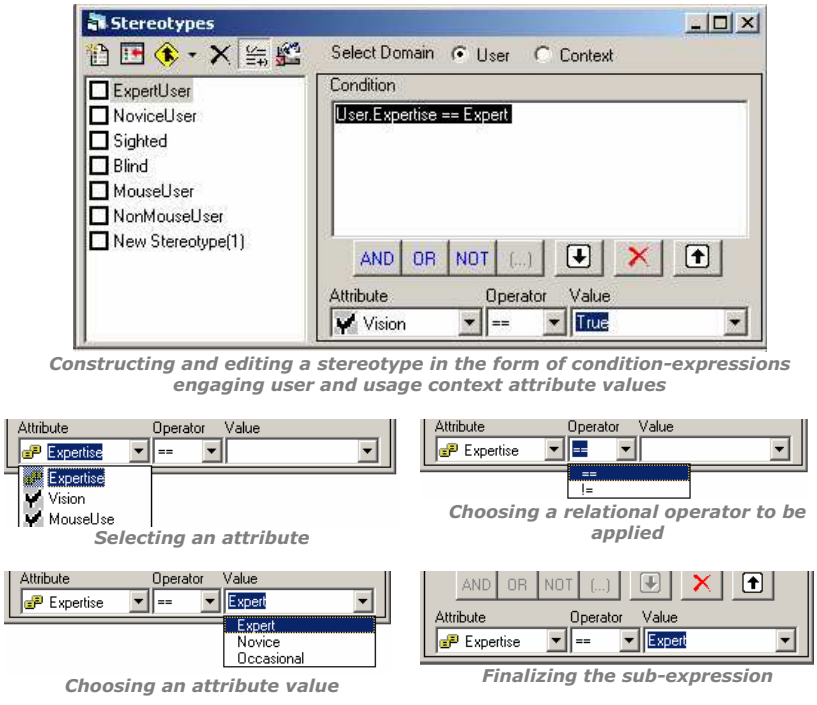

Fig. 9. Supporting user and usage-context profile manipulation, as well as condition editing, the latter for both styles and profile stereotypes.

alternative sub-tasks, like *DeleteFile*. The latter is defined to have multiple alternative decompositions, i.e. it is a polymorphic sub-task context, where alternative decompositions denote distinct uniquely named alternative styles, e.g. the styles named *DeleteFile_DirectManipulation* and *DeletFile_ModalDialogue* of Fig. 8. The key issue in the context of such adaptation design is the design rationale for alternative styles, i.e., why should a task-context be polymorphic. As it has been discussed earlier, alternative styles are required when, given a particular task-context and diverse user or usage-context attribute values, the delivery of a single style clearly constitutes, from a usability point of view, a less than perfect design decision.

Consequently, in interactive adaptation-design support, designers should be given the capability to manipulate: (a) user and usage-context profiles; and (b) the definition of conditions for alternative styles, in the form of expressions engaging user and usage-context profile parameters, such as the conditions of DMSL-language rules. Both of those key features are supported in the Mentor tool as illustrated in Fig. 9. The role of stereotypes is crucial in adaptation-design as it allows user-groups to be easily represented and directly referenced within style conditions. This has the following advantages:

- It relieves designers from the repetition of conditions when styles for the same user group are met in different task contexts;

```
stereotype user     Blind  : user.vision == false;
stereotype user     Novice : user.webknowledge in { low, none };
stereotype context  LowNet : context.net == low;
stereotype user     BlindNovice: Blind and Novice;

taskcontext loadconfirmation [
       if (Novice or LowNet) then
              activate “confirm dialogue;
       else
              activate “empty”;
]
```

Fig. 10. Definition and deployment of stereotypes; the decision block of Fig. 3 redefined to employ stereotypes, showing the increased readability of the adaptation rules.

- It makes design conditions self-documented, since stereotype identifiers make the design rationale far more explicit;
- It allows global changes in stereotype conditions without the need to manually change all individual style conditions.

Stereotypes are explicitly supported in the DMSL language, while the Mentor tool generates DMSL-compliant stereotype definitions, according to the interactively designed stereotypes (see upper part of Fig. 9). Stereotypes are referenced through their unique identifier while they may also be combined with other stereotypes in condition expressions. In Fig. 10, examples for stereotype definition and deployment in the DMSL language are provided.

The consistency checking of conditions is automatically performed in the Mentor tool, using knowledge of the types and value domains of all engaged user and usage-context attributes. This allows potential errors of un-satisfiability of expressions or type mismatches to be detected and reported to interface designers. Some examples showing the detection of inconsistencies for stereotype conditions are provided in Fig. 11.

4. Decision-Making Logic Verification Method

4.1. *Definitions*

To provide a vocabulary for the discussion of design logic verification, a semiformal definition of the adaptation-design task model is given, using standard first order logic. Subsequently, some formal properties of the adaptation-design logic are defined, which are considered important towards supporting designers in embedding a correct adaptation logic in their designs, and the implementation method adopted for verifying these properties is discussed.

4.1.1. *User Interface Design Task Model — UIDTM*

A *UIDTM* is a directed graph defined as a tuple:
$<$ *DESART*, art_0, *ART, U, C, UC, STYUC, STYLEDESREL*$>$ where:

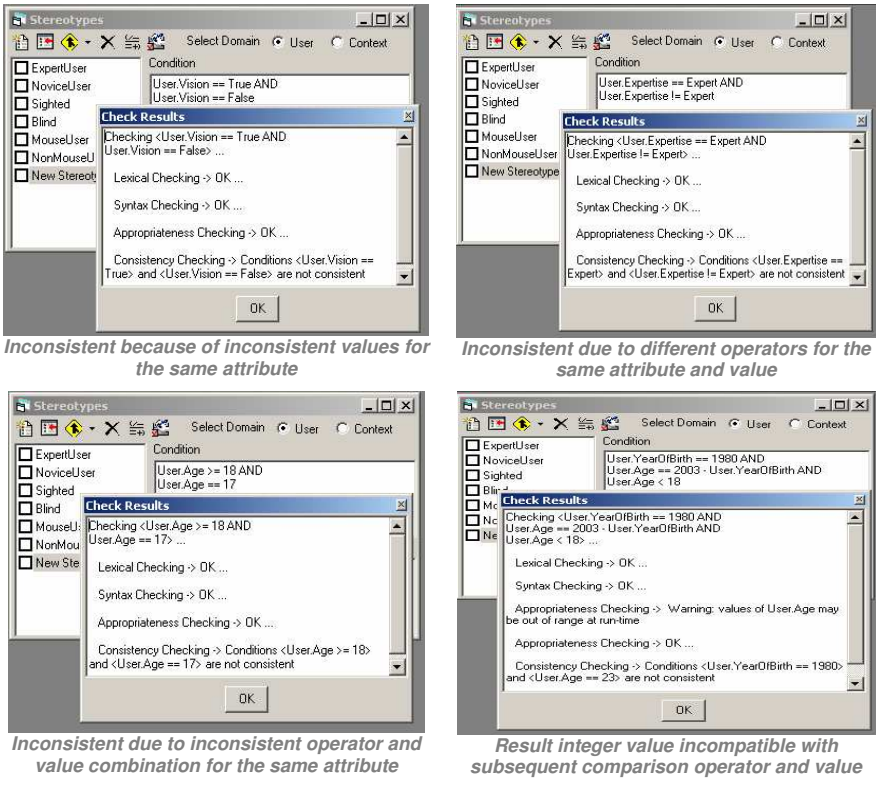

Fig. 11. Examples of condition consistency checking in the construction of stereotypes.

