7
Implementation

In this chapter we describe the Decentralized Reasoning Service (DRS), a prototype service implementation that performs the cooperative reasoning process presented before. We present also the Context Model Service (CMS), another prototype service that had to be to implemented to support the DRS providing access to up-to-date context information. Finally, to show how the use of the DRS simplifies the design of ubiquitous applications, we discuss the use of context and inference services in the implementation of a prototype application.

PUC-Rio - Certificação Digital Nº 0510970/CA

7.1
Architecture Overview

We implemented the Decentralized Reasoning Service (DRS) as a prototype service that implements our approach proposed for decentralized reasoning, performing the *cooperative reasoning process* described in Chapter 5. This rule-based inference service was designed to be executed on top of a middleware architecture aiming to provide a complete infrastructure to create context-aware applications integrating mobile devices and multiple context providers in AmI environments.

To be able to test and evaluate the DRS, it was absolutely necessary to have the functionalities provided by a service responsible for managing context information, i.e., collecting, storing and providing access to context data. As such, we implemented also the Context Model Service (CMS), a prototype service responsible for collecting context data from context providers available in a specific domain, keeping an updated representation of the assembled data according to a valid context model (an ontology), and providing access to up-to-date context information.

CMS is described in more detail in Section 7.3, while the characteristics of the DRS implementation are discussed in Section 7.4. As both CMS and DRS rely on KAON2 [103] — an OWL and reasoning API — to access ontology data and perform reasoning operations, this API is further described in the next section. In our scenario we assumed that all entities share the same context

model (ontology) and every DRS and CMS server has a well-known address (IP and port).

7.2 Ontology Management and Reasoning

Various reasoning engines have been developed for reasoning and querying OWL-DL ontologies, implementing different reasoning algorithms and optimization techniques, hence differing in a number of ways [104]. Systems such as RACER [71], Pellet [99] and KAON2 [103] provide automated reasoning support for checking concepts for satisfiability and subsumption in a *TBox*, and also for answering rule-based queries, retrieving the individuals in an *ABox* that satisfy a given rule [105].

KAON2 has been compared with RACER and Pellet [103, 106, 107] and it was found that it provides better performance for ontologies with rather simple *TBoxes*, but large *ABoxes* [108], i.e., ontologies with a large number of individuals and facts and a small number of classes and properties. In contrast, for ontologies with large and complex TBoxes, the other reasoners provide superior performance. Furthermore, among these three reasoners, KAON2 has as distinguished features its simplicity and compactness, as it includes an API for managing OWL ontologies, while Pellet and RACER require the use of specific tools as the OWL API [109]. As ontologies for ubiquitous computing scenarios tend to have large *ABoxes* while *TBoxes* are not so large nor complex [110], and considering the simplicity of KAON2, we assumed it to be more appropriate for use in ubiquitous scenarios such as the one described in Section 2.1. Hence, we selected KAON2 to implement ontology management and reasoning for our CMS and DRS.

KAON2 is an OWL-DL reasoner implemented in Java 1.5 and free for non-commercial use. Differently from RACER and Pellet, KAON2 does not implement the tableau calculus, but rather transforms OWL-DL ontologies into disjunctive datalog, and applies established algorithms for dealing with this formalism, enabling a faster processing of large *ABoxes*. The system can decide concept satisfiability, compute the subsumption hierarchy, and answer conjunctive queries in which all variables are distinguished [111].

It can be used as a stand-alone server or as a dynamic library, providing an Ontology API and a Reasoning API. The Ontology API — which is used by CMS — provides ontology manipulation services, such as adding and retrieving ontology axioms. The API fully supports OWL and the Semantic Web Rule Language (SWRL) at the syntactic level. It allows ontologies to be saved in files using either OWL-RDF or OWL-XML syntax. The Reasoning API —

PUC-Rio - Certificação Digital Nº 0510970/CA

which is used by DRS — allows to invoke various reasoning functionalities and to retrieve their results. We used the latest stable release of KAON2, published on 29th of June 2008.

7.3
Context Model Service (CMS)

In our middleware architecture for ubiquitous environments, the Context Model Service (CMS) was implemented as the basic service responsible for collecting all context data from the context providers available in a given domain and keeping an updated representation of the assembled data coherent with the adopted context model. These context provider may be any sensor, service or applications that sends data to CMS, which are interpreted and stores as facts according with the ontology. Besides that, the CMS provides access to up-to-date context data for DRS and applications that need plain context information, i.e., that does not involve reasoning.

