Abstract

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 employment of the language in an adaptation-design support tool.

1 Introduction

1.1 Dynamic interface adaptability

The need for software adaptability has been identified in (Fayad & Kline, 1996), mainly emphasizing static software properties such as extensibility, flexibility and performance tunability, without negotiating the automatic and dynamic software assembly. Similarly, in (Netinant et al., 2000), 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. 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 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 (Savidis & Stephanidis, 2001). In such a 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 Figure 1, illustrating the key principles of the Unified User Interface architecture (Savidis & Stephanidis, 2004a) for automatically adapted interactions.
As depicted in Figure 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** (Savidis & Stephanidis, 2004a) 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** (Savidis & Stephanidis, 2004b) 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, indexing uniquely 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 (Stephanidis et al. 2001), supporting interface adaptability. In Figure 2, an excerpt from the polymorphic interface component organization of the AVANTI browser is shown, to accommodate implementation of interface adaptability.
Figure 2: Parametric polymorphic containment with variant constituent components in the AVANTI browser. The indication “Empty” indicates components whose presence may have to be omitted upon dynamic interface delivery for certain user categories.

2 Decision-making specification language

2.1 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 (Cockton, 1993), 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 (Savidis & Stephanidis, 2001; 2004b). 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.

2.2 Outline of the language

The Decision Making Specification Language (DMSL) is a language specifically designed and implemented for supporting the specification of adaptations. 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 Figure 3, an example decision block is shown, being an excerpt of the implementation of the decision logic AVANTI browser (see also Figure 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 Figure 4.

```
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";  
]  
```

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

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 runtime. 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 Figure 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.
Figure 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 Figure 3. S1 is used to indicate “empty” (i.e. no load confirmation dialogue supported). S5 is the typical manual direct pointing of links using the mouse.

Figure 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.

Upon startup, the interface assembly module causes the execution of decision sessions for all polymorphic task contexts in a hierarchical manner (see Figure 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 Figure 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 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.

### 2.3 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 Figure 3 and Figure 4, the degree of the end-user web expertise leads to alternative styles for interactively supporting link selection. However, not all styles are mutually exclusive, and there are additional design relationships among alternative styles, as demonstrated in the context of unified user interface design (Savidis & Stephanidis, 2004a). These 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 Figure 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 Figure 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, alternatively, concrete rule patterns are delivered, effectively 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 Figure 7, the DMSL decision-rule patterns are provided, for the previously described style relationships.

<table>
<thead>
<tr>
<th>Exclusion($S_1, S_2$)</th>
<th>Compatibility($S_1, S_2$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>if ($S_1$.cond) then activate $S_1$; else if ($S_2$.cond) then activate $S_2$;</td>
<td>if ($S_1$.cond) then activate $S_1$; if ($S_2$.cond) then activate $S_2$;</td>
</tr>
<tr>
<td>Substitution($S_1$ by $S_2$)</td>
<td>Augmentation($S_1$ by $S_2$)</td>
</tr>
<tr>
<td>if ($S_2$.cond and isactive($S_1$)) then [ cancel $S_1$; activate $S_2$; ] else if ($S_1$.cond) activate $S_1$;</td>
<td>if ($S_1$.cond) if (not isactive($S_1$)) then activate $S_1$; else if ($S_2$.cond) then activate $S_2$;</td>
</tr>
</tbody>
</table>

Figure 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.

3 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 in supporting user and usage context interface adaptation, like the AVANTI browser (Stephanidis et al., 2001) of the AVANTI Project (see acknowledgments), and adapted information services, like the PALIO system (Stephanidis et al., 2004a) of the PALIO Project (see acknowledgments). Additionally, the DMSL language has been employed in the Mentor interactive tool (Stephanidis et al., 2004b) for adaptation-design supporting the polymorphic hierarchical decomposition of task-contexts (Savidis & Stephanidis, 2004b), 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. 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.
Acknowledgements

Part of the reported work has been carried out in the context of the: (a) 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); (b) 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).

References


