A Set of Languages for Context-Aware Adaptation

Giuseppe Giani, Fabio Paternò, Carmen Santoro, Lucio Davide Spano
CNR-ISTI, HIIS Laboratory
Via Moruzzi 1, 56124 Pisa, Italy
{giuseppe.ghiani, fabio.paterno, carmen.santoro, lucio.davide.spano}@isti.cnr.it

ABSTRACT
The creation of service front ends able to adapt to the context of use involves a wide spectrum of aspects to be considered by developers and designers. A context-aware adaptation enabled application needs a simultaneous management of very different application functionalities, such as the context sensing, identifying different given situations, determining the appropriate reactions and the execution of the adaptation effects. In this paper we describe an adaptation architecture for tackling this complexity and we present a set of languages that address the definition of the various aspects of an adaptive application.

Author Keywords
Adaptation, Context-Awareness, User-Interface models, Rule Languages.

ACM Classification Keywords
H.5.m. Information interfaces and presentation (e.g., HCI)

General Terms
Human Factors; Design; Languages.

INTRODUCTION
The creation of Service Front Ends (SFE) that are able to adapt to the context of use enhances the user experience by taking into account different interaction techniques for different situations. However, creating applications that are able to react to the different states of the context involves the management of different aspects, which may be difficult to be handled if not considered with a systematic approach. For instance, adaptive applications need to sense the context, to reason about its state in order to determine which is the current situation, to decide which actions are more appropriate given the current situation and finally to execute these actions. In this paper, we consider different aspects involved in the definition of an adaptive SFE. After discussing related work, we introduce an adaptation architecture for separating the different adaptation concerns, then provide a brief description of the languages exploited by the different architecture modules (the UI definition, the context management and the adaptation rules) and finally we discuss conclusions and future work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

RELATED WORK
In this section, we briefly summarize some related work for the presented set of languages.

The ASFE-DL language was submitted to the Model Based User Interface Working Group of the W3C, which aims to create a standard for the task modelling and for the abstract user interface languages. In this regard, the language had benefit from the comparison with the different submissions in the abstract UI field, which were MARIA [4], a universal, declarative, multiple abstraction-level, XML-based language for modelling interactive applications in ubiquitous environment, compliant with [7]; MINT-AIM [1] a language targeted for supporting multimodal setups, combining at least one media with several modes; UseDM [5] that describes abstract dialog models for context-sensitive interactive systems; UsiXML [3], which is a formal Domain-Specific Language (DSL) used that describes user interfaces from the implementation technology. The comparison with these languages allowed the improvement of ASFE-DL, which now contains modelling elements that are able to cover all the concepts proposed by these state of the art languages.

Regarding the AAL Description Language, we first mention some languages for model transformations. For instance, QVT (Query/View/Transformation) is a standard set of languages for the transformation of models, defined by the Object Management Group (OMG). The ATL (Atlas Tranformation Language) [2] is an implementation of the QVT proposal by OBEO and INRIA. It maintains the possibility to specify the transformation with a set of declarative matched rules that transform elements of the source model into elements of the target model. They are triggered with a pattern matching mechanism. XSLT (eXtensible Stylesheet Language Transformations) [7] is an XML syntax for defining transformations starting with XML files to different text formats, obviously including XML itself, based on pattern matching. A style sheet defines a set of rules that allows to take as input an XML

tree (source) and to transform it into a destination tree (result). More specifically to adaptation we can mention the work by Sottet and others [10] who presented a set of general principles relevant for supporting model-based adaptation while in our case we present a set of languages supported by engineered tools that can be applied in real world applications. Octavia et al.[9] have considered the use of a model-based approach to facilitate adaptation in virtual environments, also using the event-condition-action paradigm, we provide a more general approach for this purpose able to potentially support adaptation involving various interaction modalities.

Issues about the exploitation and the management of the context information have already previously tackled by the research community. Salber et al. [5] have introduced the concept of *context widgets*, which separate the application from the context acquisition concerns by hiding the sensor specifics to the higher layers (i.e.: to the application layer). Context widgets have a state and a behaviour, manage raw data coming from generators (which get data from sensors) and may interpreters if further abstraction is needed.

