Multi-Dimensional Context-Aware Adaptation of Service Front-Ends

Project no. FP7 – ICT – 258030

Deliverable 3.3.2
AAL-DL: Semantics, Syntaxes and Stylistics (R2)

Due date of deliverable: 30/09/2013

Actual submission to EC date: 30/09/2013

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)		
Dissemination level		
[PU]	**[Public]**	**Yes**

Document Information	
Lead Contractor	ISTI-CNR
Editor	Fabio Paternò, Carmen Santoro
Revision	Cristina González Cachón , Javier Rodríguez Escolar (CTIC)
Reviewer 1	
Reviewer 2	
Approved by	
Project Officer	

Contributors	
Partner	**Contributors**

Changes			
Version	**Date**	**Author**	**Comments**
1	05/06/2013	ISTI-CNR Carmen Santoro, Fabio Paternò	Table of Content
2	20/06/2013	ISTI-CNR Carmen Santoro, Fabio Paternò, Marco Manca	First version
3	2/07/2013	ISTI-CNR Marco Manca, Fabio Paternò, Carmen Santoro,	Revised Version
4	09/07/2013	CTIC-CT Cristina González Cachón, Javier Rodríguez Escolar	Revised Version
5	17/07/2013	CNR-ISTI Fabio Paternò, Carmen Santoro Marco Manca	Final Version

Executive Summary

The main goal of this deliverable is to present the final version of the AAL-DL, the high-level description language defined in SERENOA and intended to express declaratively Advanced Adaptation Logic. Such version is based on the extensive use done in the project of this language to specify adaptation rules.

Table of Contents

1 Introduction

1.1 Objectives

The objective of this document is to describe the last version of the AAL-DL language based on the experiences carried out in the project.

1.2 Audience

This document has a public dissemination level, so it is open to public consultation by the general public. However, a key audience is represented by project reviewers and the officer, as well as any researcher/scientist who could be interested in the topics addressed by SERENOA.

1.3 Related Documents and their Relationships with the Current Document

- **D3.3.1** (*AAL-DL: Semantics, Syntaxes and Stylistics (R1)*)*:* Deliverable D3.3.1 presents the first version of AAL-DL language (AAL-DL 1.0). The current document (D3.3.2) presents the latest updates done to the AAL-DL language, therefore, it mainly complements D3.3.1.
- **D1.1.2** (*Requirements Analysis (R2)*): Deliverable D1.1.2 provides the requirements for the various modules of the SERENOA architecture, including the AAL-DL. D1.1.2 also gives an overview of the current state of the implementation of requirements for the AAL-DL.
- **D1.2.1** (*Architectural Specifications (R1)*): Deliverable D1.2.1 provides an overview of the architecture of the platform that is currently being considered in SERENOA: D1.2.1 document describes how the AAL-DL fits this architecture and its role.
- **D3.2.1** (*ASFE-DL: Semantics, Syntaxes and Stylistics (R1)*): Deliverable D3.2.1 describes the language for specifying the interactive application (yet currently focusing only onthe abstract level). References to this D3.2.1 deliverable (and also to further releases of the ASFE-DL) are necessary so that the two description languages (AAL-DL and ASFE-DL) grow in parallel and share as much semantic and syntactic features as possible.

1.4 Organization of This Document

This deliverable is organised in the following way: Section 2 describes the AAL-DL 2.0 after analysing some related work that provided useful hints for updating the language. Section 3 provides relevant modelling examples exploiting the new version of the language, Section 4 summarises conclusions and provides possible indications for future work.

2 Advanced Adaptation Logic Description Language (AAL-DL)

2.1 Related Work

In this section, we discuss a number of contributions that we analysed in order to gather hints for extending the ECA (Event-Condition-Action)-based SERENOA AAL-DL language. It is worth pointing out that ECA rules have been used in many settings, including active databases, workflow management and specifying business processes. Therefore, when analysing such works, we mainly focused on those aspects that could be relevant also for supporting UI adaptation.

(Jung et al., 2007) presents an XML-based ECA -based framework for coordination of active, autonomous devices interacting with each other through Web Services (WSs). In authors' view, an ECA rule, when triggered by an internal/external event to the device, can result in the invocation of appropriate WSs. The proposed language for rule description, named WS-ECA, consists of events, conditions, and actions. There are four types of events: (i) *internal event* if it is generated by the internal components of a device, (ii) *external event* if it is delivered from other devices, (iii) *time event* when the timer of a device reaches some specific point in time; time events are further classified into: *absolute* (occurs once), *periodic* (occurs periodically), and *relative* (defined in relation with some other event by use of 'before' and 'after' operators) events. Finally, the (iv) *service event* is generated when invoking a service of a device; it can be either before or after. The service event of before (resp.: after) type is generated before (resp.: after) a specified device service starts (resp.: finishes). The condition is a Boolean expression that must be satisfied in order to enable some action of a device. Finally, the action represents an instruction carried out by a device, which includes primitive actions such as web service invocation and event generation. Apart from the primitive events introduced above, WS-ECA also supports specification of composite events based on them by using the following logical operators: i)*Disjunction* (e1, e2. . . en): The composite event of type ''OR'' has more than one sub-event, and it requires that at least one of the sub-events must occur during some specific time interval; ii)*Conjunction*(e1,e2,... en): The composite event of type ''AND'' has more than one sub-event, and it requires that all of the sub-events must occur during some specific time interval; iii)*Serialization* (e1;e2;. . .;en): The composite event of type ''SEQ'' has more than one sub-event, and it requires that all of the sub-events must occur sequentially during some specific time interval; iv)*Negation* (e): The composite event of type ''NOT'' has only one sub-event, and it requires that the sub-event must not occur during some specific time interval. Finally, it is worth pointing out that in this approach event composition can be done recursively to represent complex event structures.

In (Daniel et al. 2007, Daniel et al. 2008) authors address the problem of adaptive Web applications by illustrating an ECA rule-based approach intended to facilitate the management and evolution of adaptive application features. They present ECA-Web, an XML-based language for the specification of active rules expected to manage adaptivity in Web applications. The syntax of the language is inspired by another rule language for the specification of exceptions in workflow management systems. A typical ECA-Web rule is composed of five parts: scope, events, conditions, action and priority. The scope defines the binding of the rule with individual hypertext elements (e.g. pages, links inside pages). The definition of a scope allows one to couple the respective adaptivity logic with individual hypertext elements. If the definition of the scope is omitted, the rule has a global scope, i.e., the rule is applied to all the pages of the application. When it is specified for a rule, the definition of the <scope> element requires the indication of the hypertext elements the rule must be applied to. The scope can bind a rule also to multiple pages (e.g. by listing the pages that are part of the scope). Through events it is specified how the rule is triggered. In the condition part it is possible to evaluate the state of the application to decide whether the action has to be executed. The action specifies the adaptation of the application in response to a triggered event and a true condition. The priority defines an execution order for rules concurrently activated over the same scope; if not specified, a default priority value is assigned. ECA-Web provides support for the following event types: i)*Data events* refer to operations on the application's data source, such as create, modify, and delete; ii)*Web events* refer to general browsing activities (e.g. the access to a page, the submission of a form, the refresh of a page, the download of a resource), or to events generated by the Web application itself (e.g. the start/end of an operation, login/logout of the user); iii) *External events* can be configured in form of a Web service that can be called by whatever application or resource from the Web. An external event could be for example a notification of

news fed into the application via RSS; iv)*Temporal events* are divided into instant, periodic, and interval events. Interval events allow the binding of a time interval to another event (anchor event), e.g. “five minutes after the access to page X”.

In (Maatjes, 2007) the author presents how a Model-Driven Architecture approach can be used to develop a mapping between ECA-DL and the Jess language, and how to set up an automated transformation from ECA-DL rules to Jess rules, using such a mapping. A simple ECA-DL rule has a classical structure: Upon <event> When <condition> Do <action>. However, besides this basic syntax, ECA-DL supports other constructs. One is <Lifetime>, which can be used to indicate the lifetime of the rule. The following lifetimes are supported: i)always => activate the rule and keep it active; ii)once => activate the rule and deactivate the rule after it has been used; iii)<n> times => activate the rule deactivate it after it has been used <n> times; iv)from <start> to <end> => activate the rule at <start> and deactivate it at <end>. <start> and <end> are moments in time, e.g. “May 1st, 2007”; v)to <end> => activate the rule, and deactivate the rule at <end>; vi)frequency <n> times per <period> => activate the rule, and keep it active as long as it has been used fewer than <n> times during <period>. Deactivate and keep the rule inactive as long as it has been used <n> times during <period>.

