Ontology Based Adaptive Systems. A Case Study: Car Environment

Marko Ribarić
Institute Mihailo Pupin,
Volgina 15, 11060 Belgrade, Serbia
marko.ribaric@institutepupin.com

Gerd Kock
Fraunhofer FIRST,
Kekuléstrasse 7, D 12489 Berlin, Germany
gerd.kock@first.fraunhofer.de

Abstract: *Rapid development of sensor devices and methods for man-machine interaction influenced the increased research in the area of adaptive, context-aware systems. In this paper we present such a system, which aims at seamlessly collecting information from humans and their domain specific surrounding,, processing this information by dynamically creating "self-organized awareness context", and reflecting the discovered context by offering appropriate services or actions in a pervasive manner. This system is based on the use of ontologies and rules. Ontologies are used for describing the characteristics of used devices (sensors and actuators), the constraints and requirements they need to comply with, while the rules are used for defining goals (actions) that need to be fulfilled as a reaction to the discovered context. The paper shows the way this adaptive, context-aware system works in the car environment. The car environment is chosen not only for the reason that an electronically equipped car can contribute to a safer driving, but also the fact that it may influence the driver's attitude and mood by observing the driver and adapting the car settings and ambient according to the driver's psycho-physiological state.*

Keywords: adaptive, context-aware environments, ontologies, rules

1 Introduction

Recent progress in electronics, mechanics and ergonomics, as well as the progress in automatic human affect analysis has made the development of qualitatively new systems possible. The prefix smart (smart-house, smart-cars, smart-toys etc.), usually denotes pervasive and adaptive capabilities of new products which, equipped with micro-processors and sensor devices, can actively participate in everyday situations. For example, in the vehicular application domain a car is being smoothly transformed into a friendly co-driver that observes the driver, road conditions, engine settings, and actively participates in the driving process. In other application scenarios, from gaming, home ambient up to mobile and outdoor advertisement, embedded adaptive control systems silently interacts with users and support the applications' activities [8].

The system that supports affective, context-aware[1] computing generally consists of sensor devices - used to collect information from users and environment; actuators - that can be used to influence the user and the surroundings; and middleware parts. The middleware parts gather context information, process it and derive meaningful actions from it. Middleware presents the core of these systems, as it allows different agents to acquire contextual information easily, reason about it using different logics and then adapt themselves to changing contexts [3].

Developing middleware for affective, context-aware computing applications is a non trivial task because the target system must cope with dynamic properties, complex algorithms, structural design and concurrency. Plus, the development of such systems involves multidisciplinary teams with computer engineering, human science and praxis background. Often, the parties involved in such a common development do not understand each other well, as they are coming from different fields, have different educational background and sometimes even different way of thinking. These factors pose an extra burden to the already complex developing task.

One of the answers to these challenges is the REFLECTive framework [9] aiming at developing methods and tools for working with such user-centric, pervasive, adaptive systems. The first step in the research of the REFLECTive framework was to build a simulator which is capable of stepwise development and evolutionary transformation toward the final system [10]. A modular approach is selected for

[1] A system is context-aware if it can extract, interpret and use context information and adapt its functionality to the current context of use [1].

designing and implementing this simulator. By developing different simulating modules, addressing different problems that can independently be refined, the simulator improves and optimizes the collaborative work on one hand and provides means for more efficient deployment of real systems, on the other hand. The REFLECTive framework (and hence the simulator) is a service and component oriented infrastructure implemented in Java using the OSGi [7] (i.e. its Eclipse implementation Equinox). By relying on the Service Registry, along with other layers that the OSGi framework offers, our system can reliably manage context-aware services to support context acquisition, discovery, and reasoning. The REFLECTive framework also utilizes ontologies and rules. Ontologies are used for describing the characteristics of used devices (sensors and actuators), the constraints and requirements they need to comply with, as well as the current context of both the user and the environment. Rules are used for defining goals (actions) that need to be fulfilled as a reaction to the discovered context and user's state.