ADAPTATION ARCHITECTURE

Our adaptation architecture separates different concerns that are involved in the creation of an adaptive SFE. In this section, we provide a big picture of the involved modules and languages.

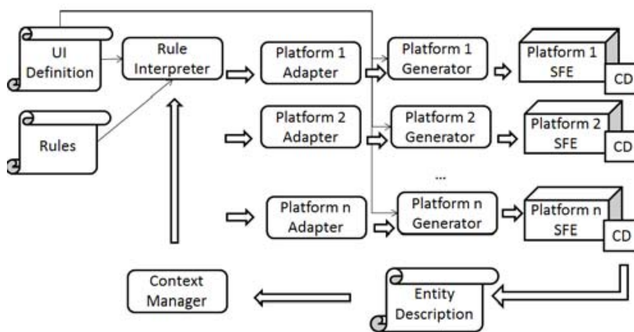

Figure 1: Adaptation Architecture

Figure 1 shows the proposed adaptation architecture. The parchment-shaped boxes represent the different languages, the rounded boxes the architecture modules, while the 3D boxes represent running SFEs. The thinner arrows represent connections at design time, while the thicker ones represent communications at runtime. At the design phase, the developers specify the initial version of the SFE, providing different versions for different platforms (e.g. graphical desktop, mobile, vocal, multimodal), using the *UI definition* languages. Such descriptions are exploited by the platform generators that create the initial executable versions of the SFE. As we better describe in section 2, it is also possible to have higher-level descriptions that are common for all considered platforms. The adaptation logic is defined by an adaptation *Rule* language, which defines the logic of the adaptation: which actions have to be executed and under which conditions.

At runtime, the architecture behaves as follows: each running SFE is provided with a Context Delegate (*CD* in Figure 1), which is able to supply the context data sensed by a given device. All the data collected by all Context Delegates is gathered by a *Context Manager*, which exposes an interface for querying and updating the different context entities, or for registering in order to asynchronously receive notifications on their state changes. The entities are described using another language, depicted as *Entity Description* in Figure 1. A module called *Rule Interpreter* registers itself to the *Context Manager* for receiving updates on the entities that are necessary for the execution of the specified *Rules*. Once an update is received, the *Rule Interpreter* selects the rules that, according to their definition, have to be executed in the current situation. Then, it notifies each different platform about the adaptation actions that need to be executed. Such actions may be very simple (e.g. changing a background colour or font) or very complex (changing the current interaction modality). Therefore, it is necessary that the actual way to execute these actions is delegated to each considered platform. Such delegation is implemented by the various *Platform Adapters*. Once the execution technique has been decided, the *Platform Adapter* sends the actual changes to the *Platform Generator*, which executes them on the running *Platform SFE*. In the following sections we describe a language for describing the SFE UIs, a language for defining adaptation rules and a language for interacting with the *Context Manager*.

FRONT-END DESCRIPTION LANGUAGE

In this section we describe the Advanced Service Front End Description Language (ASFE-DL), which is used in the Serenoa[1] EU FP7 project for describing SFEs. The language is compliant with the Cameleon Reference Framework [7]. Therefore, the modelling of a UI is defined through a set of levels of abstraction (Task & Concepts, Abstract User Interface, Concrete User Interface, Final User Interface).

Abstract-User Interface

The Abstract User Interface (corresponding to the Platform-Independent-Model – PIM in Model Driven Engineering) expresses the UI regarding its presentation units, independently of the interactors available and of the modality of interaction (graphical, vocal, haptic). A presentation unit groups a set of logically connected interactors.

[1] http://www.serenoa-fp7.eu/

Figure 2 shows the UML class diagram for the version of the ASFE-DL at the abstract level. We used different colours in order to highlight different parts of the metamodel: sky-blue for the main structure of the interface, green for the interactor hierarchy, red for the classes that model the relationships between interactors and yellow for the classes that model the UI behaviour.

