Towards the Model Driven Development of context-aware pervasive systems

Estefanía Serral*, Pedro Valderas, Vicente Pelechano

Centro de Investigación en Métodos de Producción de Software (ProS), Technical University of Valencia, Camí de Vera s/n, E-46022, Spain

ARTICLE INFO

Article history:
Received 28 February 2008
Received in revised form 30 May 2009
Accepted 1 July 2009
Available online xxxx

Keywords:
MDD
Pervasive systems
Context-aware
Ontology
OWL

ABSTRACT

In this work, we introduce a Model Driven Development method for developing context-aware pervasive systems. This method allows us to specify a context-aware pervasive system at a high level of abstraction by means of a set of models, which describes both the system functionality and the context information. From these models, an automated code generation strategy is applied. This strategy allows us to generate the system Java code that provides the system functionality and as well as an OWL specification that represents the context information and allows us to manage this information without additional burden. Furthermore, this specification is used by a reasoner at runtime to infer context knowledge that is not directly observable, and it is also used by machine learning algorithms to give support to the system adaptation according to the context information.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

Context-awareness is an essential characteristic to make Weiser’s vision a reality. In this vision, pervasive computing is a technology that “weaves itself into the fabric of everyday life until it is indistinguishable from it” [1]. This means that pervasive computing must be transparent to users. This transparency is only possible if user intervention is sought only when it is absolutely required. To achieve this, the system must be able to adapt itself without explicit user intervention and automatically execute tasks for users in a non-intrusive way.

The concept of context thus plays a main role in the development of pervasive systems. Context involves information that is related to a myriad of aspects such as users, environment, temporality, locations, etc [2,3]. Knowing these aspects in detail is crucial to provide the flexibility and adaptability that pervasive systems need in order to adapt themselves to changing conditions and dynamic environments.

One of the current problems is that most approaches for the development of context-aware pervasive systems usually use ad-hoc solutions; some of these examples are [4,5]. This type of development forces developers to work at a lower level of abstraction by directly programming devices or networks to control them. It makes developers implement the system without having a clear vision of the context which the system must interact with. In addition, if we consider that pervasive systems are characterized by a continuous evolution of hardware and software, the use of ad-hoc solutions makes maintenance and further adaptation extremely difficult.

Therefore, we think that new generations of approaches that allow developers of context-aware pervasive systems to work at a higher level of abstraction are needed. In this context, Model Driven Development (MDD) [6] seems to be the path to be followed. In the same way that the first FORTRAN compiler was a major milestone in computer science because, for
the first time, it lets programmers specify what the machine should do rather than how it should do it, MDD is a step beyond, in which software is developed not by directly writing code in implementation languages, but specifying the system using high-level abstraction models that can be transformed into code by automated code generations. By following the MDD guidelines, numerous benefits such as high productivity, high quality, easy maintenance and quick adaptability are provided. These benefits will be further explained in Section 8.

Thus, the main contribution of this paper is a MDD method for developing context-aware pervasive systems in which context is considered to be a first-order citizen. Besides, automatically obtaining the code of the system, this method generates an ontology to manage and reason about context at a semantic level without additional burden. To do this, the method provides us with:

- PervML, which is a modelling language that allows us to represent context-aware pervasive systems at a high level of abstraction. To give support to context modelling, we have identified the context information. Then, we have defined a graphical notation to represent this information in PervML models.
- A code generation strategy which is based on Model Driven Architecture (MDA) [6] and Software Factories (SF) [7]. This generation strategy obtains:
  - Java code that provides the different services that the pervasive system must offer to users. This code is based on the Open Services Gateway initiative (OSGI) [8] platform.
  - Ontology Web Language (OWL) [9] code that allows the system to infer knowledge from the context at runtime and adapt itself accordingly. We give support to this inference and adaptation with an OWL-based ontology. This ontology represents the context information and contains a set of rules to reason about it. It is used by a reasoner and machine learning algorithms that allow the system to learn from system interaction in order to further adapt itself to better support this interaction or, even to automate this, for instance, by anticipating user actions.

In the context of MDD, there are approaches such as [10–12] that develop context-aware pervasive systems. Most of these approaches focus on modelling the context data and properly manage this [10,12] but only few of them deal with the development of functional context-aware pervasive systems [11]. Moreover, none of them support automatic and complete code generation from models. We provide a more detailed overview on the related literature in Section 9.

The rest of the paper is organized as follows: Section 2 presents a detailed description of the concept of context as well as the main characteristics that a context-aware pervasive system must have. Section 3 explains the proposed method. Section 4 presents the PervML models that take as running example the case study presented in Section 8. Sections 5 and 6 explain the code generation strategy in detail. Section 7 explains the strategy to infer new context information. Section 9 presents the related work. Finally, conclusions and further work are explained in Section 10.

2. The concept of context

In this work, we use the definition of context proposed by Dey [2]. Dey defines context as “any information that can be used to characterize the situation of an entity”, and an entity as “a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves”. In order to identify this information, we have analyzed several proposals and finally we have based our approach on the SOUPA ontology [3]. SOUPA can be considered as one of the most influential published ontology model for supporting knowledge sharing, context reasoning and interoperability in pervasive systems. According to this ontology, the context information is made up of:

(1) system users (personal and contact data about system users, such as name, age, address, native language, etc.); (2) user preferences; (3) space information; (4) system services; (5) privacy and security policies that indicate what operations each user can execute; (6) temporal information (date and time, holiday, working day, etc.); (7) services available for a user in the current time; (8) user mobility; (9) user actions (what the user is doing at present moment and what the user has done in the past).

In the above context information, we can identify two types:

1. Those that are available when the system is being specified (information about users, user preferences, space information, system services, and privacy and security policies, etc.);
2. Those that are only available at runtime (temporal information, service state and available services, user mobility, and actions that users perform).

In order to properly support the context information introduced above (both the information available at design time and the information available at runtime), a context-aware pervasive system must have the following characteristics [2]:

(1) presentation of information and services to a user (the system must provide mechanisms that allow user to properly access the information and functionality provided by the system); (2) tagging of context to information to support later retrieval (the system must process the context information to allow an efficient access to it); (3) automatic execution of a service operation for a user: the system must be able to analyze user behavior and changes in the context to automatically execute operations when required.

The approach to develop context-aware pervasive systems that we introduce in this work already allows us to develop systems that have the first two features. In Section 7, we introduce some details about the strategy to be able to develop systems to support the last feature (3), which has been planned as further work.
3. A MDD method for context-aware pervasive systems development

The method that we propose to develop context-aware pervasive systems applies the guidelines defined by the MDA, which is supported by the Object Management Group (OMG), and the Software Factories, which is supported by Microsoft. Thus, the method provides the following elements:

- A domain-specific Pervasive system Modelling Language (PervML) for describing the system functionality in an abstract way independent of how this functionality is internally implemented. This language defines a set of models for specifying context-aware pervasive systems using conceptual primitives that are suitable for this domain.
- A transformation engine and an implementation framework to automatically translate the PervML models into the system Java code. The framework raises the abstraction level of the target platform to facilitate the transformation from PervML to Java/OSGi Code.
- Finally, the PervML Ontology that defines the concepts introduced in PervML and a transformation engine to automatically translate the PervML models into the system OWL specification according to the ontology. Thus, the system information is stored in an OWL specification that is updated at runtime according to the context data. This OWL specification is used for adaptation purposes.

Therefore, taking into account these elements, the MDD process is divided in two main phases: development and deployment. Next, we explain each of these phases in detail.

3.1. The development phase

The development phase obtains the code to put the system into operation. This phase consists in the following three steps (see Fig. 1):

Conceptual modelling: Firstly, the system is specified by means of PervML. This specification consists in the modelling of a set of models defined by PervML. The context information available at design time is captured by these models. They describe: the system users, the system services, which services are available for each user, and the devices that give support to the system services. These models are explained in detail in Section 4.

Code generation: From the specified PervML models, two model transformations are executed in order to translate them into both Java code and the OWL specification.

The first transformation engine automatically transforms the PervML models into Java code. This Java code consists of Java files and Manifest files that implement the functionality that supports the system services. This code is based on the implementation framework that provides a common architecture for all the systems that are developed using the method. This transformation is explained in detail in Section 5.

The second transformation automatically transforms the PervML models into the OWL specification, which describes the context aware pervasive system by using concepts of the PervML ontology. This ontology and the transformation are explained in detail in Section 6. The OWL specification is continually updated at runtime by the implementation framework according to the changes produced in the system. In addition, OWL enables automated reasoning by means of reasoners, hence, the OWL specification contains rules to infer context knowledge at runtime by using a reasoner. This knowledge is used by machine learning algorithms for supporting the adaptation to context changes and the automatic execution of service operations for users. This strategy for inferring knowledge and adapting the system to the context data and the user behavior is explained in Section 7.
Driver implementation: Finally, drivers for managing the selected devices or software systems have to be implemented. They are developed by hand, since they deal with technology-dependent issues. However, if any device or external software system has been used in a previous system, the same driver can be reused; thus, we can have a driver repository and only implement the drivers for the devices or software systems that have not been used before. Moreover, we have implemented a European Installation Bus (EIB) driver generator [13] to generate the EIB driver code from a simple description.

3.2. The deployment phase

The development phase provides us with the entire context-aware pervasive system code: the Java code to provide system functionality, and the OWL specification and the code for adaptation purposes.

