The Journal of Systems and Software 86 (2013) 2890–2905

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A domain-specific language for context modeling in context-aware systems

José R. Hoyos*, Jesús García-Molina, Juan A. Botía

Universidad de Murcia, Facultad de Informatica, Campus de Espinardo, 30100 Murcia, Spain

ARTICLE INFO

Article history:
Received 4 November 2012
Received in revised form 3 July 2013
Accepted 4 July 2013
Available online 13 July 2013

Keywords:
Model Driven Development
Context modeling
Context aware

ABSTRACT

Context-awareness refers to systems that can both sense and react based on their environment. One of the main difficulties that developers of context-aware systems must tackle is how to manage the needed context information. In this paper we present MLContext, a textual Domain-Specific Language (DSL) which is specially tailored for modeling context information. It has been implemented by applying Model-Driven Development (MDD) techniques to automatically generate software artifacts from context models. The MLContext abstract syntax has been defined as a metamodel, and model-to text transformations have been written to generate the desired software artifacts. The concrete syntax has been defined with the EMFText tool, which generates an editor and model injector.

MLContext has been designed to provide a high-level abstraction, to be easy to learn, and to promote reuse of context models. A domain analysis has been applied to elicit the requirements and design choices to be taken into account in creating the DSL. As a proof of concept of the proposal, the generative approach has been applied to two different middleware platforms for context management.

1. Introduction

Environment knowledge is an essential part of context-aware systems. They adapt their behavior to changes in context without user intervention, by combining information from various sources like, for example physical sensors, software listeners or user profiles.

The building of context-aware systems is an intricate task. Therefore, software engineering techniques are necessary for tackling its complexity, such as modeling or framework-based solutions. There are several architectures for distributed context-aware frameworks (Baldauf et al., 2007; Chen, 2004; Korpipää et al., 2003). Furthermore, as context storing and processing play a key role in context-aware systems, the creation of models assists in understanding and reasoning about contexts. Thus, a number of context modeling approaches have been proposed, which differ in the data structures used to represent the context information (Held et al., 2002; Preuveneers and Berbers, 2005; Schilit et al., 1994). However, as in other application domains, the activity of modeling in context-aware systems is currently directed towards Model-Driven Development (MDD).

MDD promotes a systematic use of models along the software lifecycle. The objectives of MDD are increasing the levels of abstraction and automation in the development of software in order to improve the software productivity and some software quality aspects as reusability and interoperability (Selic, 2008). MDD is based on four main principles (Kleppe, 2008):

i) models at different abstraction levels are created to represent aspects of the system (e.g. the context information),
ii) these models are expressed in domain-specific languages (DSL) (also known as domain-specific modeling languages, DSML),
iii) these languages are built by applying metamodeling techniques,
iv) model transformations are used to generate software artifacts from models, either directly by model-to-text transformations or indirectly by using one or more intermediate model-to-model transformations.

Recently, some approaches have been proposed to take advantage of MDD techniques in the construction of context-aware systems (Ayed et al., 2007; Sindico and Grassi, 2009) and pervasive applications (Pham et al., 2007), but an important research effort is still needed to provide appropriate languages, methods and tools to efficiently support the model-based development of this kind of systems. The best MDD practices that have emerged over the years should be taken into account in the new proposals. For instance, while most of the proposed approaches have defined a UML profile

* Corresponding author. Tel.: +34 868 88 4398; fax: +34 868 88 4151.
E-mail addresses: jose.hoyos@um.es (J.R. Hoyos), jmolina@um.es (J. García-Molina), juanbot@um.es (J.A. Botía).

0164-1212/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2013.07.008

with which to model context, DSLs created from scratch are currently considered to be more appropriate in most situations (Kelly and Pohjonen, 2009; Völter, 2009). UML profiles is not a first-class extension mechanism because it does not allow for modifying the UML metamodel, and the new elements, which specialize existing elements, must not violate the abstract syntax rules and semantics of UML. Therefore, UML profiles should be restricted to define modeling languages whose abstract syntax and semantics is close to those provided by UML (Selic, 2007).

With respect to this, here we present an MDD approach based on a textual DSL, named MLContext, which has been specially tailored to model context information. This DSL allows creating platform-independent context models which are used to automatically generate context management software artifacts for context-aware middleware platforms. This DSL has been designed to cope with the following three features:

i) to provide a *high-level abstraction* for building platform-independent models.
ii) to be a language which is *simple and easy to learn*, so that final users can write and understand context models.
iii) to promote the *reuse of context models* in different context-aware applications, by separating the implementation dependent aspects from the domain aspects.

As a proof of concept, MLContext has been applied to generate code for the OCP middleware (Botia et al., 2009) and the JCAF middleware (Bardram, 2005). This DSL is a component of an MDD solution we are developing for context-aware systems, which will include a DSL complementary to MLContext in order to specify the application-specific details.

The contribution of this paper is twofold. On the one hand, to the extent of our knowledge, MLContext is the first approach that separates the definition of the context elements from the details related to the context sources and application-dependent aspects. Moreover, unlike other proposals, we provide evidence of how context models allow us to generate software artifacts for middleware platforms, such as ontologies which represent contexts and Java code related to the context management. It is also worth noting that the domain analysis performed to define the metamodel of the MLContext abstract syntax has involved a systematic study of existing context modeling approaches. There is currently a great disparity between these proposals, particularly with regard to types of context, and our domain analysis could be useful in initiatives aimed at exploring the definition of a reference model, which would eliminate such confusion.

MLContext was presented at Hoyos et al. (2010) but the issues related to the generation of software artifacts were only briefly commented. That previous work has been extended here in several aspects. This article is focused on the generative MDD approach and includes a new section that describes in detail how software artifacts can be generated for the OCP and JCAF middleware platforms. An evaluation of our approach is presented in a separate section which shows the results of applying several metrics which have been used to measure the DSL complexity (Oliveira et al., 2009). In addition, we have identified and applied several criteria to contrast MLContext with other existing approaches. Finally, the domain analysis performed as a first stage is explained in more detail. This document is organized as follows. In Section 2, the main issues related to the representation of context are analyzed and the design choices to be taken into account when designing a DSL for modeling context are then identified; an overview of the proposed approach is also presented. In Section 3, the MLContext abstract syntax and notation are presented through an example. In Section 4, we briefly describe the OCP and JCAF middleware platforms and how we can automatically generate code from MLContext models. Section 5 shows the tool we have created to support our DSL. In Section 6 we present a brief evaluation of our work. In Section 7, a survey of current related work is presented. Finally, Section 8 shows our conclusions and future work.

2. DSL design

The main phases in the development of a domain-specific language are discussed in Mernik et al. (2005). The first phase is the domain analysis which aims to identify and describe the domain concepts and their properties. The domain knowledge is gathered from sources such as technical documents, source code or domain experts. In our case, we have analyzed the most relevant approaches and frameworks dealing with context-awareness and context representation. Next, we show the conclusions of the domain analysis, the DSL requirements elicited from such analysis, the choices that we made in the design of the language and an MLContext approach overview.

2.1. Domain analysis

In building a DSL with which to model context, one of the main obstacles is the lack of a commonly accepted definition of *what context really is*. Dozens of context definitions can be found in literature, each of which differs in what information is actually part of a context. The widely used definition of Dey and Abowd (Dey and Abowd, 2000), considers that context information is "any information that can be used to characterize the situation of any entity". An *entity* could be a person, a place, a physical or abstract object, or an event that is considered to be relevant to the interaction between a user and an application, including the user himself. For instance, a patient and a hospital room are some of the entities in the example shown in this paper. This idea of linking context information to an entity or subject is a key aspect of our approach. When entities are modeled, they can be classified in a hierarchy of *categories of entities*. For example, in the context of a hospital, patients and doctors are persons, and doctors are also hospital employees.

We must take into account that the information which is part of a context may come from several *sources* (e.g. a sensor, a tablet or smartphone, a computer system), which may use different *formats* to represent the information they provide. For example, a person's location could be expressed as address information, latitude-longitude coordinates, the ID of the location or in other ways. As indicated in Henricksen et al. (2002), there is usually a significant gap between sensor output and the level of abstraction of the information that is useful to applications, and this gap may be bridged by various types of context information processing. For example, a location sensor may supply raw coordinates, whereas an application might be interested in the identity of the building or room a user is in. Applications usually represent the context information as a set of designer-defined descriptors (e.g. numerical values of time or device IDs). While this format might be appropriate for developers, it could be difficult for the users (who are used to handling higher-level semantic descriptors) to understand and use. Moreover, requirements may vary between applications. For these reasons, the contextual information *should be modeled at a high level of abstraction*, near the user-level, as is stated in Vildjiounaite and Kallio (2007).

When modeling context, it is necessary to distinguish among different *types of context information*, so that a *taxonomy* is useful to appropriately model the concepts. However, most existing approaches do not define such a taxonomy. Instead, they define a single type of context, no matter what type of context information they are dealing with Ayed et al. (2007), Buchholz et al.

(2004), Henricksen et al. (2002), Riva (2006), Sheng and Benatallah (2005), Vildjiounaite and Kallio (2007). Some taxonomies have been proposed such as Ardissono et al. (2007), Chen and Kotz (2000), Göker and Myrhaug (2002), Hofer et al. (2003), Pires et al. (2005), Schmidt (2005), and when they are compared, the following is found. Some types of context, such as the social context or the physical context, are common to nearly all the proposals, while others like the computational context only appear in some of them. This is because most of these classifications are focused on specific domains. For example, Ardissono et al. (2007), make their classification by bearing in mind the management of context-aware web services based workflow systems, and Schmidt (2005) only considers contexts that are relative to e-learning systems in order to consider the learner's situation in an appropriate manner. Moreover, the same information is considered to be of different context types, such as the role of a user in Göker and Myrhaug (2002) and Pires et al. (2005), or the bandwidth consumption cost in Göker and Myrhaug (2002), Pires et al. (2005) and Chen and Kotz (2000). On the other hand, some contextual information should be modeled as a state of the context rather than as part of it, e.g. the "safety level of a scenario" context type defined in Alarcon et al. (2005) should be represented as a set of rules defining a range of temperatures or a level of pollution in a physical context.

A context model should normally include information on *spatial* and *temporal* aspects (Göker and Myrhaug, 2002). Spatial information describes concepts such as location, direction or places. With regard to temporal aspects, context information can be characterized as being static or dynamic. The value of static information remains unchanged over time whereas the value of dynamic information may vary depending on the time instant it is requested.