- *DESART* is the finite set of adaptation design artifacts $\{art_0, art_1, \ldots, art_n\}$ in the global hierarchical task model. Additionally, *DESART* is partitioned into two subsets, namely *UNIART* (unimorphic artifacts) and *POLYART* (polymorphic artifacts), implying that: $\forall\ \text{art}_i \in DESART \Rightarrow \text{art}_i \in UNIART \vee \text{art}_i \in POLYART$, where those two sets are defined as follows:
  - *UNIART* $\{uniart_0, uniart_1, \ldots, uniart_n\} \subseteq DESART$
  - *POLYART* $\{polyiart_0, polyart_1, \ldots, polyart_n\} \subseteq DESART$

  Additionally, the elements in *DESART* are cross-classified as follows:
  - *USERT* $\{usert_0, usert_1, \ldots, usert_n\} \subseteq DESART$, the set of user tasks in *DESART*
  - *SYST* $\{systt_0, systt_1, \ldots, systt_n\} \subseteq DESART$, the set of system tasks in *DESART*
  - *PHYSDES* $\{physdes_0, physdes_1, \ldots, physdes_n\} \subseteq DESART$, the set of physical designs in *DESART*
  - *STYLES* $\{style_0, style_1, \ldots, style_n\} \subseteq DESART$, the set of alternative design styles in *DESART*.

Value type	*Condition*	*Complement*
Boolean	$x = TRUE$	$x \neq TRUE, x = FALSE$
	$x = FALSE$	$x \neq FALSE, x = TRUE$
Enumerated	$x = val_i$	$x \neq val_i, x = \{ val_1, ...val_{1-1}, val_{i+1}, ...val_n \}$
	$x \neq val_i$	$x = val_i$
Integer	$x = a$	$x = b \wedge b \neq a$, $x \neq a$, $x > b \wedge b \geq a$ $x \geq b \wedge b > a$, $x < b \wedge b \leq a$, $x \leq b \wedge b < a$
	$x \neq a$	$x = b \wedge b \neq a$
	$x > a$	$x = b \wedge b \leq a$, $x < b \wedge b \leq a$, $x \leq b \wedge b < a)$
	$x \geq a$	$x = b \wedge b < a$, $x < b \wedge b \leq a$, $x \leq b \wedge b < a$
	$x < a$	$x = b \wedge b \geq a$, $x < b \wedge b \geq a$, $x \leq b \wedge b > a$
	$x \leq a$	$x = b \wedge b > a$, $x > b \wedge b \geq a$, $x \geq b \wedge b > a$

Fig. 12. Checking complement conditions for attribute values engaged in multiple conjucted conditions.

- art_0 is the root of the design artifacts' hierarchy, $art_0 \in USERT$
- $ART \subseteq DESART \times rel_{decomp} \times DESART$ is a set of binary relations $\langle art_i,\ rel_{decomp},\ art_j \rangle$ where rel_{decomp} is a decomposition relation for artifacts. Based on rel_{decomp}, all usual hierarchical relations (*mother, child, sister, descendant*, etc.), are also defined; for instance, $sister(art_i, art_j) \Leftrightarrow mother(art_{i-1}, art_i) \wedge mother(art_{i-1}, art_j)$.
- UC is a set of style conditions $\{uc_0, uc_1, \ldots, uc_n\}$ engaging user and usage-context attributes.
- $STYUC$ is a set of relations $\langle style_i,\ approp,\ uc_j \rangle$ where $style_i \in STYLES$, *approp* is a relationship defining if $style_i$ is an appropriate designed style for uc_j, and $uc_n \in UC$. For convenience, every relation $\langle style_i,\ approp,\ uc_j \rangle \in STYUC$ is referred with the notation $uc_{style\text{-}j}$, while the notation uc_i alone means the condition for $style_i$.
- $STYLEDESREL$ is a set of relations of the form $\langle style_i\ rel_{des}\ style_j \rangle$ where the two styles $style_i \in STYLES \wedge style_j \in STYLES$, are a pair of *sister* styles, and rel_{des} is one of the design relations of *incompatibility* (i.e. *exclusion*), *compatibility*, *augmentation* and *substitution*, as defined in adaptation design.

4.1.2. *Decomposition constraints on UIDTM*

Decomposition constraints on *UIDTM* concern the type of design artifacts that can be related through the rel_{decomp} binary relationship, and can be for formulated as follows:

4.1.2.1. *Unimorphic decomposition*

$(\forall\ art_i : art_i \in UNIART, \exists\, art_j : art_i\, rel_{decomp}\, art_j) \Rightarrow art_j \notin STYLES$
i.e., children of unimorphic tasks cannot be alternative styles.

4.1.2.2. *Polymorphic decomposition*

$(\forall\ art_i : (art_i \in POLYART \wedge art_i \in USERT)\, \exists\, art_j : art_i rel_{decomp} art_j) \Rightarrow art_j \in USERT \wedge art_j \in STYLES$
i.e., children of polymorphic user tasks can only be alternative user-task styles.

$(\forall\ art_i : (art_i \in POLYART \wedge art_i \in SYST), \exists\, art_j : art_i rel_{decomp} art_j) \Rightarrow art_j \in SYST \wedge art_j \in STYLES$
i.e., children of polymorphic system tasks can only be alternative system-task styles.

$(\forall\ art_i : (art_i \in POLYART \wedge art_i \in PHYSDES), \exists\, art_j : art_i rel_{decomp} art_j) \Rightarrow art_j \in PHYSDES \wedge art_j \in STYLES$
i.e., children of polymorphic physical designs can only be alternative physical-design styles.

4.1.2.3. *Uniqueness of physical designs*

$(\forall\ art_i\colon (art_i \in UNIART \wedge (art_i \in USERT \vee art_i \in SYST)),\ \exists art_j\colon art_i\ rel_{decomp}\, art_j \wedge art_j \in PHYSDES) \Rightarrow \neg\, \exists\, art_k : art_i rel_{decomp} art_k$
i.e., there can be only one physical design child of a unimorphic user-task or system-task.