We also use an OSGi server to run the system. OSGi is an open standard for service gateways created by the Open Service Gateway Initiative. Thus an OSGi server (a service gateway) is the platform where the software for providing services resides. It manages the devices and it communicates with external networks and defines Java APIs and several standard services like Logging, HTTP Server, Device Management, etc.

Thus, to be started, the system is deployed in the environment. Three steps (see Fig. 2), must be performed to do this:
Configuration: The Java files that use the manually implemented drivers are associated with the selected drivers. To do this, we just need to set up the driver identifiers.
Bundle installation: The files generated from the PervML to Java code transformation are compiled, packaged into bundles (JAR files) and deployed in the OSGi server along with the framework and the drivers. The reasoner and the machine learning algorithms are also compiled, packaged and deployed in the OSGi server. The generated OWL specification must be copied in the folder where the OSGi server is located.
Starting bundles: Finally, the installed bundles are started in the OSGi server. Thus, the services are already available to users who can execute them through the interfaces [13] that are provided by the implementation framework (as explained in Section 5.1).

4. PervML models and the support for the context information

PervML is a domain-specific modelling language for specifying pervasive systems using conceptual primitives (Service, Trigger, Interaction, etc.) that are suitable for this domain. This language allows us to specify context-aware pervasive systems in a platform and technology independent way. As Fig. 3 shows, PervML promotes the separation of roles in Pervasive System Analysts and Pervasive System Architects. Using this division, which is embedded in the language, different developer roles can naturally work in their expertise fields. Thus, every role must perform certain activities, which are next described.

On the one hand, Pervasive System Analysts capture systems requirements at a high level of abstraction. The analysts construct four graphical models (the Service Model, the Structural Model, the Interaction Model and the User Model), which constitute what we call the Analyst View. In these models, the analyst describes (1) the types of services that the system must support (by describing their interfaces, the relations among services and a state transition diagram that depicts the behavior of each type of service); (2) the different services of each type that the system provides; (3) how these services interact with each other; and (4) privacy policies as well as personal and contact data of each homeowner.

On the other hand, Pervasive System Architects specify which devices and/or existing software systems support the system services defined by Analysts. For instance, the Lighting Service is provided by one or more lamps. We refer to these elements (devices and software systems) as binding providers because they bind the pervasive system with its physical or logical environment. Therefore, the system architect constructs three other models (the Binding Provider Model, the Component Structural Specification, and the Functional Model), which constitute the Architect View. In these models, the
system architect describes (1) the types of binding providers (by describing its interface), (2) the binding providers that are used in each component, and (3) the actions that must be carried out when each operation of each component is invoked. All context information available at design time is captured by Pervasive System Analysts. Hence, Section 4.1 explains the PervML models of the Pervasive System Analysts view in detail, remarking how this language gives support to the context data. A more detailed description of Pervasive System Architect view can be found in [14]. Section 4.2 introduces the developed tool that gives support to the PervML models specification.

4.1. Pervasive System Analyst view

In this section, we present the models included in the Pervasive System Analyst view. In order to illustrate examples of these models, we have used a case study for a Smart Home, which is explained in detail in Section 8.

4.1.1. Service Model

The Service Model describes the kinds of services that the pervasive system provides. A service is an entity that provides a coherent set of functionality which is defined in terms of atomic operations. Note that a service can be provided by several instances in pervasive systems. For example, in the smart home, the lighting service is described once, but it is provided by several instances in the different locations of the house. Thus, the Services Model is used to specify the common characteristics of these similar services. The analyst must specify the following information in order to complete the description of the system services:

- **A UML-like class diagram** for identifying the types of service, their operations and their relationships. Each type of service is defined by means of a UML [15] class that is stereotyped with the keyword “service”. Fig. 4 shows the Service Model for the case study. For instance this figure shows that the system provides services such as Alarm, Heating and BlindManagement.

  Also, inheritance and aggregation relations can be established between services:

  - One service can aggregate other services. This means that the service needs to use the functionality provided by other services. For example, as Fig. 4 shows, the **LightingByPresenceAndIntensity** service uses the functionality from the **PresenceDetection** and **LightIntensity** services to know the context information about the inhabitant’s presence and the intensity of the room. Thus, this service regulates the light of the room according to this information.

  - One service can inherit the functionality from another service. This means that the child service will have the parent service functionality as well as own functionality. For instance, as 4 shows, the system provides services such as a **Lighting** service, which offers operations for switching on and switching off the lights, and a **GradualLighting** service, which extends the general functionality provided by the **Lighting** service and offers operations for setting the light intensity.

- **Pre and post conditions** for every service operation. These constraints, which are expressed using the Object Constraint Language (OCL), help to describe the semantics of the operation. For instance, Fig. 5 shows a constraint expressed over the **play** operation of the Player Service. This constraint specifies that a media file must be introduced before the operation is executed. The current mode of the service is changed to PLAYING as a result of the execution of this operation.

- **A State Transition Diagram** for every kind of service. This diagram indicates the service operations that can be invoked at a specific moment. For instance, Fig. 6 shows the State Transition Diagram associated to the Player kind of service. This diagram specifies that the initial state is STOPPED.


Three operation can be invoked from this state (outgoing transition): **play()**, **getCurrentMode()**, and **setMediaFile()**. The operation **play()** changes the state of the service to PLAYING. Five operations can be invoked from this state: **forward()**, **rewind()**, **getMediaFile()**, **getCurrentMode()**, and **stop()**. This last operation returns the state to STOPPED.
Triggers attached to a kind of service. The analyst can specify condition/action rules by using OCL. These rules perform the actions when the conditions are accomplished. The trigger shown in Fig. 7 is linked with the Security service type. This trigger is executed when the Security service is enabled and presence is detected or a door or window has been opened in the house, it is in charge of recording the house, activating the alarm services, and also of sending a warning email to inform users.

The services specified in the Services Model can be reused in the specification of other systems, since several services can be the same. For instance, the Lighting service is present in every smart home so we provide a services’ repository so that the system analyst only has to specify the services of the system that are not already in the repository.

4.1.2. Structural Model

The system analyst uses the Structural Model to indicate which services provide the types of service defined in the Services Model. Furthermore, we also indicate in which physical location these services are deployed. As far as the context information, this model allows us to capture information related to the locations and the services that are available at each location. To do this, the structural model is defined from two elements:
• A Location Diagram, which describes the different areas where the user can move or where services can be located. This is specified by means of a UML package diagram. Each package represents a certain area, and the hierarchy between packages symbolizes the space hierarchy between the areas represented by those packages. Also, two types of associations can exist between the areas or packages: adjacency and mobility (or accessibility). Adjacency means that the areas are next to each other, whereas mobility is the possibility to go from one area to another. Adjacency is represented by a line between two areas. Since mobility implies adjacency, it is represented by adding arrows to the line between two areas. Thus, this model allows us to infer information such as transitive relations. For example, if we can go to the corridor from the kitchen and to the bathroom from the corridor, then we can go to the bathroom from the kitchen.

Fig. 8 shows an example of the Location Diagram. It models the locations of the running example. It has seven areas: Hall, LivingRoom, Kitchen, Corridor, Bathroom, ChildRoom and ParentsRoom. This figure shows that, for instance, the Hall and the Bathroom are adjacent, but a user cannot go from one to another. In contrast, the Hall and the LivingRoom are adjacent and a user can go from one to another.

We use a UML package diagram for representing locations because it is very intuitive for the analyst, as the Fig. 8 shows, packages can contain other packages like locations can contain other locations. Also, UML components (which represent services as the following diagram shows), are deployed in packages, like the real services, which are provided in locations. This allows us to describe any space easily by simply delimiting areas and relate them intuitively.

• A UML 2.0 component diagram, which represents the components that are deployed in each physical location as well as the type of service that is provided by each component. To do this, the components are represented by means of UML 2.0 components and are associated to package that represents the location where they are deployed. The service that provides each component is depicted as a UML interface. Also, relationships between components indicate a relation of use, in other words, the source component uses the functionality of the target component.

Fig. 9 shows a partial view of the Structural Model for the running example. It shows the components associated to the ParentsRoom. We have defined several components such as the ParentsLighting that provides the Lighting service, which is used by the ParentLightingControl; and the ParentsBlind that provides the BlindManagement service.

Please cite this article in press as: E. Serral, et al., Towards the Model Driven Development of context-aware pervasive systems, Pervasive and Mobile Computing (2009), doi:10.1016/j.pmcj.2009.07.006
4.1.3. Interaction Model

This model is used to describe the communication that is produced as a reaction to a system event. An interaction is a communication between components to provide a specific functionality. Every interaction is described by an adapted UML Sequence Diagram; therefore, the Interaction Model is composed of several sequence diagrams. To describe an interaction, the analyst identifies the components of the Structural Model that participate in the interaction, defines the sequence of messages that the components must interchange, and specifies the condition (defined over property values) that triggers the interaction. The triggering condition of the interaction is specified using OCL [15]. The actions described in the diagram must be executed when the condition is satisfied.

The Systems Analyst may want the action to be ordered to every Component that provides a type of service. In this case, the Analyst can indicate that the object that receives the action is a service, with the label \langle\langle Service\rangle\rangle, instead of every component that provided it. Fig. 10 shows the interaction that is in charge of lowering every blind, winding up every awning and stopping the garden sprinklers when it starts to rain. It is also important to note that the actions specified in an interaction are not executed as an explicit user request, but are invoked as a reaction to a situation.

