Model-driven engineering of non-functional properties for Pervasive service creation

Achilleas Achilleos
Department of Computer Science,
University of Cyprus, Nicosia, Cyprus
Kun Yang
School of Computer Science and Electronic Engineering,
University of Essex, Colchester, United Kingdom
Nektarios Georgalas
Centre of Information and Security Systems Research,
BT Group, Ipswich, United Kingdom

ABSTRACT

Pervasive services are highly customizable and personalized services that must have the capability to run anytime, anywhere and on any device with minimal user attention. The creation of these dynamic services using application level approaches becomes a daunting task for the software engineering community. This necessitates changes to the way services are designed and implemented, in order to simplify and increase the agility of the service creation process. In this chapter we introduce a model-driven development process and an environment that facilitate pervasive service creation using an abstract platform independent approach. Using this approach a context modelling language is defined in the form of a metamodel and a context modelling framework is generated. The framework facilitates the definition of platform independent context models that describe the non-functional requirements of pervasive services. Subsequently, context models are mapped and transformed via the use of the generic environment's capabilities to implementation specific service code. Finally, a pervasive museum case study is presented to demonstrate the effectiveness of the approach for the definition of a context model and the generation of the service implementation.

Keywords: Model-driven; non-functional requirements; pervasive service creation; metamodelling; context modelling; context-awareness.

INTRODUCTION

In the context of software engineering, *services* are generally considered to be software applications that can be deployed and executed on a *specific* device and platform to accomplish conventional computing tasks. However, the notion of *pervasive* services is characterised by a larger degree of flexibility in that they refer to software applications capable of running *anytime, anywhere and on any device* with minimal user attention (Yang et. al., 2005). Such services should be capable of operating in a dynamic environment and provide users with a specialized and personalized behaviour that allows performing dynamic computing tasks. This means in

particular that the service must be able to adapt dynamically on the basis of changing context information and in accordance to certain predefined rules. Furthermore, the service must take into account individual user preferences in order to aid the user and undertake appropriate actions on behalf of the user with increased probability of correctness.

Service creation is a complex process that involves multiple tasks for the rapid analysis, design, implementation and validation of services (Adamopoulos et. al., 2002; Glitho et. al., 2003). The process supports the development of services commonly via the use of a high-level service creation environment. A variety of high-level service creation environments have been developed (Glitho et. al., 2003; Lennox & Schulzrinne, 2000), which attempt to simplify the service creation process. The technology-specific complexities introduced though by these kinds of environments hinder slightly the realisation of this objective. According to our views a high-level service creation environment should steer clear of implementation specific technologies. Hence, an abstract model-driven development environment, as the one proposed in (Achilleos et. al., 2007), is required in order to provide solutions to these open issues.

Context-awareness is the key characteristic feature of pervasive services that indicates the requirement to adapt the service behaviour on the basis of input context information and certain predefined rules. Typically in conventional services information is acquired mainly as input from the user and this profiled information drives the service execution. On the contrary when dealing with pervasive services input information must be acquired from a variety of context sources; e.g. repositories, sensors, users. Consequently, the complexity of the service creation process is further augmented due to the necessity to represent and manage effectively the information obtained from diverse input context sources.

The term context has been interpreted in many different ways during the course of research (Dey & Abowd, 2000). In our work we define context as: *"Any information relevant to the interaction of the user with the service where both the user and the application's environment are of particular interest"*. Context commonly refers to information such as the identity, time, location and activity of the user, together with additional information that are specific to a particular pervasive service. Therefore, understanding which information is termed as context, how to represent them and manage them, is crucial in order to simplify the pervasive service creation process and realize the overall objective of service adaptability.

Pervasive service creation has been studied during the course of research following two complementary directives, namely: (i) infrastructure-level approaches and (ii) application-level approaches. The primary directive focuses on building an infrastructure that provides the capability to sense, gather and process low-level context information required by pervasive services (McFadden et. al., 2004). Although this directive is important, our work aligns with complementary approaches that tackle pervasive service creation at the application level (Strang & Linnhoff-Popien, 2004). This is due to the fact that pervasive service creation requires an abstract model-driven approach, in order to avoid implementation specific complexities. Furthermore, a model-driven approach provides the capability to address issues such as portability and adaptability of pervasive services.

These approaches are commonly termed as context modelling techniques. They deal with management related tasks such as representation, administration and distribution of context information to pervasive services to achieve their adaptation. The principal requirement is the representation of context information at the application level, in a format that can be realised and utilised by pervasive services. Generally the representation of context information is conveyed in the form of a context model. The context model depicts the non-functional properties required

for the creation of pervasive services. These non-functional requirements introduced into the context model ensure that the functionality of the service is kept to the desired level. They define the non-functional aspects of the pervasive service, which are namely context validity, context quality and context privacy. These aspects are essential in order to ensure that the context information provided to the service is valid, is of the best quality possible and access is restricted in accordance to the defined conditions.