The paper presents one scenario of using this solution in the car environment. The car environment is chosen for the reason that an electronically equipped car can contribute to a safer driving, and for the fact that it may influence the driver's attitude and mood by observing the driver and adapting the car settings and ambient according to the driver's psycho-physiological state. The paper is structured as follows: Section 2 gives, very shortly, the research trends in the area of adaptive, context-aware systems; it also elaborates reasons for using ontologies, and gives a short overview of rule based systems. Section 3 presents the global structure of the used software architecture, while section 4 explains the role of ontologies. In section 5 one scenario in the car environment is described, while section 6 concludes the paper.

2 Background

Recently the number of papers dealing with adaptive, context-aware systems has increased. Research in this area is oriented both toward the methods of automatic human affect analysis, as well as toward addressing the complexity of such systems.

Methods that are being used in the field of automated analysis of human affective behavior are not any more concentrated on the deliberately displayed series of exaggerated affective expressions, neither are they single modal [12] (meaning that information processed by the computer system is limited to either face images or the speech signals). Researchers are now concentrating on multimodal fusion for human affect analysis, which includes audio-visual fusion, linguistic and paralinguistic fusion, and multi-cue visual fusion based on facial expressions, head movements, and body gestures [12].

On the other hand, the complexity of adaptive, context-aware systems motivated and inspired researchers to conceive many different approaches to providing architecture that would effectively deal with such systems. The paper [3] very well summarizes those different approaches, and additionally provides, based on those same approaches, a general abstract layer architecture of context-aware systems. This general abstract layer architecture consists of four layers: network layer, middleware layer, application layer, and user infrastructure layer. The authors stress that the **network layer** involves a network - supporting context-aware systems, and sensors - collecting low-level of context information. They categorize the network infrastructure layer into: internet protocol, handoff management, sensing, network requirements and network implementation. The **middleware layer** is responsible for managing processes and storing context information, and is classified as agent-based middleware, metadata based middleware, tuple space based middleware, OSGI based middleware, reflective middleware and sensor selection middleware. The **application layer** provides users with appropriate service (context-aware applications include information systems, especially decision support systems, communication systems as social community, e-commerce, etc.) while the interface of context-aware systems is managed in **user infrastructure layer**.

As previously mentioned, the framework used in this paper (and described in more detail in [9]) is based on OSGi, and the use of ontologies and rules. Ontologies play a crucial role in enabling the processing and sharing of information and knowledge on the middleware. Ontologies provide a shared and common understanding of a domain that can be communicated within a broad developing team. They also allow devices and agents, which are not originally designed to work together, to interoperate. In this paper, a somewhat loose definition of ontology is accepted - we consider ontology to be a structure of concepts or entities within a domain, organized by relationships. The primary relationships that we are interested in are class hierarchies, i.e. we can say that we are leveraging taxonomies rather then ontologies (but in the rest of the paper the term ontology is used, as we will in the future investigate other relationships between used concepts). As proposed in [6] we use UML as the language for describing the ontology. One reason is that this approach allows basing the discussion on easily understandable graphical representations, which will help in the future usage and elaboration of the ontology. Another important reason is that by using UML the ontology can be related to the software in a straight-forward manner. Partially, it will be possible to (semi-)automatically generate XML from the given UML structures, which in turn can be used to parameterize the Java implementation of the reflective framework.

Benefits of using rules in our framework are stressed in Section 5, but here we shortly explain what a rule based system is. Rule based system is a system whose knowledge base is represented as a set of rules and facts. A rule based system consists of IF-THEN rules, a collection of facts and some inference engine (interpreter). The inference engine matches facts against rules to infer conclusions which result in actions [2]. The process of matching the new or existing facts against rules is called pattern matching, which is performed by the inference engine. Each inference engine goes through three phases: matching phase, conflict resolution phase and execution phase. In the *match phase* the inference engine is comparing the fact base and the rule base. If it finds rules that are to be triggered it passes them to the *conflict resolution phase*. A system with a large number of rules and facts may result in many rules being true for the same fact assertion, these rules are said to be in conflict. To resolve this conflict a strategy is needed (this strategy is often times called heuristic strategy). When the rule engine decides which rule to fire, this rule is transferred to the next, *execution phase*. When the rule is fired, it causes the change in the fact base (new facts are added to the fact base). After execution phase again follows the matching phase, and the new cycle starts. (This cycle can continue until we exhaust our knowledge or the defined goal is met). All the rules are stored in the rule base (production memory), while the facts are stored in the fact base (working memory) where they may be modified or retracted. These two components along with the inference engine constitute a rule base system [2].