Two important observations are made in Henricksen et al. (2002). The first is that "context information is imperfect" and hence the system must therefore assure (insofar as possible) information reliability and availability. These two properties can not be really assured by the system but they should be represented in the context models to be able to reason about the levels of availability and reliability at any given time. For instance, the accuracy of the information (i.e. the distance to the true value) normally depends on the sensors' accuracy and it should be modeled with them. Then, this information could be used to choose the source of information with the required accuracy when more than one is available for the same context information. The second observation is that "context information is highly interrelated" because complex information could be composed of several simple contexts. Furthermore, the context of an entity often refers to the other entities' contexts (for example, an entity formed of other entities).

Other features of a context model such as the *user profile* and the *context history* (Hong et al., 2009) are normally considered to be optional. The user profile is a collection of personal data associated with a specific user. This information can be exploited by systems taking into account the users' characteristics and preferences. Context history becomes necessary when information is measured over time and it is necessary to keep a trace of its values in the time dimension.

Finally, note that contextual information is one of the aspects to be considered when creating the domain model of a context-aware application. Other aspects would be related to services, system adaptation, sensors, communications or security among others. This article is focused on the modeling of the context and these other aspects are not considered. We use the "context model" term to refer the model representing the entities and context sources which are part of a particular domain, as well as the entity–entity and entity–source relationships.

2.2. DSL requirements

The MLContext requirements have been identified from the performed domain analysis and our experience using the OCP platform. We have identified therefore the following requirements:

i) High-level abstraction. The language should provide a high-level abstraction by means of constructs that are close to the domain concepts (e.g. entity, context or type of context). This would thus encourage users who are not developers to participate in context modeling.
ii) Types of context. Instead of having only a single type of context, the language should support the ability to model different types of context, such as physical, social or computational context, because this is closer to reality. Moreover, separating information in multiple contexts can be exploited to achieve improvements such as more efficient storage and retrieval methods (i.e. an application might be only interested in context information of the physical type).
iii) Platform-independent models. The language should free the user from providing implementation details. Since both the context of an entity and the source from which context information is obtained can vary, a language with which to model context should allow the declarations of entities and sources to be separated in a platform-independent model.
iv) Application domain independent and model reuse. The language should promote the reuse of models. Since contexts have many similarities in different context-aware applications, MLContext should allow contexts to be modeled regardless of the context-aware application.
v) Traceability, quality and temporal mechanisms. The language should provide a traceability mechanism to support historic information, a mechanism for representing the quality of the supplied information, and it should support both static and dynamic temporal information.
vi) Ease of use. Finally, the language should be easy and intuitive to use.

2.3. Design choices

In order to satisfy these requirements, we have made some design decisions, which are discussed below. As output of the domain analysis task, we have also established the concepts and relationships between them, which have been used to build the metamodel of the abstract syntax of the language, which is described later. The three main concepts around which the definition of the language is organized are *entity*, *context* and *source of context*, and a context can also be composed of other related contexts.

Unlike other approaches, the modeling task in MLContext is *centered on entities* rather than categories. That is, in the specification of a context, we do not model categories of entities (e.g. a patient) but rather specific instances (e.g. patient 'Burt Holmes'). This is mainly because context sources are linked to entities rather than categories. In fact, two entities from the same category may have different context sources for the same property. Another reason is that the developer must give detailed information about each entity when instances of a category are created. The user groups different entities under a common name (i.e. the name of the category), then MLContext automatically creates a category with that name and infers its properties based on the properties of the grouped entities, excluding inherited properties from other categories. Changes in the properties of the entities will be automatically propagated to their categories. Therefore, the developer does not have to specify the properties of the categories. This clearly saves time and effort.

It may sometimes be necessary to explicitly define a category, as when dealing with legacy systems in which the number of entities may be very high and the information about the entities can be obtained from a database or repository to create the instances. In these situations, the developer should specify a *generic entity* (an entity with no specific values for its attributes), and assign it to the category, so MLContext can generate the category based on the properties of this entity. In the case of a legacy database, a migration application would generate each of the instances of this category from the database information.

As part of the domain analysis described in Section 2.1, we surveyed several proposals of context taxonomies. This review allowed us to define a taxonomy for our approach, which includes the most commonly used types of context. Table 1 shows the correspondence between the types of context chosen for MLContext and those proposed in other approaches. The top row of the table shows the types of context of our taxonomy organized in columns. The other rows show the types included in the different context taxonomy proposals. The columns of each row are organized to show how the types of the taxonomy correspond to the types of our proposal (top row). Note that the proposals differ in the types of context and also in their meanings. Sometimes, the context information of one context type corresponds to more than one type in our proposal. An (*) in a cell of the table indicates that the context information corresponds to several types of context in our taxonomy.

Next, we indicate the context information for each one of the types included in the proposed taxonomy.

- *Physical*: This includes all physical magnitudes (e.g. time, speed, temperature, light level, and noise level).
- *Environment*: A description of the physical space distribution by providing information on the people and objects surrounding the user (e.g. distances, user's location and objects within other objects).
- *Computational* (System): This provides information related to the software and hardware of computer systems (e.g. network traffic conditions, hardware status, information accessed by the user and memory requirements).
- *Personal*: This describes the user's profile and his/her psychological state (e.g. competences, preferences, age, gender and clothes he/she is wearing).
- *Social*: This describes the social aspects of a user, region or place, (e.g. law, rules, friends, enemies, neighbors, co-workers, relatives and role).
- *Task*: This provides information about the different tasks the user (or other entity) can perform (e.g. activities that must be carried out, activities that the user is currently performing or tasks that the user is able to perform).

The designer can use these context types to construct models which are closer to reality than the commonly used representations for context information.

Since *entities* are the first class elements, in MLContext, and a context is always linked to an entity, greater granularity is achieved in our approach than in category-centered proposals. The context of an entity is generally composed of several related contexts of different types, and a distinction can therefore be made between the complex context (the *parent context*) and simple contexts (the *child contexts*) of which it is composed. Simple contexts are formed of a set of contextual information of the same type and can refer to other entities. The information gathered in simple contexts may come from different sources. Naturally, as sources are usually a set of devices, and are potentially heterogeneous, each source can have a different information format. In MLContext models, the source descriptions are separate elements in order to offer users a high level of abstraction. In this way, context information is independent from both the sources of context and the format of raw data retrieved from sensors.

The language allows us to express whether the information is static or dynamic, whether or not it must be registered in the trace, or the accuracy of the information supplied by the sources.

MLContext models do not contain platform or application-dependent details in order to promote the reuse of context models in different context-aware applications, as explained in Section 2.4. For example, a complete and detailed context model of an art museum could include information about the location of surveillance cameras, presence detectors and fire detectors. It could also include the spatial distribution of rooms, location of emergency exits and a complete profile of every art object on display in the museum. By applying MLContext, the same context model could be used in different applications as a surveillance system, which can trigger an alarm if an intruder is detected at night; an emergency system, which can detect the presence of fire and guide the people to the nearest emergency exit; a guided-tour system, which can guide visitors through the museum and explain each object of art they are looking at.

MLContext has been defined as a textual DSL whose specifications (i.e. models) are readable and easy to learn for domain expert users who are not experienced in modeling languages such as UML, as will be shown later in Sections 3 and 6. It is applicable to any context-aware application since it only includes the basic concepts for context modeling. It is worth noting that even when a graphical notation is appropriate for a DSL, the creation of a textual DSL is recommended as a first step since the required effort is, in this case, considerably smaller and a better understanding of the problem is acquired during the implementation (Völter, 2009).

2.4. MLContext approach overview

To conclude this section we present an overview of the approach we propose in order to apply MDD techniques in the development of context-aware applications. MLContext is the core element of a model-driven strategy which aims to automatically generate software artifacts related to the management of context information.

When a model-based approach is defined for context-aware applications, context models are not the only required models, but other models are used to represent different aspects of the system. The purpose of each model and the relationships among them may vary depending on the proposal.

Tasks, goals and roles, among other aspects, are normally modeled as part of a domain model which also represents the concepts of the application domain (like an airport or a hospital). Furthermore, most existing approaches consider that the context models should not only represent the context information (i.e. information on the application entities), but they should also represent additional information such as situations, activities or sensor details (Sheng and Benatallah, 2005; Sindico and Grassi, 2009). Fig. 1 shows how a context model is one of the different aspects to be considered in a domain model of context-aware applications.

As indicated above, some information, which is normally included in the context models, is really specific to a particular application. In consequence, these models cannot be therefore reused in other application scenarios. Since the context model reuse is one of the requirements to be satisfied, our approach proposes to remove the specific details of concrete applications from the context model. Thus, if we remove these details from the context model, to completely create the application domain model the developer will need, in addition, to specify the technical details of the sensors and sources of context, and the quality parameters of these sources, among other information. The reliability and availability of the information depends on the quality of the sources of context. This information would not be part of the context model,

Table 1
Taxonomy of context.

MLContext	Physical	Environment	Computational	Personal	Social	Task
Hofer et al.		(*) physical +logical				logical
Schmidt		(*) social		(*) personal	social +organizational	organizational
Chen et al.	physical +time	(*) user +computing	computing		user	
Alarcon et al.	physical	(*) task	computational +interaction +technology		social	task
Goker et al.	environment +spatio temporal +personal			personal	social	task
Pires et al.	(*) spatio-temporal	(*) spatio-temporal+social		physiological +preferences	(*) spatio-temporal	spatio-temporal

since devices can be replaced by others with different characteristics, and should be specified in one separate application-specific model, which could not be reused by other applications.

This separation of concerns promotes reusability and keeps MLContext models simple and readable. The models also retain enough information to be useful for generating software artifacts related to the context management as shown in this article. Note that removing application details from the context model is appropriate for applications that use simple relationships (e.g. simple human-centric applications) like in our hospital example (which will be introduced in Section 3.2). However, in complex applications (like device-centric or those called self-* aware systems), we must face complex relationships. One example of this kind of relationships could be the "is near" relationship. The meaning of this relationship can vary depending on the domain of the application, so it can be "a few kilometers" for a GPS car application or "less than fifty centimeters" for an automatic door opening mechanism. As the semantics of this relationship can vary depending on the application, it must be specified separately from the context model.

We have therefore organized the information typically included in a context model into two models: the *application model* and the *context model*, as shown in Fig. 2. The *context model* represents the context information which includes the entities, their properties and the sources of context available to the context-aware application. On the other hand, the *application model* represents those aspects related to activities and situations, as well as information related to sensors (including technical details such as precision, resolution, etc.). The definition of an *application model* requires referring to elements included in the *context model*.

This approach also has some benefits regarding evolvability. For instance, we could make changes at the application level (e.g. one of the sensors is changed by another with higher precision) and we would not need to change the *context model* or the *domain model*.