The class that represents an abstract user interface model is called *AbstractUIModel*. It contains the entire specification of both the UI structure and behaviour. The interface contains an optional *DataModel*, which defines the data types manipulated by the user interface, which allow maintaining the state of the interaction with the user.

An *AbstractUIModel* consists of a composition of *AbstractInteractionUnits*. Each instance of this class represents a part of an application user interface that should be presented to the user at once. The first one among these *AbstractInteractionUnits* will be the starting point for interaction. It is possible to model the navigation among the different abstract interaction units defining instances of the *Connection* class. An *AbstractInteractionUnit* is composed of *AbstractInteractors* and *AbstractRelationships*. The abstract class *AbstractInteractor* defines a generic interactor, which represents a generic user interface object. Different categories of interactor exist, according to their semantics: *Selection*, which allows the user to select one (*SingleChoice*) or more options (*MultipleChoice*) from a predefined set of choices; *Edit*, which allows the user to manually modify an input value; *Only output*, which presents information to the user; *Control*, which allows the user to trigger actions.

The *AbstractReationship***,** the base class of all the relationships among abstract interactors. Different types of relationships are defined in ASFE-DL: *Grouping*, which represents a generic group of interactors; *Ordering***,** which represents an ordering relationship among a group of interactors; *Hierarchy*, which represents a hierarchical relationship among a group of interactors; *Repetition***,** which represents the template for a list of interactors that have to be repeated for each element of a dynamic list (e.g. a list of search results); *Dependency*, which defines a dependency relation between 1 interactor/interactor group and N interactors/interactor groups.

The behaviour part of the language is able to describe the to describe the dynamic changes in the UI according to the interaction with the user. The described concepts are shared with the Action part of the language for describing adaptation rules (see section 3) which express the changes that have to be applied to the UI (and also to other models) in order to react to context changes. In order to have a complete model for defining the dynamic behaviour, we introduced the classic set of *Statements* of the imperative programming languages. Therefore, an *EventHandler* is simply a *Block*, representing a collection of *Statement*s, which can be either basic or composite. The class names for the composite ones are self-explaining: *If, While, For, Foreach*. The possible types of basic actions (represented by the abstract class *Action*) are defined by the four classic functions of the persistent storage: *Create, Read, Update* and *Delete*, which work on all the model elements at runtime. In addition, we introduced a statement that allows the UI to invoke external functionalities, declared through the *ExternalFunction* instances.