The diversity of input context sources is the main contributing factor that introduces the non-functional requirements of the service (Henricksen & Indulska, 2006). For instance information acquired from a static repository is usually of superior quality than information captured from a sensor (Henricksen & Indulska, 2002; Simons & Wirtz, 2007). Furthermore, different context information requires to be treated differently in terms of privacy. For instance a user might want to keep its profile information accessible to all users of the service. Opposed to this, some context information such as credit card details must not be accessible to other users. Hence, different permissions should be set on each context source to restrict accordingly the access to context information.

An ideal context model shall go head to head with the service creation environment into which is to be implemented. A common software engineering technology that underpins both context modelling and pervasive service creation can naturally bring context-awareness into pervasive services at the service compilation stage; i.e. prior to service execution. The Model Driven Architecture (MDA) (Frankel 2003; Kleppe et. al., 2005) paradigm introduced by the Object Management Group (OMG MDA, 2003) comprises such a technology. In our previous wok, a preliminary MDA-based service creation environment has been proposed and verified (Achilleos et. al., 2008). The work presented in this chapter follows on our previous research outcomes to tackle context-awareness and support the engineering of non-functional properties of pervasive services. This is performed using a model-driven technology, in particular, OMG's MDA. MDA's many advantageous features such as high-level abstraction and platform independence simplify the context modelling and implementation tasks associated with pervasive service creation.

The chapter promotes the thought of incorporating context-awareness into pervasive services at the static compile time, i.e. service creation stage. These non-functional mechanisms defined into context models at the service creation stage will be triggered at the service execution phase to provide inherent and therefore much enhanced service adaptability. This is in complementation to the main-stream service adaptation methodology that is largely based on a complicated middleware infrastructure. The main technical contributions introduced in this chapter lie in twofold. Primarily, a model-driven methodology is proposed, which is based on the MDA paradigm and facilitates the service creation process. We utilise the methodology for context modelling and consider context modelling as an integral part for pervasive service creation. Secondly, we practise the methodology to design and generate a Context Modelling Framework (CMF), which is integrated into the MDA-based service creation environment as one of its components to support the design, validation and implementation of pervasive services. Finally, the effectiveness of the integrated environment is showcased and evaluated via a pervasive service case study.

The rest of the chapter is structured as follows: Section 2 presents related research work on pervasive service creation. Section 3 introduces the model-driven methodology and presents the architecture of the generic MDA-based environment. In Section 4 we extract the requirements of the context modelling domain and introduce the proposed CMF. The CMF is then integrated into

the generic environment to comprise the Pervasive Service Creation Environment (PSCE). Section 5 presents the creation of pervasive museum service and performs a preliminary evaluation of the approach using selected software metrics. In Section 6 we present the conclusions and identify future work on pervasive service creation.

BACKGROUND

Recent research work on pervasive service creation focuses on an application based solution to the problem. Several approaches have been proposed that illustrate the divergence in modelling and utilising context information for diverse application domains (Strang & Linnhoff-Popien, 2004). These approaches are commonly termed as context modelling techniques and can be categorized in accordance to the representation proposed for modelling context information. Furthermore these modelling techniques identify the non-functional properties of context-aware services that are namely context quality, context validity and context privacy. These non-functional requirements are intuitively captured in the context model proposed by different context modelling techniques.

Initially Schilit et al (1994) introduce such an approach, in an attempt to model context information using key-value pairs. Key-value models represent context in the form of a value of context information that is delivered to the application as a variable. The proposed approach is characterised by its simplicity in representing context information, something that benefits the technique in terms of its applicability. The simple representation of context information is therefore a plus from the functional management viewpoint but it is a downside if quality, validity and privacy of this information are considered.

Bauer (2003) makes use of the Unified Modelling Language (UML), to model context information relevant to air traffic management in the form of a UML extension. The strength of the approach relies on the use of graphical models for modelling context information, which makes it easy to comprehend and transform the context models. Another major benefit is the use of the very well known and widely accepted UML modelling language. Despite these benefits, Bauer's context model does not address explicitly context validity and context privacy.