3. A DSL for modeling context

A DSL, normally, consists of three elements (Kleppe, 2008): abstract syntax, concrete syntax, and semantics. The abstract syntax defines the DSL's concepts and the relationships between them, and also includes the well-formedness rules which constrain how the models can be created. In MDD, the abstract syntax of a DSL is specified in a metamodel, which is created by using metamodeling languages (e.g. Ecore). The concrete syntax defines a notation for the abstract syntax, and semantics is normally provided by means of a translation to a language whose semantics is well understood, for instance a general purpose language (e.g. Java). The

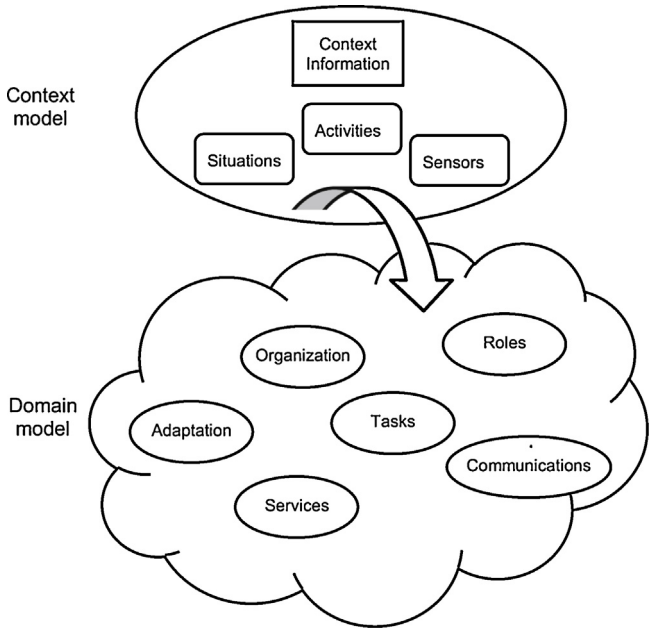

Fig. 1. Domain model and context model with application details.

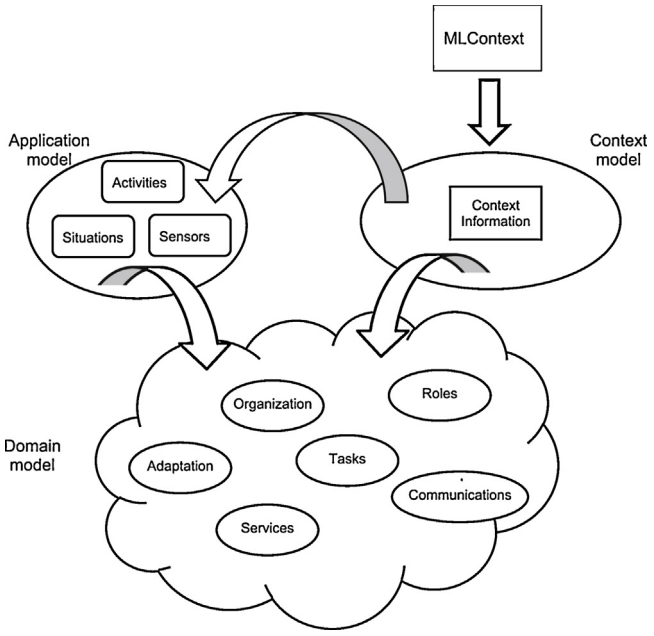

Fig. 2. MLContext approach.

MLContext DSL has been built according to the requirements and design choices discussed in the previous section. First, we show the metamodel of the language, and the notation or concrete syntax is then illustrated by means of a simple example of a context-aware application. Section 4 shows how MLContext models are transformed to software artifacts.

3.1. An example of context-aware application

We have chosen an example based on a simplified case study of a hospital, in which doctors, patients and the hospital equipment must be taken into consideration to represent the context of the application. Let us suppose that we need to develop a ubiquitous computing-based application for the management of the daily tasks of both doctors and assistant personnel (e.g. nurses and assistants). Let us also suppose that patients are identified by RFID labels attached to their personal health report. Moreover, personnel at the hospital are equipped with either smartphones or tablets with mobile communication capabilities. In this scenario, a typical ubiquitous service quickly allows the doctor to automatically access information about a patient he is dealing with by using both doctor and patient context location.

The root entity is the *hospital*, which is formed of floors and people. Each floor is formed of rooms and has a fire alarm with a fire

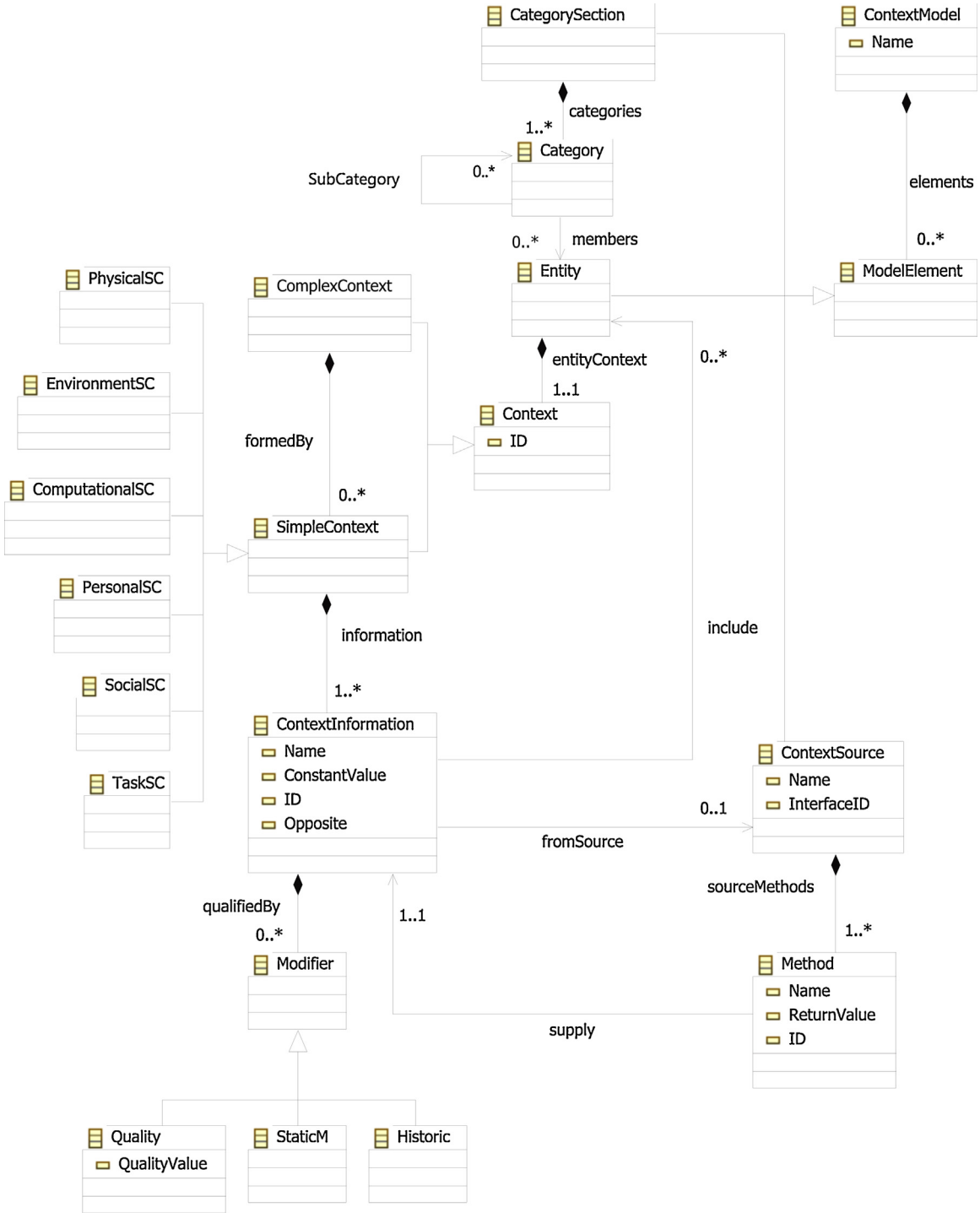

Fig. 3. MLContext's abstract syntax metamodel.

```
entity hospital_UMU context {
   environment { "contains" : floor_01 , floor_02 static }
}
```

Fig. 4. hospital_UMU entity.

```
entity floor_01 context {
   environment { "contains" : room_01, room_02 static }
   physical    { "fire_presence" source } }
```

Fig. 5. floor_01 entity.

```
entity room_01 context {
   environment { "contains" : bed_01 , tv_01 static }
   physical    { "temperature" source } }
```

Fig. 6. room_01 entity.

```
entity tv_01 context {
   computational { "status" source } }
```

Fig. 7. tv_01 entity.

```
entity bed_01 context {
   environment { "contains" : person_02 } }

entity person_02 context {
   personal { "name" :       "John"  static
              "surname" :    "Smith" static
              "age" :        "39" }
   social {   "role" :       "patient" }
   physical { "temperature" source  historic
              "pulse"       source } }
```

Fig. 8. bed_01 entity.

presence detector. For the sake of simplicity, we have represented only two floors and two rooms. *Room_01* contains a bed, a television, a temperature sensor and equipment, which can be attached to a patient in order to monitor his/her vital signs. While doctors (e.g. *Person_01*) are part of the hospital itself, patients (e.g. *Person_02*) are located on beds. MLContext allows us to use any identifier we wish, but entity and source identifiers must be unique in the model.

3.2. MLContext metamodel

The MLContext metamodel is shown in Fig. 3. It has been designed around the concept of entity represented by the *Entity metaclass*, and all context information always refers to an entity as explained above. A complex context (*ComplexContext metaclass*) is an aggregation of simple contexts (*SimpleContext metaclass*) of different types (e.g. physical and social). For example, the context of a person can be an aggregation of a personal context (e.g. name and age), a physical context (e.g. temperature and pulse) and a social context (e.g. role patient). A simple context is formed of one or more pieces of context information (*ContextInformation metaclass*) of the same type (e.g. temperature and pulse are physical information).

There are several types of simple contexts according to the defined taxonomy (e.g. *physical* and *personal*). The context information of a simple context can contain a reference to other entities of the model (e.g. the complex context of one floor of the hospital can contain a simple context with environmental information referring to the rooms it is composed of). This is represented by the *include* relationship in the metamodel. The value of a *ContextInformation* can be a constant or can come from a source of context (*ContextSource metaclass*) as indicated in the *fromSource* relationship. The information from the source is supplied through a method (*Method metaclass*) from the physical device interface. The same source can supply context information to several entities and can use a different method for each of them.