In the next section a global structure of our system is presented, being further referred to as REFLECT system.

3 REFLECT System Design

The REFLECT development system consists of three tiers (Fig. 1) [5]. The ***tangible tier*** includes basic services, which can be used to communicate with sensors and actuators. The ***reflective tier*** is the middleware of the REFLECT system, comprising a Service Registry and other central components. And the ***application tier*** adheres to software components and tools for the development of applications and related configuration items[2].

The REFLECT system can be considered as a Java based toolbox being used in the development of REFLECT applications. Any REFLECT application contains components of each of these tiers. Such an application can be considered as a constantly adapting system, analysing sensor inputs, reflecting about the actual and former results of the analysis, and reacting according to a given application scheme via controlling actuating variables. Conceptually this means, that any REFLECT application is running in a ***closed loop***, reflecting the permanent cycle ***sense-analyse-react***. However, this does not imply that at any time the same loop is performed, it only means that perpetually sensors deliver inputs, leading to outputs for actuators, which in turn lead to modified inputs, etc. Actually, there will be applications, where a number of closed loops will run in parallel.

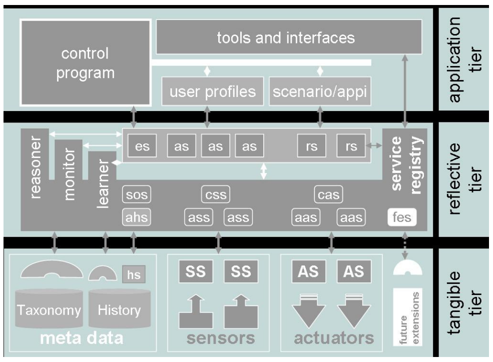

Figure 1. REFLECT system

Fig. 2 details this conceptual view. The basic idea is to hierarchically organize the sense-analyse-react cycle. At the bottom, there is the environment with sensor and actuator ***devices***. Above, the low-level software components perform the first and the last steps in any cycle; either ***features*** are extracted from sensors or are used to control actuators. Next, the high

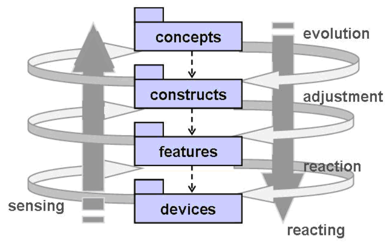

Figure 2. Conceptual view of REFLECT application

level components deal with abstract ***constructs*** describing the user context, her psychological, cognitive or physical state, or high-level system goals. The software is organized such that these high-level components build on the low-level ones. And at the top application level, according to the ***concepts*** at hand, the effectively available user constructs (obtained from the connected sensors) are related to possible system goals (depending on the involved actuators). Again, the application level directly relies only on the high-level components.

This system operates in the following way (see Fig. 3) [5]:

Each type of sensor that exists in the system (that is being used for gathering information about the environment) is described in the sensor ontology. This ontology offers, for each type of sensor, properties

[2] The goal of this section is not to give a comprehensive account of the REFLECT software, only its basic structures are explained. Details about the REFLECT system architecture can be found in [9].

that match the output from this sensor's type (usually defined in a sensor specification).

Data gathered by sensors then need to be processed in order to get features that can be used for identifying current emotional, physical and cognitive state of the user. This processing involves the use of complex computer algorithms for image processing, speech processing, voice analysis, etc. (e.g. algorithms for statistical signal processing). Today there are lots of available functional implementations in this area, and the REFLECT system encourages their reuse and provides a mechanism for their easy integration.