(Bry and Eckert, 2007) investigates issues of relevance in designing high-level languages dedicated to reactivity on the Web. It presents twelve theses on features desirable for a language of reactive rules suitable for programming Web and Semantic Web applications. One of the theses is that development and maintenance of reactive rule programs can be considerably supported by exploiting structuring mechanisms such as branching in rules. Indeed, it is more convenient to write one rule “on E if C do A1 else A2” than writing two rules “on E if C do A1” and “on E if ¬C do A2”. Rules of this kind are sometimes called ECAA rules (since they specify an action and an alternative action), and there are also more general forms such as E Cn An rules [14], which specify several condition-action pairs. This kind of rules are not only more convenient to write, but also are easier to maintain because replication (of C in this example) is avoided. Avoiding replication is also good for execution efficiency: the condition C is only tested once in an ECAA rule. Another thesis is about the need of procedural abstractions for actions: often several rules will share the same action(s). The reaction can be rather complicated and composed of many smaller actions. A procedure mechanism, where the action is specified once and given a name, is clearly a better approach than writing the same code in several rules. It is worth pointing out that in SERENOA, we judged useful to add both the latter two aspects (an else branch has been added to the specification of a rule, and a procedural mechanism has also been added) within the new version of the AAL-DL.

In (Kantere et al., 2007) authors describe a mechanism based on distributed Event-Condition-Action (ECA) rules that supports data coordination in a multi-database setting. Like other ECA languages, the proposed ECA rule language has three parts: an event language, a condition language and an action language. The event language provides a set of operators with formal semantics for a multi-database environment and which allows a wide variety of composite events. The condition language provides Boolean algebraic operators that take as operands either composite or simple conditions. The action language provides a conjunction of simple or composite actions. Furthermore, the event language that authors propose provides a set of operators with clear semantics for a multi-database environment, which allow a wide variety of composite events. A simple or primitive event can be either a database or a time event. A time event can be an absolute, a relevant or a periodic time event. A database event can be specified by one of the following four types of primitive database operations: retrieve, update, insert and delete. A composite event is an expression that is formed by applying the operators of the event algebra on simple or composite events. The operators of the proposed event algebra are shown in Figure 1.

Operator	Type	Function	Syntax
∧	Binary	Logical AND	<event1>∧<event2>
∨	Binary	Logical OR	<event1>∨<event2>
!	Unary	Logical NOT	!<event>
«	Binary	Loose Sequence	<event1> » <event2>
*	Unary	Zero or more occurrences	*<event>
+	Unary	One or more occurrences	+<event>
#	Binary	Exact number of occurrences	<number of occurrences>#<event>
&	Binary	Maximum number of occurrences	<number of occurrences>&<event>
$	Binary	Minimum number of occurrences	<number of occurrences>$<event>
>	Binary	Strict sequence	<loose sequence expression> > <not expression>

Figure 1: The operators of the event algebra proposed in (Kantere et al., 2007)

In (Ngeow et al., 2007) each JECA rule contains four components: i) Justification (J), which forms the reasoning context in which evaluation of the JECA rule has to be performed. It is frequently used as a disqualifier. For example, the rule is disqualified if J is evaluated to true, ii)Event (E): If event occurs, related JECA rules will be evaluated; iii)Condition (C): Condition is used as a logic constraint to be satisfied so that the action (A) in the rule can be executed if the rule is not disqualified; iv)Action (A): If the rule R is not disqualified, where an event occurs with condition (C) is met, and justification (J) is not satisfied, then actions (A) will be carried out.

In (Paschke, 2006) authors address how to correctly and efficiently capture and process the event-based behavioural, reactive logic represented as ECA rules in combination with other conditional decision logic which is represented as derivation rules. Thus, they elaborate on a homogeneous integration approach which combines derivation rules, reaction rules (ECA rules) and other rule types such as integrity constraint into the general framework of logic programming. A reactive rule in ECA-LP is formalized as an extended ECA rule, represented in the KB as a 6-ary: eca (T,E,C,A,P,EL), where T (time), E (event), C (condition), A (action), P (post condition), EL(se).

The work (Balme et al., 2005) focuses on context-aware adaptation rules, i.e. rules that drive adaptation depending on the context of use, by providing a meta-model of such adaptation rules expressed according to the ECA format, and defined according to a UML Class diagram. That work also provides a taxonomy of rules according to four criteria for characterizing rules for context-aware adaptation: nature, centrality, type and directivity: i)The nature of the rule differentiate between rules according to whether they consist in proposing, favouring, or prohibiting reactions; ii) The centrality of the rule distinguishes between rules whose application is necessary versus luxurious for the interaction; iii) The type of the rule characterizes the contribution of the rule whether it is functional or non-functional; iv) The directivity of the rule makes the difference between rules that set objectives versus rules that specify the physical actions to perform. A rule that prescribes a solution to master the change of context of use is said an evolution rule, whereas a transition rule aims at accompanying the user in the change when the evolution rule is applied. Figure 2 shows the meta-model developed for evolution rules in this approach.

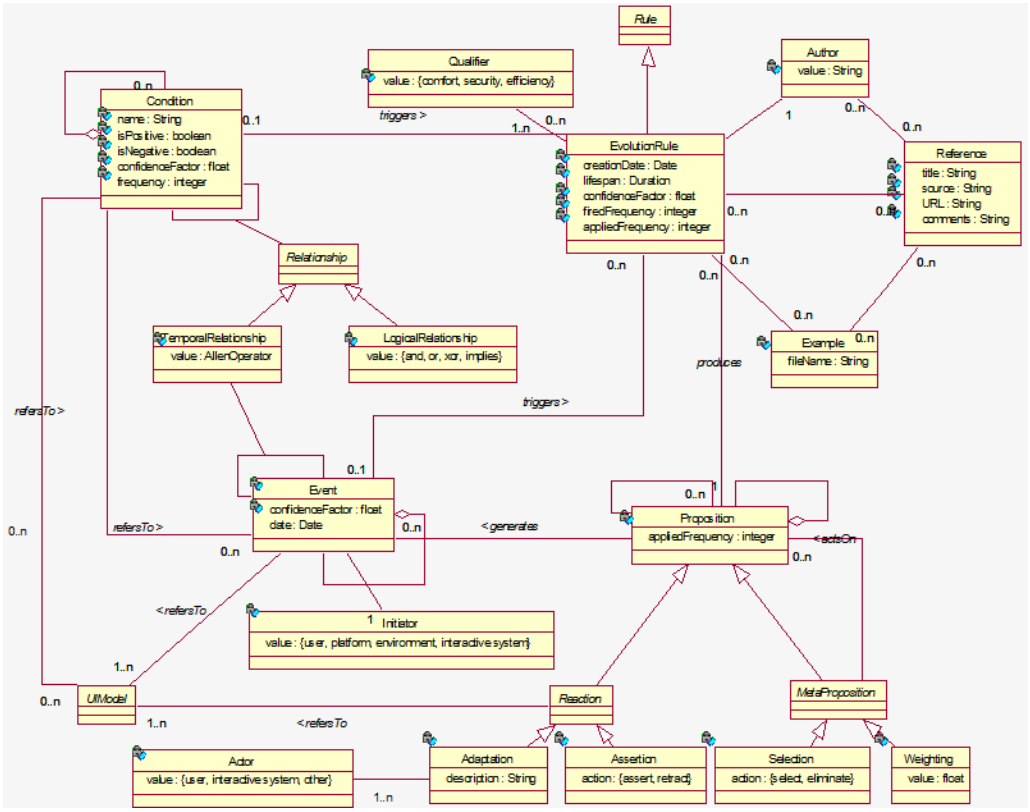

Figure 2: The meta-model for adaptation rules proposed in (Balme et al., 2005)

Relevant aspects to mention are: the inclusion of a qualifier in order to qualify each rule in terms of some aspect (e.g. comfort, security, efficiency), the fact that events are composed through some temporal relationships (e.g. through Allen operators), and the fact that each evolution rule can put in place a number of propositions (classified into reactions and meta-propositions). A proposition is a definition of a potential action or reaction that may occur in the future provided that some predefined conditions are satisfied. Since several propositions may occur simultaneously, a mechanism is introduced to identify the proposition which is the most likely to be triggered. Therefore, a proposition has two subclasses: a normal reaction and a meta-proposition, which is a proposition helping how to choose among propositions. Furthermore, a reaction could be of two types: *adaptation* or *assertion*, while the meta-proposition could be either *selection* or *weighting*.