4.1.4. User Model

The User Model is used to specify the context data about users and policies. It is defined from two elements:

- A Policy Diagram, which is used to specify the policies of the system. A policy describes a type of user by defining its allowed operations. Thus, a policy is associated with users (as Fig. 11 shows the Father policy is associated with the Peter user), which means that these users can perform the operations specified in the policy. Then, in order to associate the allowed operations to each policy, the analyst can: (1) associate a service (then every operation of every component that provide that service will be allowed); (2) associate a component (then every operation of this component will be allowed); (3) associate a service operation (then this operation will be permitted for every component that provides this service); or (4) associate a component operation (then this operation of this component will be permitted).

Furthermore, inheritance relations can also be established in this model. These relations allow us to define capacities of a policy taking the capacities of a defined policy as a basis. This can be done in two ways: (1) By adding new capacities to capacities of the parent policy. This is used if the child policy can execute more operations than the parent policy. The actions that the child can execute but the parent can not are denoted by “+”.

For instance, in the example of Fig. 11 the parent can execute operations related to the Lighting, GradualLighting and BlindManagement services. The child can execute the same operation except the operations of the BlindManagement service.

- A set of User Characterization Templates, which specify the users of the system. Pervasive System Analysts must indicate the following information for each user:
  - The policy associated to the user, which has been previously specified in the Policy Diagram.
  - The following personal data: name, surname, gender, date of birth and marital status.
  - The following contact data: email, telephone number, mobile phone and address.
  - Social relations, i.e., information related to people that the user knows.

Both Personal and Contact data are those proposed by SOUPA to characterize users. We allow analysts to add new properties if required. Fig. 11 shows an example of User Characterization Templates. This example represents the Peter user in the system, whose policy is Father.

Finally, note that this model provides support for the privacy, the security and the views of the system, since users will be able to see and execute only system actions that they are authorized to use. On the other hand, the construction of this model
is optional. The model must be defined only in the case that a personalized system behavior must be provided. If Pervasive System Analysts do not create this model, three policies are defined (with a default user for each one): (1) the administrator, who can execute all the operations available in the system including configuration operations; (2) the limited user, who is able to execute all the operations available, except configuration operations; and (3) the guest, who can only consult the state of the system, but not execute any operation that modifies it.

4.2. Tool support

Model driven methods must be supported by tools in order to be applicable in an effective way. Thus, we have developed the PervML Generative Tool (PervGT), which supports the presented model driven method for the development of context-aware pervasive systems. The tool is based on the Eclipse platform [16]. Eclipse is a flexible and extensible platform with many plug-ins that provide functionality for specific purposes, such as the creation of modelling tools or code generation. Mainly, we have used two plug-ins from the Eclipse platform: the Eclipse Modelling Framework (EMF) plug-in and the Graphical Modelling Framework (GMF).

EMF is a modelling set of tools and code generation facilities for specifying metamodels and managing model instances. We have used EMF to first create the PervML metamodel and next to automatically generate the following from this metamodel: (1) A set of Java classes representing each one of the PervML metamodel concepts; these Java classes provide methods to modify PervML models according to the PervML metamodel. (2) A basic tree editor that facilitates the development of PervML models according to the metamodel.

The GMF plug-in provides an editor to specify graphical editors in a declarative way, and also provides a runtime where common functionality related to graphical editors is already implemented, like model printing or automatic layout algorithms. We have used GMF to develop the PervGT graphical editor. Since PervML proposes different models, the PervGT tool provides a graphical editor for each one. For instance, Fig. 12 shows the graphical editor for the state transition diagram of the Service Model. In addition, we have used GMF in order to specify model checking constraints in OCL [15] that keep developers from creating models with syntactic or semantic mistakes.

Please cite this article in press as: E. Serral, et al., Towards the Model Driven Development of context-aware pervasive systems, Pervasive and Mobile Computing (2009), doi:10.1016/j.pmcj.2009.07.006
5. Java code generation

We have defined and implemented a model to Java code transformation to automatically generate the system Java code from the PervML model.

Following the MDA [6] guidelines, different transformation approaches can be selected to achieve this goal. One key point of these approaches is the number and nature of intermediate models, refined Platform Independent Models (PIMs) or Platform Specific Models (PSMs) [6], that must be produced while executing the transformation. In this work, we have selected a straight approach where a PervML model is directly transformed into OSGi bundles [8] in source code format (Java code and Manifest files) that constitute the pervasive system. This approach provides us with some advantages because only one model-to-text transformation is carried out. The advantages are:

- There are fewer assets (just one) to update and keep synchronized than in other approaches with intermediary models. Therefore, the potential modifications are isolated and completely focused. This characteristic implies that the debugging task is easier than with other approaches.
- The overall development time of the transformation is considerably reduced, since fewer assets need to be created and maintained.
- The resulting transformation is faster than the approaches that use PSMs since this approach does not uses intermediate steps or intermediate products.

However, the selected approach has one main drawback, too. The drawback is that the abstraction gap is dealt with only one step, so the transformation could be quite complex if the gap is wide. To solve this, we have followed the Software Factories guidelines. We have developed an implementation framework (which is explained in the next subsection), that increases the abstraction level of the target technology and, thereby, considerably reduces the abstraction gap.

Thus, as Fig. 13 shows, in our approach, first the system must be specified using PervML. Next, this specification is transformed automatically into the Java code that conforms the system and that extends the implementation framework code built on top of the OSGi middleware. We have selected the OSGi middleware as the implementation technology, since it has bridges to many of the technologies used in home automation systems and provides high-level implementation constructs. This middleware helps us significantly to fill the abstraction gap between the domain-specific language and the target implementation technology.

Next, we explain how the system Java code is obtained from PervML models. However, since the obtained code extends the implementation framework, we present it first.

5.1. Implementation framework for building context-aware pervasive systems

The proposed implementation framework raises the abstraction level of the target platform by providing similar constructs to those defined by the primitives proposed in the PervML models. Moreover, the framework encapsulates the common functionality and structure of the elements that are generated by the development method. Therefore, the amount of code that must be generated is significantly reduced.

The framework applies the Layers Architectural Pattern [17], which allows us to organize the system elements in layers with well defined responsibilities. Hence, in order to provide facilities for integrating several technologies (EIB networks, web services, etc.) and for supporting multiple user interfaces, the framework architecture has been designed in three layers:
The Driver Layer, which is in charge of managing the access to the devices and external software;

The Logical Layer, which is in charge of giving support to the generation of the system logic code. It was subdivided into three sub layers: the Communications sub-layer, which gives support to code generation about binding providers, the Services sub-layer, which provides the functionality that is required, and the Security sub-layer, which gives support to the code generation about users and policies. This sub-layer is also in charge of ensuring that a user only executes the operations that are allowed.

The Interface Layer, which manages the access to the system by any kind of client.

A more detailed description of these layers can be found in [18].

As far as context-aware characteristics of the system, the implementation framework gives support to (1) the presentation of information and services to a user and (2) the context information processing for later retrieval (that is, the updating of the OWL specification with runtime context information). In the following, we explain the layers that support these context-aware characteristics in detail. These layers are: the logical layer and the interface layer.

5.1.1. Logical layer implementation

The logical layer provides the set of classes that facilitates the generation of the logic of the system. To do this, these classes provide us with constructors that are similar to the PervML concepts such as service, binding provider, component, user, etc. These classes are extended in order to generate the final system. These classes define the attributes that must take value when the framework is instantiated and implement the execution strategies of each element using the Template Method pattern [17].

Fig. 14 shows a partial view of the classes of the logical layer as well as the different relationships among them. Classes in this layer can be classified in three functional groups:

- Classes for mapping the PervML conceptual primitives. This functional group is composed of five classes. The PervMLInteraction, PervMLService, and BProvider classes provide support to the Communication and Services sub-layers. For instance, they provide support to implement the binding providers that communicate the system with the physical devices. The PervMLUser and PervMLPolicy classes provide support to the Security sub-layer. These classes facilitate the code generation from policies and users specified in the user model. For instance, the User class provides mechanisms for checking security policies: when a system service is activated by a user, the corresponding User object checks if the user can activate the operation.

- Classes for encapsulating OSGi-related functionality. The goal of this group is to isolate some OSGi-related functionality that is inherited by the classes in the previous functional group. Classes in this group provide facilities for logging events (Logger), for participating in the event notification mechanism supplied by OSGi (WireParticipant) and for search services in the OSGi framework (FrameworkSearcher).

- Classes for dealing with the system life-cycle. This functional group is composed of the InteractionActivator, ServiceActivator, BProviderActivator, UserActivator and PolicyActivator classes. An activator is an OSGi concept which describes the class that is in charge of registering and unregistering the services in the OSGi framework when a bundle is started or stopped. In our case, the mechanisms for notifying and receiving notification of changes in the OSGi services (Wires in OSGi terminology) are created, too. Most of the functionality supplied by the activators is shared by five elements, so an abstract class (FrameworkActivator) has been implemented.

Updating Context Information at runtime. The PervMLService class of the implementation framework, which is the class from which every service inherits, is in charge of capturing all the changes in the context information produced at runtime. To do this, this class registers the information about each and every interaction in the OWL specification since any change in context information is produced by an interaction. This makes the approach flexible and powerful enough to facilitate the management and the processing of the context information. Thus, an interaction with the system can be due to either one of the following:

- A change in the environment. This interaction is detected by the sensors of the system. For instance, when a user goes into the parents’ room, a state change is produced in the ParentsDetection service (Fig. 9). As a consequence, the operation for checking the current state is activated (the presenceDetected operation).