In a similar approach Simons and Wirtz (2007) defined a UML based Context Modelling Profile (CMP) that allows to model context information for mobile distributed systems. UML stereotypes have been defined for the context modelling domain and Object Constraint Language (OCL) (OMG OCL, 2005) constraints are enforced to ensure the correctness of models. In this work context quality, validity and privacy are addressed via the definition of temporal constraints and the classification of diverse context sources. Consequently, these non-functional aspects are defined accordingly in the proposed CMP. In overall the approach benefits from the use of the widely accepted UML, since the CMP can be used in various UML tools. Despite that fact, these tools do not provide a standard way to access model stereotypes and enforce constraints (Simons & Wirtz, 2007). Hence, constraints are imposed and enforced in this approach using the Eclipse Modelling Framework (EMF) (EMF, 2008). Moreover, mapping context models to an implementation, which enforces non-functional requirements during the service execution, is considered as future work.

Henricksen and Indulska (2004; 2006) propose an infrastructure and a framework to gather, manage and disseminate context to services. Context modelling concepts are introduced that facilitate the generation of a context management system from models. These are namely the context modelling language, the situation abstraction, the preference, branching and programming models. The approach is mainly based on the taxonomy of context sources used in

this work. The taxonomy presents a clear-cut classification that segregates context sources as static, profiled, sensed or derived. In particular it is denoted that information obtained from a static source is usually of superior quality than information acquired from a dynamic source (profiled, sensed, derived). Moreover it is stated that dynamic context sources usually provide information that is outdated in comparison to static sources, which are commonly more reliable. Context privacy is also addressed in the approach. Furthermore, formality of models is considered and validation capabilities are provided. The approach is slightly hindered by the absence of a context modelling editor and the degree of automation provided for software generation.

The absence of a clear cut approach that follows the MDA paradigm as defined by the OMG increases the service creation overheads in terms of cost and effort required to develop new services (Azmoodeh et. al., 2005). In this chapter we propose a model-driven approach that is strictly based on the MDA paradigm and provides a higher level of automation in software generation. Via the use of the generic MDA-based service creation environment we facilitate the semi-automatic generation of service creation environments for different application domains. Using this approach the CMF is semi-automatically generated in the form of an Eclipse plug-in. The plug-in is then integrated into the generic service creation environment, comprising a new software capability of the resulting PSCE. Consequently, via the use of the PSCE the definition and validation of context models can be effectively carried out. In addition the capability to transform the context models to different implementations is also provided. This simplifies the process and enables the rapid creation of pervasive services.

MODEL DRIVEN METHODOLOGY FOR PERVASIVE SERVICE CREATION

The model-driven development process presented in this section aims to provide a systematic methodology, which facilitates primarily the generation of service creation environments (Achilleos et. al.; 2008) and supports in overall the service creation process. Table 1 shown below illustrates the mapping between the MDD process phases and the service creation phases. The mapping demonstrates how each phase of the MDD process can be effectively used to perform the task associated with the corresponding phase of the service creation process. In this chapter we make explicit use of the methodology to facilitate the pervasive service creation process.

Table 1. Mapping the MDD process to service creation.

MDD process	***Service creation process***
Domain specific language definition	Service analysis
Domain model definition	Service design
Domain model validation	Service validation/testing
Domain model-to-model transformation	Service implementation/ management
Domain model-to-code generation	Service implementation

Domain Specific Language Definition

The domain specific language definition phase reflects fully the service analysis phase. In the service analysis phase a requirements analysis is performed to identify the semantics of the domain and provide an overall understanding of the service to be implemented. Correspondingly, as illustrated in Figure 1, during the domain specific language definition phase the domain semantics are identified and mapped to language constructs (step 1). These language constructs are defined in the form of a metamodel that comprises the abstract syntax of the domain specific

language (DSL). Furthermore, domain rules that govern the language need to be identified and mapped to corresponding OCL constraints. The constraints are then imposed onto the language definition (metamodel) to restrict the language and allow the definition of valid model instances (step 2). The constraints are most commonly applied separately of the domain metamodel definition, but it is the case that some constraints are implicitly stated in the definition.

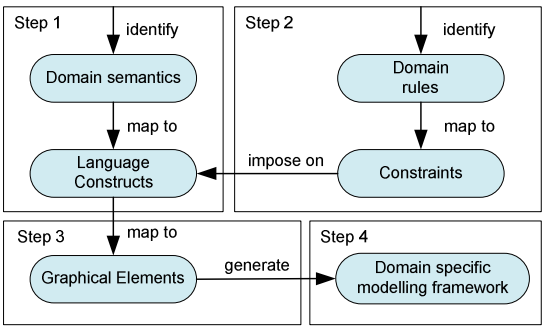

Figure 1: Domain specific language definition; service analysis.