PUC-Rio - Certificação Digital Nº 0510970/CA

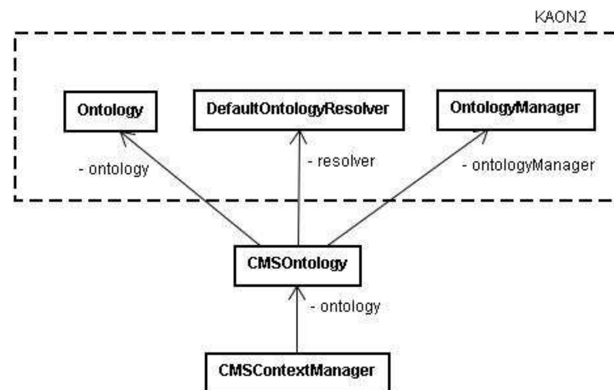

Figure 7.1: Class diagram showing the implementation of the CMS server.

CMS was implemented in Java (version 1.6.0) as a multi-threaded server that supports synchronous communication using both TCP and UDP transport layer protocols. Since it does not implement context monitoring (which is implemented only by the DRS), CMS does not provide asynchronous (event-based) communication. CMS uses KAON2 API — discussed in the last section — to manipulate the context ontology database and MoCA communication APIs [35] to implement synchronous communication. Figure 7.1 shows a simplified class diagram of the CMS service implementation, in which the main dependencies on KAON2 classes are represented. The main class of the service is the *CMSContextManager*, which implements the communication and the server loop to receive message from client applications. This class relies on class *CMSOntology* to access and manage a specific ontology. Class *CMSOntology*

uses KAON2 classes *Ontology*, *DefaultOntologyManager* and *OntologyManager* to manage an OWL ontology file.

In practice, CMS is a server that when started loads up a configuration file ("cms.properties") that defines the IP address, port and protocol (UDP or TCP) for running the server, and an ontology file to be loaded. The server will load the context model and data stored in the ontology file and wait for messages from client applications consulting or updating this context data.

To facilitate the work of developers that implement context provider or context consumer applications we created a client API for CMS. The *CMSClient* API implements the methods enumerated below, providing a greater abstraction level than the KAON2 API for describing the providing or consulting context information.

- `ArrayList getAllClasses()` - Used by a client application to retrieve, as an array list of RDF tuples, the names of all classes.
- `ArrayList getAllIndividuals()` - Used by a client application to retrieve, as an array list of strings, the names of all individuals.
- `ArrayList getAllProperties()` - Used by a client application to retrieve, as an array list of RDF tuples, the names of all properties.
- `ArrayList getIndividualsOfClass(String C)` - Used by a client application to retrieve, as an array list of strings, the names of all individuals belonging to a class C, whose name is passed as parameter.
- `ArrayList getIndividualsOfProperty(String P)` - Used by a client application to retrieve, as an array list of strings, the names of all individuals having a property P, whose name is passed as parameter.
- `ArrayList getPropertiesOfIndividuals(String I)` - Used by a client application to retrieve, as an array list of RDF tuples, all the properties of an individuals I, whose name is passed as parameter.
- `void register(String pred, String subj, String obj)` - Used by a context provider to register at CMS and provide some context data regularly, in the form of a RDF tuple.
- `void include(ArrayList L)` - Used by a context provider to send a list of context data, as an array list of RDF tuples, to be included in CMS database.
- `void remove(ArrayList L)` - Used by a context provider to send a list of context data, as an array list of RDF tuples, to be removed from CMS database.

PUC-Rio - Certificação Digital Nº 0510970/CA

– `void update(String pred, String subj, String obj)` - Used by a context provider to update a specific context data piece, in the form of an RDF tuple.

7.4 The Decentralized Reasoning Service (DRS)

The Decentralized Reasoning Service (DRS) was implemented to provide reasoning services for application clients, not only in synchronous mode (queries) but also in asynchronous mode (publish/subscribe interactions), according to the design strategies enumerated in Section 4.3. It relies on the CMS server to access context data and monitor context data changes, and is capable of reasoning about rules provided by client applications.