- An explicit request by a user. This interaction is produced when a user executes a specific operation of a service (for example, the open operation of the ParentsBlind service) through the user interfaces that the system provides (for instance, the web interface for desktop browsers or for PDA browsers, shown in Fig. 2).

Thus, when an interaction is produced, the updateOWLSpecification method from the PervMLService class (Fig. 14) is executed. This method creates the corresponding OWL description of this change (an instance of the Action class and its related data: the date and time of that action performance, the executed operation, etc.) and inserts it in the OWL specification. This is further explained in Sections 6.1 and 7.
5.1.2. Interface layer implementation

The Interface Layer gives support to the presentation of information and services to users. In order to implement this layer, we have used the Model–View–Controller (MVC) pattern [17], which provides us with support for having different interfaces to interact with the system. By following this strategy, the components of the Services Layer correspond to the Model; the Controller class, which is reusable for every interface, has been implemented; and specific views must be implemented for each supported Interface. Thus, as well as implement the Controller class, we have implemented some views, for example, the corresponding to a Web Interface. Fig. 15 shows the Controller class and the classes that implement this Web interface. This Web interface has been implemented for access from desktop web browsers.

Please cite this article in press as: E. Serral, et al., Towards the Model Driven Development of context-aware pervasive systems, Pervasive and Mobile Computing (2009), doi:10.1016/j.pmcj.2009.07.006
The Controller class provides methods that allow users: (1) To select the component that is the specific service with which the user wants to interact. For instance, the method `getServicesFromLocation(String location)` returns all the services that are available for a user in a specific location. (2) To interact (get information and request functionality) with a single component. For instance, `ManageOperation(String ComponentPID, String Operation, Object []parameters)` is in charge of executing a Component operation using Java reflection capacities. These methods take the user identification into account. They only show the services and the operations specified in the policies of the User Model for the identified user.

Fig. 15 also shows the `ServicesListingServlet` and the `ServicesUIServlet` classes that implement Java Servlets. These classes provide the view of the web interface and invoke the Controller operations in order to generate the web pages that will be shown to users. The `ServicesListingServlet` class invokes the first set of methods, whereas the `ServicesUIServlet` class invokes the second set of methods. It is worth noting that every class in Fig. 15 is a concrete class. This means that classes do not need to be extended. In addition, these elements are reusable for all the pervasive systems that are developed using the proposed approach. This feature is feasible since (1) every `PervMLComponent` implements an interface that is known by the Controller and (2) the Java reflection capabilities have been used to invoke previously unknown methods.

5.2. From PervML to Java code

In this subsection, the transformation to obtain the system java code is explained. The input of the transformation must be a PervML model, which is composed by the models described in Section 4 (each of them describes a view of the system), and the output must be OSGi bundles in source code format (Java code and Manifest files), which extend the framework code and follow its guidelines. Although the input of the transformation is the PervML model (which is composed by both the models from the Analyst view and the models from the Architect view), we only focus on the models from the Analyst view because the context information is described in these models. First we describe the mappings between the Analyst view models and the output in an intuitive way. Then we explain how to automate the mappings between the PervML model and Java/OSGi code, this is done in the same way for each model.

5.2.1. Mappings

We describe the mappings in an intuitive way defining the outputs produced from each PervML model. These outputs are the followings:

- **From the Services Model:** Every type of service produces an abstract class that implements methods that contain the information that is specified for that type; for instance, methods for checking the pre and post conditions of every type of service operation. This abstract class extends the PervMLService class of the framework (Fig. 14). The state machine is implemented as a set of linked classes for dealing with every state and every transition. Triggers are also implemented in specific classes with methods for checking the triggering condition and executing the actions.

- **From the Structural Model:** Every component from the component diagram produces a concrete class that extends an abstract class that was produced from the Services Model (depending on the type of service that the component implements). This class is in charge of implementing the operations that were defined in the Services Model. It also contains the location where the services are located. It maintains links with the components that it uses, and it provides mechanisms for reacting to changes.

- **From the User Model:** Every policy produces a concrete class that extends the abstract `PervMLPolicy` class of the framework (Fig. 14). This class is in charge of restricting the operations that can be access with this policy. Every user produces a concrete class that extends the abstract `User` class of the framework (Fig. 14). This class contains the user properties (contact data and personal data), the policy assigned to the user, the relationships with other users, and the system access data.

- **From the Interaction Model:** Every interaction produces a concrete class that extends the abstract `PervMLInteraction` class of the framework (Fig. 14). This concrete class implements methods to check the triggering condition and to execute the actions that are specified in the sequence diagram.

5.2.2. Automatic application of mappings

We have implemented a set of rules from the mappings in order to obtain the code automatically. To do this, we have used the MOFScript tool, which is included in the Generative Model Transformer (GMT) Eclipse project. The MOFScript tool is an implementation of the MOFScript model-to-text transformation language. MOFScript allows us to create model-to-code transformation rules that are applied over the different elements of a source model to generate the corresponding code. Using MOFScript, we have implemented the mappings described above by means of model-to-code rules that generate Java code from the PervML models.

Fig. 16 shows an example of these rules and the code that this generates. This rule generates the code of a Policy class. It generates the code that is in charge of initializing the name of the policy and the service operations allowed for the policy.

We have integrated these MOFScript rules in the implemented tool. In order to transform PervML model into Java code we first define these models by using the editor developed in Eclipse and next we apply the defined MOFScript rules automatically. The final code generated by these rules is structured in a set of packages that contain java classes and a manifest file.
6. OWL code generation

In this section, we introduce a model-to-model transformation that allows us to automatically obtain an OWL specification from PervML models. By means of concepts of the PervML ontology, this specification represents the context information of the system in a machine interpretable language. This facilitates the automated reasoning about context information. Furthermore, there are several reasoners or inference engines such as Racer [19] or Pellet [20] that allow us to infer knowledge from OWL. Thus, the OWL specification can be used by these reasoners to infer knowledge about the system (this aspect is explained in detail in Section 7). In addition, OWL [9] has the capability of supporting semantic interoperability to exchange and share context knowledge between different systems, it is also more expressive than other ontology languages such as RDFS, and is an open W3C standard.

Additionally, the OWL specification provides us with persistence support since OWL allows us to store both information available at modelling time and information available at runtime.

Next, we explain how OWL specifications are obtained from PervML models. However, since this specification is based on the PervML ontology, we present this ontology first.

6.1. PervML ontology

The PervML ontology is detailed in this section through the description of the most important concepts, their properties and the relations among these concepts. We graphically show these concepts in Fig. 17 by using the approach presented by Al-Muhammed et al. [21]. According to this approach, two kinds of concepts can be defined: lexical concepts (enclosed in dashed rectangles), which represent the properties of each class; and nonlexical concepts (enclosed in solid rectangles), which represent the ontology classes. Fig. 17 also shows a set of relationships among concepts (which are represented by connecting lines), such as Component has Trigger. The arrow connection represents a one-to-one relationship or a many-to-one relationship (the arrow indicates a cardinality of one), and the non-arrow connection represents a many-to-many relationship. A small circle near the source or the target of a connection represents an optional relationship.

The main conceptual primitive of PervML is the service metaphor. It describes the service types of the Pervasive System. A service is specified through its operations, its behavior and its relations. For example, a service type could be the gradual illumination service, by means of which you can switch on the light, switch off it, or graduate its intensity. As Fig. 17 shows, the properties that describe this primitive are: (1) Its name, which identifies it in the system; (2) The category to which the service belongs (illumination, multimedia...) as shown in Fig. 17, which indicates that its belongsTo relationship is a many-to-one relationship (i.e., each service only can belong to one category, but a category can group many services); (3) The set of operations provided by a service; (4) A set of triggers that allows the specification the proactive behavior of the services; (5) The valid sequence of operations, which indicates the operations that can be invoked at a specific moment; (6) Its generalization service, if it exists; (7) And the set of services that the service aggregates, i.e., the set of services that the service needs to work, as shown in Fig. 17, this is a many-to-many relationship (i.e. a service can aggregate several services and a service can be aggregated by several services). The small circle near the source and the target of the connection represents that it is not obligatory for a service to aggregate or be aggregated by other services.
Other important concepts are the Component and the Interaction. A Component is a concrete service that provides a service type. As Fig. 17 shows, components can have several relations, such as Used relation, which means one component requires the functionality of other components. The Interaction allows a set of components to communicate with each other as a reaction to an event. It is described by means of a condition, which represents the trigger condition of the interaction, and a set of messages, which indicate the operations of the components that have to be executed.

The PervMLOntology also includes the Person, Policy, Location and Action classes. The Person class is defined to represent the users of the pervasive system. The Policy class is a set of rules that is used to restrict the execution of actions. The properties of the Policy class are its author, its creation time, the set of people to whom the policy is applied, and an operation set that the policy permits. The Location class represents the different areas of the environment. Lastly, the Action class constitutes the execution of an operation. Its properties are the operation that is carried out, the moment in which the action has been executed, and the person who executes it.

This ontology has been manually developed in OWL by using Protégé [22], which is an open source ontology editor. It is important to note that the implementation in OWL of the PervMLOntology does not have to do again because is valid for every system. In addition, we have specified some details and constraints in classes and properties of the ontology that facilitates the later retrieval of information (e.g., we have indicated that the mobility and adjacency relations are symmetric).