In (Alferes et al., 2006), the authors introduce ERA, an ECA language based on and extending the framework of logic programs updates that also exhibits capabilities to integrate external updates and perform self-updates to its knowledge and behaviour. Among the relevant aspects, we mention the fact that ERA also includes inhibition rules of the form: When B Do not Action; where B is a conjunction of literals and events. Such an expression means: when B is true, do not execute Action. Inhibition rules are useful for updating the behaviour of reactive rules. If the inhibition rule above is asserted all the rules with Action in the head are updated with the extra condition that B must not be satisfied in order to execute Action. ERA language also allows combining basic events to obtain complex ones by an event algebra. The operators are: $\Delta \mid \nabla \mid A \mid$ not. More specifically, *e1* Δ *e2* occurs at an instant i if both e1 and e2 occur at i; *e1* ∇ *e2* occurs at instant i iff either e1 or e2 occur at instant i; *not e* occurs at instant i iff e does not occur i. A(e1, e2, e3) occurs at the same instant of e3, in case e1 occurred before, and e2 in the middle. This operator is very important since it allows to combine (and reason with) events occurring at different time points. Actions can also be basic or complex, and they may be internal actions or affect the external environment (external actions). Complex actions are obtained by applying algebraic operators on basic actions. Such operators are: $\triangleright \mid \mid \mid$ IF. The first is for executing actions sequentially and the second for executing them concurrently. Executing IF(C, a1, a2) means executing a1 in case C is true, or executing a2 otherwise.

In (Alferes et al., 2009) a general framework for reactive Event-Condition-Action rules in the Semantic Web is presented to deal with the heterogeneity of behaviour of the Semantic Web. Authors report on describing their language XChange, which is built on the idea that an expressive event query language must cover the

following four orthogonal dimensions: i) *Data extraction*: events contain data that is relevant to whether and how to react. The data can be structured as quite complex. The data of events can be used to construct new events or trigger reactions; ii) *Event composition*: to support composite events, i.e., events that are made up out of several events, event queries must support composition constructs such as the conjunction and disjunction of events; iii) *Temporal (and causal) relationships*: time plays an important role in event-driven applications. Event queries must be able to express temporal conditions such as "events A and B happen within 1 hour, and A happens before B." For some applications, it is also interesting to look at causal relationships, e.g., to express queries such as "events A and B happen, and A has caused B."; iv) *Event accumulation*: event queries must be able to accumulate events to support non-monotonic features such as negation (absence) of events, aggregation of data, or repetitive events. Furthermore, authors allow composite event specifications: given an event algebra, event algebra expressions form a tree term structure over atomic event specifications.

As a sample composite event language, a variant of the SNOOP event algebra (Chakravarthy and Mishra, 1994) was developed. In particular, (Chakravarthy and Mishra, 1994) discusses the SNOOP event specification language for active databases. They define an event, distinguish between events and conditions, classify events into a class hierarchy, identify primitive events, and introduce a small number of event operators for constructing composite (or complex) events. SNOOP supports temporal, explicit, and composite events in addition to the traditional database events. They recursively define a composite event expression, as an event expression formed by using a set of primitive event expressions, event operators, and composite event expressions constructed so far. The event operators identified in this work are: 1) *Disjunction* (v). Disjunction of two events E1 and E2 (E1 v E2) occurs when E1 or E2 occurs. This is useful when the occurrence of one or more (i.e. exclusive-Or) events out of a set of events may fire a rule. 2) *Sequence* (;). Sequence of two events E1 and E2 is a composite event, which is denoted El; E2, and occurs when E2 occurs provided E1 has already occurred. This implies that the time of occurrence of E1 is guaranteed to be less than the time of occurrence of E2. This constructor is useful when a predefined order has to be imposed over the occurrences of events. 3) *Conjunction* (Any, All). The conjunction event, denoted Any (I, El, E 2 , . . . En) where I < = n, occurs when I events out of the n events (corresponding to n distinct events specified) occur, ignoring the order of their occurrence. 4) *Aperiodic operators* (A, A*). The Aperiodic operator A allows one to express the occurrence of an aperiodic event bounded by two arbitrary events (for providing an interval). There are situations when a given event is signalled more than once during a given interval (e.g. within a transaction), but rather than firing the rule every time the event occurs, the rule has to be fired only once. To meet this requirement, they provide an operator A*(E1, E2, E3) that occurs only once when E3 occurs and accumulates the parameters for each occurrence of E2. 5)Periodic event operator (P, P*). They define a periodic event as an event E that repeats itself within a constant and finite amount of time. A periodic event can be represented by a triplet, consisting of an event El, the time period t after which a temporal event takes place and a terminating event E3 that marks the end of the periodic event.

2.2 From AAL-DL 1.0 to AAL-DL 2.0

In this section, the latest updates done to AAL-DL are described. In the following, for brevity, we refer to the specification of AAL-DL contained in Deliverable D3.3.1 as AAL-DL 1.0, while the one presented in this document will be referred to as AAL-DL 2.0.

In AAL-DL 2.0 two main types of changes have been done with respect to the previous language version: one mainly targeted the format of the language (now it does not only covers XML, but also JSON); another change is focused more on the semantic aspects of the language: we added/changed some features in order to improve the expressivity of the language.

2.2.1 The Support for JSON Format

A recent addition to the work done on AAL-DL was the possibility of having this language specified

according to JSON format[1]. This format is often used for serializing and transmitting structured data over a network connection (e.g. between a server and a web application), serving as an alternative to XML. Actually, JSON was originally designed for serializing/un-serializing data being sent to/from JavaScript applications, thus the advantages of JSON related to its over other means of serialization, e.g. XML. However, XML can be rather cumbersome as the sender must encode the data to be serialized based on a document type definition that the recipient needs to receive and decode: this creates some "extra padding" around the actual data regardless of the particular DTD or XSD used. So, the size of XML documents is often fairly large in comparison with the actual set of values they contain.

In comparison, the serialization of data using JSON by the sender is relatively quicker/more compact because the structure of JSON reflects that of standard programming data types and the encoding mechanism adds only the minimum information regarding contained data. Once the recipient receives the JSON data, the only processing needed is to evaluate the text of the string using e.g. JavaScript's built-in *eval* function or another equivalent function.

In SERENOA, this choice was done to make lighter the communications with some architectural SERENOA modules (e.g. the Adaptation Engine) which need to manage and exchange information on AAL-DL adaptation rules. In particular, at ISTI, the JSON specification of AAL-DL has been used together with two relevant libraries. The first one – *jsonschema2pojo* library[2]– is used to generate Java types/classes from JSON Schema. The second one –*jackson* library[3] – is used to serialise from Java objects to JSON. The specification of AAL-DL 2.0 in JSON format can be found in an appendix.

2.2.2 Latest Updates to the Language

After analysing some other ECA languages, we decided to introduce new features in the AAL-DL, in order to improve its flexibility and expressivity. The changes of AAL-DL2.0 with respect to AAL-DL 1.0 are:

- An *"else"* branch has been added to each ECA rule to better express the actions to do when the associated condition is not verified.
  It is worth pointing out that this change does not add much expressivity power to the language, but it aims at making the AAL-DL rules more structured and readable. Indeed, AAL-DL 1.0 can already express the semantics associated to the "else" branch: it would be sufficient to write two separate rules having as a condition part two opposite conditions (namely: the actual condition and its negation). However, in this way, it is lost the underlying logical organisation/structure, thus, this possibility is now included in AAL-DL 2.0.
- *New operators to compose events* are also added. In AAL-DL 1.0, it was possible to compose events only through Boolean operators: AND (two events both occur in any order), OR (at least one between two events occurs), XOR (only one between two events occurs), NOT (an event does not occur). However, we realised that further temporal combinations of events could be useful, for instance that two events (or two expression of events) occur one after the other (*sequence* operator), or that the same event occurs multiple times (namely: zero or more times, or one or more times, or at least a fixed number of times). For compositions of events, we also judged useful to specify (optionally) a *time interval* in which the involved events should occur.
- There are also *new types of actions*. It is possible to specify an *update action* also in "relative" terms. In AAL-DL 1.0, it was only possible to specify an update action in absolute terms (e.g. by specifying the new attribute values of an element). In AAL-DL 2.0, the possibility to specify an update action (targeted to a particular UI element) with respect to/in terms of another UI element is added. In particular, the operators that we have introduced are: *before* (e.g. to put an element before another element) and *after* (e.g. to put an element after another element).
- In the action part of a rule, the possibility to specify a *reference* to a sequence of actions (e.g. like a "procedure") that are defined somewhere else (e.g.: in another rule) has been added. This can be useful when the same set of actions that can be triggered in different rules appear: instead of

[1] http://tools.ietf.org/html/rfc4627
[2]: Available at: https://code.google.com/p/jsonschema2pojo/
[3]: Available at: http://jackson.codehaus.org/http://www.mkyong.com/java/how-to-convert-java-object-to-from-json-jackson/)

redundantly repeating the same set of actions in different points (with also the risk of introducing possible inconsistencies), this set of actions are specified – and named - in one rule and ,then, this name is referred in other rules without repeating its definition.

- An *alphabet of event types* is specified in order to better define the list of possible events that can be included in an adaptation rule.

In section 2.3.2, it is better defined how these changes have been actually implemented in the XML schema of the AAL-DL 2.0 language. In section 2.3.1, the meta-model of the AAL-DL 2.0 language is provided.