Each feature gained as an output of these processors is defined in the features ontology. Based on those features REFLECT system has to infer the state (emotional, physical and cognitive) of the user. There are lots of solutions that can be used for emotion recognition from available features – the majority of which fall into group of so called classifiers (machine learning methods for classification): e.g. support vector machines, neural networks, or rule-based classifiers. The REFLECT system does not offer its own classifiers, but again provides a way for easy integration of existing solutions.

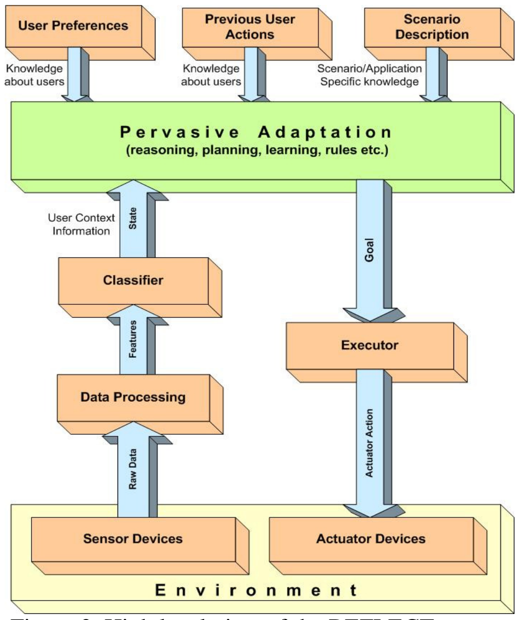

Figure 3. High level view of the REFLECT system

The output of the classifier describes the user's current state – each state is again defined in the state ontology. At this point the REFLECT system contains information about the user's state, the user's preferences and about the history of previous user actions – as the system keeps records of previous choices a user made. Based on this information the system can decide on the future course of action, trying to adapt itself to the user needs. For this the REFLECT system relies on the use of rules for scenario description. These scenarios define abstract (i.e. regardless of present actuators) goals (or rules actions) that aim at changing the state of environment.

The goals are then forwarded to the component called Executor that decomposes these goals to the specific actuator actions. The Executor knows what actuators are available in the system, and how a concrete actuator needs to be changed in order to satisfy the given goal. The Executor is aware of all the actions each actuator is able to perform, as all actuators are described in the actuator ontology. This ontology classifies actuators and defines each actuator through actions that it can perform.

4 The role of ontology

The global structure of the ontology adheres to the conceptual view as depicted in Fig. 2, i.e. the ontology is hierarchically organized into four modules being named ***devices***, ***features***, ***constructs*** and ***concepts***, and these modules incorporate those items or concepts, which make the ontology suited for discussing adaptive, context-aware applications. The ontology is under permanent development, as in the course of elaborating or specifying particular applications, there will be the need for adding further notions. In the next subsections, for each of the four levels introduced in Fig. 2, we sketch the internal structure of the associated modules.

4.1 Modelling Devices

The *devices* module contains entries for sensors, actuators, or other devices. In Fig. 4 the top level UML concepts of the *devices* package are presented. *Sensor* examples are *Camera*, *Thermometer* or *BloodPressureSensor*, *Actuato*r examples are *RoomLight* or *Radio*, and a hard disk is an example for *OtherDevice*.

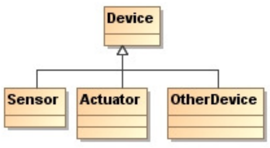

Figure 4. Modelling Devices

4.2 Modelling Features

Features may be measurements like *Pulse* or *Temperature* stemming from sensors, actuating variables like *Tone* or *Volume* belonging to actuators, or facts like *Age* or *Gender* stemming from a hard disk. In Fig. 5, the top level UML diagrams of the *features* package are presented.

Each *Feature* is realized and/or implemented by one or more *Device* items. A *Feature* can be a

UserFeature or a *ContextFeature*, plus one has to distinguish between *SimpleFeature* and *ComplexFeature*.