6.2. From models to OWL

In this subsection, we explain the transformation to obtain the OWL Specification of the system. To obtain this specification, we have implemented a transformation that allows us to instantiate the PervMLOntology from the PervML model. This transformation creates the individuals that represent the information contained in the PervML model. Individuals represent instances of a class; for example, individuals of the Location class can be Kitchen, Living room, Bedroom, etc.

The transformation is shown in Fig. 18. Its input must be a PervML model (which is composed of the models described in Section 4) and the PervMLOntology in OWL; and the output must be the instantiated PervMLOntology. Therefore, to do the transformation, we first describe the mapping between the input and the output in an intuitive way, and then we explain how to automate these mappings.

6.2.1. Mappings

We describe the mappings in an intuitive way defining the outputs produced from each input. On the one hand, the output from the PervML Ontology is one to one, in other words, the output and the input are the same. On the other hand the outputs produced from each PervML element are the followings:

Please cite this article in press as: E. Serral, et al., Towards the Model Driven Development of context-aware pervasive systems, Pervasive and Mobile Computing (2009), doi:10.1016/j.pmcj.2009.07.006
ARTICLE IN PRESS

E. Serral et al. / Pervasive and Mobile Computing 1 (2009) 1–11

Conforms To

PervML OWL
(ODM)

myHome.owl

myHome.pervml

ATL transformation

pervmlOntology.owl OWL2OWL

PervML2OWL

Fig. 18. Transformation to obtain the system OWL specification.

• From every instance of a class C of the PervML metamodel, an individual of the OWL class that represents the PervML metamodel class C is produced. For instance, the individual Lighting of the Service OWL class is produced from the service Lighting in the PervML Service Model.

• From every instance of a PervML metamodel class attribute A, an instantiation of the corresponding OWL property is obtained. For instance, from the attribute name of the Service GradualLighting whose value is “GradualLighting”, the datatype property name in the GradualLighting individual is instantiated with the value “GradualLighting”.

• From every PervML metamodel relationship between an instance of a class A and an instance of a class B, the OWL object properties of the individuals that represent these instances are instantiated. This is, the object property defined in the class A to represent the relationship with the class B, is instantiated with the individual that represents the instance of the class B; and the object property defined in the class B to represent the relationship with the class A, is instantiated with the individual that represents the instance of the class A. For instance, from the child attribute of the GradualLighting service whose value is the Lighting service instance, the child object property of the GradualLighting individual is instantiated with the Lighting service individual.

6.2.2. Automatic application of mappings

In this section, we have implemented the rules based on the mappings to automatically obtain the OWL specification. To do this, we have used the Atlas Transformation Language (ATL) Eclipse plug-in. It provides us with a set of model-to-model transformation tools and can be integrated into the Eclipse framework. In order to automate the transformation, we have also used the EODM plug-in [16]. This plug-in is built on top of EMF and conforms to the ODM (Ontology Definition Metamodel) standard of OMG. We have used this plug-in because it provides us with the OWL metamodel. Also, it allows creating, modifying, and navigating RDF/OWL models. Using ATL, we have implemented the mappings described above by means of model-to-model rules that generate the OWL specification from the PervML Ontology (defined according to the OWL metamodel provided by EODM) and the PervML model (defined according to the PervML metamodel developed using EMF). Examples of these rules are shown in Fig. 19. The first rule copies the OWL classes from the PervML Ontology into the OWL specification and the second rule transforms the PervML locations in OWL individuals of the Location Class.

We have integrated these ATL rules in the implemented tool. Therefore, once the PervML models have been specified by using the editor developed in Eclipse or PervGT, we can apply the ATL rules to obtain the complete information about the system in OWL automatically. Using the EODM plug-in, we can see the result with either the tree editor that provides this plug-in or in OWL source code. The Specification must be compiled with the reasoner, packaged into a bundle (JAR files), and then installed in the OSGi server.

7. OWL inference strategy

In this section, we explain the proposed strategy to infer indirect observable knowledge from the System OWL Specification. To allow this inference, we must obtain all information about the system, i.e., the set of individuals that describe the system and the context information related to the system. Hence, the OWL specification of the system must contain the information available both at modelling time and at runtime.

On the one hand, in order to store the information available at modelling time in OWL we use the transformation explained in the previous section. This transformation allows us to obtain the specification of the system specified by PervML in OWL (Fig. 20A).

On the other hand, the runtime information is extracted by the Pervasive System itself from its interaction with users and the environment. In Section 5.1.1, we have explained how the framework provides mechanisms to update the OWL specification according to the runtime information. For instance, when the Peter user switches on the kitchen lighting, the
A user class that represents him, checks if the policy assigned to Peter allows him to execute this operation. If the operation is allowed by his policy, the user class that represents Peter makes use of the service in order to carry the operation out. Finally, if the operation is performed successfully, the PervMLService class creates the OWL action description and adds it to the OWL specification, as shown in Fig. 20. Fig. 20A shows the initial ontology. When Peter switches on the kitchen lighting successfully, the system adds the On_KitchenLighting Individual of the Action class (Fig. 20B). Thus, the OWL specification is continually extended at runtime storing the context history.

Additionally, we have a rule repository that contains a set of rules in SWRL [23] that allows us to infer additional or indirectly observable context information. To do this, these rules are applied to the OWL specification by the Pellet reasoner every time that the specification is updated. This reasoner can reason about OWL classes, their properties and relationships, however, in this work, we focus on reasoning about ontology individuals because it allows us to deduce new context data that not is possible to obtain directly from the system, such as the activity that a user is performing, or even the state of mind of a person according to the data about the tone of voice, the expression of the face… In Fig. 21, for example, a rule is shown. This rule tells the system where a user can go in every moment, which depends on its current location. For now, the rules are manually added in the rule repository, but we are currently working to obtain them from the models, too.

Please cite this article in press as: E. Serral, et al., Towards the Model Driven Development of context-aware pervasive systems, Pervasive and Mobile Computing (2009), doi:10.1016/j.pmcj.2009.07.006
Furthermore, the OWL specification can be used by machine learning algorithms [24], such as neural network algorithms, decision tree, etc. in order to:

- Predict the next action that a user is going to execute and to automatically execute the corresponding operation of the service.
- Detect actions patterns that always occur at a regular time interval and to update the system to add an interaction that is triggered at the right time.
- Detect action patterns where a set of actions is always executed after a certain action and to update the system to add an interaction that is triggered when the corresponding action is executed.

These actions provide the system with mechanisms to adapt itself according to user behavior and context information. As further work, we are studying how these actions can be automatically performed.

8. A case study: Evaluation of our method

In this section, we evaluate our method and show its ability to facilitate the development of context-aware pervasive systems. To do this evaluation, we have followed the guidelines presented by Kitchenham et al. in [25] and Basili et al. in [26]. We present the objectives, development and outcomes of the evaluation below.

8.1. Objectives

The goal of this evaluation is to demonstrate the advantages that our method provides in the development of context-aware pervasive systems over manual development. The software quality measures and metrics primarily considered to evaluate both developments are the following [27]:

- Development time: time needed to develop the case study.
- Reusability: software artifacts that can be reused.
- Maintenance and evolution: time used to correct faults, to improve performance and to adapt services to changes in their requirements.

This set of measures is not exhaustive, but reflects the principal quantitative advantages that our model-driven approach provides developers.

8.2. Development of the evaluation

To achieve our goal, we performed a replicated project case study [25], which consists of several teams developing the same case study by following two approaches in order to be able to measure and compare them. Thus, we selected a suitable case study and two different teams to develop it first manually, and then using our method (which was presented in Section 3).

The case study selected to be developed was a pervasive system for a smart home because it is a typical project in this field. This case study is the running example used throughout the entire paper. The principal aim of this case study is to
improve the everyday life of a family by addressing vital aspects such as home care and safety, comfort, entertainment, etc. Mainly, this case study provides the following services (which were shown in Fig. 4):

- **Multimedia management**: allows inhabitants to store, manage and reproduce multimedia archives.
- **Intelligent lighting management**: controls the lighting according to both user presence and light intensity.
- **Security management**: when activated, if it detects presence inside or detects that a door or window has been opened, the system goes off the alarm, starts to record and sends a warning to users.
- **Heating and Air conditioning management**: keeps the temperature optimum in the room where the user is, keeps a temperature close to optimum in the locations where the user can go and puts the Heating and Air conditioning in saving energy mode in the rest of the house.
- **Blind Management**: allows inhabitants to control the blinds of the home.

The teams selected to develop the case study were the following:

- **Team 1**: composed by one computer science engineer that worked full-time. He had 6 months of work experience in UML modelling and Java programming.
- **Team 2**: composed by two 5th-year students of Computer Science at the Technical University of Valencia (Spain), who worked part-time. They could program in C++ and knew UML.

All three of them were beginners in the development of pervasive systems. We selected these teams to properly show the differences regarding the established measures and how working in parallel affects to them.

In order to measure the reusability and support for maintenance and evolution for each development, we provided each team with a manual implementation of a case study for a smart meeting room as well as its modelling using PervGT. This case study, which was presented in [14], provided the following services:

- **Multimedia management**: allows inhabitants to store, manage and reproduce multimedia archives.
- **Intelligent lighting management**: controls the lighting according to inhabitant presence.
- **Security management**: starts to record when activated if it detected presence inside.