4.1.2.4. *Physical design decomposition*

$(\forall\, art_i : art_i \in PHYSDES, \exists\, art_j : art_i rel_{decomp} art_j) \Rightarrow art_j \in PHYSDES$
i.e., physical designs only decompose to physical designs.

In the Mentor tool introduced in Sec. 3.4, these constraints do not need to be verified, since their satisfaction is interactively enforced through context-sensitive decomposition toolbars and menus (i.e., forbidden combinations are excluded since they are not supported through interface operations).

4.1.3. *User Interface Design Adaptation Model — UIDAM*

UIDAM is the subset $\langle STYLE,\ U,\ C,\ UC,\ STYUC,\ STYLEDESREL \rangle$ of *UIDTM*, and constitutes its adaptation model (i.e., adaptation-design logic). The following sections describe properties that are assumed over *UIDAM* as critical towards ensuring the reliability and soundness of the adaptation behaviour of the designed interface(s) and present how these properties can be verified.

4.2. *Satisfiability of conditions for alternative styles*

Requirement: $\forall uc_i \in UC,\ uc_i$ *is satisfiable*

Satisfiability entails that a condition expression engaging user and context attributes, denoting the adaptation-design rationale of a particular style, holds true for at least some instantiations of the user and usage-context profiles. For example, if

a DMSL adaptation rule has a condition *User.vision == TRUE AND User.vision == FALSE*, such an antecedent is contradictory and therefore un-satisfiable. In terms of adaptation logic, this would mean that there are no run-time situations in which a rule with such an antecedent would be triggered, i.e., that part of the rule will never apply. The reason for imposing style conditions satisfiability is therefore directly related to the rationale basis of the adaptation-design logic. Clearly, an adaptation logic in which some antecedents of adaptation rules are always false is a design error. For example, some user groups may never get the interface instance appropriate for them. The implementation method for satisfiability checking is based on the approach described in [12], the latter based on [13], and takes full advantage of the quasi-variable-free nature of DMSL-language condition expressions. In this context, the following steps are carried out.

4.2.1. *DNF reduction*

Concerns the typical simplification of the condition to Disjunctive Normal Form (DNF). A formula is said to be in DNF if it only contains disjunctions of conjunctions. This is achieved by eliminating negation operators from conditions, by recursively applying the following equivalences for the condition expressions:

- *De Morgan laws*
  $\neg (C_i \vee C_j) \Leftrightarrow \neg C_i \vee \neg C_j, \qquad \neg (C_i \wedge C_j) \Leftrightarrow \neg C_i \wedge \neg C_j$
- *Double negation elimination*
  $\neg\neg C \Leftrightarrow C$
- *Simple negation elimination*
  $\neg f = v \Leftrightarrow f \neq v, \qquad \neg f \neq v \Leftrightarrow f = v, \qquad \neg f > v \Leftrightarrow v \geq f$
  $\neg f \geq v \Leftrightarrow v > f, \qquad v > f \Leftrightarrow f \geq v, \qquad \neg v \geq f \Leftrightarrow f > v$

Subsequently, distributivity is applied top-down over conjunction and disjunction operators to eliminate conjunctions of disjunctions: $C_i \wedge (C_j \vee C_k) \Leftrightarrow (C_i \wedge C_j) \vee (C_i \wedge C_k)$. The result is effectively the production of a disjunction of conjunctions of the form:

$$(C1_0 \wedge \cdots \wedge C1_i \wedge \cdots \wedge C1_n) \vee \cdots \vee (Ci_0 \wedge \cdots \wedge Ci_i \wedge \cdots \wedge Ci_n) \vee \cdots \vee (Cn_0 \wedge \cdots \wedge Cn_i \wedge \cdots \wedge Cn_n).$$

4.2.2. *Satisfiability of individual disjuncts*

Each disjunct $D_i = (C_1 \wedge \cdots \wedge C_i \wedge \cdots \wedge C_n)$ in the resulting disjunction of conjunctions is checked for satisfiability until one satisfiable disjunct is found. Satisfiability of the disjunct is equivalent to the *consistency* of all atomic conditions in it. If no disjunct is consistent, the condition is un-satisfiable. Consistency for each disjunct is verified as follows:

- *Orthogonality*: if no pair of atomic conditions C_i and C_j in the disjunct involves the same attribute, the disjunct is consistent (e.g., the $f = v \wedge g = q$ is consistent since it holds $f \neq g$, i.e., the two conjuncted conditions in the disjunct are orthogonal).
- *Syntactic checking for constant values*: for each attribute occurring in more than one conjuncted condition, possible inconsistent values are searched pair wise by comparing the second value with the complements of the first, through the rules depicted in Fig. 12. For integer attributes, the value, if expressed as an arithmetic expression, may need to be calculated before applying the checking; e.g., in the DMSL language syntax, *User.YearOfBirth == 1985 AND User.Age == (2003 - User.YearOfBirth)* $\Leftrightarrow$ *User.Age == 18*. As the example shows, instantiated variables are substituted with constant values.
- (*Partial*) *Syntactic checking for variable values*: (i) consistency of atomic conditions involving one integer variable value is checked by comparing the constant value with the minimum/maximum values allowed in the user or context attribute declaration, to verify that the constant falls in the allowed range; (ii) consistency of atomic conditions involving two integer variables is checked by comparing the constant the minimum/maximum values allowed for the first attribute with the minimum/maximum values allowed for the second, to verify that the two ranges (partially) overlap.

A limit of the current approach is that it does not detect potential redundancy in conditions, and therefore does not provide support to designers on simplifying the redundant conditions to equivalent non-redundant conditions.

4.3. *Hierarchical consistency of style conditions*

Requirement: $(\forall\ style_i,\ style_j\colon style_i \in STYLES \wedge style_j \in STYLES \wedge antecedent(style_i, style_j)) \Rightarrow uc_{style-i}$ *and* $uc_{style-j}$ *are consistent*

This property states that style conditions along the same path in the polymorphic task hierarchy must be consistent, i.e., their conjunction must be satisfiable. For example, the DMSL conditions *User.Vision = TRUE* and *User.Expertise = EXPERT* are consistent, since their conjunction *User.Vision = TRUE AND User.Expertise = EXPERT* is satisfiable. Vice versa, the DMSL conditions *User.Vision = TRUE* and *User.Vision = FALSE* are not consistent, since their conjunction *User.Vision = TRUE AND User.Vision = FALSE* is a contradiction (it violates uniqueness of values). The hierarchical consistency property imposes a hierarchical discipline on style conditions, based on the structure of the polymorphic task-context hierarchy. In other words, each polymorphic decomposition point in the hierarchy determines partitions (as many as its styles are) corresponding to hierarchically descending *design paths*. For example, if at some point in the hierarchy a style is associated to the condition *User.Vision = TRUE*, a sighted-user path is explicitly established downwards to the task-context hierarchy.

All subsequent decompositions of such a style implicitly inherit its design condition, and thus refer unambiguously to sighted users.