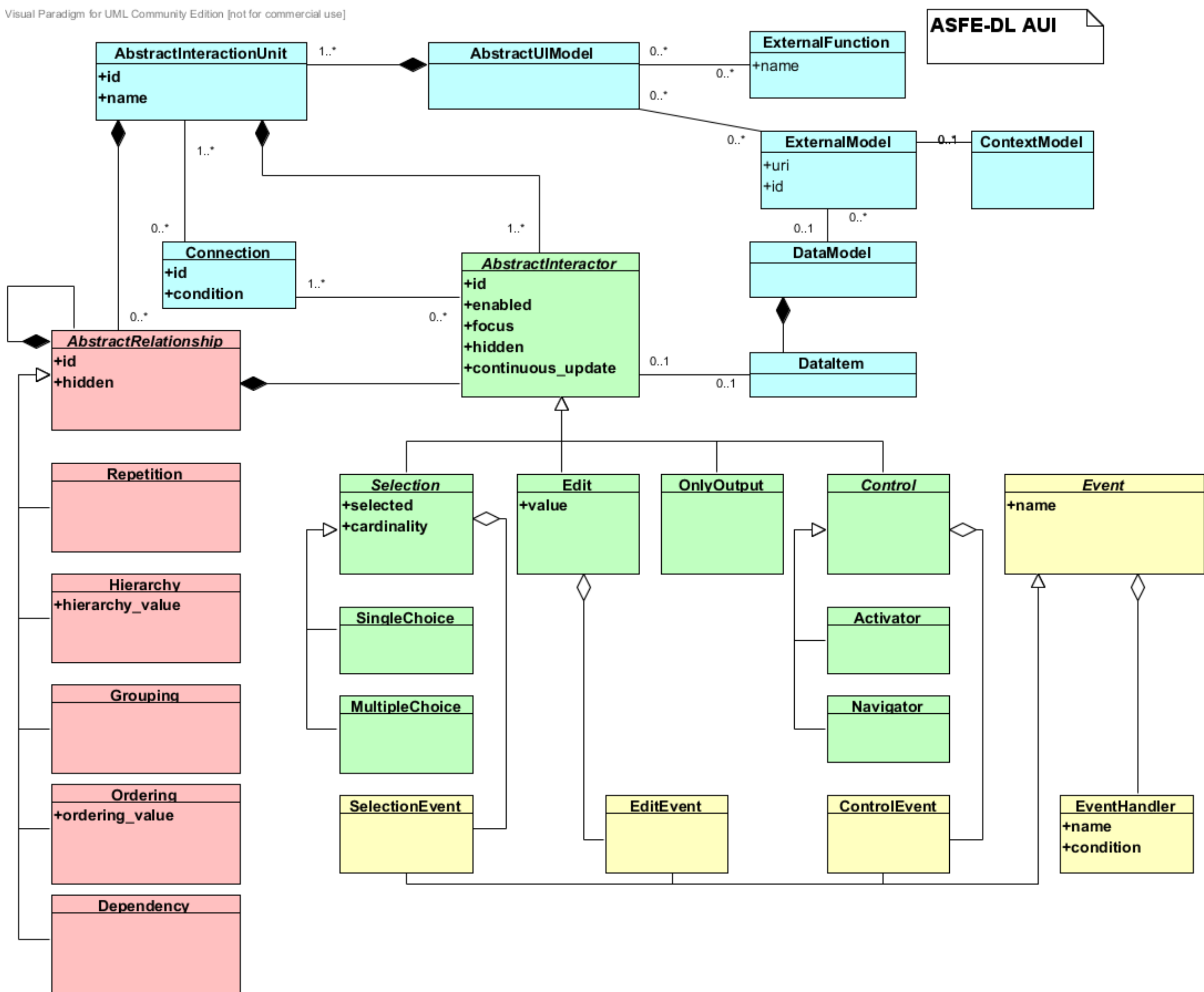

Figure 2: ASFE-DL UML diagram (abstract level)

Concrete Desktop User Interface

The definition of a Concrete Language according to [7] consists mainly of the refinement of the abstract classes with the elements that can be used in the specific platform in order to build the UI, which is depicted in Figure 3, the concrete refinements are highlighted in dark blue. Different style properties can be associated to the different concrete interactors, which can be grouped in the following categories: borders, position, display (height, width etc.), margins, padding, text formatting, list formatting and table formatting.

The input devices provided by the Desktop platform are the pointer and the keyboard. Each model element may receive notifications about the status of pointer position and keyboard or mouse buttons. Therefore, in the ASFE-DL Concrete Desktop model we introduce two classes that represent such events: the *MouseEvent* (enter, over, leave, click, double-click, down, up) and the *KeyboardEvent* (down, up, pressed).

The concrete refinements for the *Activator* and the *Navigator* class are the *Button*, *Link* and *Image Map*. While the first two elements contain a combination of *Text* and *Images* that provide their label, the *Image Map* contains an image and a set of polygons which represent the clickable areas.

The *Edit* interactor can be refined into *Text Field* (a field for entering text on a single row), *Text Area* (a field for entering text on multiple lines), *Spin Box* (a field for entering numerical values buttons for increasing and decreasing the values), *Track Bar* (a bar with a draggable knob for entering values in a given range) and *Map* (a map control for entering positions). All of them may contain a list of texts and images that represents the element labels.