PUC-Rio - Certificação Digital Nº 0510970/CA

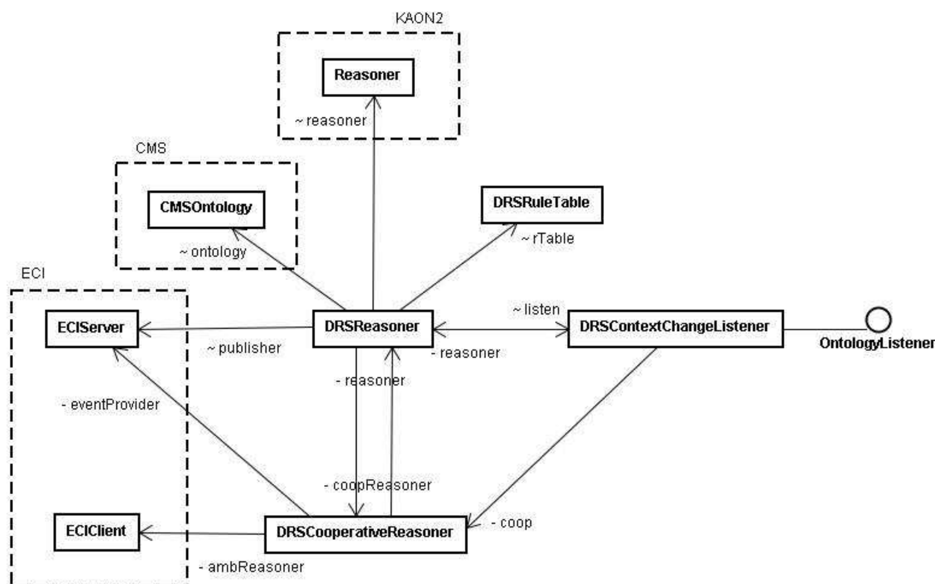

Figure 7.2: Class diagram showing the implementation of the DRS server.

DRS was implemented in Java (version 1.6.0) as a multi-threaded server supporting both synchronous and asynchronous (event-based) communication, using either TCP or UDP protocols. DRS uses *CMS* API to access the corresponding ontology and KAON2 API to implement the reasoning operations over the context ontology database. Besides that, it uses MoCA's communication API [35] to implement event-based communication. Figure 7.2 depicts the class dependency showing a simplified class diagram of the DRS service implementation. The main class of the service is the *DRSReasoner*, which implements the communication and the server loop to cope with messages received from client applications. This class relies on class *CMSOntology* to access data from specific ontology, class *ECIServer* to provide an event-based interface to client

applications and the *peer reasoner*, class *ECIClient* to subscribe at the *peer reasoner* and class *Reasoner* to perform reasoning operations. While synchronous queries are immediately managed by DRS, subscriptions require the use of specific data structures for keeping the information associated with each active subscription. Class *DRSRuleTable* is used to keep all information related with a received subscription, such as the associated rule and the latest result found. Class *DRSCooperativeReasoner* is responsible for managing operations related with rules that are not local, such as *pre-evaluation*, *forwarding* and *update* (discussed in Subsection 5.1.2). Finally, class *DRSContextChangeListener* is used to trigger the *reevaluation* of any rule associated with context data that was subject of changes. It is implemented extending the *OntologyChangeListener* interface, part of the KAON2 API.

When executing the DRS server, after start up it loads up a configuration file ("drs.properties"), which defines the IP address and protocol (UDP or TCP) for running the server, assigning different ports for receiving synchronous queries and for receiving subscriptions.

The DRS Client API is provided to facilitate the work of developers who want to implement client applications that use the inference services implemented by DRS. Using this API, a client application may interact with a DRS server to check a rule (synchronously), or to post or remove a subscription. The *DRSClient* class implements the methods describe ahead.

- `ArrayList checkRule(DRSRule R)` - Checks the result of a query for the rule R, passed as parameter, in a synchronous interaction. The result is an array list of RDF triples corresponding to binary or unary facts correlating individuals that satisfy the rule.
- `subscribe(DRSRule R, EventListener e)` - Subscribes at DRS, with a request to be notified about the result of the rule R passed as parameter, together with a listener that represents a callback routine.
- `unsubscribe(DRSRule R, EventListener e)` - Removes the subscription related to rule R.

In the next section we exemplify how these methods can be used to build a prototype application.

PUC-Rio - Certificação Digital Nº 0510970/CA

PUC-Rio - Certificação Digital Nº 0510970/CA

7.5 Prototype Application

To show how DRS may be used to support the implementation and execution of a ubiquitous application, we present the main steps of the design of a prototype application. We chose to implement a simplified version of the application proposed in our scenario, the Conference Companion (ConfComp), which was discussed in Section 2.1. This application aims to help the user with his agenda during a conference event and to stimulate the collaboration and social interaction with other researchers attending the event, by helping the user to locate people with similar interests.