Correct definition of the modelling language and successful imposition of constraints allows defining complete and unambiguous models. Since subsequent phases of the development process rely heavily on these models, it is crucial that the language allows the definition of coherent models. This saves considerably on time and minimises costs, since it does not require performing heavy testing and applying extensive corrections onto the generated implementation. Domain models definition requires a graphical modelling framework, which should consist part of the overall service creation environment. Implementing a modelling framework from scratch is often cumbersome and costly. Especially its maintenance introduces quite an overhead on the development process and subsequently increases costs. Therefore the capability to generate service creation environments for different application domains must be essentially provided.

The generic service creation environment provides the capability to map the metamodel definition to graphical elements comprising the language's concrete syntax (step 3). This is performed by automatic interpretation of the metamodel elements and relationships to visual constructs (graphical metamodel) of the domain specific modelling framework. Moreover, the domain metamodel artefacts are translated in an automatic manner to components (tooling metamodel) that form the palette of the modelling framework. Subsequently, the domain, graphical and tooling metamodels are merged into a common mapping metamodel that facilitates generation of a domain specific modelling framework (step 4).

Besides the definition of a new modelling language the potential to utilise and customise accordingly existing modelling languages is addressed in the methodology. Given that the existing modelling language can be represented using the standard based XML Metadata Interchange (XMI) format then the capability to import it into the generic service creation environment is effectively provided. The XMI standard comprises of an XML like syntax that is utilised for exchanging models in the form of metadata information. Consequently, each modelling language represented using an XMI format can be imported into the environment. This provides the capability to perform the subsequent steps (step 2, 3 and 4) of the language definition and generate a domain specific modelling framework for the language. Furthermore, the framework is integrated to the generic service creation environment to support the following phases of the MDD process.

Domain Model Definition and Validation

The second phase of the MDD process facilitates the definition of domain models via the use of the generated domain specific modelling framework; as illustrated in Figure 2. The framework comprises of a modelling editor with drag and drop capabilities used for the design of domain models on the basis of the modelling language (step 5). This phase corresponds to the service design phase since models can resemble services; if the DSL(s) targets the application domain of services.

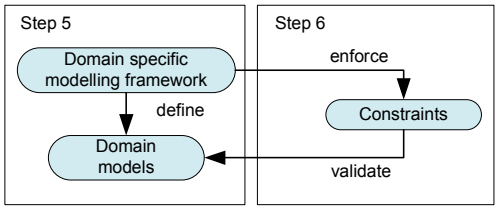

Figure 2: Domain model definition and validation; service design and validation.

The third phase involves domain models validation against the rules imposed during the language definition phase. Figure 2 illustrates that via the use of the generated modelling framework the defined OCL constraints are enforced to validate the domain models (step 6). This is performed in order to ensure that only non erroneous implementations can be automatically generated from these models. Models validation reflects the service validation/testing phase if again we consider that the domain models defined resemble services.

Model-to-Model Transformation

The model-driven development process entails a model-to-model transformation phase (step 7), which assists either the service implementation or the service lifecycle management. As illustrated in Figure 3, via the use of the transformation language the mapping of the source language to the target language is defined. The transformation takes as input a model conforming to a metamodel and produces an output model conforming to another metamodel. In the case of the service implementation phase the transformation accepts as input a platform independent model (PIM) and generates an output platform specific model (PSM). The PSM includes implementation specific details and conforms to a metamodel, which reflects the operational semantics of a programming language. Besides the service implementation phase, transformations can be used for the configuration management of services when porting from one service version to another version; PIM to PIM transformation.

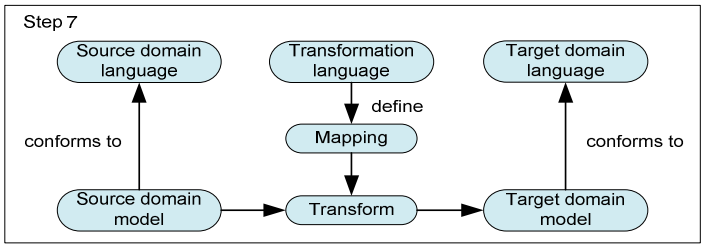

Figure 3: Model-to-model transformation; service implementation/management.

Model-to-Code Generation

The final phase is model-to-code generation (step 8), which corresponds to the service implementation phase. In case the intermediary PIM to PSM transformation phase is omitted, implementation specific details are hard-coded within the code generator. The implementation is

obtained via a semi-automatic process that transforms the models (PIM or PSM) to code. Figure 4 illustrates the definition of the code generator in the form of templates, via the use of a template language. The code generator accepts as an input the domain model defined in accordance to the language and generates the corresponding software application code.