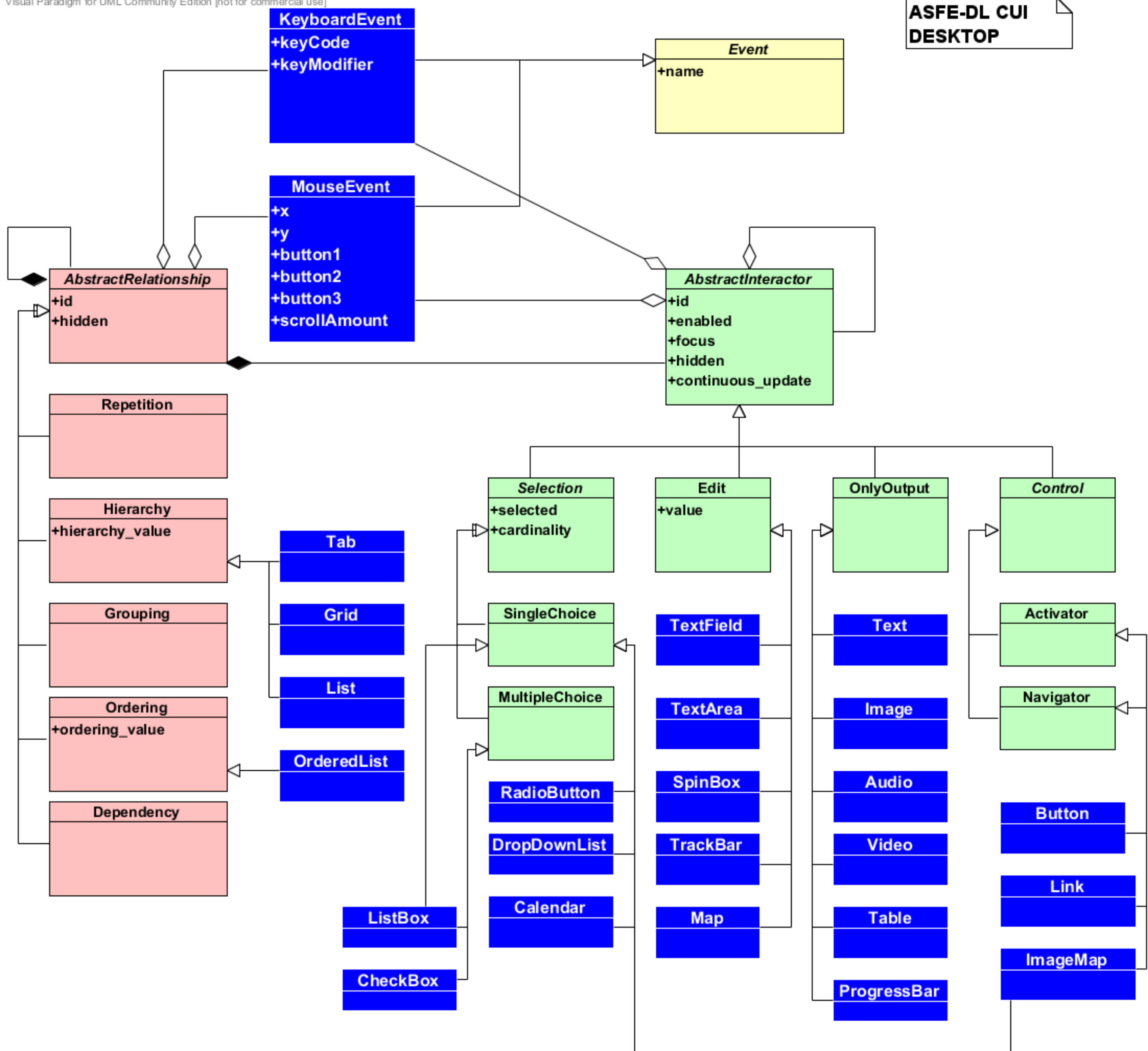

Figure 3: ASFE-DL UML class diagram (concrete desktop level)

The *Only Output* interactor can be refined into a *Text* (which represent a textual content provided by the content attribute), an *Image* (which represent an image content) *Table* (which represent a set of contents displayed in a tabular form), an *Audio* (which represents an audio content, together with the controls for playing, pausing, stopping and change the position in the stream), a *Video* (which represents a video content, again together with the controls for playing, pausing, stopping and to change the position in the stream), or a *Progress Bar* (which represents a bar that can be displayed in order to show the progress of an operation that may take a long time).