Figure 7.3: Conference Companion icon at the Windows toolbar and the pop-up menu.

ConfComp is a simple application that — after started and configured — runs in background, occasionally providing notifications for the user. Figure 7.3 shows the icon of the application at the Windows toolbar and the menu that pops up when the user clicks the right button of the mouse with it over this icon. In the menu we see the options "Exit", "About", "Configure" and "Pause". If the "Exit" option is selected, the program is terminated. The selection of the "About" option causes a window showing information about the program to pop up. The option "Configure" shows a window that allows the user to select the sessions of the conference in which he is interested. After that, each time a session in which the user is interested is about to start, the application shows a pop up window warning the user about the event.

In order to demonstrate how the use of the DRS service and APIs simplify the design of ubiquitous applications, we will discuss some aspects of the implementation of the prototype related to the use of context and the inference services. As such, the implementation of the user interface will not be in the scope of this text. Figure 7.5 shows a block diagram of the prototype system, where applications interact directly with the middleware services in the same

```
1   try {
2     client = new CMSClient(cmsServerIp, cmsServerPort, localIp, localPort, CMSConstants.UDP);
3     ArrayList act = client.getIndividualsOfClass("activity");
4     if (act.size() > 0) {
5       for (int i = 0; i < act.size(); i++) {
6         String at = (String) act.get(i);
7         System.out.println("activity(" + at + ")");
8       }
9     } else {
10      System.out.println("no individual found...");
11    }
12  } catch (CMSException ex) {
13    Logger.getLogger(ConfComp.class.getName()).log(Level.SEVERE, null, ex);
14  }
```

Figure 7.4: Code snippet showing the query to get all "activities" from CMS.

side. In our example, ConfComp sends context queries (about activities) and updates (about the user's preferences) to the local CMS and subscribes at the local DRS for having a rule inferred. The local services are in charge of interacting with the remote services.

PUC-Rio - Certificação Digital Nº 0510970/CA

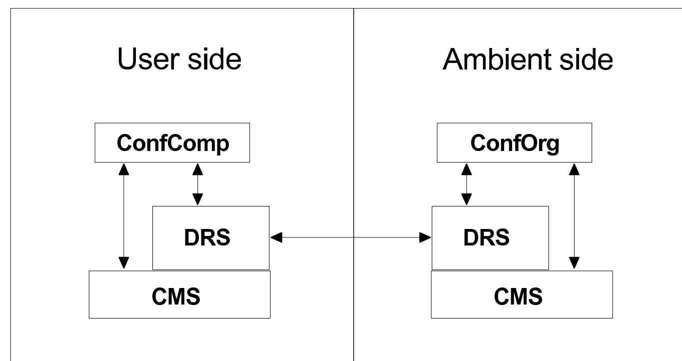

Figure 7.5: Block diagram representing the interaction among the applications and the middleware services DRS and CMS on the *user side* and the *ambient side*.

The first step performed by this application is to query the CMS running at the *user side* to get a list of activities, i.e., scheduled conference sessions. Figure 7.4 shows the piece of Java code that corresponds to this operation. At Line 2 an object *client*, instance of the class *CMSClient*, is created, having as parameters the IP and port of the CMS server and the application, and the communication protocol. At Line 3 we can see a call to the method *getIndividualOfClass* in which the parameter "activity" is used to recover all individuals of such class (e.g., *Session_1*, *Session_2*, etc). At Line 12 we see the statement for catching the *CMSException*, that may be thrown if the communication with CMS fails.

When the user wants to set the list of sessions that he wishes to attend,

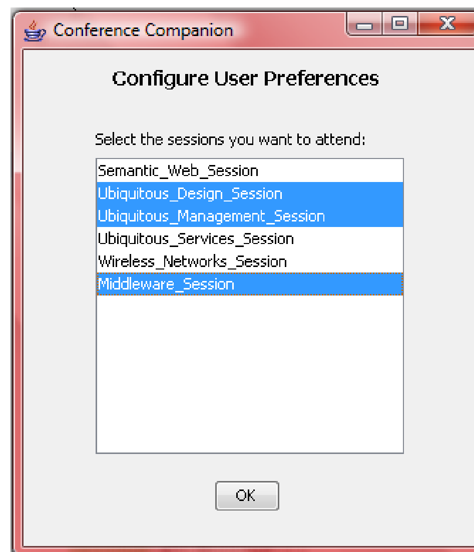

Figure 7.6: GUI for the user to set the list of sessions he wants to attend.