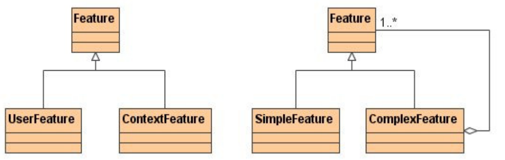

Figure 5. Modelling Features

The term *UserFeature* is used, if the respective feature refers to the emotional, cognitive, or physical state of the user (e.g. *Pulse* or *Movement*). The term *ContextFeature* refers to the user context (e.g. *Time*). Note however that some features will be user and context features at the same time (e.g. *Temperature*). Anyway, in many cases the final category depends on the application under consideration.

Notions like *Pulse*, *Movement*, or *Temperature* are simple features. Complex features are items being derived from one or more other (simple or complex) features, an example would be *MeanTemperature*.

As noted above, features are used for sensing and reacting. Consequently, in modelling applications they will be related to sensor or actuator items, or to both. Examples for the first case would be *FacialExpression* and *HeartRate*, and examples for the last case would be *RoomTemperature* and *RoomLightning*. (A camera might be used to observe the facial expression, but there is no device to directly manipulate it. On the other hand, the room temperature can be measured by a thermometer and can be adjusted by a heater control.)

4.3 Modelling Constructs

Any REFLECT application considers the user's current context and state and adapts itself to it. As far as the user's state is concerned, one has to distinguish between the emotional state (e.g. annoyance) and cognitive engagement (e.g. high mental workload) of the user and physical conditions and actions (e.g. temperature and movement). Dealing with the system reaction to a given context and user state requires a notion of "goal", which is used with environment and user data to infer actions to be taken.

In Fig. 6, the top level UML concepts of the *constructs* package are presented. Each *Construct* item is an aggregate of one or more *Feature* items, and one has to distinguish between *UserContext*, *EmotionalState*, *CognitiveState*, *PhysicalState*, and *SystemGoal*. Examples for emotional or cognitive states are *Motivation*, *MentalOverload*, *Comfort*, *Effort*, *Mood*.

Again, in the process of modelling applications, notions from the *constructs* module are being related to notions from the *features* module, to reflect the sensing or reacting stage respectively. As an example, consider the *Mood* construct. Determining the actual mood level belongs to the sensing stage and would be reflected in relating the mood notion for example to features like *FacialExpression* or *HeartRate*. On the other hand, for the reacting stage, i.e. to influence the mood, *SystemGoal* items would build on features like *RoomTemperature* or *RoomLight*, or on musical features.

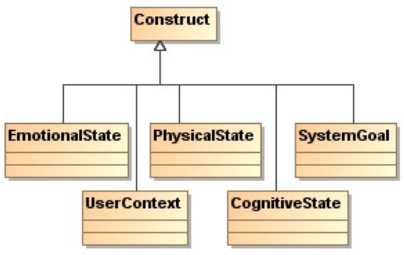

Figure 6. Modelling Constructs

4.4 Modelling Concepts

The idea of the *concepts* module is to provide a means for easy description of application scenarios. One example of a simple scenario is: "If the driver has been driving for a long time and he frequently changes his sitting positions then give him some visual/audio warning telling him to rest a bit". This module effectively connects available user constructs (obtained from the connected sensors) with possible system goals (depending on the involved actuators) and the application description. Development of this module is something to be undertaken in the future.

5 Defining goals

This section describes one scenario of defining several goals in a car environment. These goals are defined manually based on the user state, user and environment context, and user preferences. The goals are basically actions that need to be fulfilled, and are ignorant of the actuators present in the system. Chosen actions that are to be taken (depending on the user state, context and preferences) later are sent to the next component in the REFLECT system, called Executor, that tries to fulfill these goals depending on the present actuators (see Section 3).