The *Single Choice* interactor (*Selection*) can be refined into a *Radio Button* (which shows the choice elements together with an associated button for the selection), a *List Box* (which shows the choice elements grouped inside a box where it is possible to highlight one of them), a *Drop Down List* (which shows the currently selected object together with a button that allows the visualization of the other options), an *Image Map* (which allows the graphical selection of an area in an image), or a *Calendar* (which allows the selection of a date).

The *Multiple Choice* interactor (*Selection*) may be refined into a *Check Box* (which displays the choice elements

together with a check for selecting them) or *List Box* (this time with the possibility to highlight more than one option).

With respect to the *Abstract Relationships,* some classes from the abstract level can be directly used also in the concrete desktop platform (e.g., *Hierarchy, Repetition, Dependency*). Other may be used either directly or through their refinements. The *Grouping* class can be directly used or, in addition, it is possible to use the *Grid*, the *Tab* and the *List* refinements that specify a different layout for the group. The *Grid* allows the designer to display the different contents in a table-like layout, without using a table. Therefore, in the grid specification, there are no headings for the rows and columns. The *Tab* element allows to display only one of the different *TabElement* contents at a given time. The *List* displays its inner contents using a bullet list. Also the *Ordering* Relation can be used directly or through its *OrderedList* subclass, which represents an ordered list of items.

ADAPTATION RULE LANGUAGE

In this section, we describe the Advanced Adaptation Logic Description Language (AAL-DL), which is used in the Serenoa EU FP7 project for describing when and how the applications should react to the context changes. Such adaptation logic should define the transformations affecting the interactive application when some specific situations occur both at the context level (e.g. an entity of the context changes its state), and in the interactive application (e.g. an UI event is triggered).

In Figure 4 there is the UML class diagram representing the main classes, interrelationships, and attributes of the AAL-DL. At the highest level we have the element *RuleModel*, which will contain a number of *Rules*, each one consisting of a triplet <*Event, Condition, Action*>. In addition, a rule could have an optional priority attribute (modelled through an integer value), to identify the rule which is the most likely to be triggered when multiple, conflicting rules occur simultaneously.

The *Event* part of a rule describes the event whose occurrence triggers the evaluation of the rule. It could specify elementary events occurring in the interactive application, or even a composition of events. The *agent* (user, platform or environment) responsible for the activation of the event can be optionally specified. The *Condition* of a rule is represented by a Boolean condition that has to be satisfied in order to execute the associated rule action(s). The *Action* part of the rule, as mentioned in the previous section, is shared with the ASFE-DL language. This part allows to define which changes should be performed at runtime in order to react to the context change. It is worth pointing out that the actions may be expressed either at the abstract or at the concrete level. This means that it is possible to concentrate the adaptation behaviour, which is shared between all the different platforms into a single rule expressed at the abstract level. When the *Rule Interpreter* module in Figure 1 selects a rule that contains abstract actions, forwards it to all the *Platform Adapters*, which are able to interpret the action part binding the abstract interactors with their concrete counterparts.

CONTEXT MANAGEMENT LANGUAGE

Context information refers to the situation where the application is operating: user-related parameters (e.g., geographical position), capabilities of the software/hardware, environment light, etc., are considered to be part of the context of use.

Among the benefits of context-aware applications, there is the possibility of retrieving information based on the user's situation and of adapting the system behaviour. In general, context-awareness leads to an optimal utilization of the available resources.