he has to select the "Configure" option of the pop-up menu and the window depicted in Figure 7.6 will be shown. In this window, the user must select the names of the sessions that are of his interest and press the "Ok" button. The application will send this information to local CMS. Figure 7.7 shows the Java code that performs this operation. From Lines 1 to 6 an array list is created containing the new facts to be added to the ontology in the local CMS. These facts are of class *CMSAtom*, that describe ontology facts as RDF tuples, with a predicate, a subject and an object. At Line 8 we can see the invocation of the method *include* in the *CMSClient* object, which had been previously instantiated (Fig. 7.4, Line 2). The parameter for this method is the variable *pref*, an array list of *CMSAtom* objects representing facts $wantsToAttend(Silva, Session_i)$, where $Session_i$ is each session indicated by the user. *CMSException* may be thrown if the communication with CMS fails.

```
1   ArrayList pref = new ArrayList();
2   try {
3     CMSAtom a;
4     for (int i = 0; i < o.length; i++) {
5       a = new CMSAtom("wantsToAttend", "Silva", (String) o[i]);
6       pref.add(a);
7     }
8     client.include(pref);
9   } catch (CMSException ex) {
10    Logger.getLogger(ConfComp.class.getName()).log(Level.SEVERE, null, ex);
11  }
```

Figure 7.7: Code snippet showing an update of data in the CMS.

After the user has set the list of sessions, the next step for the application is to subscribe at the DRS running at the *user side*, providing a rule to be

PUC-Rio - Certificação Digital Nº 0510970/CA

monitored by the reasoner. For our application, Rule 7.1 below describes the situation in which "a session that the user wants to attend is about to start and he is outside the respective room".

Rule 7.1:

$$isInterestedIn(\text{"Silva"}, ?s) \wedge takesPlace(?s, ?r) \wedge isAboutToStart(?s) \wedge$$
$$isLocatedIn(\text{"Silva"}, ?t) \wedge isDifferentFrom(?t, ?r) \Rightarrow isStartingIn(?s, ?r)$$

Figure 7.8 shows the Java code used to create a rule object corresponding to Rule 7.1. At Line 1 a new *DRSRule* object is instantiated. From Lines 2 to 6 new atoms are added to the antecedent of the rule, each corresponding to one of the five atoms presented in Rule 7.1. At Line 7 the consequent of the rule is defined.

```
1  DRSRule R = new DRSRule(2);
2  R.addAtomtoAntecedent(new DRSAtom("wantsToAttend", new DRSTerm(1, "Silva"), new DRSTerm(0, "S")));
3  R.addAtomtoAntecedent(new DRSAtom("takesPlace", new DRSTerm(0, "S"), new DRSTerm(0, "R")));
4  R.addAtomtoAntecedent(new DRSAtom("isAboutToStart", new DRSTerm(0, "S")));
5  R.addAtomtoAntecedent(new DRSAtom("isLocatedIn", new DRSTerm(1, "Silva"), new DRSTerm(0, "T")));
6  R.addAtomtoAntecedent(new DRSAtom("differentFrom", new DRSTerm(0, "S"), new DRSTerm(0, "T")));
7  R.setConsequent(new DRSAtom("isStartingIn", new DRSTerm(0, "S"), new DRSTerm(0, "R")));
```

Figure 7.8: Code snippet showing the description of a rule.

Figure 7.9 shows the Java code used to subscribe at a DRS, having the rule R to be monitored as a parameter. At Line 2 an object *reasoner*, instance of the class *DRSClient*, is created, having as parameters the IP and port of the DRS server and the application, and the communication protocol to be used. At Line 3 we can see an invocation of the method *subscribe* of the object *reasoner*, in which the parameters are the *DRSRule* object R and a *EventListener* object *MyEventListener*. As a result, this rule is sent to DRS as a subscription and any notification will trigger the event listener *listener*, which will cope with the received result. At Line 4 a *DRSException* exception is caught if the communication with DRS fails.

```
1  try {
2    reasoner = new DRSClient(drsServerIp, drsServerPort, localIp, localPort, DRSConstants.UDP);
3    reasoner.subscribe(R, MyEventListener);
4  } catch (DRSException ex) {
5    Logger.getLogger(ConfComp.class.getName()).log(Level.SEVERE, null, ex);
6  }
```

Figure 7.9: Code snippet showing the subscription.