The process of defining, choosing and firing adequate goals in the REFLECT system relies on the use of rule engines. Advantages of using rule engine are numerous [2]: rules facilitate declarative programming – rule engines allow you to say "What to do" not "How to do it"; they allow for logic and data separation - the logic can be much easier to maintain when there are changes in the future, as the logic is all laid out in rules; rules allow the centralization of knowledge - by using rules, you create a repository of knowledge (a knowledgebase) which is executable. Rules are also more understandable to domain experts (possible non-technical people) as they can be expressed in the language that is easily understandable by them. These advantages are also the main reasons for using rule engines in the REFLECT system, which benefits greatly from the fact that rules can be defined by domain experts (in this case users of the system which are not necessary technical people), and also from the fact that rules are located in one centralized place (when a need for a rule change emerges, only one place in a system needs to be updated). Currently we are using the Jess[3] rule engine, but we are considering the use of Drools[4] rule engine because of its somewhat better tool support.

The process of defining goals begins with the ontology. We mentioned that goals (which constitute the right hand side of a rule, or consequence/action part of a rule) are triggered based on the user state, user and environment context, and user preferences (these factors constitute the left hand side of a rule, or when/conditional part of a rule). In the REFLECT system the user state, context and preferences, among other things, are defined in ontologies. As we define our ontologies in UML (for the reasons mentioned in Section 2), we need a way to serialize them in the XML format, as well as to get corresponding Java objects so we could use them as a left hand side (LHS, or conditional part) of a Jess. For this purpose we use the Eclipse Modeling Framework (EMF)[5].

EMF is a framework and code generation facility that allows defining a model in any of the following forms: Java interfaces, UML diagram or XML schema, from which the other forms and corresponding implementations classes can be generated [11]. The model used to represent models in EMF is called Ecore. Ecore is itself an EMF model, and thus is its own metamodel. Ecore is a small and simplified subset of full UML [11]. The serialized form of an Ecore model is XMI. (XMI stands for XML Metadata Interchange and is a standard for serializing metadata concisely using XML).

The process of getting XML files and Java objects from UML diagrams is as follows (Fig. 7). First, we define our ontologies (UML class diagrams) in the MagicDraw[6] tool. MagicDraw has an option of exporting an UML diagram into the EMF UML2 XMI format, which can then be converted into the Ecore metamodel by using the UML2 plug-in for Eclipse[7] (in the Eclipse environment, with the installed EMF support, we choose an option "EMF UML import"). After getting the Ecore metamodel, we leverage the feature of EMF that offers exporting EMF models to XML schema (simply by choosing an option "EMF Export Model to XML schema"). At this point we have XML schema that is generated from the starting UML diagram, so now all we need are corresponding Java implementation classes. For getting these classes we again rely on a nice, built-in feature that EMF offers (option "Generate Model Code" in the Eclipse environment). The code generated like this can be used as a LHS for the Jess rule engine, more specifically can be used in the Java application that has embedded Jess engine.

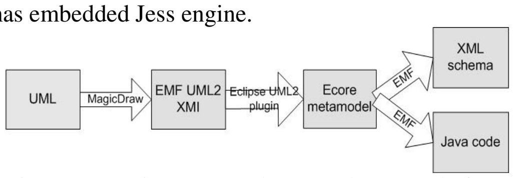

Figure 7. Getting XML schema and Java code from UML

Now that we have our classes generated, all that is left is the rules definitions. In this scenario we presume the availability of the following sensors: camera, microphone, CAN-bus[8], finger thermometer, vest (that integrates ECG electrodes and a Respitrace band - Respiration Plethysmography Band), seat force sensors, and GPS device. Based on these sensors we can infer the following states of a user (in this case a user is a driver as we are talking about the car environment): driver's physical state, emotional state, and cognitive state. Also these sensors tell us something about the current environment context, e.g. weather and car context: is the driver alone in the car, what is the noise in the car, speed of a car, type of road, the driving time. Person's context (driver's age, gender) and preferences (favorite radio station, music genre, CD, etc.) is something we acquired earlier. In this scenario, we also presume the presence of following actuators: manettino[9], seat (adjustment), music player, dashboard, and iPhone.