The *Modifier* hierarchy represents the three optional modifiers for the context information: quality (*Quality metaclass*) to specify the accuracy of the information, historic (*Historic metaclass*) to keep trace of the information, and static (*Static metaclass*) to specify that the information does not change over time (if this is omitted, then dynamic information is assumed).

The *CategorySection metaclass* represents the set of categories of the model. Each category (*Category metaclass*) contains all the entities belonging to that category, and can also be a subcategory. An entity or subcategory can belong to more than one category (i.e. multiple classification).

MLContext's abstract syntax has been modeled using the Ecore metamodel language supported by the Eclipse Modeling Framework (EMF) (EMF, 2012).

3.3. MLContext concrete syntax

A textual concrete syntax has been created for the previous metamodel which can be used to create MLContext models. A declaration must be written for each model element. Since the *hospital_UMU* entity is formed of two floor entities, the context information of the hospital must include the context information of each of its floors. The declaration of the *hospital_UMU* entity appears in Fig. 4. It shows that *hospital_UMU* has a complex context formed of an environment context, which only has a "contains" property, which expresses that the hospital contains two floors (*floor_01* and *floor_02*). These entities must be specified later in the model.

The "contains" property references the entities *floor_01* and *floor_02*. The modifier *static* indicates that this information does not change its value. Note that doctors are not included in the declaration of any room because they can change their locations from one room to another in the model. They are not attached to a specific room or floor but they are in the model. Each floor could be specified as in Fig. 5.

The specification of the *floor_01* entity shows a complex context formed of two simple contexts of the *environment* and *physical* types. The environment information expresses that the floor contains two rooms and the physical information expresses that it has a "fire_presence" property. The *source* keyword indicates that the value of this property comes from a source of context which needs to be declared later in the model (a *fire_alarm*). It would be now necessary to model each of the rooms in the hospital.

In the *room_01* (Fig. 6), there are a bed named *bed_01* and a television named *tv_01*. We are interested in knowing the temperature of the room in order to activate the central heating if the room becomes too cold. This is also context information of a physical type that we can obtain from a source (which we will model later). If there is a high number of rooms with the same characteristics, in order to avoid modeling all of them, we should only model one generic entity "rooms" and specify that the floors contains "rooms" as stated in Section 2.3, but this is not the case in this example.

We might also be interested in knowing if the room's television is turned on (Fig. 7). The context of the television is of the *computational* type. The status of the television can be *on* or *off* (we suppose it can be sensed, so we can obtain this information from a source of context).

Now, let us suppose that there is a patient *person_02* in *bed_01* (Fig. 8). The *person_02* entity has a lot of context information which is useful in our modeling example. This entity would have a complex context formed of the aggregation of *personal*, *social* and *physical* contexts. The *personal* context refers to the profile of the patient which is context information of the personal type. Note that there is no context source associated with the "name", "surname" or "age". This is considered to be information provided by the user.

```
entity person_01 context {
   personal { "name" :    "Bob"     static
              "surname" : "Steward" static }
   social {   "role" :    "doctor" }
   task {     "current" : "none"
              "task_01" : "patients_attending"
              "task_02" : "surgery" } }
```

Fig. 9. person_01 entity.

```
contextSource equip_01 {
   interfaceID : "AdapterX670"
      methodName : "getTemp" {
          supply: person_02.temperature
          returnValue : "float"
      }
      methodName : "getPulse" {
          supply : person_02.pulse
          returnValue : "float"
      }
}
```

Fig. 10. equip_01 source of context.

As indicated in the social context information section, the "role" this person is playing in the hospital is "patient". Other people can play other roles such us "visitor" or "doctor". To monitor the state of the patient, we need to know his "temperature" and "pulse". This information is enclosed in a *physical* context. We assume that the patient is connected to an *equip_01* source, which can provide us with the required context information. Note that, in this case, we will associate the same source with two different pieces of physical context data as we will see in the *equip_01* source definition.

While the patient is in *bed_01*, the source *equip_01* could be associated with the physical equipment in room *room_01* containing this bed, but if the patient is moved to *room_02* the source *equip_01* will be attached to the physical monitor of this new room. The binding between an entity and its context sources is expressed in the definitions of the sources, as will be shown later. We have specified that "temperature" is contextual information of the *historic* type because we are interested in keeping a trace of its values over the time.

The declaration for a *doctor* entity (Fig. 9) provides an example of *task* context. In this case, the *task* context provides us with information about the tasks that this person can perform and the task he is currently performing ("none" in the above example). Notice that we are specifying at design-time a property whose value will be changed at run-time by the middleware.

Once we have modeled all the entities, we have to specify the context sources, which correspond with the source elements, in order to establish a match between each source and its physical device. For the sake of space we only show one example of a source definition (Fig. 10).

The *interfaceID* keyword refers to the identification of the physical device adapter (e.g. an X10 standard address), which will provide us with the desired information. This source provides values for the temperature and pulse properties of the *person_02* entity. The *methodName* keyword refers to the method to be invoked to retrieve information from the adapter, and the *returnValue* to the type of the information returned. These methods provide the interface for accessing the source independently of the actual interface of the source (i.e. the Adapter design pattern is applied).

In order to specify the categories to which entities belong, it is necessary to add a *categories section* to our model as in Fig. 11. Note that we could specify the category in each entity definition but we have decided to put all the definitions together instead. A change in the name of a category does not, therefore, force us to change it for each of the entities. Moreover, we can use the same declaration to discover all the entities belonging to one category rather than searching through the whole model.

```
categories {
   Hospital : hospital_UMU
   Floor : floor_01 , floor_02
   Room : room_01, room_02
   Bed : bed_01
   Person : person_02 , Doctor
   Doctor : person_01
}
```

Fig. 11. Categories section.

The aforementioned definition states that the *hospital_UMU* entity belongs to the *Hospital* category, the *floor_01* and *floor_02* entities belongs to the *Floor* category, and so on. A category can have subcategories resulting in a category hierarchy. In the previous example the *Doctor* category is a subcategory of the *Person* category.

We do not need to specify the properties for each category because MLContext can infer them from its entities (the union of the properties of the entities belonging to that category). In MLContext we can define relationships between entities by defining properties whose values are also entities. MLContext automatically establishes relationships between categories based on the relationships of their entities. Note that it is possible to explicitly define a category as we explained in Section 2.3.

MLContext takes into account the properties inherited from other categories (for example the *Doctor* category will have only the "task" properties included on *person_01* because *Doctor* is a subcategory of *Person* and the "name", "surname" and "role" properties are inherited from it). A property in a category indicates that a member of that category could exhibit the property, but this is not mandatory. When generating code, the data-type of a property is determined by its source of context or by analyzing its constant value. In the case of a property name collision (two entities which have the same property but different data-type under the same category), a warning is shown so that the user can make the appropriate decisions.

When an MLContext model is compiled an executed to generate software artifacts, if the designer did not specify a category for some of the entities, a warning message is shown and the categories are automatically created from the name of the entities. For example, a *C_tv_01* category would be created for entity *tv_01* in our model because we have not specified it.

4. Generating code from the MLContext models

As we previously mentioned, MLContext is a DSL specially designed to be used following an MDD approach. A DSL engine normally includes model-to-text transformations which convert the DSL models into software artifacts of the final application. When the semantic gap between the source model and the target artifact is high, one or more intermediate model-to-model transformations can be applied to implement the transformation process in several steps. In our case, MLContext models have been converted into software artifacts using only model-to-text transformations, since we have not encountered great difficulty in the mapping between MLContext and code (e.g. Java code or ontologies). Moreover, we have created an API to facilitate writing model-to-text transformations for MLContext models.

The MLContext models represent the context information at a high-level of abstraction and they are independent of any concrete middleware. With regard to the formalisms provided by existing context-aware middleware for representing context information, MLContext makes modeling context easier to developers. An MLContext engine (also known as code generator) generates therefore the context representation used by a concrete middleware (e.g. an ontology in OCP and Java classes in JCAF. The OCP

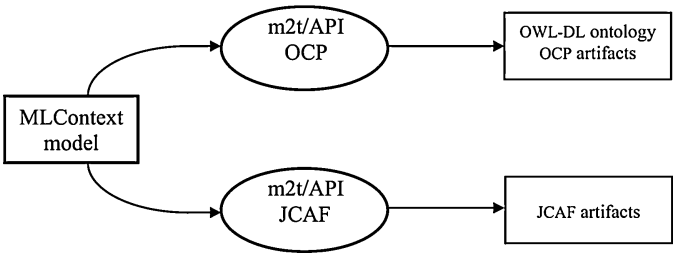

Fig. 12. Code generation from MLContext models.

and JCAF middleware platforms will be explained in Section 4). In addition, other software artifacts required by a middleware (e.g. Java code of producers in OCP), could be derived from MLContext models. As indicated above, an MLContext engine for a particular middleware is composed of several model-to-text transformations, one for each kind of target artifact.

Since MLContext models are independent of the platform, the same model could be used in different middleware platforms to generate software artifacts. Naturally, this require the existence of an MLContext engine for a particular middleware. For instance, we have generated OCP and JCAF artifacts from the same MLContext model. On the other hand, MLContext models could be reused for different applications on the same platform. For instance, the MLContext model for the hospital example could be reused in different applications by combining it to specific application models as explained in Section 2.4. In addition, artifacts generated as an ontology are useful for applications based on the same context.

It should be noted that the code generation will be limited by the capabilities of the target platforms. For example, the Context Toolkit middleware (Dey, 2000) does not have any representation for entities or categories, so we can not generate any code to manage these concepts.

We have chosen OCP and JCAF middleware in order to assess the potential of MLContext in the generation of code. OCP is a framework we are developing for adaptive applications, and JCAF is a well-known Java-based middleware. Fig. 12 shows how a separate code generator has been created for each middleware.

The model-to-text transformations, which are part of each generator, have been implemented with the MOFScript language (MOFScript, 2012). To facilitate the implementation of MLContext engines, we have created a MOFScript API which simplifies the writing of model-to-text transformations by avoiding a complex navigation of the MLContext models. In the following sections we will explain briefly the main characteristics of the OCP and JCAF middleware platforms and then, we will show how we have automatically generated code for them.

4.1. The OCP middleware

OCP (Open Context Platform) (Botia et al., 2009) is a framework for the development of adaptive applications. Adaptiveness is the capacity of a system of reacting in order to appropriately respond to changes in the environment. Context-awareness is a necessary property of these systems since it permits these changes to be characterized.