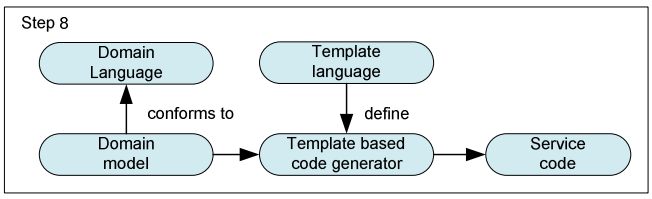

Figure 4: Model-to-code generation; service implementation.

GENERIC SERVICE CREATION ENVIRONMENT ARCHITECTURE

In order to apply the methodology in practice a generic MDA-based service creation environment is required. The proposed service creation environment is composed by existing model-driven frameworks integrated together on top of the Eclipse platform. The platform provides an extensible component-based architecture for the environment that supports the integration of additional modelling frameworks in the form of Eclipse plug-ins. The core components integrated into a common generic framework (Achilleos et. al.; 2008) are namely the *Eclipse Modelling Framework (EMF)* (EMF, 2008), the *Graphical Modelling Framework (GMF)* (GMF, 2008), the *Atlas Transformation Language (ATL)* (ATL, 2008) and *openArchitectureWare (oAW)* (oAW, 2008). Figure 5 presents the Eclipse platform as the foundation and the container of the combined modelling frameworks.

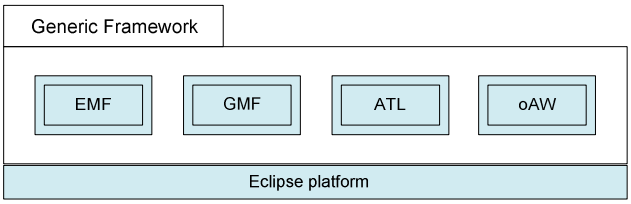

Figure 5: Generic service creation environment architecture.

The Eclipse Modelling Framework is the core software component of the generic environment that interlinks and enables the functionality of the additional frameworks. This means in particular that these software components rely on EMF, to perform their individual operations. For instance transformations written in the ATL language are defined on the basis of the EMF-based domain metamodel. Similarly, template-based code generators are defined in accordance to the EMF-based domain metamodel. Therefore, EMF acts as the bridge and ensures horizontal integration of the environment's individual frameworks.

In this work we utilise the capabilities of the generic framework to define the Context Modelling Language (CML) and generate in a semi-automatic manner it's supporting Context Modelling Framework. The CMF is the new software component generated as an Eclipse plug-in, which is integrated with the main frameworks to deliver the Pervasive Service Creation Environment (PSCE). It provides its own context modelling capabilities for the definition and validation of non-functional requirements for pervasive services. Moreover, it makes use of the software capabilities of the existing components to execute the rest of the phases of the MDD process.

CONTEXT MODELLING

Non-Functional Properties of Context Information

Context-awareness is the main characteristic feature of pervasive applications that denotes the requirement to adapt to dynamic changes in context information. This dynamic nature arises from the diversity of context sources from which context information can be obtained. Although early research on context-awareness focused mainly on sensed context sources current work identified and defined a taxonomy of context sources (Henricksen & Indulska, 2006). According to the taxonomy the nature of the context source determines the persistence of context information. In particular the class of the context source depicts the soundness of the acquired information and can be considered as a measure of context *quality* (Simons & Wirtz, 2007). For instance sensed context sources are commonly recognised as error prone due to sensing errors that can occur and subsequently these sources can provide imperfect or improper information to the user. On the other hand information stored within a repository is generally considered as reliable. Consequently the overall *quality* of the service is affected by the quality of context information obtained from the context source and supplied to the user.

Another non-functional property that also arises from the dynamic nature of context information is context *validity* (Simons & Wirtz, 2007). Context might refer to information that is constantly changing through time, such as the current location of the user of the service. Other pieces of information are rather static and changes are very infrequent; for example a person's date of birth never changes. Therefore, it can be realised that the validity of the information that refers to the current location of the user is generally lower than the validity of static information. In particular context validity denotes the reliability and the accuracy of context information obtained from diverse context sources.

Privacy of context information is an additional issue that must be addressed in order to gain the acceptance of pervasive services by users (Simons & Wirtz, 2007). The protection of the users' privacy indicates another important non-functional property that must be addressed in the context model. Once again the context source provides the target where access restrictions to context information must be applied. Consequently each context source linked to particular information must be assigned a permissions property in order to control and restrict the access to this information. For instance a user of a calendar service might require keeping his work schedule open to all users but he would like to keep his personal schedule restricted to family members.