In addition, in order to properly validate both developments of the case study, we built an execution environment that allows real scenarios to be emulated. Fig. 22 shows this execution environment which is made up of a barebone (where the OSGi server runs), an EIB network and a device simulator. The barebone was a Pentium IV with 512 Mb RAM and connectivity by Ethernet 802.g and two serial ports. It was used as the central server running a Windows XP Professional Edition. The OSGi server was the Prosyst Embedded Server 5.2. The deployed EIB network was physically connected to the barebone by the serial ports. The pervasive system accessed this network by means of the EIB bundle provided by Prosyst. The Device Simulator simulates the behaviour of the devices needed for the case study that are not in the EIB network. This simulator was previously developed [13] in our research group and allows developers to define virtual devices and manage them by means of an intuitive user interface. We also provided the teams with the drivers for the virtual devices that they needed to emulate the devices that there were not in the EIB network. Thus, to deploy the system, the teams had to package the code of the system and the simulator into bundles (with their corresponding manifest files), and install and start them in Prosyst.

8.2.1. Development of the case study by hand

Since it was not possible to directly program the functionality required by the proposed case study into the devices, both teams used a three-layer architecture:

1. **Driver Layer**: contains the classes that allow devices to be used. To implement the drivers for controlling the EIB devices, the Java language, the OSGi technology and the EIB API that Prosyst (the OSGi server) provides were used.
2. **Logical Layer**: contains the classes that give support to the required functionality by using the classes of the driver layer. The logical layer also supports the context management that is explicitly required for the services of the case study (without saving a context history). In addition, this layer also contains the mechanisms for ensuring the security of the system. To implement this layer, Java, OSGi, Java DB (as database [28]), Eclipse 3.3 (as IDE), and Prosyst (to run the system) were used.
3. **The Interface Layer**: contains the classes to allow users to access the functionality. To implement this layer, HTML and Eclipse 3.3 were used.

Prior to implementing the case study with this architecture, the teams had to learn the basics to start to program in the above-mentioned technologies (HTML, Java, OSGi, EIB, Eclipse and Prosyst) that they did not know. To do this, we provided them with the necessary tutorials and tools. In this learning phase, what took them the most time was understanding the main concepts of OSGi and how to program using these concepts in the Prosyst server (it took them between 98 and 105 h). The teams also found it difficult to properly use the EIB API to communicate the services installed in Prosyst with the EIB drivers (it took them between 18 and 24 h). However, Team 2 was able to start to program in Java with Eclipse in two days, and both Team 1 and Team 2 only needed a few hours to start to program in HTML and to use Java DB. Nonetheless, both teams needed to continue learning throughout the development of the case study.

Thus, the computer science engineer of Team 1 learned the basics of the required technologies and sequentially implemented the three layers. First, he implemented the EIB drivers needed for control the devices of the EIB network,
except the ones that had been already implemented in the meeting room since all of them could be reused. Therefore, it took him 18 days to develop the driver layer.

He then implemented the Java/OSGi classes that provided the required services and managed the context information. To do this, he first studied the code of the meeting room to see which services he could reuse. Since multimedia service of the meeting room provided the same functionality required for the smart home, it could be reused with very few changes. However, the lighting and security services of the smart home required more complex functionality. The lighting service of the meeting room only took the presence into account to switch on/off the light, whereas in the smart home the lighting needed to be graduated according to presence and the room lighting intensity. Also, the security service of the meeting room only started to film when it detected presence, whereas in the smart home required filming the house, activating the alarm, and sending a warning email when it detected presence or an opened door or window. Furthermore, since these manual services were very dependent on the drivers that they used, both the lighting service and the security service had to be completely implemented again.

The engineer also had some problems implementing the functionality required for the Heating and Air conditioning management because this service required user information as well as information about the locations of the house. Thus, to implement this service, he directly added the preferred user temperature and the distribution of the house as attributes of the service class. He also implemented methods to obtain the context required to manage the Heating and Air conditioning by means of these attributes and by directly accessing the presence detector devices.

When he finished implementing the service functionality, he implemented mechanisms to ensure that the functionality of each service could only be executed by registered users with the corresponding permission. This was done implementing a small data base (in Java DB) that contained the users and the operations they were allowed to perform. Thus, he took 42 days to finish the logical layer.

Then he implemented the Web interface in 8 days. He could not use the web interface implemented for the meeting room because it was dependent on the provided services.

During the development of these three layers, the engineer spent 23 h to understand the code of the meeting room, and 57 h to debug and correct errors. Last, he spent 12 h to deploy the system because he had to develop the manifest files for each bundle to be able to run them in Prosyst.

The students of Team 2 worked in parallel. Since they considered that the Logical Layer was the hardest to develop, one of them was in charge of developing the drivers and the Web interface, and the other one was in charge of developing the logical layer. They had some problems working in parallel; for instance, the interface and the driver layers could not be properly proved until the logical layer was implemented. Thus, when the student in charge of the drivers and the interface finished, he had to wait until the other student finished. Also, both students had to interact with each other continually to be able to use the components implemented in the lower layer. It took them 113 days to finish all the layers.

This team was able to reuse the same code of the meeting room as Team 1 and had similar problems implementing the Heating and Air conditioning management. In addition, during the development of the case study they spent 32 h to understand the code of the meeting room and 46 h to debug and correct errors (17 for drivers, 25 for the logical layer, and 4 for the web interface). Last, they spent 8 h to deploy the developed system.
8.2.2. Development of the case study using our method

In order to develop the case study by following our method, both Team 1 and Team 2 followed the steps described in Section 3. Prior to developing the case study, they had to learn PervML (the modelling language) and how to use PervGT (the tool provided to support the development).

The first step to be done according to our method is the system modelling. Thus, the engineer of Team 1 played the role of both System Analyst and Architect. He learned PervML and how to use PervGT in 7 days. Afterwards, he studied the models of the meeting room to determine which services and binding providers could be reused. Since multimedia service of the meeting room provided the same functionality, this service and its binding providers could be completely reused. In addition, even though the lighting and security services of the smart home required more complex functionality, the specification of all the services (the PresenceDetection, VideoRecording, GradualLighting, Lighting and Activation services) and its binding providers could be reused since PervML allows the composition of services and is independent of the drivers used. Thus, Team 1 spent 5 days to specify the system.

The students of Team 2 worked in parallel. One of them played the role of System Analyst and learned its models in 6 days. Then, this analyst captured the requirements of the system and specified them in their corresponding three models in another 6 days. The other student played the role of System Architect and learned its models in 6 days. Then, this architect specified the types of devices that would provide support for the services by using the binding provider model. It took 1 day. Finished this model, the architect waited a day until the analyst finished the Service and Structural Models (which are needed to be able to specify the Component Structure Model and the Functional Model). When these models were finished, the architect specified the remaining two models, which took 4 days. This team was also able to reuse the same services and binding providers of the meeting room as Team 1.

In order for the services of the case study to take into account the needed context information, each team specified the use of this information in the Functional Model (where the service behavior is modelled) by using the names of the corresponding ontology properties. For instance, the areas where a user can go are stored as the canGo property of each user in the ontology. Therefore, this property can be used by any service by specifying user.canGo in its behavior modelling. This information is automatically obtained by means of a SWRL rule (Fig. 21) contained in the ontology.

Next, both teams executed the code transformations using PervGT to translate the specified models into both the Java–OSGi code (which extends the implementation framework) and the OWL specification (which stores and reasons about the context information and gives support for the adaptation of the system). By using these transformations, not only did the teams obtain the code of the specified context-aware pervasive system, but they also obtained the graphical user interface. This interface allows controlling the corresponding services as well as dealing with security and privacy concerns. Both the implementation framework and the OWL context ontology are reused for every system.

Once the code was generated, the teams had to develop the drivers that control the EIB devices (lights, sensors, etc.) of the provided EIB network, except the meeting room drivers that could be reused. To do this, they had to specify the corresponding drivers and generate their code by using the EIB driver generator [13]. In addition, the system had to be deployed in the execution environment following the steps of the deployment phase of our method (which are: packaging the code into bundles, and installing and starting these bundles in Prosyst). Thus, Team 1 specified and generated the EIB drivers in 2 days and then deployed the system in 30 min because the generated system code was already prepared to be packaged into bundles.

Team 2 carried out both tasks at the same time. To do this, the architect specified and generated the EIB drivers, while the analyst deployed the system and proved its services by using the Device Simulator. These tasks took them 4 days.

During the development of the system, Team 1 needed less than two hours to understand the meeting room models and Team 2 needed less than three hours to do this task. In addition, neither of them had to correct faults (because the code is generated free of errors) or change any functionality after deploying the system (because during the modelling phase they discussed the functionality using the models).

Note that neither of the teams had to implement anything to manage the context information of the system since it is automatically managed by the framework and the context ontology. By means of them, the generated systems also store the user actions, which can be used for later action automation. Moreover, neither of the teams had to spend time on ensuring the privacy and security of the system because the code that provides this functionality is also generated from the User model by the code generation strategy.

8.3. Analysis and results

In this subsection, we analyze and compare the results obtained regarding the measures established in Section 8.1. These results are summarized in Table 1.