Adaptive systems can be developed as Java applications on top of OCP in a layered arrangement. These applications use OCP to generate and obtain up-to-date context in a variety of configurations ranging from totally decentralized to simple distributed and monolithic systems. Thus, applications are not aware of the communication infrastructure used to distribute context information. They only see an API through which they may play two different roles: *producers* and/or *consumers*.

A *producer* is a software element (which usually represents a hardware element, e.g. a sensor) which produces data and sends it to OCP to fill up the context of entities involved in the system (usually users). A *consumer* is a service or application which consumes context information (e.g. a location service which delivers location information to other services or applications which need the context of the users that must be located in the system). Different versions of the middleware have been developed, including a non-distributed application (monolithic), a totally decentralized version of the middleware over an ad hoc network and an OSGi (OSGi, 2012) based version of the middleware (Navarro et al., 2009). An approach based on using a DSL as MLContext would thus help a developer ignore some particular aspects of specific versions of OCP, focusing only on modeling aspects. Four elements are required to develop an OCP application:

i) an OWL-DL ontology to describe the contextual information of the application, which constitutes the knowledge base.
ii) a Java class for each of the information producers.
iii) a Java class for the consumers of the information, which depends on the context-aware application to be developed.
iv) an initialization Java class that generates the individuals (entities in the MLContext model).

We have been able to automatically generate the ontology as well as the Java code for basic producers and consumers from an MLContext model. While the ontology and the consumer code are totally generated, the producer code is partially generated. Therefore, the developer must later complete the producer code with application-dependent details. This code generation process saves a significant amount of time and effort in developing with OCP as we discuss in Section 6, where we provided an estimation of the level of automation. Next, we describe how the ontology and producer/consumer code has been generated through model-to-text transformations.

4.2. Generating code for the OCP middleware

The transformation from an MLContext source model to an OWL ontology is carried out in four steps. Each one of the steps corresponds to a mapping between an MLContext element and an ontology element:

i) **Generate ontology headers**. In this step, a common XML header and a set of entity definitions are generated as part of a document type declaration (DTD). These entities are used to define the RDF namespaces of the ontology that will be used as part of the XML tags in the next step. The *name of the ontology* is created from the *name of the MLContext model*. A comment, a label for the ontology and some OCP imports are also generated in this phase.
ii) **Generate ontology classes**. The *OWL classes* are generated from the *categories* of the MLContext model, specifying subclasses when necessary and bearing in mind the inheritance hierarchy. An OWL class is generated for each category.
iii) **Generate ontology datatype properties**. A *DatatypeProperty* is generated for each *property of the model's categories*. The XML Schema datatype of the property is obtained from the return value of the source method that supplies the property value, or is inferred from the constant value of the property if it has no source of context. At this moment, the supported types are: *string*, *boolean*, *integer* and *float*. The domain of the property will be one of the OWL classes generated in the previous step.
iv) **Generate ontology object properties**. When a property in the MLContext model refers to other entities (a *relationship*), an OWL *ObjectProperty* is generated, rather than a

DatatypeProperty. The domain of this property is the OWL class generated from the MLContext category containing it or, if the property is present in more than one category, the union of these classes. The range of the *ObjectProperty* is the union of all the classes from each entity belonging to the relationship.

Finally, we close all labels and include some *comments* in the generated ontology.

Fig. 13 shows a partial view of the ontology generated for the running example. We have used our own graphical notation rather than XML code for the sake of clarity, where rounded squares represent classes and squares represent individuals. The individuals are created during run-time as instances of the OWL classes (referenced by the io relationship) by the main Java program. The individual *person_02* has all the properties of the *Doctor* class and the inherited properties of the *Person* class. The values of the properties are shown in the rectangles.

The OCP information producers are Java classes which are executed as a thread. As indicated above, these classes can be partially generated from the information contained into an MLContext model. This code generation is organized as follows:

i) A producer class is generated for each information source in the model. The information needed for the Java code of the producer class is obtained from the context information sources of the model, which supply the values of some properties of the entities. Fig. 14 shows the run() method for the equip_01 source producer.
ii) Each producer class includes the methods needed to update the value of the entities' properties based on the methods imported from the adapter (Fig. 14, lines 23–28), and creates a context service to obtain the context of the entity (not shown in the figure). Then, the producer class adds (as a context item) the value of the supplied information to the context of the entity (Fig. 14, lines 32–34).
iii) The name of the Java class is obtained from the name of the source in the MLContext model (in Fig. 14 it is equip_01 class).
iv) There is also a call to an Adapter object method to obtain the value of the information supplied by the physical adapter (Fig. 14 line 25). The Adapter Java class must be implemented by the developer or the manufacturer of the physical adapter, because the Java code only includes a call to the adapter method based on the name of the source method of the MLContext model.
v) We also generate an additional Java file (not shown in the example) which creates the skeleton of the main program that initializes all context instances and producers.

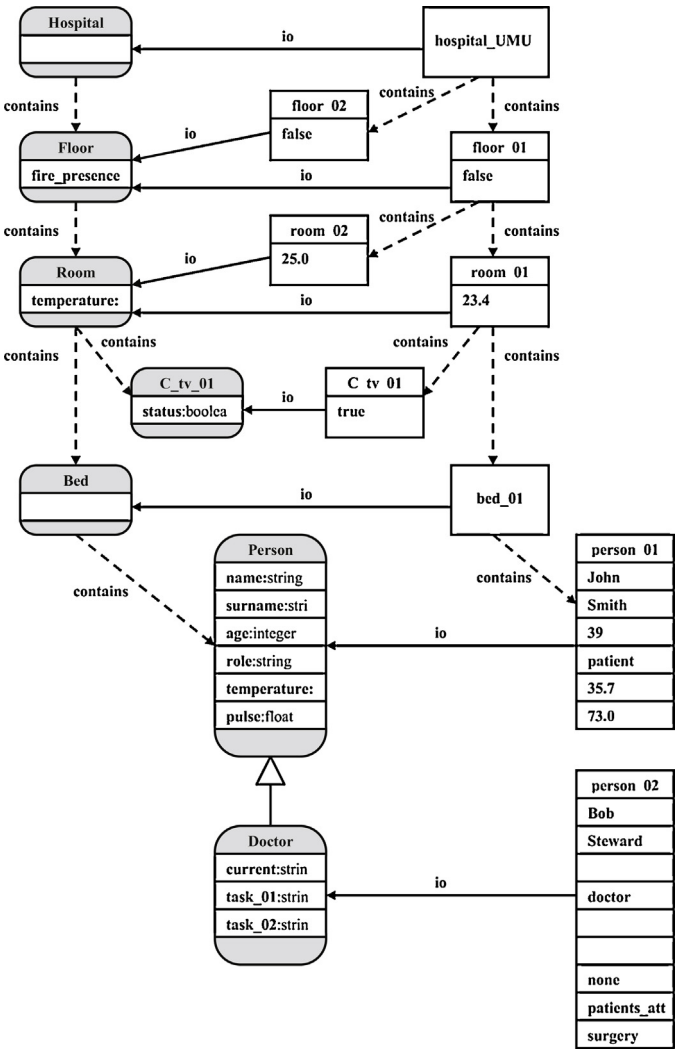

Fig. 13. An excerpt of the OCP generated ontology for the hospital example.

4.3. The JCAF middleware

The Java Context-awareness Framework (JCAF) (Bardram, 2005) is a Java-based infrastructure and programming API for creating context-aware applications. This framework provides a Java API for the definition of the context of an application as an

```
12 public void run() {
13
14             int i = 0;
15             while (i < NUM_OPERATIONS) {
16                 i++;
17                 try {
18                     Thread.sleep( (long)(WAIT_TIME) );
19                 } catch (InterruptedException ex) {
20                 i = NUM_OPERATIONS; continue;
21                 }
22
23          //---- Add adapter interface code here ----
24                 float returnValue;
25                 //returnValue=AdapterObject.getTemp();
26                 float r = (float)Math.random()*100.f;
27                 returnValue=r;
28          //---- End Adapter interface section ----
29
30                // check that context service is active
31                waitForServiceActive();
32                // update context entity property
33                cs.setContextItem( entityClass, entityId, "temperature",
34                                   returnValue);
35             }
36             // finalize
37             ((OCPService)cs).terminate();
38             System.out.println(id + " closed.");
39         }
40 }
```

Fig. 14. Code example from the equip_01 source producer.

object-oriented model. This API includes some core modeling interfaces, such as *Entity*, *Context*, *Relation*, and *ContextItem*, as well as default implementations of these core interfaces. For example the *GenericEntity* class implements the *Entity* interface and can be used to create concrete entities (e.g. patients, beds, pill containers) using specialization. A context (e.g. a hospital context) would be modeled as a class implementing *Context*. Physical location and the status of an operation are examples of context items which could be modeled as instances of a class that implements *ContextItem*. Examples of relations are "using" or "located". Hence, we can model that "doctorX is located in room 123" where "doctorX" is the entity, "located" is the relation, and "room 123" is the context item. An entity also implements the *ContextItem* interface. Hence, one entity can be in the context of another. The JCAF framework can also handle the acquisition and transformation of context information. A *context monitor* can continuously supply context information (i.e. items) to an entity. Finally, the JCAF API provides the *ContextService* interface which has methods for adding, removing, getting and setting entities.

A JCAF developer can represent the context information as MLContext models, instead of using the API provided for this purpose. From these models, an MLContext engine can automatically generate the Java classes which represent the context. Next, we describe the model-to-text transformation implemented to carry out this code generation process.

4.4. Generating Java code for the JCAF middleware

The model-to-text transformations are organized in the following way:

i) A JCAF entity is generated from each one of the categories of an MLContext model. They are Java classes which extend the *GenericEntity* (or other entity if they come from a subcategory). For example (see Fig. 15, line 07) the entity *Person* extends *GenericEntity* and the entity *Doctor* is created as a subclass of Person.
ii) Each of the entities has several attributes (e.g. Fig. 15, lines 08–13) which are generated from the category properties of the context model, taking into account the inheritance relationships. For example, the name and surname attributes are not generated for the *Doctor* entity because it inherits those attributes from the Person entity.
iii) We also generate public setter and getter methods for each of the attributes (e.g. Fig. 15, lines 31–36), which are declared as private.
iv) A JCAF entity also needs a special constructor with an id parameter (Fig. 15, lines 22–24), which is instantiated when an object of the class is created, and is used internally by the middleware. This constructor is also automatically generated.
v) A JCAF relationship class is created to establish a relation between an entity property and the information source that provides it a value. Each of these classes implements the Relationship interface (not showed in the example code).
vi) Entities are notified when changes to their context happens. If this occurs, the context service calls the *contextChanged()* method on the entity (Fig. 15, lines 68–85). Therefore, when the value of some properties vary along the time, depending on a source of context, a *contextChanged()* method is automatically generated for these entities, but not for static properties. This method keeps an entity attribute up-to-date when events occur. The new value is a context item based on a relationship with a source of context.