2.3 The AAL-DL 2.0

2.3.1 The Meta-Model

The meta-model of AAL-DL 2.0 is shown in Figure 3. As commented before, a rule model is composed of one or more rules. Each rule can have one event part, zero or one condition part, one or more actions, zero or more actions to carry out if the condition is not true ("else" action part). Each event part can be either a simple/elementary event, or a composition of events obtained by applying a 1-ary operator to a single event element or by applying a n-ary operator to multiple event elements. Composition of events could also (optionally) specify a time interval in which the involved events should occur. Conditions can be elementary (e.g. something like: environment_noise<50), or complex (e.g. something like (environment_noise<50)AND(environment_light=high)). The specification of elementary conditions involves the specification of at least two elements between the following ones: an entity reference (e.g. the reference to a variable), a constant (to specify a constant value), and an expression (to be used when the entity referred by the condition is not directly available but it is the result of a calculation through some operations e.g. +, -, *, /, .. used within expressions). The specification of complex conditions involves the use of Boolean operators like AND, OR, XOR. The operators to be used within the specification of conditions have been modelled through a suitable attribute ("BooleanOp" in case of complex conditions; "ComparisonOp" in case of elementary condition).

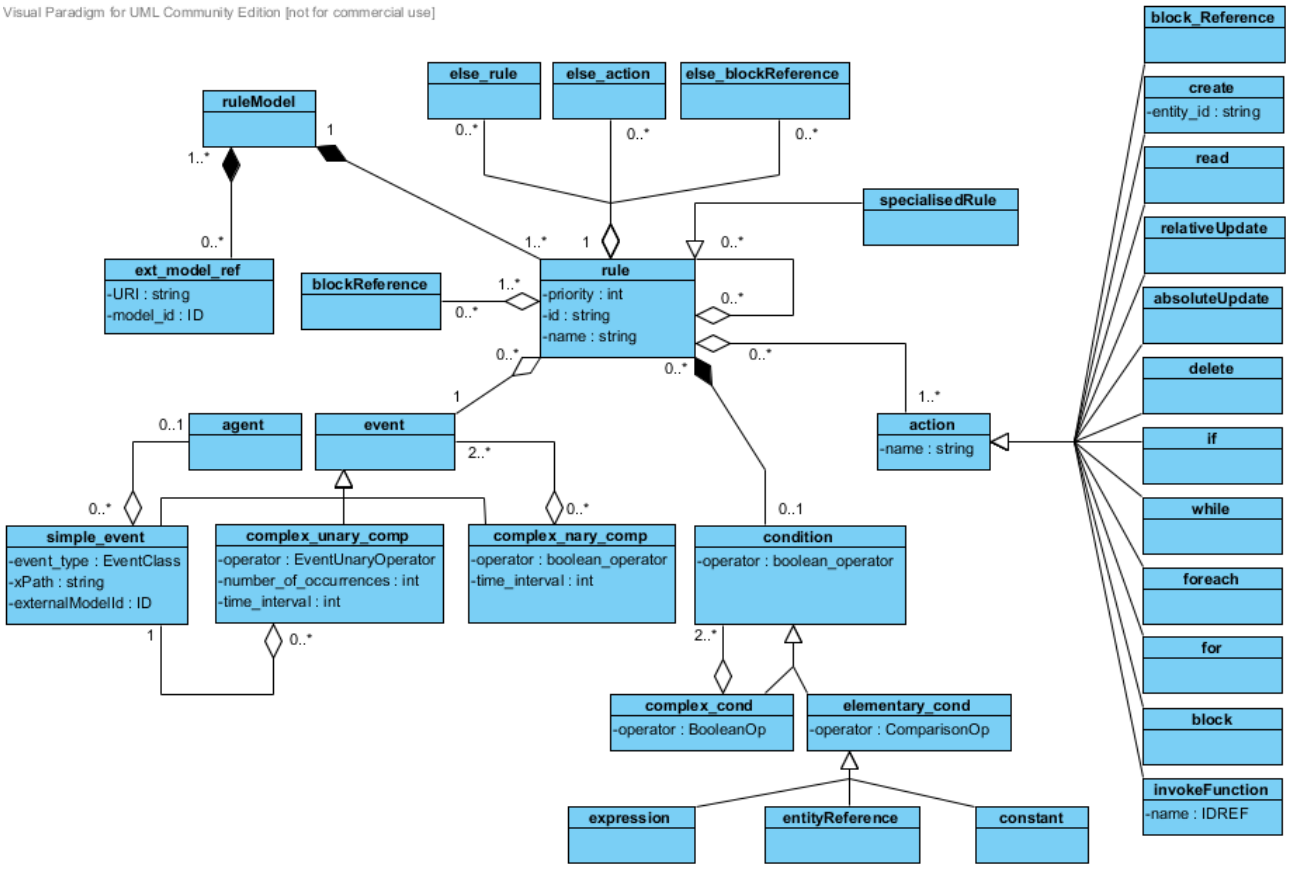

Figure 3: The main elements of the meta-model of AAL-DL 2.0

A rule can specify one or more actions. We have identified a series of possibilities for such actions: create, read, delete, if, while, for each, for, block, invoke a function, absolute update, relative update (just introduced in AAL-DL 2.0, the possibility to specify the update of an element in terms of another element). However, apart from activating one of such actions, additional possibilities are allowed to specify what the rule should do: either activate other rules (see the self-association on the *rule* class), or refer to a block of actions that was defined and named somewhere else in the rules document (see the aggregation between *rule* and a *blockReference*).

2.3.2 The XML Schema

In this section, the additions/changes done to the language will be analysed with respect to its previous version. Thus, only some relevant excerpts of the XSD schema of the new AAL-DL 2.0 version will be described (the ones not described here are unchanged with respect to AAL-DL1.0).

In Figure 4, the new definition of *RuleType* type is shown. An *else* branch has been added (rows 61-65, Figure 4), specifying the actions to be done if *condition* does not hold. This branch is actually a choice between an *else_action* (to specify a single action), an *else_rule* (to trigger another rule), and an *else_blockReference* (to refer to a block of actions which has been defined and named elsewhere in the rules definition document).

```
39    <!-- Rules -->
40     <xs:complexType name="RuleType">
41         <xs:annotation>
42             <xs:documentation> [9 lines]
52         </xs:annotation>
53            <xs:sequence>
54                <xs:element name="event" type="EventType" minOccurs="1" maxOccurs="1"/>
55                <xs:element name="condition" type="ConditionType" minOccurs="0"/>
56                <xs:choice minOccurs="1" maxOccurs="unbounded">
57                    <xs:element name="action" type="BlockType"/>
58                    <xs:element name="rule" type="RuleType"/>
59                    <xs:element name="blockReference" type="NamedEntityReferenceType"/>
60                </xs:choice>
61                <xs:choice minOccurs="0" maxOccurs="unbounded">
62                    <xs:element name="else_action" type="BlockType"/>
63                    <xs:element name="else_rule" type="RuleType"/>
64                    <xs:element name="else_blockReference" type="NamedEntityReferenceType"/>
65                </xs:choice>
66            </xs:sequence>
67            <xs:attribute name="priority" type="xs:int" />
68            <xs:attribute name="id" type="xs:string" />
69            <xs:attribute name="name" type="xs:string" />
70    </xs:complexType>
```

Figure 4: The new definition of RuleType

The new definition of the elements of type Event (*EventType*) is shown in Figure 5. Events can be simple events (namely: atomic events) and complex events (composite expressions of events).

On the one hand, *SimpleEventType* defines the possible *elementary*/*atomic* events that can be referred within a rule. Such events can be either events of the UI (e.g. on_edit, on_select,..) or events of the context (e.g. noise_increased, light_decreased, user_position_changed, ..): the type *EventClass* (which defines the alphabet of possible events) has been specified in a separate XSD Schema, which is described in the Appendix.