In a scenario like this we can define these three rules (expressed in the Jess rule language):

```
(defrule sport-drive
"if the driver is in high emotional state,
and the weather is nice, and she is driving
on a highway, set the car in the sport mode"
(Person  (lastName x) (firstName y)
(emotionalState high) )
(Weather {temperature > 15} (rain false) )
(Car (roadType highway) {speed > 90} )
=>
(add (new Goal "set sport mode" ) )
)

(defrule tired-driver
"if the driver is driving constantly more
than 4 hours, and hers cognitive state is
high, and she is frequently changing her
sitting position (i.e. her physical state is
low) then give her visual alerts (i.e.
warning light on a dashboard, plus message on
an iPhone) and start playing load music"
(Person  (lastName x) (firstName y)
(cognitiveState high) (physicalState low) )
(Car {drivingTime > 4} )
```

[3] http://www.jessrules.com/
[4] http://jboss.org/drools
[5] http://www.eclipse.org/modeling/emf/
[6] http://www.magicdraw.com/
[7] http://www.eclipse.org/uml2/
[8] http://en.wikipedia.org/wiki/Controller_Area_Network
[9] http://en.wikipedia.org/wiki/Manettino_dial

```
=>
(add (new Goal “set visual fatigue alert” ) )
(add (new Goal “play random music high
volume” ) )
)

(defrule sad-driver
“if the driver is sad and she is alone in the
car and the noise in the car is high then
play her favorite music and give her visual
feedback”
(Person  (lastName x) (firstName y)
(emotionalState low) )
(Car (noise high) (alone true) )
=>
(add (new Goal “play favorite music medium
volume” ) )
(add (new Goal “set visual happy feedback” )
)
)
```

We see in these examples that the actions come after the "=>" symbol in the rule. E.g. if the first rule applies to LHS (the conditions of this rule are met), a new *Goal* object is created (with a constructor parameter "sport mode"), and that *Goal* object is added to working memory, using the *add* Jess function. *Add* function adds the given object to working memory, and also creates a "shadow fact" [4] (shadow facts are facts that serve as bridges to Java objects) representing the given Java object, using the template whose name is the same as the given object's class. If this template doesn't exist, Jess creates it. We also mentioned that Jess library can be used from Java. To embed Jess in Java application one simply has to create one or more *jess.Rete* objects and manipulate them appropriately (the *jess.Rete* class is the rule engine itself – each object of this class has its own working memory, agenda, rules, etc. [4]).

This simple scenario has the objective to show conceptually the process of generating/defining rules in the REFLECT system. This process is liable to changes, as our further efforts will be oriented toward total automation of goal definition (user history will be also considered in the decision process of choosing adequate goals).

6 Conclusion

Developing software to control affective, context-aware computing applications is a complex task. Our efforts are concentrated toward the development of a simulator where the low-level devices (sensors and actuators) as well as user emotional, cognitive and physical states, are simulated and represented as services. Later as the system develops each simulated service will be substituted with a real one, thus allowing for a gradual final system implementation. Our solution is based on the use of ontologies. The adopted approach to first design the ontology and then to design and develop reflective simulator brought several benefits. Some of the advantages can be summarized in the following:

- abstract and generic approach to system design
- separate development of low-level psycho-physiological measurement services (hidden in the tangible tier)
- separate development of the service and component oriented platform to support context-awareness
- separate development of high level adaptive and reflective components that combine kernel primitives, user profiles and application scenario (hidden in the application tier)
- early prototyping – allowing for different scenario probation and re-designing in an almost real framework
- easy interfacing to other existing taxonomies and ontologies of similar systems
- efficient final implementation of embedded control systems which requires only assembly of already tested modules from all three tiers.

Lot of research efforts in the field of adaptive, context-aware systems are recently being conducted, but because of complexity of these systems their scope is still limited only to small regions, e.g. smart rooms, hospitals, cars, toys, etc. Researchers are trying to find answers to some questions that haven’t been answered yet [3]:

How to effectively extract user context in context-aware application? In the past researchers focused just on a physical context of a user, but that was not enough to build effective context-aware, adaptive systems. Our solution takes in consideration cognitive, emotional and physical context of a user. But, the question remains: can we capture the real state of a user with non intrusive devices that are present today.