5. Tooling for MLContext

The definition of a textual DSL involves the development of the tools needed to assist to the language users. Some basic tools are:

i) an editor to create DSL specifications,
ii) a parser (commonly named "model injector") with capabilities to extract models from a DSL textual specification,
iii) a code generator which is able to transform the DSL models into software artifacts.

Several tools currently exist which automatically generate editors and model injectors from either the DSL metamodel (e.g. TCS (Jounault et al., 2006) and EMFText (EMFText, 2012)) or the DSL extended grammar (e.g. Xtext (Xtext, 2012)). We have used EMFText to apply a metamodel-based approach because it can generate an editor which does not contain dependencies to this tool.

EMFText aims to define textual syntax for languages whose abstract syntax is described by an Ecore metamodel. The syntactic information (e.g. keywords) is attached to metamodel elements through a specification formed of a set of rules expressed in a simple language provided by the tool. EMFText uses this specification to generate:

1. an Eclipse editor, which features syntax highlighting, an outline and hyperlinks.
2. a model injector in Java that takes a source specification expressed in its textual concrete syntax and generates a model conforming to the DSL metamodel.
3. an extractor that performs the reverse operation to the injector, and generates textual specifications from the models.

Fig. 16 shows a schema of the tooling for MLContext. Firstly, MLContext textual specifications are written by using the editor, then this text is converted into an MLContext model, by using the injector, and finally this model is then transformed into a software artifact by using a model-to-text transformation.

Given the textual specification of the hospital example described in Section 3.1, the generated model is shown in Fig. 17 in the form of a tree model. This model conforms to the MLContext metamodel (see Fig. 3) and would be the input to the code generator in order to generate the middleware artifacts.

6. Evaluation

The benefits and disadvantages of using DSLs are well-known (Völter et al., 2013) and they must be contrasted to decide if the creation of a DSL is worthwhile. It is necessary to measure to what extent the productivity will be increased with the use of the DSL, as well as other software quality factors as maintainability, portability, integrability and reusability. Both, the productivity improvement and maintainability level reached, mainly depend on easiness of writing and reading, and on the understandability of the DSL code. With regard to the disadvantages, the main factors to be assessed are the costs of development and maintenance, and the cost of training the developers. Next, we evaluate these factors in the case of MLContext and show how a proper balance has been achieved by using the hospital example.

1. **Easy to read and understand**. We have measured some classical metrics to calculate an estimation of the complexity of the language, in order to show some measurement that supports our claims on the simplicity, readability and understandability of MLContext. The metrics applied are TERM, VAR and HAL (Power and Malloy, 2004), which will allow us to obtain a brief

```
07 public class Person extends GenericEntity {
08    private String name;
09    private String surname;
10    private int age;
11    private String role;
12    private float temperature;
13    private float pulse;

...

22    public Person(String id) {
23       super(id);
24    }
25
26    @Override
27    public String getEntityInfo() {
28       return "Person entity";
29    }
30
31    public String getName() {
32       return name;
33    }
34    public void setName(String x) {
35       name = x;
36    }

...

68    @Override
69    public void contextChanged(ContextEvent event) {
70       float new_temperature;
71       float new_pulse;
72
73       if (event.getRelationship() instanceof equip_01) {
74          if (event.getItem() instanceof Temperature) {
75             new_temperature = ((Temperature) event.getItem()).getTemperature();
76             this.setTemperature(new_temperature);
77          }
78       }
79       if (event.getRelationship() instanceof equip_01) {
80          if (event.getItem() instanceof Pulse) {
81             new_pulse = ((Pulse) event.getItem()).getPulse();
82             this.setPulse(new_pulse);
83          }
84       }
85    }
86 }
```

Fig. 15. Excerpt of code generated for the Person JCAF entity.

description of the grammar complexity. We shall additionally use the LRS and LAT/LRS metrics proposed in Ĉrepinŝek et al. (2010), which will allow us to provide a better characterization of the language. For the sake of concreteness, we shall not detail each metric, but simply discuss their meaning with regard to our DSL. These values were calculated by analyzing the DSL grammar structure and rules, using the *gMetrics* tool from Ĉrepinŝek et al. (2010). Table 2 shows the results obtained, along with a brief explanation of each metric, and compares them with those of Java 1.5 and ANSI C.

According to the values in the table, MLContext is clearly simpler than Java or C programming languages (see HAL and LRS).

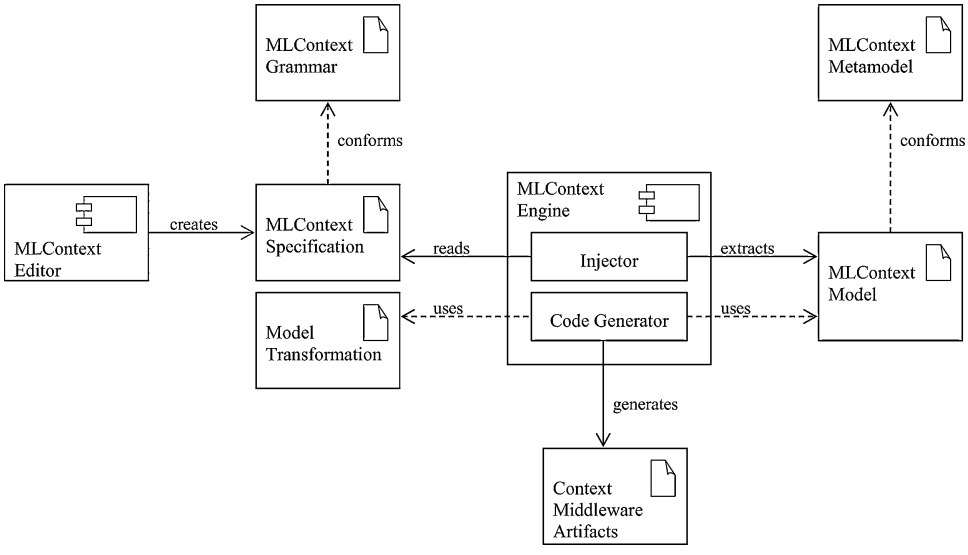

Fig. 16. Tooling for MLContext.

Table 2
Results of applying grammar metrics.

Metric	Explanation	Java 1.5	C	MLContext
TERM	Number of grammar terminals.	102	83	24
VAR	Number of grammar non-terminals.	129	66	8
HAL	Designer effort to understand the grammar.	140.38	42.34	3.54
LRS	Grammar complexity independent of its size.	7741	2512	49
LAT/LRS	Facility to learn the language. Lower value is easier to learn.	0.17	0.18	0.09

Moreover, the low value of LAT/LRS, denotes that MLContext is much easier to learn than the other languages.

2. **Productivity improvement**. The increase in productivity when using DSLs is often measured by calculating the effort saved in writing code for the final application. This effort is normally calculated as lines of code, and it is measured as the difference between the effort to write a DSL specification (i.e. the model) and the effort to manually write the artifacts which are automatically generated. Obviously, this effort should take into account the measurements on the complexity of the language showed in Table 2.

   In the case of MLContext, the number of lines of code are 70 for the abstract syntax, 27 for the concrete syntax, 500 for the MOFScript API, 190 for the OCP transformation and 100 for the JCAF transformation.

   For the small hospital example, we have automatically obtained 17 Java files for the OCP ontology, main program and producers, as well as the JCAF entities, relationships and monitors, with about 700 lines of code. The generated OWL-DL ontology is an XMI file with about 100 lines of code, which can be used by other tools like Protégé 2000. In contrast, the MLContext model source is a file with only 90 lines of code, which are written with a simple language. This is a significant saving in programming effort. Note that the benefit obtained for this small example almost compensate the cost of the MLContext development. For other OCP or JCAF new projects we only need to write the context model.

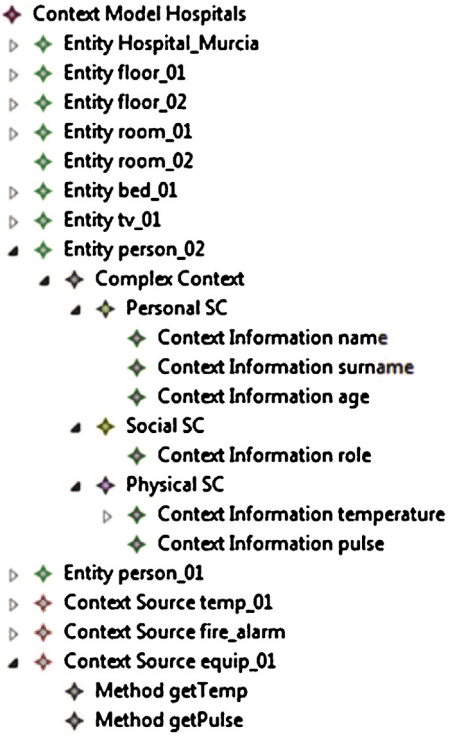

Fig. 17. An excerpt of the generated context model for the hospital example.

   The model-to-text transformations should be written by experts in the target technology in order to generate high-quality code. The MLContext API has been written to help those developers write their own transformations. OCP has been chosen because we have developed it, therefore we are able to write high-quality model transformations. On the other hand, a third-party middleware helps strengthen the evaluation of our approach. This is the motivation to select JCAF, a middleware in which we had no experience. In this case, the transformation was written using examples available in the JCAF web site as implementation patterns.

3. **Portability and reusability**. There are a number of middleware platforms for context-aware systems, and each of them has its own format for representing the context information (e.g. an ontology or Java classes). Our approach proposes the use of a generic Ecore metamodel to represent the context information in a platform-independent way. As proved with OCP and JCAF, the platform-specific context models can be generated from MLContext models. Therefore, MLContext models could be used in any middleware for which a transformation engine can be written (i.e. a mapping between the MLContext metamodel and the target platform can be defined). With regard to the model reuse, the MLContext models do not contain specific application code, so it is easy to reuse the same context model within other context-aware applications.
4. **Maintainability**. This is a typical benefit of using DSLs as they are simpler and more legible than general purpose languages. MLContext models are more maintainable than, for example, JCAF code or XML code.
5. **Integration with other tools**. MLContext is integrated into the Eclipse EMF framework, one of the most widely used MDE development platforms, so the MLContext models can be used by a wide variety of Eclipse modeling tools (parsers, editors, report generators,. . .) and the context model can be automatically serialized in XMI format.
6. **Middleware migration**. Most approaches proposed for representing context information do not support code generation for different middleware platforms (see Section 7). Through a separation of concerns, our approach allows the same context information to be used in different platforms, which could be useful to save effort in a middleware migration.
7. **Cost of training developers**. MLContext is easy to learn and favors the interaction between domain experts and developers. Actually, an MLContext specification could be written by a domain expert (i.e. a person with great knowledge on a given domain but, normally, with any or little expertise on programming).