In this context, the addition in the path of a new alternative style associated to the DMSL condition $User.Vision = FALSE$ would lead to inconsistent results. Semantically, hierarchical consistency of style conditions means that:

- Assume a pair of conditions $uc_{style\text{-}i}$ and $uc_{style\text{-}j}$, respectively associated to the pair of styles $style_i$ and $style_j$, where $style_i$ is hierarchically higher than $style_j$, meaning that a path exists between the two. Then:
  — $\forall$ set of values V such that $V(uc_{style\text{-}j}) = true \Rightarrow V(uc_{style\text{-}i}) = true$

For example, if $uc_{style\text{-}i}$ is $User.Vision = TRUE$ and $uc_{style\text{-}j}$ is $User.Expertise = EXPERT$, the satisfiable set V for $uc_{style\text{-}i}$ is that of sighted users, and of $uc_{style\text{-}j}$ is inherently that of expert sighted users. Should $uc_{style\text{-}j}$ be $User.Vision = FALSE$, the satisfiability set V of $uc_{style\text{-}j}$ becomes empty since $V(User.Vision = FALSE) \Rightarrow V(User.Vision = TRUE)$ is a contradiction. It is critical that such automatic hierarchical inconsistencies are detected during design-time, since at run-time they typically cause the inapplicability of a rule due to a conflict with the particular inherited conditions from hierarchically higher styles. The hierarchical consistency of a style condition uc_i in the polymorphic task hierarchy is verified through the same mechanism used for satisfiability. In fact, the following steps are performed:

- For each style $style_i$ in the polymorphic task hierarchy, an inherited condition iuc_i is defined as: $iuc_{i\text{-}1} \wedge uc_{i\text{-}1}$, i.e., the conjunction of the inherited condition and the normal condition of the parent style. The styles of top-level polymorphic artifacts have an empty inherited condition, and do not need consistency checking.
- Satisfiability is verified for the condition $iuc_i \wedge uc_i$.

If this conjunct is satisfiable, the condition locally associated to the current style is not incompatible with any of the conditions attached to styles in the same upgoing path of the task-context hierarchy, and therefore hierarchical consistency is preserved. Hierarchical consistency verification of a style presupposes that for all predecessor styles, their conditions have been already specified and passed successfully the consistency checking. Consistency is a necessary but not sufficient condition for the correct definition of style conditions, and it is complemented by hierarchical subsumption of style conditions as follows.

4.4. *Hierarchical subsumption of style conditions*

Requirement: $(\forall\ style_i, style_j\colon\ style_i \in STYLES \in style_j \wedge STYLES \wedge antecedent(style_i, style_j)) \Rightarrow subsumes(uc_i, uc_j)$

Effectively, all DMSL condition expressions are partial descriptions of sets in the considered semantic domain that encompass value assignments to user and usage-context attributes (i.e., instantiations, informally called *profiles*), for adapted user

interfaces. In other words, every DMSL condition denotes a particular set of user and context profiles, all of which satisfy the condition. *Subsumption* is a term frequently used in knowledge representation approaches (see [14, 15]) and in feature-based logics (see [12, 13]), indicating relationships of information generality or specificity, between logical descriptions of entities, which establish a partial order between the consistent entity-descriptions. For DMSL expressions, the subsumption relationship is defined as follows (by adapting the original definition provided in [12]):

- Let E be a condition with attributes $\alpha_1, \ldots, \alpha_n$ and $V(E)$ to be defined as the set of m n-tuples $\langle v_1, \ldots, v_n \rangle_1, \ldots, \langle v_1, \ldots, v_n \rangle_m$, such that: $\forall\ i \in (0, m]$, $E(\langle v_1, \ldots, v_n \rangle_i)$ is satisfied. That is, $V(E)$ is the set of all value assignment tuples for the attributes of E that satisfy E.
- Let E be a condition and α an attribute $\alpha \in E$. Then, $V(E, \alpha)$ is defined as the set: $\{\alpha_1, \ldots, \alpha_n\}$: $\alpha_i \in \langle v_1, \ldots, v_n \rangle_j\ \forall\ \langle v_1, \ldots, v_n \rangle_j \in V(E)$. That is, $V(E, \alpha)$ is the set of all values of α that appear in the satisfiability set of E.
- Let D_1 and D_2 be two satisfiable independent disjuncts of a DNF expression. The following relationships are defined:
  $stronger(D_1, D_2) \Leftrightarrow \forall\ \alpha \in D_2$: $\alpha \in D_T \wedge V(\alpha, D_1) \subseteq V(\alpha, D_2)$
  $weaker(D_1, D_2) \Leftrightarrow stronger(D_2, D_1)$
  According to the *stronger* relationship, satisfiability of D_1 implies satisfiability of D_2, since due to $V(D_1) \subseteq V(D_2)$, any satisfiability evaluation of D_1 is also a satisfiability evaluation for D_2.
- Let the conditions F and G in DNF, where $F = A_1 \vee \ldots \vee A_{nF}$, $G = B_1 \vee \ldots \vee B_{nG}$. The following relationship is defined if F *subsumes* G or G is *subsumed* by F:
  $subsumes(F, G) \Leftrightarrow (\forall\ A_i \in F\ \exists\ B_j \in G$: $stronger(B_j, A_i)) \wedge (\forall\ B_j \in G\ \exists\ A_i \in F$: $stronger(B_j, A_i)) \wedge V(F) \neq V(G)$
  $subsumed(F, G) \Leftrightarrow subsumes(G, F)$
  Effectively, this relationship states that every satisfiable disjunct of G is *stronger* than some satisfiable disjunct of F and every satisfiable disjunct of F is *weaker* than some satisfiable disjunct of G, and F does not imply G.
- Following the above definitions, it follows easily that:
  $subsumes(F, G) \Rightarrow (G \Rightarrow F \wedge \neg\,(F \Rightarrow G))$, i.e., G implies F, but not the other way round.

As an example of subsumption reflecting the above definition, the DMSL expression *User.Vision = TRUE* subsumes the expression *User.Vision = TRUE AND User.Expertise = EXPERT*, since the attribute *User.Vision* of the first condition, is also part of the second, and with the same satisfiable value set (i.e., *TRUE* value). It is intuitively evident that in the above example the two descriptions are consistent and the first is more general (i.e., provides less information) than the second (which explicitly specifies an additional user characteristic). In the hierarchical organization of alternative styles, the adaptation condition of a style, at any point in the hierarchy, must:

- *Be subsumed by the conditions of predecessor styles along the same path;*
- *Subsume the conditions of all decendent styles along the same path.*

This property essentially formalises the incremental specification nature of adaptation-oriented user interface design, as it enforces that polymorphic decomposition always proceeds by hierarchically specialising the condition of designed styles from the most generic to the most specific. At run-time, subsumption targets towards making sure that adaptation rules are contextually appropriate, and do not apply in vacuum. To provide an example combining both hierarchical consistency and subsumption, consider a particular rule specifying that "*in a non-visual interface component, if the user does not have visual ability a certain style should apply, else a different style should be delivered*". This rule is clearly useless (i.e., redundant) in the first part, since in the specific task context considered, the dialogue is already non-visual. Additionally, in its second part, the rule is completely out of context, since it makes little sense to hierarchically attach a style for a sighted user under non-visual style. Such a rule is effectively ruled out by subsumption on the first style (i.e., no need to re-specify), and by hierarchical consistency on the second (i.e., the conflicting condition will be detected). Clearly, the above subsumption relation between two conditions C_1 and C_2 is equivalent to the following definition given in terms of satisfiability by [13]:

- C_1, C_2 are consistent, i.e. C_2 implies C_1;
- $C_1, \neg C_2$ are consistent, i.e. C_1 does not imply C_2.