Which is the best algorithm to use in order to extract high level user context from low level sensors? Also, what algorithm to use for defining goals, i.e. for increasing user satisfaction by recommending service that user wants to receive? Researchers use different algorithms to solve this problem: Bayesian networks, probabilistic logic, fuzzy logic, decision tree, neural network and support vector machine are applied. The REFLECT system, presented in this paper, does not offer a solution for this question, rather it tries to provide a mechanism for painless integration of existing solutions.

How to provide the users with the automatic personalized services? There were some limitations in previous research for providing the personalized services on context-aware systems: the users had to input their preferences directly, and automated services were not provided for them.

There are many sensors involved in gathering data about the user and environment. Each of these sensors uses different scale, unit, data format, so the question is how to deal with this variety of information. Our solution tries to address this issue by using ontologies, where every sensor used in a system is presented in the ontology along with properties that define it (information about expected input/output to/from this sensor).

When the context of users is conflicted how to choose the best solution? For example, if we have many persons in a room, how to know whose state is measured by some sensor, or how to know whether different sensors measure the same person. Also if preferences of these persons differ (which is very likely) how to know what goals to define, i.e. whose preferences have a greater priority. The problem of conflicts has been approached by researchers by using information fusion, time stamps and fuzzy algorithm, but it is still not solved perfectly.

Security issue. Adaptive, context-aware systems store and handle sensitive and personal data, so if they want to be publicly accepted they need to consider user privacy and security.

And finally, can some design patterns be extracted in the area of context-aware systems.

Our further work will be devoted to solving these problems.

7 References

[1] Byun H. E, Cheverst K: Utilizing context history to provide dynamic adaptations. Applied Artificial Intelligence, 18(6), 2004, pp. 533–548.

[2] Drools documentation, available at: http://www.jboss.org/drools/documentation.html

[3] Hong J-y, Suh E-h, Kim S-j: Context-aware systems: A literature review and classification. Expert Systems with Applications, 2008.

[4] Jess documentation, avalable at : http://www.jboss.org/drools/documentation.html

[5] Kock G, Ribaric M, Serbedzija N: Modeling User-Centric Pervasive Adaptive Systems – The REFLECT Ontology, Intelligent Systems for Knowledge Management, Springer, 2009, submitted.

[6] Kogut P, Cranefield S, Hart L, Dutra M, Baclawski K, Kokar M, Smith J: UML for Ontology Development. The Knowledge Engineering Review 17(1), 2002, pp. 61-64.

[7] OSGI Alliance: *About OSGI Service Platform, OSGI* White paper, available at: http://www.osgi.org/wiki/uploads/Links/OSGiTechnicalWhitePaper.pdf , November 2005.

[8] Serbedzija N, Calvosa A, Ragnoni A: Vehicle as a Co-Driver. Proc. of the First Annual Int. Symposium on Vehicular Comp. Systems, ISVCS 2008, Dublin, Ireland , 2008.

[9] Serbedzija N, Kock G,Ribaric M, Tomasevic N, Stanojevic M, Schroeder A: REFLECT Deliverable D1.1, First Year Report: Requirements and Design, 2009.

[10] Serbedzija N, Ribaric M, Tomasevic N, Beyer G: Simulating Adaptive Control in Multimedia Applications, 1st PerAda Workshop at SASO 2008, Venice, Italy, 2008.

[11] Steinberg D, Budinsky F, Paternostro M, Merks, E: EMF Eclipse Modeling Framework, Second Edition, Addison Wesley, 2009.

[12] Zeng Z, Pantic M, Roisman G, Huang T: A Survey of Affect Recognition Methods: Audio, Visual, and Spontaneous Expressions, In ICMI '07: Proceedings of the 9th international conference on Multimodal interfaces, 2007, pp. 126-133.