In order to define a DSL that is easy to learn, we have taken into account some of the criteria and guidelines proposed in Kolovos et al. (2006), Oliveira et al. (2009), Völter (2009) such as:

- The language constructs must correspond to important domain concepts (conformity). In this sense, MLContext has constructs for Entities, Sources of context, Context information, Categories and properties.
- In MLContext, each construct in the language is used to represent exactly one distinct concept in the domain (orthogonality).
- The language should be as simple as possible in order to express the concepts of interest and to support its users and stakeholders

in their preferred ways of working. To keep our DSL simple, the MLContext grammar has a low number of terminals.
- The semantics of the domain is implicit in the language notation. This allows their users to easily create mappings between the syntax of the language and the objects of the problem domain.
- Notations that only express concepts of the domain makes the language efficient for being read and learned by the domain experts.

7. Related work

Since the emergence of pervasive computing, great research efforts have been devoted to the subject of context information modeling and several approaches have been proposed. A survey of context model approaches is presented in Baldauf et al. (2007) and in Bolchini et al. (2007). However, the number of proposals for Model-Driven Development of context-aware and pervasive systems is considerably small, although it is expected to grow significantly in coming years. Most of these works include a solution for representing context information.

To compare the available context model approaches, Bolchini et al. (2007) propose an analysis framework providing a rich set of features to evaluate: what type of formalism is used; whether it is possible to represent location and temporal aspects, user profile, context history, and subject described; whether time and space are represented absolutely or relatively; what kinds of domains can be addressed; what granularity is provided and whether constraints can be expressed, but that survey does not consider context modeling proposals in MDD approaches. The most relevant approaches will be analyzed below.

ContextUML (Sheng and Benatallah, 2005) is a UML-based graphical language aimed at the design of context-aware web services, which include constructs for modeling context information. ContextUML distinguishes between atomic and composite context. However the notion of composite context is different from other approaches which also consider it. ContextUML defines a composite context as a state of a context depending on two or more atomic contexts, for example "(*temperature*>40^{o}C) OR (*rainLikehood*>80%)", and models it by using statecharts, while in MLContext a complex context is the aggregation of two or more simple contexts. Because ContextUML does not include the notion of entity, the contextual information is scattered through the model rather than being related to an entity as in MLContext. The sources of the contextual information are context services. ContextUML does not use a taxonomy or different types of context and does not support code generation.

PervML (Serral et al., 2008) is a DSL for specifying pervasive systems, which has been defined for a MOF metamodel and its concrete syntax is based on UML 2.0 notation. In contrast to MLContext, PervML is more oriented towards representing services than context and the context information is not separated from its sources. The context information is mixed with domain-specific technical details from the sources, making the model application-dependent and non-reusable by other applications. PervML uses a transformation engine to translate the PervML models to an OWL ontology. It does not use a taxonomy or types of context but makes use of context information that could be classified by type.

An MDD approach for the development of context-aware systems is described in Ayed et al. (2007), where a process with six phases is proposed. This process takes into account both the collection of the context and adaptation mechanisms, and the languages needed for each phase are defined by means of an UML profile. With regard to the context modeling, the identification of the required context information must be performed during the first phase, specifying several context elements such as the context types or the required context quality. These specifications must be made independently of the platform that will be used to collect the information. The UML profile for the context model includes the elements for collecting information (starting time of collection, number of samples, rate of sampling,...) which depends on a particular application. This approach does not propose different types of context, and it does not allow to generate code but makes use of model-to-model transformations to create platform specific models.

CAMEL (Sindico and Grassi, 2009) is an Ecore metamodel for representing the behavior and adaptation of context-aware systems, but it lacks a concrete syntax and no code generation has yet been defined. In CAMEL the context-awareness concern is handled by means of three separated parts: context sensing, context adaptation triggering and context adaptation. It does not use a context taxonomy. Instead, it refers to generic contextual information. The adaptation and behavior specified in the model makes it non-reusable, because it depends on the application which it was created for.

WebML (Ceri et al., 2002) is a visual language for specifying the context structure of web applications and its presentation in one or more hypertexts. Firstly, an Entity-Relationship model or a UML class diagram is used to specify how the application contents are organized. Then, a WebML hypertext model describes how contents specified in the data schema are published in the application hypertext. WebML can generate code for adaptive web applications that can be manually adapted by web programmers. This proposal cannot be applied to model context-aware systems in general. The WebML model is also dependent on the application for which the hypertext was created.

Henricksen et al. (2002) have attempted to model a context-aware communications scenario by using both entity-relationship (ER) models and class UML diagrams. The agents in this scenario rely upon information about the participants and their communication devices and channels. The authors experienced difficulties in distinguishing between different classes of context information (for example static versus dynamic information or sensed information versus information supplied by users). They found UML constructs to be more expressive than those provided by ER, but also more cumbersome. They suggest that the most appropriate approach for modeling context information is that of using special constructs designed with the characteristics of context in mind. In this respect, MLContext constructs are specially designed to model context information.

Table 3 compares MLContext with other approaches considering several properties. The first column indicates how the context modeling language has been implemented, the second indicates whether it has a code generation engine, the third whether types of context are supported, the fourth whether platform independence is promoted and the last whether the language is independent of the kind of context-aware application (i.e. if the context model can be reused by other applications).

Most context model approaches have proposed a UML profile as their modeling language, but one of the MDD current best practices is the creation of DSLs, since their design and implementation is not restricted by the limitations of the profiling mechanism (Kelly and Pohjonen, 2009; Völter, 2009). Unlike MLContext, UML profiles offer a limited extension mechanism since totally new types can not be added to the language. The use of these profiles for code generation therefore necessitates accessing the extended language concepts via mandatory and possibly unnecessary UML concepts.

To the best of our knowledge, all current approaches include target platform or application-dependent aspects in the context model. Since MLContext proposes to represent this information in a separated model, the models are thus more readable and closer to the user's conceptual domain. For instance, ContextUML does

Table 3
Comparison of different approaches.

	Implementation technique	Code generation	Types of context	Platform independent	Application independent
MLContext	Textual DSL	Yes	Taxonomy	Yes	Yes
ContextUML	UML Profile	No	No	Yes	No
PervML	UML Profile	Yes	No	Yes	No
CAMEL	UML Extension	No	No	Yes	No
Ayed et al.	UML Profile	No	No	Yes	No

not allow the assignation of a different context source for the same attribute in each of the instances of its classes to be clearly expressed. In addition to this, the contextual information is not related to an entity and is scattered throughout the model. This makes it difficult to generate code from a ContextUML model to some middleware platforms like JCAF, which makes use of entities. In JCAF, an entity is a small Java program that is executed in a context service. An MLContext model contains the information required to generate the Java code of the entities programs and part of the code of the context services.

It is also worth noting that ContextUML models can be expressed in MLContext with the exception of information specifying services, which must be specified in a separate application model. With regard to the artifacts generation, other approaches such us the Context Toolkit middleware (Dey, 2000), do not make use of context representation and focus on services. In this case, the information of the MLContext model can be used only to generate a skeleton of the widgets programmed in Context Toolkit.

We conclude this section by considering the Standard Ontology for Ubiquitous and Pervasive Applications (SOUPA) (Chen et al., 2005). The SOUPA ontology consists of nine ontology documents that use the OWL language. SOUPA includes some context aspects which are commons to most of proposal commented in Section 2 such as user profiles or spatial and temporal aspects. Regarding the context taxonomy, SOUPA has types of contexts such as: personal information, time and space. However, SOUPA was defined bearing in mind the Context Broker Architecture (CoBrA), so it includes concepts for interfaces, agents, security policies, and agents priority, which are specific of certain types of applications, and it is intended for user-centric applications and context reasoning. The authors of the ontology do not propose any DSL for context modeling, and it could be very difficult to create a mapping from SOUPA to other context modeling approaches because context information is spread through several ontologies. Nevertheless it would be possible to write a transformation to create part of some of the ontology documents of SOUPA, like the user profile, from an MLContext model.

8. Conclusions and future work

The MLContext DSL is the principal component of a generative architecture for context-aware applications currently under development. To the best of our knowledge, this DSL presents several innovative aspects such as providing a textual language tailored to context modeling and the generation of software artifacts from abstract models which do not include implementation details. The possibility to express the relationships between the entities in terms of roles, which is proposed in the approach presented in Coutaz et al. (2005), is a remarkable characteristic of MLContext. Another characteristic is the separation of the modeling of context processing from the modeling of the matching to context sources.

We have shown a case study illustrating how software artifacts for a middleware platform can be generated from MLContext models. The MLContext tooling and some examples can be downloaded and tested.[1] Since MLContext models do not include information related to the platforms or the specific implementation, these models can be reused in different context-aware applications which are based on the same context. In contrast to existing approaches, MLContext is entity-centered rather than category-centered, and it supports a taxonomy of types of context. These design choices have helped provide a high level of abstraction which allows MLContext to be used by both developers and domain experts. MLContext is being used to generate OCP code in the CADUCEO project[2] intended to create a platform of context-aware services for hospitals. From now on, our work will continue in two different directions. On the one hand, we are working on developing model-to-text transformations to automatically generate code for other context management middleware platforms for ubiquitous computing (e.g. Context Toolkit). On the other hand, there are practical aspects regarding context-awareness that should be considered in the future. One of these aspects is the dynamics of context sources and data flows (i.e. sample rate, starter time of collection, number of samples, etc.). The other kinds of aspects include not only services provided by the middleware and directly related to context information (e.g. context discovery, reasoning), but also adaptive services which should be built on top of the corresponding middleware (e.g. OCP) in order to create complete ubiquitous computing applications on top of such middleware.

We are working on an MLContext extension which let us specify technical details and quality parameters for the sources of context, as well as complex relationship which are dependent on the domain application. This extension will help the developer create an application-specific model that complements the context model.

References

Alarcon, R., Guerrero, L.A., Ochoa, S., Pino, J.A., 2005, July. Context in collaborative mobile scenarios. In: Proc. of the Fifth International and Interdisciplinary Conference on Modeling and Using Context (Context 2005), Workshop: Context and Groupware.

Ardissono, L., Furnari, R., Goy, A., Petrone, G., Segnan, M., 2007. Context-aware workflow management. In: Int. Conf. on Web Engineering (ICWE 2007). No. 4607. LNCS, pp. 47–52.