With regard to development time, the difference between the two types of development is clear. With our approach Team 1 (composed of one computer engineer working full-time) took 14 days and Team 2 (composed of two students working part-time) took 16 days, whereas with the manual development Team 1 took 85 days and Team 2 took 113 days (see Table 1). Therefore, the development of a pervasive system using our approach is faster than using the manual one. Moreover, working in parallel turns out to be more efficient by using our method than in the hand development. This is because technologies like OSGi or EIB are complicated and learning them takes longer than learning PervML, which provides intuitive and high level...
primitives to specify the system. Furthermore, our approach allows developers to focus on satisfying system requirements instead of solving technological problems. Last, the teams also saved time deploying the system using our approach, because the code generated by PervGT is ready to be packaged and installed in OSGi.

With regard to reusability, our approach facilitates reuse to a large extent. First of all, PervML allows the composition of services so that, even though the requirements of a service change, the specifications of the services used by it can be completely reused. For instance, in the smart home case study, security and lighting services provided more complex functionality than in the meeting room, therefore, they had to evolve to offer the new functionality. In the manual development only a few lines of these services could be reused, while using our method all the specifications of the services and the binding providers used for the lighting and security services could be completely reused. In addition, although the specification of these services had to be changed, they could be extended to provide the new functionality because PervML specifies them independently of the drivers. Moreover, both teams found the meeting room models much easier to understand than the meeting room manual code. Furthermore, the complete framework and the ontology that our approach provides are always reused. Table 1 summarizes the software artifacts that were reused in each approach.

With regard to maintenance and evolution, we have shown that this is a very complicated task using handwritten code. Adapting the lighting and security services from the meeting room to make them fit the requirements of the smart home meant implementing them again. On the contrary, as Table 1 shows, using our approach, adapting these services to the new requirements took the teams very few hours. This is because updating the system (such as adding, modifying or deleting services or context information) is as easy as modifying the specified models in PervGT to make them fit the new requirements since the new system code is automatically generated with the changes.

In addition, our MDD method provides other benefits over the manual approach such as the following [29–31]:

- **Systematization of the software development process**: Our method defines a precise sequence of steps to follow in order to develop and deploy a context-aware pervasive system. It also provides tools to develop it. This facilitates considerably the development process of systems of this type.

- **Automation of the software development**: Our approach provides a high level of automation in the development of context-aware pervasive systems, which considerably reduces the development time.

- **Simulation and early requirement validation**: Our approach allows the system code to be automatically and completely obtained from models, which also facilitates the early validation of the requirements through prototyping. In addition, our approach allows the functionality to be simulated without having the real devices by using the device simulator.

- **Quality**: Our approach promotes the separation of roles with well-defined responsibilities. Each task can be developed by the most qualified developers. For instance, specialized analysts can capture the requirements of the system according to user needs, and specialized architects can specify the devices that will provide their support. In addition, specialized

---

**Table 1**

Summary of the results obtained from the comparative evaluation.

<table>
<thead>
<tr>
<th></th>
<th>Development time (days worked by the team)</th>
<th>Reusability (software artifacts)</th>
<th>Maintenance and Evolution (hours worked in adapting lighting and security services)</th>
</tr>
</thead>
<tbody>
<tr>
<td>By hand</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Team 1 (1 engineer/full-time)</td>
<td>Learning: 16</td>
<td>▪ Services: Multimedia</td>
<td>124 (time used by the engineer to implement again the two services)</td>
</tr>
<tr>
<td></td>
<td>Development: 68</td>
<td>▪ All the meeting room drivers</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Deployment: 1.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total 85.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Team 2 (2 students/part-time)</td>
<td>Learning: 35 (1st student)</td>
<td></td>
<td>154 (time used by the student to implement again the two services)</td>
</tr>
<tr>
<td></td>
<td>Development: 62 (1st student)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Deployment: 2 (1st student)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total max</td>
<td></td>
<td></td>
</tr>
<tr>
<td>By following our method</td>
<td>Learning: 7</td>
<td>▪ Framework &amp; Ontology</td>
<td>2</td>
</tr>
<tr>
<td>Team 1 (1 engineer/full-time)</td>
<td>Development: 7</td>
<td>▪ Services: PresenceDetection, VideoRecording, GradualLighting, Lighting, Activation &amp; Multimedia</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Deployment: 0.2</td>
<td>▪ All the Binding Providers used by the above services</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total 14.2</td>
<td>▪ All the meeting room drivers</td>
<td></td>
</tr>
<tr>
<td>Team 2 (2 students/part-time)</td>
<td>Learning: 6</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Development: 9.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Deployment: 0.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total 16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
programmers can develop PervGT in order to generate high quality code in a specific technology. Furthermore, our approach facilitates the systematic reuse of know-how knowledge, software best practices and development assets.

- **Technological change support:** The development of the system is independent of implementation technologies because it is the specification of the system in PervML.

9. Analysis of the related work

In this section, some of the most important projects for developing context-aware pervasive systems are analyzed. We focus our analysis on those projects that carry out context modelling in their development process, as we do in this work. Next, we present comparative analysis of the described projects and our work.

9.1. Related work

In this subsection, we first present some approaches that, in spite of modelling context, only use these models as documentation. Therefore, they do not provide support for the development of pervasive systems like the one presented here. In these approaches, the systems must be developed ad-hoc, which makes their implementation and maintenance difficult due to the permanent technological innovations and the complexity and dynamism of pervasive computing systems.

The CASS (Context-awareness sub-structure) project [32] proposes a middleware for context-aware mobile applications. CASS supports high-level context data abstractions by using a logic model and the separation of both context-based inferences and behaviors from the application code. The CORTEX project has also built a context-aware middleware. It is based on the Sentient Object Model [5] and is suitable for the development of context-aware applications in ad-hoc mobile environments. It also allows developers to use data from disparate sensors, represent context application, and reason about the context efficiently. In [33], Korpipää et al. present the Context Managing Framework for developing context-aware mobile applications. Its architecture is comprised of four main functional entities: the context manager, the resource servers, the context recognition services, and the application. This Framework deals with context by using an ontology described in RDF. CoBrA (Context Broker Architecture) [34] is an agent-based architecture that supports context-aware computing in intelligent spaces. CoBrA has adopted an OWL-based ontology approach, and it offers a context inference engine that uses rule-based ontology reasoning. These works (CASS, CORTEX, Context Managing Framework and CoBrA) cannot be used to develop services like the ones proposed in our case study because they only give support to manage context.

The SOCAM (Service-oriented Context-Aware Middleware) project introduced by Gu et al. [35] is another architecture for building context-aware mobile services. The SOCAM authors divide a pervasive computing domain into several sub-domains and then define each sub-domain in OWL to reduce the complexity of context processing. SOCAM has also implemented a context reasoning engine that reasons over the knowledge base. In SOCAM, the interactions of the case study could be developed because it allows specify rules to invoke methods when a condition becomes true. However, this approach does not manage the development of pervasive services to be executed by user request. The Gaia middleware [4] also not only gives support to manage context, but also gives a little support for developing the services of the system since it extends typical operating system concepts to include context-awareness. However, to develop the case study, Gaia gives support for developing pervasive services in a way similar to our framework, with the difference that in Gaia these services directly use devices to provide the pervasive services. This makes both the reusability and the development of complex services more difficult, such as the services in our case study like the Intelligent Lighting (which uses context information captured by other services as presence detectors and light intensity sensors to save energy).

Second, we present some projects that go a step further by using these models to develop the system. They propose Model Driven Development approaches. For instance, in [10], Ayed et al. proposed a MDD that allows developers to graphically specify context using a UML profile. This approach is based on several phases that approach the context platform step by step and allow designers to partially obtain the system code for managing context through the definition of modular transformations. Another example of these approaches is the one presented in [12] by Henricksen and Indulska. They propose a graphical context model called CML, which is an extension to Object-Role Modelling for context modelling purposes. The authors also propose a model driven approach to develop context-aware applications based on CML. They propose a semi-automated procedure to map their context models to context management systems that use relational databases.

As in the first group, these approaches do not provide support for developing the pervasive services that the presented case study must offer. Thus, although they propose semi-automated methodologies to generate context-aware pervasive systems from the models that describe the systems, they cannot automatically obtain a complete functional context-aware pervasive system. Unlike all the works presented above, the work presented in [11] by Ou et al., attempts to obtain complete functional context-aware pervasive systems by following a MDD strategy. They state that it is necessary to specify the application logic and provide graphical user interfaces for completely developing context-aware pervasive systems, as our approach tries to do. They propose a pure MDA approach for ontology-based context-aware applications development. To do this, they define a set of metamodels and a Model Driven Integration Architecture to integrate these metamodels and generate CAA implementations either semi-automatically or automatically. However, as we explained in Section 5, one of the drawbacks of this approach is that the abstraction gap is dealt with in only one step, so the transformation can be very
Table 2
Summary of requirements to support context-aware pervasive.