On the basis of the context sources diversity, the aforementioned non-functional requirements are identified and addressed in this work. Context model associations (sources) are enriched with properties that provide the capability to incarcerate the non-functional requirements within the service implementation. These properties incorporated in the context model support merely the decision making for selecting different context sources. This is performed in order to provide to the user the best information available for a particular pervasive service.

Requirement Analysis

The pervasive service creation process relies heavily on context-awareness, which is considered the principal feature of pervasive services. Context-awareness is a dynamic feature that depicts the capability of devices to detect changes in context information and react accordingly to adapt the service execution. Different categorisations of context information have been defined during the course of research. We acknowledge and build upon the categorisation defined by Dey and Abowd (2000) that enumerate context information as *Identity, Time, Location and Activity*. In

addition to these categories of context information we introduce also the *Preference* category. This particular context information is of significant importance since it depicts user preferences that facilitate the personalisation, customisation and adaptation of the service behaviour.

Moreover, by denoting user preferences as explicit context information we can monitor and provide a rating to individual preferences in order to enable the service to undertake the correct actions on behalf of the user. For instance, by monitoring how a user contacts a particular user (e.g. telephone, email, SMS), the service can realise at a later stage and recommend the appropriate communication channel between the two users. Furthermore, by recording and providing a rating to user preferences the pervasive service can undertake appropriate actions with increased probability of correctness (Henricksen & Indulska, 2006).

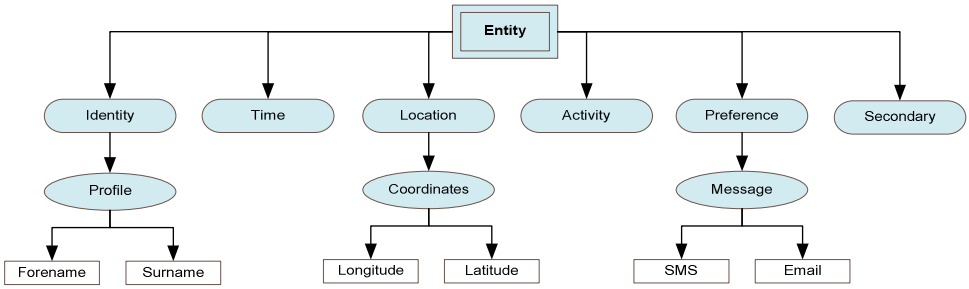

Figure 6: Context categories hierarchy.

Apart from the primary context categories, explicit information that is specific to a context-aware service is introduced in the context model. This context information is termed in this work as secondary categories. Secondary categories describe information specific to a service such as the details of a restaurant; e.g. name, address, and telephone. Both primary and secondary categories are composed by simple properties that are mainly described as primitive datatypes. For instance, if a person's *Identity* is known we can realise primitive context information about that individual such as his *name, email and gender*.

In accordance to the aforementioned categorisation a hierarchy for modelling context categories associated to a particular entity has been defined. Figure 6 illustrates this hierarchy presenting at the top level the *Entity* element that represents any type of real world object (e.g. Person, Device), which can be associated with particular context information. For example a *Person* has an *Identity*, which is successively specialised by the *Profile* complex datatype. The complex datatype is also refined into primitive datatypes such as *Forename and Surname*. A similar context hierarchy is also depicted in the case of the *Location* category. Furthermore, categories can be specialised via an enumeration rather than a complex datatype. For instance, in this example the *Preference* category is refined by the *Message* enumeration, which contains two enumeration literals namely *SMS and Email*.

Context information for pervasive services is acquired typically from different input sources (e.g. repositories, sensors), in contrast to conventional services where information is obtained mainly as input from the user. Consequently, apart from context categorisation a classification of context sources is essential, to extract and define the non-functional properties of pervasive services in the context model. The classification defined by Henricksen and Indulska (2006) and presented next provides a clear-cut taxonomy for context sources:

- Static: information of high persistence (e.g. date of birth).
- Profiled: user-supplied information.
- Sensed: information captured from sensors.
- Derived: derived on the basis of other context information.

The primary non-functional requirement addressed by this classification is *context validity*, which can be determined by the persistence of context information. Persistence defines the frequency with which context information is subject to change, which is essentially different for diverse context sources. Conventionally, static context sources disclose a permanent correlation between the entity and its associated context information. Profiled sources reveal infrequent (seldom) context changes since information remain fixed over long periods of time; unless altered by the user. Conversely, sensed and derived context sources denote information that change frequently and are extremely unstable (frequent or volatile). This is because context obtained from sensors or derived from other information is highly unpredictable. Furthermore, context validity can be defined in the context model using temporal constraints, which are either set as comparative or absolute time constraints. Comparative constraints define a valid expiration time for context information acquired from a context source. On the other hand, absolute temporal constraints designate both the starting and expiring time for context information. Temporal constraints are of prime importance since they designate when information becomes outdated.