In the DMSL verification mechanism only the second of the two previous steps needs to be performed for determining subsumption, after the hierarchical consistency has been checked. Therefore, for a style condition uc_i in the polymorphic task-context hierarchy, consistent with its inherited condition iuc_i, hierarchical subsumption is verified in a straightforward manner by checking that the condition $iuc_i \wedge \neg\, uc_i$ is consistent. It is important to note that, contrary to satisfiability, which is always locally evaluated using the style condition independently of hierarchical context, both hierarchical consistency and subsumption are evaluated for a style condition in a hierarchical-context dependent fashion, always engaging the style's particular position in the task context hierarchy.

4.5. *Appropriateness of style relationships*

As it has been previously mentioned, style relationships concern the interplay of style adaptation conditions and adaptation-design relationships among alternative styles. Reflecting the semantic definitions from Sec. 3.3, $\forall\ style_i, style_j$: $style_i \in STYLES \wedge style_j \in STYLES$, the style relationships are defined as follows:

- $incompatible(style_i, style_j) \Rightarrow \neg\, consistent(uc_i, uc_j)$
  — The conditions of incompatible styles must not be consistent.

- $augments(style_i, style_j) \Rightarrow subsumes(uc_i, uc_j)$
  — If a style augments another style, its condition must be more specific than the condition of the first style.
  — $substitutes(style_i, style_j) \Rightarrow \neg\, consistent(u_{-i}, uc_j) \vee subsumes(uc_i, uc_j)$
- If a style augments another style, its condition must be either not consistent or more specific than the condition of the first style.

The reason for introducing the checking of design relationship properties is that they ensure that adaptation rules provide sufficient information for appropriately performing decision-making. For an incompatibility relation, it is required that the two style conditions are not consistent. In semantic terms, this means that style incompatibility implies that the set of user and context profiles denoted by the involved styles are disjoint. This is based on the rationale that incompatibility is a decision point in the overall adaptation decision making process in which the interface takes one "irreversible" adaptation action, whose choice should be based on unambiguously formulated design parameters. It should be noted that no verification checking is required when two styles are defined as compatible, since compatibility in adapted user interface design simply means that two styles can be activated simultaneously independently from their conditions as a design decision (e.g., in the case of multi-modal user interfaces).

4.6. *Complexity issues*

The verification mechanisms discussed above offer a simple and effective way of preventing interface adaptability development problems related to incorrect or ambiguous adaptation logic in the design phase, being an important feature towards the provision of adequate support for adaptation design. The approach described above has proved to be adequate and appropriate for the purposes of the work described in this paper, through mapping the necessary verification operations to well-known and algorithmically sound operations such as satisfiability, consistency and subsumption verification. The current discussion has not addressed relevant issues of computational complexity and optimisation, which are extensively discussed for similar approaches in [16] and [12]. The efficiency of the proposed solution remains to be ascertained, and is likely to relate to factors which depend on the large scale practice of adaptation design (such as, for example, the maximum size of style conditions, i.e., the number of atomic clauses, and the number of disjunctions and conjunctions involved, as well as the frequency of use of variable integer values for conditions), which impact the computational complexity of satisfiability verification, and, as a consequence, of all other verification operations that are based on condition satisfiability.

5. Dynamic Software Adaptability

5.1. *Dynamic polymorphic containment hierarchies*

The key architectural implication due to the functional requirement for dynamic interface assembly is the specific organization of implemented interface components to enable dynamically established containment hierarchies.

In non-adaptable unimorphic interactive applications developers typically program the hierarchical structure of the user interface through hard-coded parent-child associations that are determined during development time. However, in the context of adapted interface delivery, the component containment hierarchies should support two key features: (a) parent-child associations are always decided and applied during runtime; and (b) multiple alternatively candidate contained-instances are expected for container objects. The interface-component organization method of dynamic polymorphic containment hierarchies is illustrated in Fig. 13. This model has been employed for the implementation of the AVANTI browser, as shown in Fig. 2. Following Fig. 13, PL indicates the polymorphism factor, which provides the total number of all potential different run-time incarnations of an interface component, recursively defined as the product of the polymorphic factors of constituent component classes.

The dynamic interface assembly process reflects the hierarchical traversal in the polymorphic containment hierarchy, starting from the root component, to *decide, locate, instantiate* and *initiate* appropriately every target contained component (see Fig. 14). This process primarily concerns the interface components that implement polymorphic task-contexts. From the implementation point of view, the following software design decision have been made:

- The task-context hierarchy has been implemented as a tree data structure, with polymorphic nodes triggering decision making sessions (see also Fig. 5);
- Interface components have been implemented as distinct independent software modules, implementing generic containment Application Programming Interfaces (APIs), while exposing a singleton control-API for dynamic instantiation and name-based lookup;
- The interface assembly procedure is actually carried out via two successive hierarchical passes:
  - Execution of decision sessions, to identify the specific styles for polymorphic task contexts, that will be part of the eventually delivered user interface;
  - Interface construction, through instantiation and initiation of all interface components for the decided styles.