Exploiting context information, however, can be problematic due to some issues. Firstly, context information may change dynamically and thus needs to be monitored in real time. In addition, information often needs to be abstracted from data coming from several sensors, each of which has its own peculiarities

In general terms, context management is responsible for the collection and processing of the information that characterizes the environment defined by the interaction between the user, the services and the devices. The context information is collected from various sources and should be represented formally in accordance with ontological resources described in order to be effectively understood and used for reasoning. High variability of contexts must be considered as these are changing over time. In this regard, in the Serenoa FP7 EU project, the *Context Manager* in Figure 1 provides the functionality for storing the various context entities into a centralised repository, where all the SFE may add, modify or delete information. Subscription for receiving asynchronous notifications about a value change for the specified context entity is also possible. Therefore, the context management language contains elements that allows different applications through different communication protocols to:

- Insert an entity, i.e. adding it on the context management core. A type and a data field must be specified in order to create the entity. The context manager returns the entity identifier.
- Query an entity, i.e. asking the context management core for the current definition of the entity. It returns the description of the entity.
- Query all entities, i.e. getting the list of stored entities.
- Update an entity with the specified data. This method merges the data of the entity with the provided data.
- Delete an entity..
- Reset an entity, i.e. clearing its data field.

- Subscribe for an entity update, specifying the entity identifier and the network address for receiving the notifications. This operation creates a subscription on entity updates for the specified network address (IP-port). After the subscription, a message is sent back to the subscriber, for every modification of the entity (update, reset, delete), that specifies the current entity description.
- Unsubscribe from an entity update, which removes a previously created subscription.