Ayed, D., Delanote, D., Berbers, Y., 2007. MDD approach for the development of context-aware applications. In: Kokinov, B., et al. (Eds.), CONTEXT 2007. Vol. 4635. LNAI. , pp. 15–28.

Baldauf, M., Dustdar, S., Rosenberg, F., 2007. A survey on context-aware systems. International Journal Ad Hoc and Ubiquitous Computing 2 (4), 263–277.

Bardram, J.E., 2005. The Java Context Awareness Framework (JCAF) – a service infrastructure and programming framework for context-aware applications. In: Pervasive 2005. Vol. 3468. LNCS.

Bolchini, C., Curino, C., Quintarelli, E., Schreiber, F.A., Tanca, L., 2007, December. A data-oriented survey of context models. SIGMOD Record 36 (4), 19–26.

Botia, J.A., Villa, A., Palma, J.T., Perez, D., Iborra, E., 2009. Detecting domestic problems of elderly people: simple and unobstrusive sensors to generate the context of the attended. In: First International Workshop on Ambient Assisted Living. IWAAL, 2009. Vol. 5602 of LNCS.

Buchholz, T., Krause, M., Linnhoff-Popien, C., Schiffers, M., 2004. CoCo: dynamic composition of context information. In: Proceedings of the First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous'04), pp. 335–343.

[1] http://www.modelum.es/MLContext.

[2] CADUCEO Project (IPT-2011-1080-900000), INNPACTO Program, 2011 founded by Ministry of Economy and Competitiveness (Spain) and the European Regional Development Fund (ERDF).

Ceri, S., Fraternali, P., Matera, M., July–August 2002. Conceptual modeling of data-intensive web applications. IEEE Internet Computing 6 (4), 20–30.
Chen, G., Kotz, D., 2000. A survey of context-aware mobile computing research. Tech. Rep. TR2000-381. Dartmouth College, Dept. of Computer Science, Hanover, NH.
Chen, H., 2004. An Intelligent Broker Architecture for Pervasive Context-Aware Systems. University of Maryland, Baltimore, USA (Ph.D. Thesis).
Chen, H., Finin, T., Joshi, A., 2005. The SOUPA ontology for pervasive computing. In: Ontologies for Agents: Theory and Experiences. BirkHauser, Berlin, Germany, pp. 233–258.
Coutaz, J., Crowley, J.L., Dobson, S., Garlan, D., 2005. Context is key. Communications of the ACM 48 (3), 49–53.
Ĉrepinŝek, M., Kosar, T., Mernik, M., Cervelle, J., Forax, R., Roussel, G., 2010. On automata and language based grammar metrics. Journal on Computer Science and Information Systems 7, 310–329.
Dey, A.K., 2000, December. Providing Architectural Support for Building Context-Aware Applications. College of Computing, Georgia Institute of Technology (Ph.D. Thesis).
Dey, A.K., Abowd, G.D., 2000. Towards a better understanding of context and context-awareness. In: Proceedings of the Workshop on the What, Who, Where, When and How of Context-Awareness, New York, 2000. ACM Press.
EMF, 2012. Eclipse Modeling Framework Project (EMF). http://www.eclipse.org/modeling/emf/
EMFText, 2012. EMFText Concrete Syntax Mapper. http://www.emftext.org/
Göker, A., Myrhaug, H.I., 2002. User context and personalisation. In: 6th European Conference on Case Based Reasoning, ECCBR 2002, Aberdeen, Scotland, UK, September 4–7, 2002, Workshop Proceedings, pp. 1–7.
Held, A., Buchholz, S., Schill, A., 2002, July. Modeling of context information for pervasive computing applications. In: Proc. of the 6th World Multiconference on Systemics, Cybernetics and Informatics.
Henricksen, K., Indulska, J., Rakotonirainy, A.,2002. Modeling context information in pervasive computing systems. In: Pervasive 2002. Vol. 2414 of LNCS. Springer-Verlag, pp. 167–180.
Hofer, T., Schwinger, W., Pichler, M., Leonhartsberger, G., Altmann, J., Retschitzegger, W.,2003. Context-awareness on mobile devices – the hydrogen approach. In: Proc. of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03). Vol. 9 of HICSS'03. IEEE Computer Society, pp. 292.1.
Hong, J., Suh, E., Kim, J., Kim, S., 2009. Context-aware systems for proactive personalized service based on context history. Expert Systems with Applications: An International Journal 36 (4), 7448–7457.
Hoyos, J.R., García-Molina, J., Botía, J.A., 2010. MLContext: a context-modeling language for context-aware systems. In: 3rd International DisCoTec Workshop on Context-Aware Adaptation Mechanisms for Pervasive and Ubiquitous Services (CAMPUS 2010).
Jounault, F., Bézivin, J., Kurtev, I., 2006. TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: 5th International Conference on Generative Programming and Component Engineering, Portland, OR, USA, pp. 249–254.
Kelly, S., Pohjonen, R., 2009. Worst practices for domain-specific modeling. IEEE Software 26 (4), 22–29.
Kleppe, A., 2008. Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Addison-Wesley, Boston, United States.
Kolovos, D.S., Paige, R.F., Kelly, T., Polack, F.A., 2006, July. Requirements for domain-specific languages. In: 1st ECOOP Workshop on Domain-Specific Program Development (DSPD 2006), Nantes, France.
Korpipää, P., Mäntyjärvi, J., Kela, J., Keränen, H., Malm, E.-J., 2003, July–September. Managing context information in mobile devices. IEEE Pervasive Computing 2 (3), 42–51.
Mernik, M., Heering, J., Sloane, A., 2005, December. When and how to develop domain specific languages. ACM Computing Surveys 37 (4), 316–344.
MOFScript, 2012. MOFScript (Meta-Object Facility) Model-to-Text Transformation Tool. http://www.eclipse.org/gmt/mofscript
Navarro, S., Alcarria, R., Botia, J.A., Platas, S., Robles, T., 2009, April. CARDEA: service platform for monitoring patients and medicines based on SIP-OSGi and RFID technologies in hospital environment. In: Primer simposio OpenHealth, Spain.
Oliveira, N., Pereira, M.J.V., Henriques, P.R., Cruz, D., 2009. Domain specific languages: a theoretical survey. In: 3rd Compilers, Programming Languages, Related Technologies and Applications (CoRTA'2009).
OSGi, 2012. Open Services Gateway Initiative. OSGi Alliance Home Page. http://www.osgi.org/Main/HomePage
Pham, H.N., Mahmoud, Q.H., Ferworn, A., Sadeghian, A.,2007. Applying model-driven development to pervasive system engineering. In: SEPCASE '07: Proceedings of the 1st International Workshop on Software Engineering for Pervasive Computing Applications, Systems, and Environments. IEEE Computer Society, p. 7.
Pires, L.F., Sinderen, M.v., Munthe-Kaas, E., Pokraev, S., Hutschemaekers, M., Plas, D.J., 2005, October. Techniques for Describing and Manipulating Context Information. Tech. rep., Freeband A-Muse Project, Deliverable D3.5.
Power, J., Malloy, J., 2004. A metrics suite for grammar-based software. Journal of Software Maintenance and Evolution: Research and Practice 16, 405–426.
Preuveneers, D., Berbers, Y., 2005. Semantic and syntactic modeling of component-based services for context-aware pervasive systems using owl-s. In: MCMP-05. First International Workshop on Managing Context Information in Mobile and Pervasive Environments, pp. 30–39.
Riva, O., 2006. Contory: a middleware for the provisioning of context information on smart phones. In: Middleware 2006 ACM/IFIP/USENIX 7th International Middleware Conference Proceedings. Vol. 4290. LNCS, pp. 219–239.
Schilit, B.N., Adams, N.L., Want, R.,1994. Context-aware computing applications. In: IEEE Workshop on Mobile Computing Systems and Applications. IEEE Computer Society, pp. 85–90.
Schmidt, A., 2005. The knowledge maturing process as a unifying concept for E-learning and knowledge management. In: Proceedings of the 5th International Conference on Knowledge Management (I-KNOW 2005).
Selic, B.,2007, May. A systematic approach to domain-specific language design using UML. In: Tenth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2007). IEEE Computer Society, pp. 2–9.
Selic, B., 2008. Personal reflections on automation, programming culture, and model-based software engineering. Automated Software Engineering 15 (3/4), 379–391.
Serral, E., Valderas, P., Pelechano, V., 2008. A model driven development method for developing context-aware pervasive systems. In: UIC 2008. Vol. 5061. LNCS, pp. 662–676.
Sheng, Q.Z., Benatallah, B., 2005. ContextUML: a UML-based modeling language for model-driven development of context-aware web services. In: 2005 International Conference on Mobile Business (ICMB 2005), 11–13 July 2005, Sydney, Australia. IEEE Computer Society, pp. 206–212.
Sindico, A., Grassi, V., 2009, July. Model driven development of context aware software systems. In: COP'09: International Workshop on Context-Oriented Programming, Genova, Italy. ACM, pp. 1–5.
Vildjiounaite, E., Kallio, S., 2007. A layered approach to context-dependent user modelling. In: Advances in Information Retrieval ECIR 2007. Vol. 4425. LNCS, pp. 749–752.
Völter, M., 2009. MD* best practices. Journal of Object Technology 8 (6), 79–102.
Völter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L.C.L., Visser, E., Wachsmuth, G., 2013. DSL Engineering - Designing, Implementing and Using Domain-Specific Languages. dslbook.org
Xtext, 2012. Xtext – A Programming Language Framework. http://www.eclipse.org/Xtext/

José R. Hoyos is an Associate Professor at the University of Murcia (Spain) where he leads the Computer Forensics unit. He holds a BS degree in Computer Science and MS degree in Computer Science and Mathematics applied in Science and Engineering from the University of Murcia. His research interests include model-driven development and ambient intelligence.

Jesús García-Molina is a full professor in the Department of Informatics and Systems at the University of Murcia (Spain), where he leads the Modelum group, an R&D group focused on Model-Driven Engineering with a close partnership with industry. His research interests include model-driven development, domain-specific languages and model-driven modernization. He received his PhD in physical chemistry from the University of Murcia.

Juan A. Botía is an Associate Professor at Universidad de Murcia. He received his Engineer Degree in Computer Science from the University of Murcia (Spain). He was teaching at the Alcalá University in Madrid, Spain, where he finished his PhD at the Department of Communications and Information Engineering, in the group ANTS. His research interests include software agents and multi-agent systems, intelligent data analysis and social simulation. Currently, he focuses the application of his research to ubiquitous computing and ambient intelligence.