5.2. *Architectural generalization from dynamic interface assembly*

The architectural generalization of the dynamic interface assembly method is a normal manual transformation of the basic architectural entities of interface adaptability to the general software architecture domain. As it has been discussed in

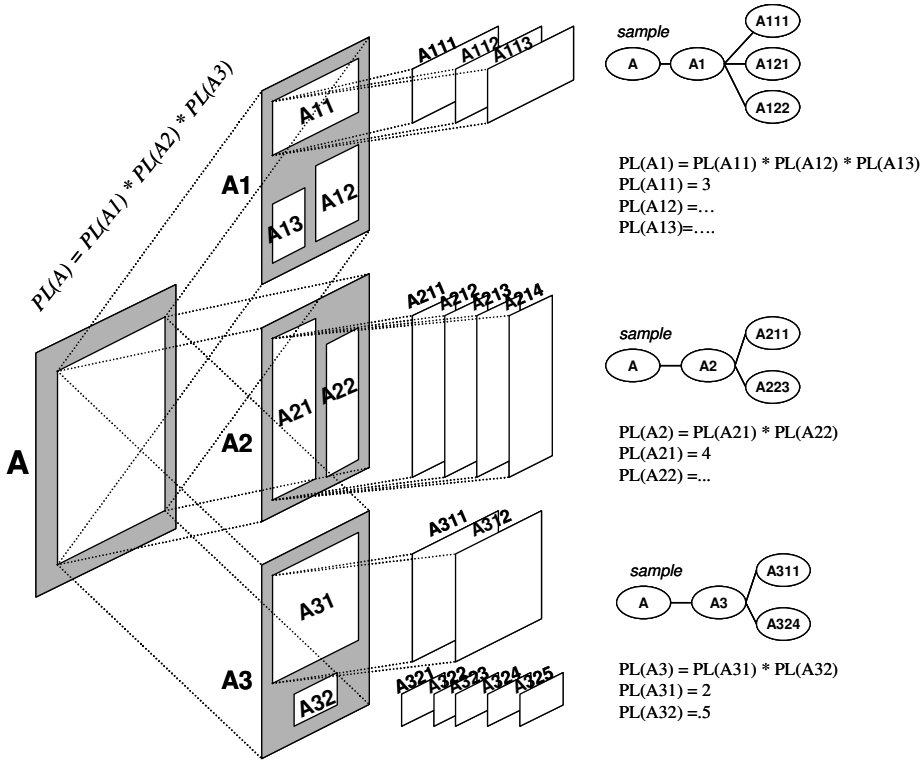

Fig. 13. The notion of dynamic polymorphic hierarchical containment in interface adaptability, to cater for the runtime interface assembly process.

Sec. 2.2, the adopted notion of software adaptability reflects the functional properties of automatic software assembly, through decision-making that relies upon runtime software adaptation parameters. It should be noted that this is a fundamentally different target from methods related to software evolution, which focus on the automated transformation and evolution of software structures at development-time, according to diverse software requirements. The key architectural generalization aspects are detailed in Fig. 15.

This generalization leads to an augmented vocabulary for the software architecture domain, mainly introducing the meta-elements necessary to accommodate runtime software assembly driven by decision-making for deployment adaptation, as illustrated in Fig. 16.

Containment APIs are effectively transformed to functionality abstraction APIs, the latter normally differentiating per polymorphic architectural context and distinct application domain, which all alternative candidate components should thoroughly implement. The generalization of interface-adaptation decision parameters, i.e., user and user-context profiles, concerns the *software deployment parameters*

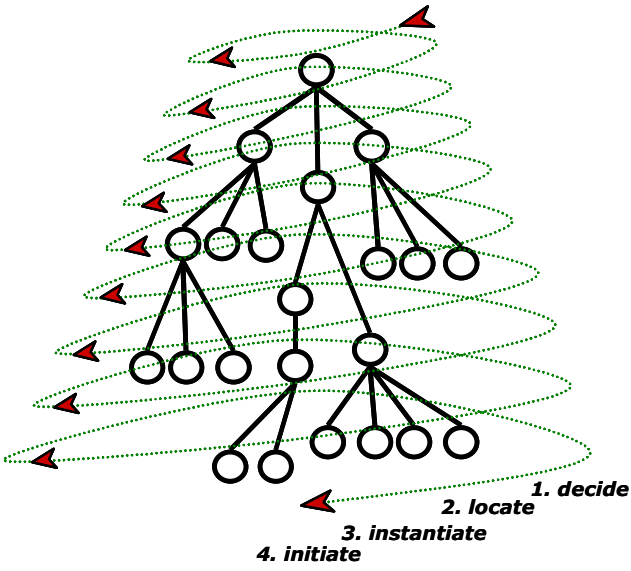

Fig. 14. Illustration of the dynamic interface assembly process as an incremental hierarchical construction procedure.

Original architectural semantics	***Generalized architectural semantics***
Hierarchical interface structure	Hierarchical architectural structure
Task context	Architectural context
Interface component	Software component
User and usage-context attributes	Software deployment parameters
User and context stereotypes	Software deployment scenarios
Polymorphic task context	Polymorphic architectural context
Alterative styles	Alternative encapsulated components
Task decomposition	Architectural decomposition
Task-based component indexing	Architectural role component indexing
Interface component containment	Architectural containment
Generic containment APIs	Functional-role abstraction APIs

Fig. 15. Generalizing the architectural semantics of dynamic interface assembly from the interface development domain to the software development domain.

of Figs. 15 and 16, conveyed as generic profile structures with a domain-specific interpretation. From the DMSL language point of view, such an extension can be trivially accommodated, by accumulating *user* and *context* built-in profiles in a generic profile-structure named *params*. For instance, the user and context profiles become syntactically visible as `params.user` and `params.context` respectively. In the Appendix at the end, the original DMSL grammar for interface adaptability is provided, together with the slightly updated version for decision-making in dynamic software assembly.

Domain specific parameters may reveal software deployment characteristics, which constitute the basis for choosing alternative best-fit software components,

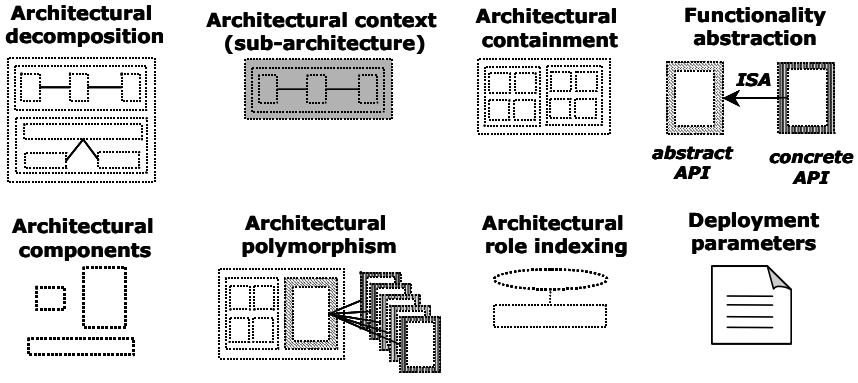

Fig. 16. The key architectural meta-elements in the context of dynamic software adaptability, as transformations of the corresponding elements from the interface adaptability architectural domain.

while stereotypes may reflect particular deployment scenarios. For example, `params.memory` and `params.storage` could be used to choose alternative implementations of algorithms with varying main and secondary memory requirements. Naturally, the same behavior can be accomplished by injecting decision-making within the software itself. However, through the separation and externalization of the design logic from the implementation components, better reusability is accomplished through orthogonal component combinations. Apart from interface assembly and the potential for dynamic software assembly, the same approach has been effectively employed in the context of adapted information delivery, realized through dynamic content assembly, as discussed in the next section.