On the other hand, composition of events are in turn divided into *ComplexUnaryEventType* (composition of events that result from applying 1-ary operators to events), and *ComplexNaryEventType* (resulting from applying n-ary operators to multiple events). The difference between the latter two consists in the type of the operators applied (n-ary or 1-ary) and, consequently, the number of the involved operands. Indeed, the operators of type *EventUnaryOperatorType* require only one element of class *EventType* (see row 118, Figure 5), while the n-ary operators (*EventNaryOperatorType*) require at least two operands (see row 137, Figure 5). In addition, in case of compositions of events, an optional *time_interval* in which the specified composition of events should occur can be specified. Finally, within the definition of

ComplexUnaryEventType it has also been included an attribute named *number_of_occurrences*, which is used in association with the event operator *at_least_occur* (see Figure 6), as it specifies the minimum number of times that an event must occur.

```
76    <!-- Simple or complex event -->
77    <xs:complexType name="EventType">
78        <xs:annotation> [4 lines]
83        <xs:choice>
84            <xs:element name="simple_event" type="SimpleEventType"/>
85            <xs:element name="complex_unary_comp_event" type="ComplexUnaryEventType"/>
86            <xs:element name="complex_nary_comp_event" type="ComplexNaryEventType"/>
87        </xs:choice>
88    </xs:complexType>
89
90    <!-- Simple event -->
91    <xs:complexType name="SimpleEventType">
92        <xs:annotation> [7 lines]
100       <xs:sequence minOccurs="0" maxOccurs="1">
101           <xs:element name="agent" type="EntityReferenceType"/>
102       </xs:sequence>
103       <xs:attribute name="event_type" type="EventClass" use="required"/>
104       <xs:attributeGroup ref="EntityReferenceAttributeGroup"/>
105   </xs:complexType>
106
107   <!-- Complex Expressions of Events composed through 1-ary Operators -->
108   <xs:complexType name="ComplexUnaryEventType">
109       <xs:annotation> [8 lines]
118       <xs:sequence minOccurs="1" maxOccurs="1">
119           <xs:element name="event" type="SimpleEventType" />
120       </xs:sequence>
121       <xs:attribute name="operator" type="EventUnaryOperatorType"/>
122       <xs:attribute name="number_of_occurrences" type="xs:int" use="optional"/>
123       <xs:attribute name="time_interval" type="xs:int" use="optional"/>
124   </xs:complexType>
125
126   <!-- Complex Expressions of Events composed through N-ary Operators -->
127   <xs:complexType name="ComplexNaryEventType">
128       <xs:annotation> [8 lines]
137       <xs:sequence minOccurs="2" maxOccurs="unbounded">
138           <xs:choice>
139               <xs:element name="simple_event" type="SimpleEventType"/>
140               <xs:element name="complex_unary_comp_event" type="ComplexUnaryEventType"/>
141               <xs:element name="complex_nary_comp_event" type="ComplexNaryEventType"/>
142           </xs:choice>
143       </xs:sequence>
144       <xs:attribute name="operator" type="EventNaryOperatorType"/>
145       <xs:attribute name="time_interval" type="xs:int" use="optional"/>
146   </xs:complexType>
```

Figure 5: The definition of *EventType* in AAL-DL 2.0

In Figure 6, it is shown the definition of the operators used to compose events. On the one hand, in rows 495-497 (Figure 6),the definition of *EventNaryOperatorType* is visualized, which is, basically, the union of *BooleanNaryOperatorType* (e.g. the classical Boolean operators like AND, OR, XOR), and *TemporalNaryOperatorType*. On the other hand, in rows 498-500 (Figure 6), it is specified thedefinition of *EventUnaryOperatorType*, which is basically the union of *BooleanUnaryOperatorType* (e.g. the Boolean operator NOT) and *TemporalUnaryOperatorType* (which includes the *sequence* operator, namely the possibility of composing two events in sequence). The *TemporalUnaryOperatorType* type includes an operator for identifying events occurring zero or more times (*zero_more_occur*), an operator for expressing the occurrence of an event one or more times (*one_more_occur*), an operator to express the occurrence of an event at least N times (*at_least_occur*) and an operator for identifying the first occurrence of an event from the beginning (*first_of*).

```
476    <xs:simpleType name="TemporalNaryOperatorType">
477        <xs:annotation> [2 lines]
480        <xs:restriction base="xs:string">
481            <xs:enumeration id="temp_operator_value_sequence" value="sequence" />
482        </xs:restriction>
483    </xs:simpleType>
484    <xs:simpleType name="TemporalUnaryOperatorType">
485        <xs:annotation> [2 lines]
488        <xs:restriction base="xs:string">
489            <xs:enumeration id="temp_operator_value_zero_more_occur" value="zero_more_occur" />
490            <xs:enumeration id="temp_operator_value_one_more_occur" value="one_more_occur" />
491            <xs:enumeration id="temp_operator_at_least_occur" value="at_least_occur" />
492            <xs:enumeration id="temp_operator_first_of" value="first_of" />
493        </xs:restriction>
494    </xs:simpleType>
495    <xs:simpleType name="EventNaryOperatorType">
496        <xs:union memberTypes="BooleanNaryOperatorType TemporalNaryOperatorType" />
497    </xs:simpleType>
498    <xs:simpleType name="EventUnaryOperatorType">
499        <xs:union memberTypes="BooleanUnaryOperatorType TemporalUnaryOperatorType" />
500    </xs:simpleType>
```

Figure 6: Event Operators of AAL-DL 2.0

Another modification regarded the possibility to specify a *relative update* action is done. The change in the language was to replace the original (single) update action by two subtypes (see rows 580-581 in Figure 7): *AbsoluteUpdateActionType* (which was the update action already contained in AAL-DL 1.0) and *RelativeUpdateActionType*, which was added in AAL-DL 2.0.

```
576    <xs:complexType name="BlockType">
577        <xs:choice minOccurs="1" maxOccurs="unbounded">
578            <xs:element name="create" type="CreateActionType"/>
579            <xs:element name="read" type="ReadActionType"/>
580            <xs:element name="absoluteupdate" type="AbsoluteUpdateActionType"/>
581            <xs:element name="relativeupdate" type="RelativeUpdateActionType"/>
582            <xs:element name="delete" type="DeleteActionType" />
583            <xs:element name="if" type="IfThenElseType"/>
584            <xs:element name="while" type="WhileType" />
585            <xs:element name="foreach" type="ForEachType"/>
586            <xs:element name="for" type="ForType"/>
587            <xs:element name="block" type="BlockType"/>
588            <xs:element name="invokeFunction" type="InvokeFunctionType"/>
589        </xs:choice>
590        <xs:attribute name="name" type="xs:string" />
591    </xs:complexType>
```

Figure 7: The new definition of *BlockType* (which includes now an action of type "*RelativeUpdateActionType*")

The definition of the type named *RelativeUpdateActionType* is shown in Figure 8. It includes the specification of mainly two elements: *EntityReferenceToUpdate* (which is the reference to the entity that has to be updated), *EntityReferenceToRefer* (which is the reference of the entity that should be referred to). In addition, the attribute *operator* identifies the type of relative update operator to apply (either *before* or *after*).

```
<!-- Relative Update Action -->
<xs:complexType name="RelativeUpdateActionType">
    <xs:annotation>
        <xs:documentation> [2 lines]
    </xs:annotation>
    <xs:sequence>
        <xs:element name="entityReferenceToUpdate" type="EntityReferenceType" minOccurs="1" maxOccurs="1">
            <xs:annotation>
                <xs:documentation> [2 lines]
            </xs:annotation>
        </xs:element>
        <xs:element name="entityReferenceToRefer" type="EntityReferenceType" minOccurs="1" maxOccurs="1">
            <xs:annotation>
                <xs:documentation> [2 lines]
            </xs:annotation>
        </xs:element>
    </xs:sequence>
    <xs:attribute name="operator" type="RelativeOperatorType"/>
</xs:complexType>
```

Figure 8: The specification of the type *RelativeUpdateActionType*

The specification of the type named *RelativeOperatorType* is shown in Figure 9.

```
<xs:simpleType name="RelativeOperatorType">
    <xs:annotation> [2 lines]
    <xs:restriction base="xs:string">
        <xs:enumeration id="relat_operator_value_before" value="before" />
        <xs:enumeration id="relat_operator_value_after" value="after" />
    </xs:restriction>
</xs:simpleType>
```

Figure 9: The specification of the type *RelativeOperatorType*

3 Modelling Examples

3.1 Example 1: Adaptation of a Multimodal UI in a noisy environment

The first example regards adaptation in multimodal user interfaces. In particular, Figure 10 shows an example rule related to environment noise. If there is a variation in the noise level (e.g. the noise goes beyond a certain threshold of decibels), then this rule is triggered and the presentation is adapted in a way that only graphical interaction is supported: the corresponding action updates the presentation CARE value (Coutaz et al., 1995) by setting the graphical assignment value to the output attribute (rows 13-17, Figure 10). It is worth pointing out that actually two rules have been used in this case: the first one is to specify the possibility that at the beginning the increase of noise is already beyond the threshold. The second rule detects a variation in the noise level between two consecutive events of type *noise_increased* in which the first one is *under* the threshold and the next one is *above* the threshold.

```
11  <rule>
12      <event>
13          <complex_unary_comp_event operator="first_of">
14              <event event_type="noise_increased" xPath="/context/environment/@noise_level" externalModelId="ctxModel"></event>
15          </complex_unary_comp_event>
16      </event>
17      <condition>
18          <elementary_condition operator="gt">
19              <entityReference xPath="/context/environment/@noise_level"/>
20              <constant value="25" type="integer"/>
21          </elementary_condition>
22      </condition>
23      <action>
24          <absoluteupdate>
25              <entityReference xPath="/interface/presentation/@output" externalModelId="cuiModel"/>
26              <value>
27                  <constant value="graphical_assignment" type="string"/>
28              </value>
29          </absoluteupdate>
30      </action>
31  </rule>
32
33  <rule>
34      <event>
35          <complex_nary_comp_event operator="sequence">
36                  <simple_event event_type="noise_increased" xPath="/context/environment/@noise_level" externalModelId="ctxModel"></simple_event>
37                  <simple_event event_type="noise_increased" xPath="/context/environment/@noise_level" externalModelId="ctxModel"></simple_event>
38          </complex_nary_comp_event>
39      </event>
40      <condition>
41          <complex_condition operator="and">
42                  <elementary_condition operator="lt">
43                      <entityReference xPath="/context/environment/@noise_increased"/>
44                      <constant value="25" type="integer"/>
45                  </elementary_condition>
46                  <elementary_condition operator="gt">
47                      <expression operator="next">
48                          <entityReference xPath="/context/environment/@noise_increased"/>
49                      </expression>
50                      <constant value="25" type="integer"/>
51                  </elementary_condition>
52              </complex_condition>
53      </condition>
54      <action>
55          <absoluteupdate>
56              <entityReference xPath="/interface/presentation/@output" externalModelId="cuiModel"/>
57              <value>
58                  <constant value="graphical_assignment" type="string"/>
59              </value>
60          </absoluteupdate>
61      </action>
62  </rule>
```