PUC-Rio - Certificação Digital Nº 0510970/CA

As a mean of notifying the user, the application shall pop up a window with a message warning the user about the session that is going to start, so that he can go to the respective room where the activity will take place. Figure 7.10 shows the Java code that describes the class *MyEventListener*, which implements the interface *EventListener* defining the action to be taken when an notification arrives from the DRS.

```
1   public class MyEventListener implements EventListener {
2     public void onReceiveData(Event receivedEvent) {
3       DRSReply q = (DRSReply) receivedEvent.getData();
4       ArrayList matches = q.getMatches();
5       for (int i = 0; i < matches.size(); i++) {
6         DRSAtom atom = (DRSAtom) matches.get(i);
7         String m = "User should go to "+atom.getObj().getTerm()+" now to attend "+atom.getSubj().getTerm()";
8         JOptionPane.showMessageDialog(null,m,"Conference Companion",JOptionPane.WARNING_MESSAGE);
9       }
10    }
11  }
```

Figure 7.10: Code snippet showing the implementation of the event listener.

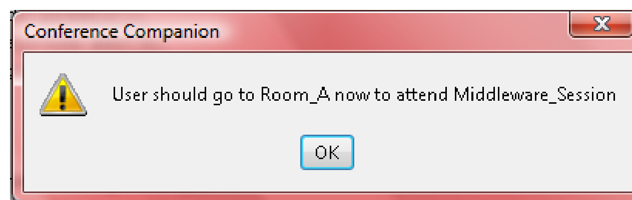

Figure 7.11: Window that pops up to warn the user that a session he wants to attend is going to begin.

The notification triggers the *listener*, having an *Event* object as parameter. At Line 3 of Figure 7.10, the method *getData* is used to get the content brought in the *Event* object. This content is a *DRSReply* object, which contains an arraylist of atoms, each corresponding to a fact in the ontology database that satisfies Rule 7.1, i.e., assertions in the form *isStartingIn(?s,?r)*. At Line 6, an atom from the list is selected and, at Line 7, the subject and object of each atom representing a binary property assertion — e.g. *isStartingIn(MiddlewareSession, Room_A* — are used to compose the message to be displayed to the user. At Line 8, we show the window described in Figure 7.11. This warning message will pop up each time a new notification about an activity arrives at the client application.

PUC-Rio - Certificação Digital Nº 0510970/CA

7.6 Discussion

In this chapter, we described our prototype implementations of the Context Model Service (CMS) and the Decentralized Reasoning Service (DRS). The CMS is the service responsible for collecting context data from context providers available in a specific domain, keeping an updated representation of the assembled data according to a valid context model (an ontology), and providing access to up-to-date context information. The DRS is the service that implements the *cooperative reasoning process*, providing reasoning services for application clients in synchronous mode (queries) and in asynchronous mode (publish/subscribe interactions).

To be used in real world AmI scenarios, dealing with the dynamic and heterogeneous characteristics of such environments, these services should be executed on top of a more complex middleware architecture, capable of providing complementary functionalities such as service discovery [112], or support to semantic interoperability [113, 114]. In the absence of such services, we greatly simplified the model of our system, assuming that all entities shared a same context model and the DRS and CMS servers had well-known communication addresses.

CMS and DRS were implemented using the KAON2 reasoning API to access ontology data and perform reasoning operations. Although our implementation of the services has a small memory footprint — 20.2 KBytes —, as KAON2 was available only for J2SE environment, it was not possible to implement our services targeting mobile devices, which execute only J2ME based applications. We believe, however, that in the future these implementations may be ported to the mobile environment. In this case, the interfaces provided by our CMS and DRS APIs will not be modified, and the implementation applications for mobile devices will follow the same model discussed in Section 7.5.

A programmer who wants to design ubiquitous applications will have his work facilitated by the CMS and DRS services and APIs, as he will be able to use rules as an abstraction to describe the situations of interest for his application. On the other hand, in a system where the context model is not as simple as the one presented in our scenario, formulating the necessary rules may be a hard task for the programmer, demanding a great knowledge about the context model of the target system and some acquaintance with description logics. In this case, tools or interfaces that help the programmer to formulate and validate these rules would be an important complement to be developed and added to DRS.

PUC-Rio - Certificação Digital Nº 0510970/CA