5.3. *Dynamic query assembly for content adaptability*

In the context of the PALIO project (see acknowledgments), the DMSL language and the software engineering method for dynamic software assembly have been effectively employed for adaptable information delivery [6] over mobile devices to tourist users. The decision-making process was based on parameters such as nationality, age, location, interests or hobbies, time of day, visit history, and group information (i.e. family, friends, couple, colleagues, etc.). The information model reflected a typical relational database structure, while content retrieval was carried out using XML-based SQL queries. In this context, in order to enable adapted information delivery, instead of implementing hard-coded SQL queries, query patterns have been designed, with specific polymorphic placeholders filled in by dynamically decided concrete sub-query patterns. For instance, as seen in Fig. 17, particular data categories or even query operations may be left "open", with multiple alternatives, depending on runtime content-adaptation decision making.

The implementation of dynamic query assembly has been realized through: (a) the hierarchical representation of the polymorphic query structure as a tree

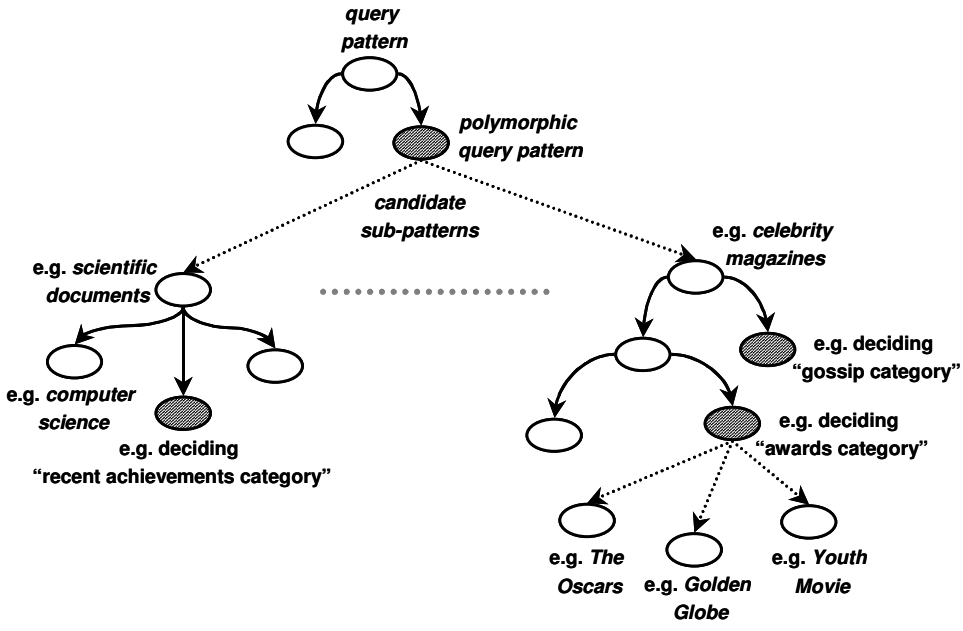

Fig. 17. Query patterns with polymorphic placeholders having multiple alternative candidate sub-patterns, selected through decision-making.

(i.e., in the same way as hierarchical task-contexts); (b) the easy implementation of the alternative sub-query patterns as text fragments, due to the textual nature of XML-based SQL queries; and (c) the incremental assembly of the eventual query from its constituent textual elements.

6. Summary and Conclusions

This paper has presented the DMSL language for adaptation-oriented decision-making specification, accompanied with a design-logic verification method, and an appropriate software engineering approach to accommodate dynamic software adaptability. The language has been intensively applied and tested in the course of various developments targeted to supporting user and usage context interface adaptation, like the AVANTI browser [4] of the AVANTI Project (see acknowledgments), and adapted information services, like the PALIO system [6] of the PALIO Project (see acknowledgments). Additionally, the DMSL language has been employed in the Mentor interactive tool [6] for adaptation-design supporting the polymorphic hierarchical decomposition of task-contexts [3], generating decision rules in the DSML language.

In all these developments, the DMSL language played a crucial software engineering role, effectively enabling the separation between the decision-making logic from the repository of interface components and their runtime coordination. As a

result, the decision-making logic has been made editable and extensible directly by interface designers, while interface component reuse has been largely promoted due to the orthogonal combinations inherent in the organization model of dynamic polymorphic containment hierarchies. The DMSL language reflects the polymorphic hierarchical design model, in which alternative design-parameter values (e.g., user/context/deployment attributes) require the presence of alternative design solutions (e.g., styles, software components) in different design contexts (e.g., task contexts or architectural contexts). Additionally, the DMSL language comes with a specific software meta-architecture to better encapsulate its decision-making facilities for adapted software delivery.

The verification method presented for the adaptation-logic focuses on adaptation-design conditions for alternative styles, emphasizing: satisfiability, hierarchical consistency, hierarchical subsumption for inherited conditions, and appropriateness of conditions for formally related styles. Complexity analysis and study of optimisation methods, which were not addressed in the design of the verification approach, are also planned. In particular, it is planned to investigate the simplification of redundant DMLS conditions, as well as the implementation of mechanisms for keeping track of conditions for which satisfiability has been already computed in a given design, and of condition pairs for which subsumption has already been computed, in order to avoid duplicating computational effort. The effectiveness, but also the necessity, of the above mechanisms will depend on empirical data on the practice of adaptation design (such as, for example, the maximum size of style conditions, i.e., the number of atomic clauses, and the number of disjunctions and conjunctions involved, as well as the frequency of use of variable integer values for conditions), which impact the computational complexity of satisfiability verification, and, as a consequence, of all other verification operations that are based on condition satisfiability.

The binding of decision parameter attributes, decision block identifiers, and style names is always performed at runtime, enabling rules to be edited independently of the software implementation of interface components. Such loose coupling between the decision logic and the coordinated components allows decision kernels to be effectively reused across applications of the same domain, as far as the same design-time naming conventions are applicable. This accounts not only for the whole decision logic as a single reusable unit, but also for specifically selected decision blocks. For instance, the decision blocks related to the "Help" task-context, and all its related sub-task contexts, may be directly reused across applications offering interactive help facilities. The latter necessitates reuse of the corresponding implemented interface components, something that can be directly accommodated when those particular components implement abstract APIs for dynamic containment and "Help" facilities, as opposed to "hard-wiring" with the overall interface implementation.

The DMSL language and the dynamic software assembly approach have been also employed for adapted information retrieval. In this case, the implemen-

tation model has been radically different in comparison to the typically programmed interface components in adapted interface delivery, being actually realized through XML-based SQL queries. The first step was the identification of the elements that had to dynamically vary due to adaptation (i.e., query sub-patterns), followed by the definition and study of the super-structure encapsulating all alternative elements. From this initial analysis, the emergence of a polymorphic hierarchy became quickly evident, while the assembling of adapted query-instances through combinations of textual patterns was easily crystallized as an appropriate implementation technique.