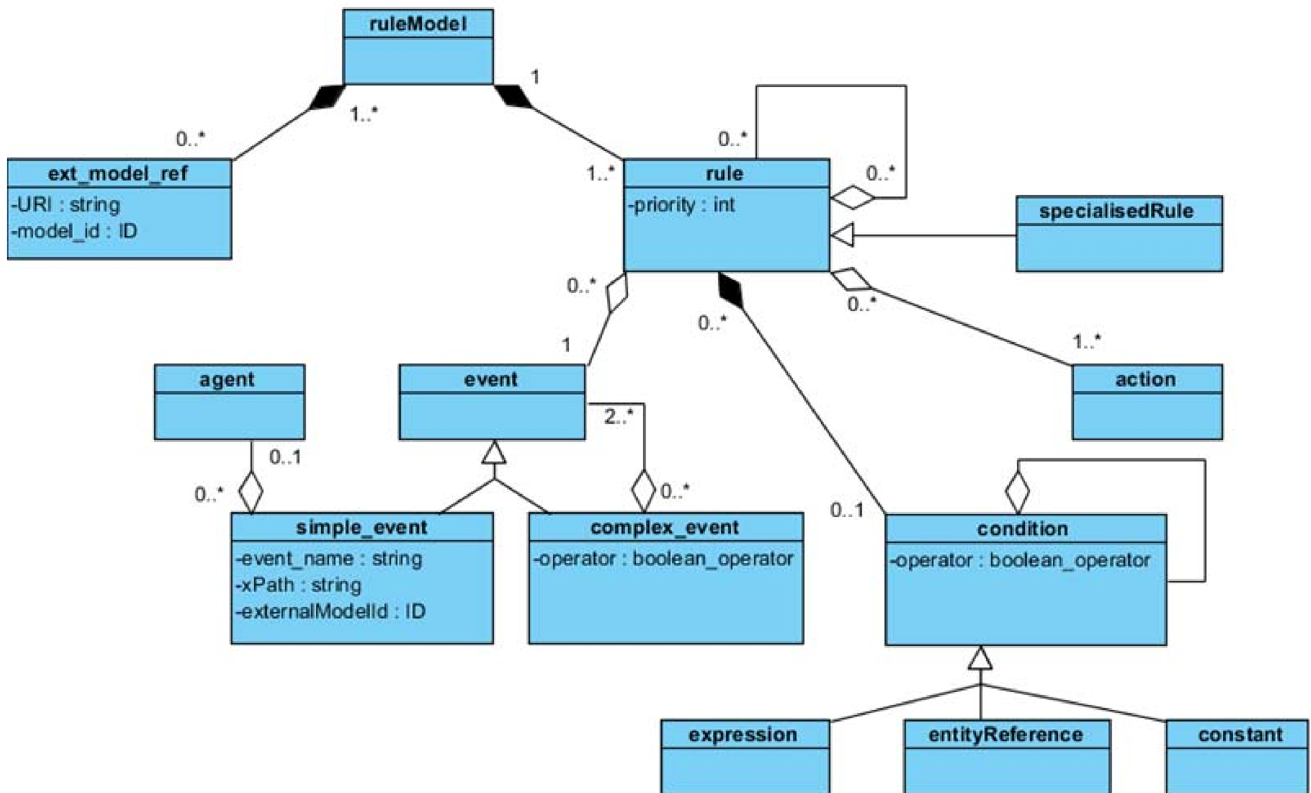

Figure 4: AAL-DL UML class diagram

CONCLUSION AND FUTURE WORK

In this paper we described an architecture for creating Service Front Ends that are able to adapt to different context of use. We described a set of modules that manage the different aspects of adaptation, such as the design-time definition of the initial application configuration, the definition of the adaptation logic, the adaptation decision making process, and the execution of the adaptation logic on different platform. In addition, we described a set of languages that can be used for i) defining the User Interface structure ii) describing when the adaptation should take place and what are its effects and iii) interconnecting different context sensing devices and applications in order to create a shared representation of the context of use.

In the future, we will extend the UI description language in order to cover more platforms. In addition, we will consider the inclusion in the AAL-DL of declarative rules the may be used for maintaining the consistency of the adaptation across various platform. Indeed, such kind of rules are suitable in order to express relationships that should hold when passing from a level to another.

REFERENCES

1. Feuerstack S., Pizzolato, E.; Building Multimodal Interfaces out of Executable, Model-based Interactors

and Mappings; HCI International 2011; 14th International Conference on Human-Computer Interaction; J.A. Jacko (Ed.): Human-Computer Interaction, Part I, HCII 2011, LNCS 6761, pp. 221-228. Springer, Heidelberg (2011), 9-14 July 2011, Orlando, Florida, USA.

2. Jouault, F. and Kurtev, I. Transforming Models with ATL. Lecture Notes in Computer Science, 2006, Springer, Volume 3844/2006, 128--138
3. Limbourg, Q., Vanderdonckt, Q., Michotte, B., Bouillon, L., López-Jaquero, V.: USIXML: A Language Supporting Multi-path Development of User Interfaces. EHCI/DS-VIS 2004: 200-220
4. Paternò, F., Santoro C., Spano, L.D., MARIA: A Universal Language for Service-Oriented Applications in Ubiquitous Environment", ACM Transactions on Computer-Human Interaction, Vol.16, N.4, November 2009, pp.19:1-19:30, ACM Press.
5. Salber, D., Anind, D., Abowd, G. 1999. The context toolkit: Aiding the development of context-enabled applications. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. ACM (1999), pp. 434–441
6. Seissler, M., Breiner, K., Meixner, G.: Towards Pattern-Driven Engineering of Run-Time Adaptive User Interfaces for Smart Production Environments. Proceedings of the 14th International Conference on Human-Computer Interaction. Springer (2011)
7. The CAMELEON Reference Framework, G. Calvary, J. Coutaz, D. Thevenin, L. Bouillon, M. Florins, Q. Limbourg, N. Souchon, J. Vanderdonckt, L.Marucci, F.Paternò, and C.Santoro, CAMELEON Project, September 2002.
8. XSL Transformations (XSLT) Version 2.0. W3C recommendation, W3C, Feb. 2007. http://www.w3.org/TR/xslt20/
9. Octavia, J.; Vanacken, L.; Raymaekers, C.; Coninx, K.; Flerackers, E., Facilitating Adaptation in Virtual Environments Using a Context-Aware Model-Based Design Process, Proceedings TAMODIA 2009, LNCS 5963, pp.58-71.
10. Sottet, J.S., Ganneau, V., Calvary, G., Coutaz, J., Demeure, A., Favre, J.M., Demumieux R.: Model-Driven Adaptation for Plastic User Interfaces. INTERACT (1) 2007: 397-410