Figure 10: An example of adaptation rule for a multimodal UI

3.2 Example 2: Adaptation of a UI for a user with low vision

The following example of adaptation rule deals with accessibility issues (Minon et al.'13). In this case, the event is triggered when the page is rendered as shown in Figure 11 row 2. The condition to be verified is that the user suffers from low vision problems AND inside the presentation checks whether there is some text with font settings size less then 14px. In this case the action part of the rule will increase the font size of all the text elements inside the presentation with font size less than 14px.

```
1  <rule>
2      <event><simple_event event_name="render" xPath="/interface/presentation"/></event>
3      <condition operator="and">
4          <condition operator="lt">
5              <entityReference xPath="/*//font_settings/@size" externalModelId="cui"/>
6              <constant value="14" type="string" />
7          </condition>
8          <condition operator="eq">
9              <entityReference xPath="//context/user/disability/@blindness" externalModelId="ctx"/>
10             <constant value="low_vision" type="string" />
11         </condition>
12     </condition>
13     <action>
14         <foreach>
15             <in><entityReference xPath="//font_settings[@size]"/></in>
16             <do>
17                 <if>
18                     <condition operator="lt">
19                         <entityReference xPath="./@size" externalModelId="cui"/>
20                         <constant value="14" type="string" />
21                     </condition>
22                     <update>
23                         <entityReference xPath="./@size" externalModelId="cuiModel" />
24                         <value><constant value="14px" type="string" /></value>
25                     </update>
26                 </if>
27             </do>
28         </foreach>
29     </action>
30 </rule>
```

Figure 11: Example of an adaptation rule dealing with accessibility issues (user suffering low vision problems)

3.3 Example 3: Adaptation of UI depending on user activity

The following example regards the user activity. The event considered is onActivityChange, which is related to the change of the user activity type. If the user is walking or running (rule condition part) then the rule is triggered and each only graphical element in the UI becomes multimodal, which means with a graphical and a vocal part in order to better support the fast moving user.

```
<rule>
    <event><simple_event event_name="onActivityChange" xPath='/user/userActivity/@activityType"
                         externalModelId="ctx"/></event>
    <condition operator="or">
        <condition operator="eq">
            <entityReference xPath="/user/userActivity/@activityType" externalModelId="ctx"/>
            <constant value="walking" type="string"/>
        </condition>
        <condition operator="eq">
            <entityReference xPath="/user/userActivity/@activityType" externalModelId="ctx"/>
            <constant value="running" type="string"/>
        </condition>
    </condition>
    <action>
        <foreach>
            <in><entityReference xPath="//*" externalModelId='cui"/></in>
            <do>
                <if>
                    <condition operator="eq">
                        <entityReference xPath="./@input" externalModelId="cui" />
                        <constant value="graphical_assignment' type="string" />
                    </condition>
                    <then>
                        <update>
                            <entityReference xPath="./@input"
                                externalModelId="uiModel" />
                            <value><constant value="equivalent" type="string"/></value>
                        </update>
                    </then>
                    <elseIf>
                        <condition operator="eq">
                            <entityReference xPath="./@prompt' externalModelId="uiModel"/>
                            <constant value="graphical_assignment" type="string"/>
                        </condition>
                        <then>
                            <update>
                                <entityReference xPath="./@prompt"/>
                                <value><constant value="redundancy" type="string"/></value>
                            </update>
                        </then>
                    </elseIf>
                </if>
            </do>
        </foreach>
    </action>
</rule>
```

Figure 12: An example rule (related to user activity)

3.4 Example 4: Adaptation of UI when the user enters a noisy and bright environment

In the following example, we consider the situation when the user changes its current environment while accessing a Web application through the smartphone. In particular, the user moves from an indoor environment to an outdoor environment: this change is detected by two main events occurring in the current context: an increased noisy, and an increased light. In this case, the UI adapts in such a way to increase the contrast of the Web page (i.e. the background colour is set to white, while the colour of the text of the content inside the page is set to black) and videos are replaced by videos having subtitles (in order to solve the problem of the noisy environment). Figure 13 and Figure 14 show the adaptation rules that support this behaviour. In particular, it is worth noting that such scenario is specified by using i) the possibility in one rule to refer to a block of actions defined in another rule and ii)a composition of events. In particular, the adaptation rule shown in Figure 13 defines the block of actions to apply (and names it), while the adaptation rule shown in Figure 14, in its "action" part just refers to the name of that block (without redefining it).

```
<rule>
   <event> [ ... ]

   <condition> [ ... ]

   <action>
          <block name="block_of_actions_for_updating videos_and_contrast">
          <if>
               <condition>
                   <elementary_condition operator="gt">
                       <entityReference xPath="count(/interface//video)"
                           externalModelId="uiModel" />
                       <constant value="0" type="int" />
                   </elementary_condition>
               </condition>
               <then>
                   <foreach>
                       <in>
                           <entityReference xPath="/interface//video" />
                       </in>
                       <do>
                          <create entity_id="mySubtitledVideo">
                              <containingEntityReference xPath="/interface//./video" externalModelId="cuiModel"/>
                              <complexType xPath="description/subtitledVideo"/>
                          </create>
                           <absoluteupdate>
                               <entityReference xPath="/interface//subtitledVideo[@id='mySubtitledVideo']/@source" externalModelId="cuiModel"/>
                               <value>
                                  <!--Assuming that the video with subtiltles is allocated in the same
                                   path that video without subtitles and the its name is the same with the word
                                   'Subtitled' concatenated -->
                                   <entityReference xPath="concat(/interface//video/@name, 'Subtitled')" externalModelId="cuiModel"/>
                               </value>
                           </absoluteupdate>
                           <absoluteupdate>
                               <entityReference
                                   xPath="//interface//subtitledVideo[@id='mySubtitledVideo']/@name" externalModelId="cuiModel" />
                               <value>
                                   <entityReference xPath="concat(/interface//video/@name, ' subtitled')" externalModelId="cuiModel" />
                               </value>
                           </absoluteupdate>
                           <delete>
                               <entityReference xPath="/interface//video" externalModelId="cuiModel" />
                           </delete>
                       </do>
                   </foreach>
               </then>
          </if>
          <absoluteupdate>
               <entityReference
                   xPath="/interface/presentation/default_settings/@background_colour" externalModelId="cuiModel" />
               <value>
                   <constant value="white" type="string" />
               </value>
          </absoluteupdate>
          <absoluteupdate>
               <entityReference
                   xPath="/interface/presentation/default_settings/@text_colour" externalModelId="cuiModel" />
               <value>
                   <constant value="black" type="string" />
               </value>
          </absoluteupdate>
          </block>
   </action>
</rule>
```

(1)

Figure 13: An Adaptation rule which defines a block of adaptation actions – see (1)

```
<rule>
    <event>
        <complex_nary_comp_event operator="and">
                <simple_event event_type="noise_increased" xPath="/context/environment/@noise_level" externalModelId="ctxModel" />
                <simple_event event_type="light_increased" xPath="/context/environment/@light_level" externalModelId="ctxModel" />
        </complex_nary_comp_event>
    </event>
    <condition>
        <complex_condition operator="and">
                <elementary_condition operator="gt">
                    <entityReference xPath="/context/environment/@noise_level" externalModelId="ctxModel" />
                    <constant value="25" type="int" />
                </elementary_condition>
                <elementary_condition operator="eq">
                    <entityReference xPath="/context/environment/@light_level" externalModelId="ctxModel" />
                    <constant value="high" type="string" />
                </elementary_condition>
        </complex_condition>
    </condition>
    <blockReference xPath="block_of_actions_for_updating_videos_and_contrast"/>
</rule>
```

(2)

(3)

Figure 14: Example of a rule exploiting a composition of events (2) and a reference (3) to a block of actions defined in the rule shown in Figure 13

4 Conclusions

In this deliverable, the second version of the AAL Description Language for expressing adaptation rules is described. In particular, the main updates done to the language have been reported, namely, the availability of AAL-DL language also in JSON format and the inclusion of some new features in the language in order to improve its expressivity and readability: the possibility to compose events in more flexible manners, to specify a relative update action, to define an “else” branch in the adaptation rules, to define a block of action in one rule and refer to this block in another rule without repeating its definition and to specify an alphabet of possible events.