Overall, the described decision-making language is suited to address the development of systems that have to accommodate diversity on the fly, through dynamically decided and delivered software artifacts. In such cases, runtime decision-making relying upon diversity-parameters becomes mandatory, while the software organization of the system's implementation around architectural polymorphism is considered to be a fundamental architectural property.

Appendix

DMSL grammar for interface adaptation

```
logic ::= { block | stereotype }
block ::= 'taskcontext' (string | name) compound
compound ::= '[' { stmt } ']'
stmt ::= ifst | ('activate' | 'cancel' | 'evaluate') expr ';' | compound
ifst ::= 'if' '(' boolexpr ')' 'then' stmt [ 'else' stmt ]
expr := primary | boolexpr | arithexpr
primary ::= const | param | funccall | '-' expr | 'not' expr | name
param ::= ('user' | 'context') '.' (string | name)
funccall ::= libfunc '(' [ expr { ',' expr } ] ')'
const ::= 'true' | 'false' | number | string | name
boolexpr ::= expr boolop expr | expr 'in' set
arithexpr ::= expr arithop expr
arithop ::= '+' | '-' | '*' | '/' | '%'
boolop ::= 'or' | 'and' | '<' | '>' | '<=' | '>=' | '=' | '!='
set ::= '{' [ expr '{,', expr } ] '}'
libfunc ::= 'isactive | 'tonumber' | 'hasattr'
stereotype ::= 'stereotype' ('user' | 'context' ) name ':' boolexpr ';'
```

DMSL grammar modifications for software adaptation

```
block ::= 'architecturecontext' (string | name) compound
param ::= 'params' '.' (string | name)
stereotype ::= 'stereotype' name ':' boolexpr ';'
```

Acknowledgments

Part of the reported work 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 (1995–1998). 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).
- PALIO IST–1999–20656 (Personalized Access to Local Information and Services for Tourists, partially funded by the IST Program of the European Commission, and lasted 36 months (2000–2003). The partners of the PALIO consortium are: ASSIOMA S.p.A. (Italy) — Prime Contractor; CNR-IROE (Italy); Comune di Firenze (Italy); ICS-FORTH (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 S.A. (Greece).

References

1. A. Savidis and C. Stephanidis, The unified user interface software architecture, in C. Stephanidis (ed.), *User Interfaces for All — Concepts, Methods, and Tools*, Lawrence Erlbaum Associates, Mahwah, NJ, 2001.
2. A. Savidis and C. Stephanidis, Unified user interface design: Designing universally accessible interactions, *Int. J. Interacting with Computers*, Issue 16 (2004), pp. 243–270. Available electronically from: http://authors.elsevier.com/sd/article/S0953543804000025.
3. A. Savidis and S. Stephanidis, Unified user interface development: Software engineering of universally accessible interactions. To appear in the *Universal Access in the Information Society* **3**(3-4) (2004) 165–193.
4. C. Stephanidis, A. Paramythis, M. Sfyrakis, and A. Savidis, A case study in unified user interface development: The AVANTI web browser, in *User Interfaces for All*, C. Stephanidis (ed.), Lawrence Erlbaum, NJ, 2001, pp. 525–568.
5. C. Stephanidis, A. Savidis, and D. Akoumianakis, Tutorial on Engineering Universal Access: Unified User Interfaces. Tutorial in 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, Louisiana, USA, 5–10 August. [On-line]. Available at: http://www.ics.forth.gr/proj/at-hci/files/uahci_2001.pdf.
6. C. Stephanidis, A. Paramythis, V. Zarikas, and A. Savidis, The PALIO Framework for Adaptive Information Services, in A. Seffah and H. Javahery (eds.), *Multiple User Interfaces: Cross-Platform Applications and Context-Aware Interfaces*, John Wiley & Sons, Chichester, UK (2004), pp. 69–92.
7. C. Stephanidis, M. Antona, and A. Savidis, Design for All: Computer assisted design of user interface adaptation, in G. Salvendy (ed.), *Handbook of Human Factors and Ergonomics*, 3rd edition, John Wiley & Sons, 2005, pp. 1459–1485.

8. D. Benyon, MONITOR: A self-adaptive user-interface, in *Proc. IFIP Conf. on Human-Computer Interaction: INTERACT '84* (vol. 1), North-Holland, Elsevier Science, Amsterdam, 1984, pp. 335–341.
9. J. Cote Muñoz, AIDA — An adaptive system for interactive drafting and CAD applications, in M. Schneider-Hufschmidt, T. Köhme, and U. Malinowski (eds.), *Adaptive User Interfaces — Principles and Practice*, North-Holland, Elsevier Science, Amsterdam, 1993, pp. 225–240.
10. A. Kobsa and M. Pohl, The user modeling shell system BGP-MS, *User Modeling and User-adapted Interaction* **4**(2) (1995) 59–106. http://www.ics.uci.edu/∼kobsa/papers/1995-UMUAI-kobsa.pdf. Program code: ftp://ftp.gmd.de/gmd/bgp-ms.
11. M. Fayad and M. Cline, Aspects of software adaptability, in *CACM Journal* **39**(10) (1996) 58–59.
12. P. Netinant, C. A. Constantinides, T. Elrad, and M. E. Fayad, Supporting aspectual decomposition in the design of adaptable operating systems using aspect-oriented frameworks, in *Proc. 3rd Workshop on Object-Orientation and Operating Systems*, Sophia Antipolis, France, June 2000, pp. 36–46.
13. G. Cockton, Spaces and distances — Software architecture and abstraction and their relation to adaptation, in M. Schneider-Hufschmidt, T. Köhme, and U. Malinowski (eds.), *Adaptive User Interfaces — Principles and Practice*, North-Holland, Elsevier Science, Amsterdam, 1993, pp. 79–108.
14. P. F. Patel-Schneider, B. Owsnicki-Klewe, A. Kobsa, N. Guarino, R. MacGregor, W. S. Mark, D. L. McGuinness, B. Nebel, A. Schmiedel, and J. Yen, Term subsumption languages in knowledge representation, *AI Magazine*, **11**(2) (1990) 16–23.
15. A. Borgida and P. F. Patel-Schneider, A semantics and complete algorithm for subsumption in the classic description logic, *J. Artificial Intelligence Research* **1** (1994) 277–308.
16. G. Smolka, Feature-constrained logics for unification grammars, *J. Logic Programming* **12** (1992) 51–87.
17. C. Stephanidis, A. Savidis, and D. Akoumianakis, Tutorial on Universally accessible UIs: The Unified User Interface development. Tutorial in the ACM Conference on Human Factors in Computing Systems (CHI 2001), Seatle, Washington, 31 March–5 April. [On-line]. Available at: http://www.ics.forth.gr/proj/at-hci/files/CHI_tutorial.pdf.
18. A. Zeller and G. Snelting, Unified versioning through feature logic, *ACM Transactions on Software Engineering and Methodology* (*TOSEM*) **6**(4) (October 1997).