<table>
<thead>
<tr>
<th>Context model</th>
<th>Context management</th>
<th>Context reasoning</th>
<th>Context history</th>
<th>Privacy &amp; security</th>
<th>Support for heterogeneity</th>
<th>Systematization of the Software-guidance</th>
<th>Ease to deploy a system-guidance</th>
<th>Only Context (C)/Full functional system (F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ayed et al.</td>
<td>Graphic</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>C</td>
</tr>
<tr>
<td>CML</td>
<td>Graphic</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>C</td>
</tr>
<tr>
<td>CORTEX</td>
<td>OQ</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>C</td>
</tr>
<tr>
<td>CASS</td>
<td>Logic</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>C</td>
</tr>
<tr>
<td>CoBrA</td>
<td>Ontologies</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>C</td>
</tr>
<tr>
<td>SOCAM</td>
<td>Ontologies</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>C</td>
</tr>
<tr>
<td>Context managing framework</td>
<td>Ontologies</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>C</td>
</tr>
<tr>
<td>Gaia</td>
<td>Logic &amp; ontologies</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>C</td>
</tr>
<tr>
<td>Ou et al.</td>
<td>Graphic &amp; ontologies</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>F</td>
</tr>
<tr>
<td>Our approach</td>
<td>Graphic &amp; ontologies</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>F</td>
</tr>
</tbody>
</table>

complex. Furthermore, this approach does not ensure the security and privacy of the system and does not save the context history; thus, they do not give support to the adaptation of user behavior. Moreover, they do not give any detail about how the context reasoning can be carried out.

9.2. Analysis

In order to analyze the related works, we first evaluate and compare them with our approach according to a set of requirements. Next, we propose a set of relevant software quality measures and metrics to evaluate these works and compare them with ours.

After analyzing the published literature [3,36–40], we have established a set of requirements that every approach for developing context-aware systems should satisfy. These requirements are the following:

- **Context management**: indicates whether the support for the capture and storage of context for later retrieval is provided.
- **Context reasoning**: indicates whether the context model enables reasoning on context data in order to infer properties or more abstract context information (e.g., to deduce user activity by combining sensor readings).
- **Context history**: indicates if the history of previous contexts that are relevant to context itself is provided.
- **Privacy and security**: indicates if the protection of user privacy is provided by establishing and enforcing user defined policies so that any user can only execute the services allowed by its policy.
- **Support for heterogeneity**: indicates if the approach is independent of the hardware components used. They must support a wide range of sensors and actuators, mobile client devices to high-performance servers, networking interfaces, legacy components, etc.
- **Systematization of the Software**: indicates if a precise and efficient sequence of well-defined steps for developing a context-aware pervasive system is provided.
- **Ease of deployment and configuration**: indicates if the hardware and software components of the system are easily deployed and configured to meet user and environmental requirements, potentially by non-experts.

Table 2 summarizes the analysis of the related works with respects to the above requirements. To do this, this table indicates which approaches fulfill each one of these requirements. It also shows which approaches focus on supporting context and which approaches also provide support to the development of complete Context-aware Pervasive Systems. In this table, we also indicate the context model that each approach uses, i.e., the class of the conceptual tool that is used to capture context. The information shown in this table has been extracted from the publications of the studied works [34,4,33,35,32,10–12] and the most important surveys about context-aware pervasive systems carried out to date [39,40].

As Table 2 shows, all the presented approaches permit context management, and almost all provide facilities to derive or interpret knowledge from the captured context information. However, in only a few cases is the context history stored. Also, very few of these approaches provide support for privacy and security or a comprehensive support for heterogeneity, which are both important requirements of systems of this type.

Most of these approaches are focused on providing context-awareness, but not on developing full context-aware pervasive systems, as our approach does. Our approach is the only one that provides a precise sequence of well-defined steps to both deploy and develop the system. These steps guide developers to automatically generate the context-aware pervasive system code from the system specification. This specification is performed at a high level of abstraction by using intuitive primitives to specify both the system requirements and the context available at modelling time independently of the hardware components. In addition, to manage context at runtime, our approach provides a context ontology, which is one of the best choices for modelling context. Ontologies guarantee a high degree of expressiveness, formality and semantic richness, as well as facilitating reasoning, interoperability and reuse of context [41,4]. Furthermore, our approach also
provides a transformation engine to automatically obtain an OWL context repository based on this ontology from the specified models and an infrastructure that, by using the generated repository, automatically manages context, derives knowledge from it and provides support for the automation of user actions. This infrastructure is also in charge of ensuring the privacy and security of the system. Thus, our approach fulfils all the established requirements.

In order to evaluate the studied works in a quantitative way and compare them with our work, we have established a set of measures and metrics. This set has been established on the basis of the MDD guidelines [42,29–31] since all these works use models in their development process. This set is composed of the following measures and metrics:

- **Automation of the software development**: indicates the support provided by the approach to automatically obtain the system code specified in the models. We consider three degrees of automation:
  - High: the approach generates the complete system code from models.
  - Medium: the approach generates the partial system code from models (the system code has to be completed) and/or provides a framework that encapsulates common functionality of pervasive systems.
  - Low: the whole system has to be developed by manual code.

- **Technological change support**: indicates the support provided by the approach to change the implementation platform or programming language. We consider three degrees of technological change support [29,31]:
  - High: the modelling language allows the system to be specified independently of the technology and the system code can be generated completely.
  - Medium: the modelling language allows the system to be specified independently of the technology but the system code can be only partially generated.
  - Low: the models are not independent of the technology.

- **Development time**: indicates the time required to create a new system. The more hand-written code there is, the longer development time [30]. We established the following metrics to measure it:
  - High: the entire system is programmed by hand.
  - Medium: some hand-written code is required.
  - Low: no hand-written code is necessary.

- **Support for the simulation and early requirement validation**: indicates the support provided by the approach to simulate and validate the system during the development process. This feature greatly benefits from the application of MDD techniques. We established the following metrics to measure it [29]:
  - Very high: the code of the system is automatically and completely obtained from models and the approach provides developers with a device simulator to simulate the functionality of pervasive devices.
  - High: the code of the system is automatically and completely obtained from models.
  - Medium: the code of the system is partially obtained from models.
  - Low: the approach does not provide a strategy for generating the system code.

- **Support for maintenance and evolution**: indicates the cost of modifying the system after delivery to correct fault, to improve performance or other attributes, or to adapt the system to a modified environment [42]. This feature benefits from the application of MDD techniques. We consider three degrees of maintenance [30]:
  - High: the approach does not allow code generation so that both models and code have to be modified.
  - Medium: the approach allows partial code generation so models have to be modified and the system code has to be completed by hand.
  - Low: only models have to be modified.

- **Reusability**: indicates the potential of reuse at both modelling time and development time. It will be higher if the approach uses ontologies [40], allows the composition of services [43], and/or provides a framework. We consider the following degrees of reusability:
  - at modelling time:
    - High: the approach allows the composition of services and uses ontologies.
    - Medium: the approach allows the composition of services or uses ontologies.
    - Low: the approach neither allows the composition of services nor uses ontologies.
  - at development time:
    - High: the approach allows the composition of services and provides a framework that encapsulates common functionality of pervasive systems.
    - Medium: the approach allows the composition of services or provides a framework that encapsulates common functionality of pervasive systems.
    - Low: the approach neither allows the composition of services nor provides a framework that encapsulates common functionality of pervasive systems.

Table 3 summarizes the analysis of the related works with respect to the above measures. The information shown in this table has been extracted from the publications of the studied works [3,34,4,5,33,35,32,10,12] and relevant studies of the established measures for pervasive systems [40,42,29–31,43]. Section 8 also shows the evaluation of these measures for our approach.

As Table 3 shows, the approaches that generate code from the models used for specifying the system (such as the CML proposal or the approaches proposed by Ayed et al. and Ou et al.), usually obtain a higher score in most of the
measures. However, our method obtains the highest score since our modelling language allows the system to be specified independently of the technology and automates the entire development process by allowing complete code generation from the system specification. Therefore, with our method the development time comes down considerably. Our approach also makes the maintenance and evolution of the system easier because the system code is generated free of errors and only models have to be modified to adapt the functionality. In addition, our modelling language and the proposed framework favor reusability as the table shows. Furthermore, our approach facilitates the simulation of the system functionality and the early validation of the system requirements. Moreover, since a device simulator is provided, the system can be proved without having to have the real devices.

10. Conclusions and further work

In this work, we have presented a model driven development method for generating context-aware pervasive systems. The proposed method applies the MDA and SF guidelines and allows us to properly specify the context information in a set of models and to automatically generate the code of context-aware pervasive systems from these models. The proposed method provides developers with:

- A set of models that allow a context-aware pervasive system to be specified.
- Mechanisms for storing the context information in OWL so that it can be accessed by the system at runtime.
- An implementation framework to give support to translation from PervML models into code and the updating of the OWL specification according to the context information produced, at runtime.
- A transformation engine to translate the context models into Java code.
- A strategy to give support to the adaptation of the system according to the context information.

As further work, we plan to: (1) develop tools that allow users to reconfigure the system (e.g. tools for allowing users to create new users or policies at runtime); (2) study and select the adequate learning machine algorithms to predict next actions and extract behavior patterns in order to be able to automatically execute them; (3) automatically generate the SWRL rules from PervML models; (4) provide developers with quality metadata for context modelling; and (5) develop more case studies for context-aware pervasive systems (we are currently developing two case studies, one system to be deployed in a car and another to support the daily needs of the members of a researcher department).

In summary, we believe that model driven approaches can provide many benefits to the heterogeneous and rapidly changing pervasive systems field. The method presented in this work to generate context-aware pervasive systems is our contribution to this vision.

References


Please cite this article in press as: E. Serral, et al., Towards the Model Driven Development of context-aware pervasive systems, Pervasive and Mobile Computing (2009), doi:10.1016/j.pmcj.2009.07.006


Please cite this article in press as: E. Serral et al., Towards the Model Driven Development of context-aware pervasive systems, Pervasive and Mobile Computing (2009), doi:10.1016/j.pmcj.2009.07.006