The second non-functional requirement that is determined via the sources classification is *context quality*. Typically static information contained within a repository is of superior quality than profiled information inputted by the user. This is due to the fact that a user might neglect to input or update the necessary context information. Respectively, sensed information is of inferior quality than profiled information since context information obtained from sensors is generally inaccurate (or erroneous) and thus unreliable. Moreover, context information derived on the basis of other information is considered of the lowest quality. This is because derived information quality is based on the quality of other information and on the defined derivation rule. For example, potential restaurants for dining can be derived from the food preference of the person.

Context privacy is another important non-functional property that must be depicted in the context model, in order to achieve the acceptance of pervasive services by users. Different permissions must be set to restrict access to different context information and preserve the privacy of individual users. For example, a user should be able to limit access to personal context information such as his credit card details but on the other hand keep his profile information open to all users of the service. Consequently, different access restrictions must be imposed on each context source to restrict the access to context information in accordance to the user's requirements.

In addition to context information and non-functional properties, contextual situations are crucial for the development of pervasive services. These situations depict explicit actions that should be executed when a context event occurs. For instance the change of context information related to a person (*e.g. location*) results in the alteration of the person's situation (*e.g. in office or at a meeting*). If the person is currently at a meeting its mobile phone device must be set automatically to silent mode in accordance to the occurring contextual situation. Therefore, it can be realised that contextual situations are imperative for modelling explicit behaviours, which are valuable to the interaction between the user and the service.

The definition of context categories, non-functional properties and contextual situations facilitate the mapping of the context model and the generation of the corresponding pervasive service implementation. Primarily categorisation supports the generation of information classes in accordance to the requirements of each distinct category. Moreover, context classification aids the generation of different context management classes for dealing with non-functional properties as required in a distributed environment. Also the definition of contextual situations

supports the generation of distinct situation classes that handle explicit service behaviours. Via the generated classes the developer can query, obtain and distribute context information to the pervasive service to achieve its adaptation. Hence, the generated implementation aids and simplifies the pervasive service creation process.

PERVASIVE SERVICE CREATION ENVIRONMENT

Context Modelling Language Definition

The requirements analysis facilitates the identification of the necessary properties of the context modelling domain for accomplishing the domain specific language definition phase. Figure 7 presents the definition of the CML and the generation of the CMF in accordance to steps 1-4 (Figure 1) of the proposed model-driven development process. The creation of the CMF is delivered via the use of the software components of the generic service creation environment. The CMF component is then integrated to the service creation environment to compose the Pervasive Service Creation Environment.

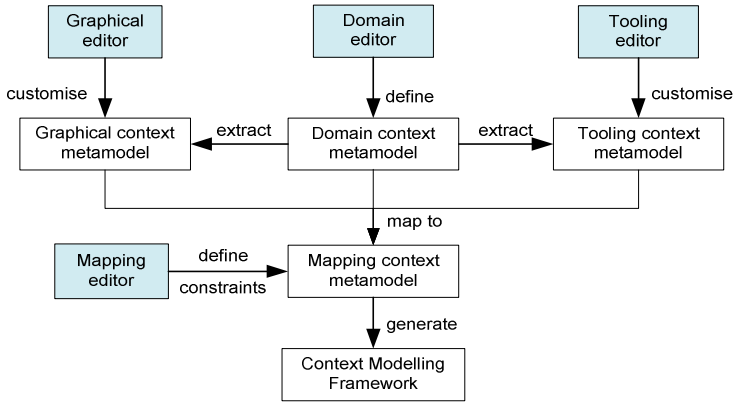

Figure 7: Creating the Context Modelling Framework.

The model-driven development process defines that the semantics identified during the requirement analysis must be mapped to language constructs (step 1 – Figure1). In practice this denotes the definition of the CML in the form of a context metamodel that captures the abstract syntax of the modelling language. The context metamodel is defined on the basis of the Meta Object Facility (MOF) (OMG MOF, 2005) formal specification. Given that the MOF specification does not contribute any software tools to support the metamodel and the concrete syntax definition, the Ecore meta-modelling language of the EMF component is used instead. The motive for selecting EMF as the core of the generic environment is its one-to-one mapping with MOF (Gerber & Raymond, 2003; Mohamed et. al. 2007). Furthermore, both EMF and GMF influenced heavily the MOF 2.0 specification towards the critical direction of software tools integration to achieve the overall objective of model-driven development. The context metamodel definition can be performed either using the EMF or the GMF based editor. The GMF domain editor is preferred, as shown in Figure 7, since it facilitates the context metamodel definition using a comprehensible graphical representation.