Some modelling examples are also provided to show the suitability of the language to specify relevant adaptation rules.

References

(Alferes et al., 2006) J. J. Alferes, F. Banti, A. Brogi, An Event-Condition-Action Logic Programming Language. JELIA 2006: 29-42.

(Alferes et al., 2009) J. J. Alferes, M. Eckert, W. May, Evolution and Reactivity in the Semantic Web. REWERSE 2009: 161-200.

(Balme et al., 2005) L. Balme, N. O. Bernsen, G. Calvary, B. Collignon, A. Coyette, J. Coutaz, A. Demeure, L. Dyckbaer, J.-M. Favre, M. Florins, V. Ganneau, B. Michotte, F. Paternò, C. Pribeanu, C. Santoro, J.-S. Sottet, A. Stanciulescu, J.Vanderdonckt, Repository of Design Knowledge: context-aware adaptation rules, SIMILAR Deliverable D19, November 2005.

(Bry and Eckert, 2007) F. Bry, M. Eckert, Twelve Theses on Reactive Rules for the Web. Event Processing 2007.

(Chakravarthy and Mishra, 1994) S. Chakravarthy, D. Mishra, Snoop: An Expressive Event Specification Language for Active Databases. Data Knowl. Eng. 14(1): 1-26 (1994).

(Coutaz et al., 1995) J. Coutaz, L. Nigay, D. Salber, A. Blandford, J. May, R. Young, 1995. Four Easy Pieces for Assessing the Usability of Multimodal Interaction: the CARE Properties. Proceedings INTERACT 1995, pp.115-120.

(Daniel et al., 2007) F. Daniel, M. Matera, A. Morandi, M. Mortari, G. Pozzi, Active Rules for Runtime Adaptivity Management. AEWSE 2007.

(Daniel et al., 2008) F. Daniel, M. Matera, G. Pozzi, Managing Runtime Adaptivity through Active Rules: the Bellerofonte Framework. J. Web Eng. 7(3): 179-199 (2008).

(Jung et al., 2007) J.-Y. Jung, J. Park, S.-K. Han, K. Lee, An ECA-based framework for decentralized coordination of ubiquitous web services. Information & Software Technology 49(11-12): 1141-1161 (2007).

(Kantere et al., 2007) V. Kantere, I. Kiringa, J. Mylopoulos: Supporting Distributed Event-Condition-Action Rules in a Multidatabase Environment. Int. J. Cooperative Inf. Syst. 16(3/4): 467-506 (2007).

(Minon et al., 2013) R.Minon, F.Paternò, M.Arrue, An Environment for Designing and Sharing Adaptation Rules for Accessible Applications, Proceedings ACM EICS'13, pp.43-48, London, June 2013, ACM Press

(Maatjes, 2007) N.C. Maatjes, Automated transformations from ECA rules to Jess, Master Thesis, University of Twente, Available at http://essay.utwente.nl/559/1/scriptie_Maatjes.pdf

(Ngeow et al., 2007) Y.C. Ngeow, A.K. Mustapha, E. Goh, H.K. Low, Context-aware Workflow Management Engine for Networked Devices. Int. J. of Multimedia and Ubiquitous Engineering (IJMUE) 2(3), 33-47 (2007).

(Paschke, 2006) A. Paschke, ECA-LP / ECA-RuleML: A Homogeneous Event-Condition-Action Logic Programming Language. CoRR abs/cs/0609143 (2006).

Acknowledgements

- TELEFÓNICA INVESTIGACIÓN Y DESARROLLO, http://www.tid.es
- UNIVERSITE CATHOLIQUE DE LOUVAIN, http://www.uclouvain.be
- ISTI, http://giove.isti.cnr.it
- SAP AG, http://www.sap.com
- GEIE ERCIM, http://www.ercim.eu
- W4, http://w4global.com
- FUNDACION CTIC http://www.fundacionctic.org

Glossary

- http://www.serenoa-fp7.eu/glossary-of-terms

Appendix I

A first set of detectable contextual events included in the *Event.xsd* XSD Schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
    xmlns="http://www.serenoa-fp7.eu/"
    targetNamespace="http://www.serenoa-fp7.eu/">

    <xs:simpleType name="EventClass">
        <xs:union memberTypes="EnvironmentEventClass UserEventClass TechnologyEventClass
            SocialRelationshipsEventClass UIEventClass" />
    </xs:simpleType>

    <xs:simpleType name="EnvironmentEventClass">
        <xs:restriction base="xs:string">
            <xs:enumeration id="light_changed_env_event" value="light_changed" />
            <xs:enumeration id="light_increased_env_event" value="light_increased" />
            <xs:enumeration id="light_decreased_env_event" value="light_decreased" />
            <xs:enumeration id="noise_changed_env_event" value="noise_changed" />
            <xs:enumeration id="noise_increased_env_event" value="noise_increased" />
            <xs:enumeration id="noise_decreased_env_event" value="noise_decreased" />
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="UIEventClass">
        <xs:restriction base="xs:string">
            <xs:enumeration id="select_ui_event" value="on_select" />
            <xs:enumeration id="change_ui_event" value="on_change" />
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="UserEventClass">
        <xs:restriction base="xs:string">
            <xs:enumeration id="position_changed_user_event" value="user_position_changed" />
            <xs:enumeration id="knowledge_changed_user_event" value="user_knowledge_changed" />
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="TechnologyEventClass">
        <xs:restriction base="xs:string">
            <xs:enumeration id="connectivity_changed_tech_event" value="tech_connectivity_changed" />
            <xs:enumeration id="connectivity_increased_tech_event" value="tech_connectivity_increased" />
            <xs:enumeration id="connectivity_decreased_tech_event" value="tech_connectivity_decreased" />
            <xs:enumeration id="browser_changed_tech_event" value="tech_browser_changed" />
            <xs:enumeration id="browser_closed_tech_event" value="tech_browser_closed" />
            <xs:enumeration id="browser_open_tech_event" value="tech_browser_open" />
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="SocialRelationshipsEventClass">
        <xs:restriction base="xs:string">
            <xs:enumeration id="friendship_changed_social_event" value="social_friendship_changed" />
        </xs:restriction>
    </xs:simpleType>

</xs:schema>
```

Appendix II

The JSON Schema of AAL-DL.

```
{
        "id":                   "http://giove.isti.cnr.it/entry-schema#",
   "$schema":              "http://json-schema.org/draft-04/schema#",
   "description":  "schema                 for                 rules                 model",
        "type" :           "object",
        "properties"                                    :                                       {
                "ext_model_ref" : {
                        "type"                          :                                       "array",
                        "items"     :     {     "$ref"     :     "#/definitions/ExternalModelReferenceType"     }
                },
                "rule"                                  :                                       {
                        "type"                          :                                       "array",
                        "items"         :         {         "$ref"         :         "#/definitions/RuleType"         }
                }
        },
        "definitions"                                   :                                       {
                "ExternalModelReferenceType"                    :                               {
                        "type"                          :                               "object",
                        "properties"                    :                               {
                                "URI"           :           {           "type"           :           "string"},
                                "model_id"          :          {          "type"          :          "string"}
                        },
                        "required"                      :                               ["URI"]
                },
                "RuleType" : {
                        "type"                          :                               "object",
                        "properties"                    :                               {
                                "event"                 :                               {
                                        "$ref"          :                       "#/definitions/EventType"
                                },
                                "condition"                     :                       {
                                        "$ref"          :                       "#/definitions/ConditionType"
                                },
                                "action"                        :                       {
                                        "type"          :                               "array",
                                        "items"     :     {     "$ref"     :     "#/definitions/BlockType"     }
                                },
                                "rule"                  :                               {
                                        "type"          :                               "array",
                                        "items"         :         {         "$ref"         :         "#"         }
                                },
                                "priority"          :          {          "type"          :          "integer"          },
                                "id"            :           {           "type"           :           "string"           },
                                "name"          :          {          "type"          :          "string"          }
                        },
                        "required"                      :                               ["event"]
                },
                "EventType"                             :                               {
                        "type"                          :                               "object",
                        "properties"                    :                               {
                                "simple_event"                  :                       {
                                        "$ref"          :                       "#/definitions/SimpleEventType"
                                },
                                "complex_event"                 :                       {
                                        "$ref"          :                       "#/definitions/ComplexEventType"
                                }
```

```
        }
    },
    "SimpleEventType" : {
        "type" : "object",
        "properties" : {
            "agent" : {
                "$ref" : "#/definitions/EntityReferenceType"
            },
            "event_name" : { "type" : "string"},
            "xPath" : { "type" : "string" },
            "externalModelId" : { "type" : "string"}
        },
        "required" : ["event_name", "xPath"]
    },
    "ComplexEventType" : {
        "type" : "object",
        "properties" : {
            "event" : {
                "type" : "array",
                "minItems": 2,
                "items" : { "$ref" : "#/definitions/EventType" }
            },
            "operator" : {
                "$ref" : "#/definitions/BooleanOperatorType"
            }
        }
    },
    "BooleanOperatorType" : {
        "enum" : [ "and", "or", "xor", "contains", "starts", "ends",
                   "gt", "lt", "gteq", "lteq", "eq", "neq", "not"
                 ]
    },
    "EntityReferenceType" : {
        "type" : "object",
        "properties" : {
            "xPath" : { "type" : "string"},
            "externalModelId" : { "type" : "string"}
        },
        "required" : ["xPath"]
    },
    "ConditionType" : {
        "type" : "object",
        "properties" : {
            "entityReference" : {
                "items" : { "$ref" : "#/definitions/EntityReferenceType" }
            },
            "constant" : {
                "items" : { "$ref" : "#/definitions/ConstantType" }
            },
            "condition" : {
                "type" : "array",
                "items" : { "$ref" : "#" }
            },
            "expression" : {
                "type" : "array",
                "items" : { "$ref" : "#/definitions/ExpressionType" }
            },
            "operator" : {
                "$ref" : "#/definitions/BooleanOperatorType"
            }
```

```
        }
    },
    "ExpressionType" : {
        "type" : "object",
        "properties" : {
            "entityReference" : {
                "$ref" : "#/definitions/EntityReferenceType"
            },
            "constant" : {
                "type" : "array",
                "items" : { "$ref" : "#/definitions/ConstantType" }
            },
            "expression" : {
                "type" : "array",
                "items" : { "$ref" : "#" }
            },
            "operator" : {
                "$ref" : "#/definitions/ExpressionOperatorType"
            }
        }
    },
    "ExpressionOperatorType" : {
        "enum" : [ "+", "-", "*", "/", "%", "concat"]
    },
    "ConstantType" : {
        "type" : "object",
        "properties" : {
            "value" : { "type" : "string"},
            "type" : {
                "$ref" : "#/definitions/xsdSimpleTypeEnum"
            }
        },
        "required" : ["value", "type"]
    },
    "xsdSimpleTypeEnum" : {
        "enum" : [ "anyURI", "base64Binary", "boolean", "byte",
                   "date", "dateTime", "decimal", "duration",
                   "float", "gDay", "gMonthDay", "gYear", "gYearMonth",
                   "hexBinary", "int", "integer", "language", "long",
                   "negativeInteger", "nonPositiveInteger", "normalizedString",
                   "positiveInteger", "short", "string", "time", "token",
                   "unsignedInt", "unsignedLong", "unsignedShort"]
    },
    "BlockType" : {
        "type" : "object",
        "properties" : {
            "create" : {
                "type" : "array",
                "items" : { "$ref" : "#/definitions/CreateActionType" }
            },
            "read" : {
                "type" : "array",
                "items" : { "$ref" : "#/definitions/ReadActionType" }
            },
            "update" : {
                "type" : "array",
                "items" : { "$ref" : "#/definitions/UpdateActionType" }
            },
            "delete" : {
                "type" : "array",
```

```
                "items" : { "$ref" : "#/definitions/DeleteActionType" }
            },
            "if" : {
                "type" : "array",
                "items" : { "$ref" : "#/definitions/IfThenElseType" }
            },
            "while" : {
                "type" : "array",
                "items" : { "$ref" : "#/definitions/WhileType" }
            },
            "foreach": {
                "type" : "array",
                "items" : { "$ref" : "#/definitions/ForEachType" }
            },
            "for" : {
                "type" : "array",
                "items" : { "$ref" : "#/definitions/ForType" }
            },
            "block" : {
                "type" : "array",
                "items" : { "$ref" : "#/definitions/BlockType" }
            },
            "invokeFunction" : {
                "type" : "array",
                "items" : { "$ref" : "#/definitions/InvokeFunctionType" }
            }
        }
    },
    "CreateActionType" : {
        "type" : "object",
        "properties" : {
            "containingEntityReference" : {
                "$ref" : "#/definitions/EntityReferenceType"
            },
            "complexType" : {
                "$ref" : "#/definitions/EntityReferenceType"
            },
            "simpleType" : {
                "$ref" : "#/definitions/xsdSimpleTypeEnum"
            },
            "valueType" : {
                "$ref" : "#/definitions/ReadActionType"
            },
            "entity_id" : { "type" : "string"}
        },
        "required" : ["containingEntityReference"]
    },
    "ReadActionType" : {
        "type" : "object",
        "properties" : {
            "entityReference" : {
                "$ref" : "#/definitions/EntityReferenceType"
            },
            "constant" : {
                "$ref" : "#/definitions/ConstantType"
            }
        }
    },
    "UpdateActionType" : {
        "type" : "object",
```

```
        "properties"                    :                    {
            "entityReference"               :                    {
                "$ref"          :          "#/definitions/EntityReferenceType"
            },
            "value"                     :                    {
                "$ref"          :          "#/definitions/ReadActionType"
            }
        },
        "required"      :      [      "entityReference",      "value"      ]
    },
    "DeleteActionType"                  :                    {
        "type"                  :                    "object",
        "properties"                    :                    {
            "entityReference"               :                    {
                "$ref"          :          "#/definitions/EntityReferenceType"
            }
        },
        "required"          :          [          "entityReference"          ]
    },
    "IfThenElseType"                    :                    {
        "type"                  :                    "object",
        "properties"                    :                    {
            "condition"                 :                    {
                "$ref"          :          "#/definitions/ConditionType"
            },
            "then"                  :                    {
                "$ref"          :          "#/definitions/BlockType"
            },
            "elseIf"                    :                    {
                "type"                  :                    "array",
                "items" : { "$ref" : "#/definitions/IfThenType" }

            },
            "else"                  :                    {
                "$ref"          :          "#/definitions/BlockType"
            }
        }
    },
    "IfThenType"                    :                    {
        "type"                  :                    "object",
        "properties"                    :                    {
            "condition"                 :                    {
                "$ref"          :          "#/definitions/ConditionType"
            },
            "then"                  :                    {
                "$ref"          :          "#/definitions/BlockType"
            }
        }
    },
    "WhileType"                     :                    {
        "type"                  :                    "object",
        "properties"                    :                    {
            "condition"                 :                    {
                "$ref"          :          "#/definitions/ConditionType"
            },
            "do"                    :                    {
                "$ref"          :          "#/definitions/BlockType"
            }
        }
    },
```

```
        "ForEachType"                                   :                                   {
                "type"                                  :                                   "object",
                "properties"                                    :                                   {
                        "in"                                    :                                   {
                                "$ref"                  :                   "#/definitions/ReadActionType"
                        },
                        "do"                                    :                                   {
                                "$ref"                  :                   "#/definitions/BlockType"
                        },
                        "alias"         :               {       "type"          :       "string"}
                }
        },
        "ForType"                                       :                                   {
                "type"                                  :                                   "object",
                "properties"                                    :                                   {
                        "do" : {
                                "$ref"                  :                   "#/definitions/BlockType"
                        },
                        "alias"         :               {       "type"          :       "string"},
                        "from"          :               {       "type"          :       "integer"},
                        "to"            :               {       "type"          :       "integer"}
                },
                "required"              :               [       "alias",        "from",         "to"]
        },
        "InvokeFunctionType"                                    :                                   {
                "type"                                  :                                   "object",
                "properties"                                    :                                   {
                        "input"                                 :                                   {
                                "type"                                  :                                   "array",
                                "items" : { "$ref" : "#/definitions/InvokeFunctionParameterType" }

                        },
                        "output"                                        :                                   {
                                "type"                                  :                                   "array",
                                "items" : { "$ref" : "#/definitions/NamedEntityReferenceType" }

                        },
                        "name"                                  :                                   "string"
                }
        },
        "InvokeFunctionParameterType"                                   :                                   {
                "type"                                  :                                   "object",
                "properties"                                    :                                   {
                        "value"         :       {       "$ref"          :       "#/definitions/ReadActionType"          },
                        "name"          :               {       "type"          :       "string"}
                }
        },
        "NamedEntityReferenceType"                                      :                                   {
                "type"                                  :                                   "object",
                "properties"                                    :                                   {
                        "xPath"         :               {       "type"          :       "string"},
                        "externalModelId"               :               {       "type"          :       "string"},
                        "name"          :               {       "type"          :       "string"}
                },
                "required"                                      :                                   ["xPath"]
        }
    }
}
```