The metamodel describes the elements (*e.g. Entity*), properties (*e.g. multiplicity*) and relationships (*e.g. ECAsource*) of the context modelling domain as identified during the requirement analysis. Figure 8 illustrates the *DocumentRoot* metaclass as the container of the elements of the context model. Its aggregation associations (*e.g. contexts*) define the containment relationships with the rest of the elements.

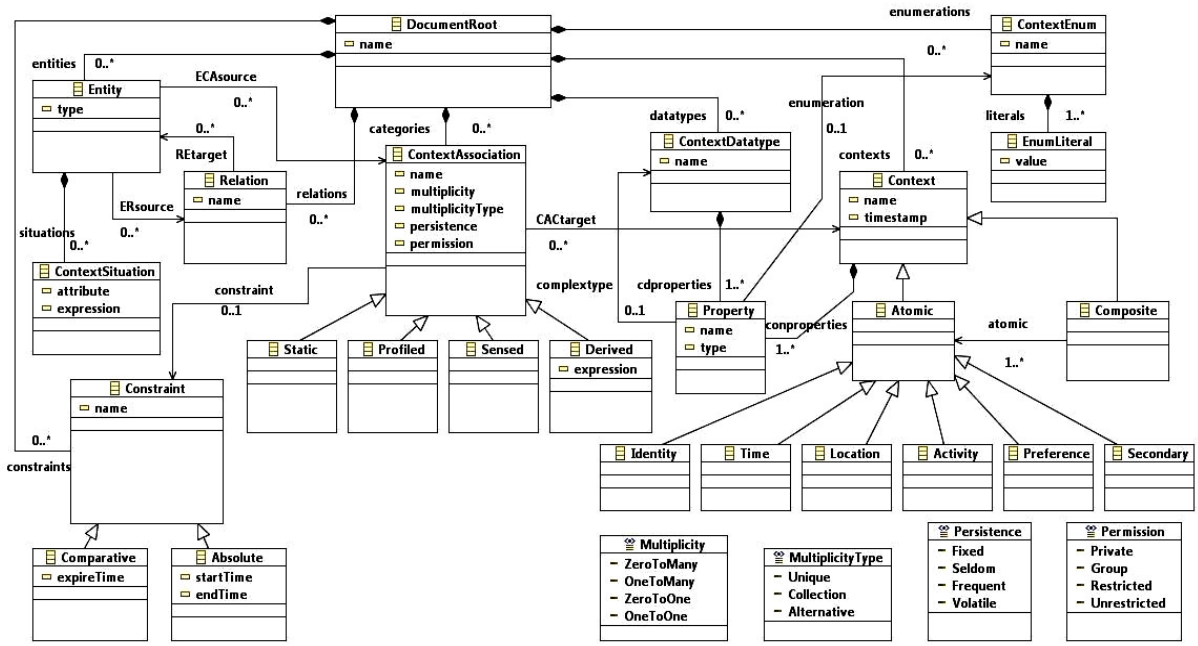

Figure 8: Context modelling language.

The *Entity* metaclass represents objects that can be associated via the *ContextAssociation* metaclass to a variety of context information. The object type (*e.g. Person, Device*) can be defined via the *type* property designated in the metaclass. Entities can contain one or more situations defined by the *ContextSituation* metaclass. Contextual situations are defined via the *attribute* and *expression* properties of the *ContextSituation* metaclass. The *attribute* property describes a Boolean variable. In accordance to the expression defined as an OCL constraint the variable value is evaluated either to true or false, which denotes the occurrence or not of the contextual situation. Finally an entity can be associated to other entities via the *Relation* metaclass and the *ERsource* and *Retarget* associations. Conceptually the *Relation* metaclass denotes a relationship between two entities and it is interpreted as a specific behaviour bound to the two entities; e.g. *Person → owns → Device.*

ContextAssociation is defined as an abstract parent metaclass from which the *Static, Profiled, Sensed and Derived* metaclasses inherit their properties. These metaclasses depict the classification between context sources and facilitate the definition of non-functional requirements in the context metamodel. The primary property depicts the name of the context source. Secondly the *multiplicity* property designates a collection of information and can be assigned the values defined by the *Multiplicity* enumeration. Moreover, the *multiplicityType* property determines the number of simultaneous valid occurrences of context information and obtains its values from the *MultiplicityType* enumeration. The *persistence* property is bound to the *Persistence* enumeration, which describes the frequency with which context is subject to change. Finally the *permission* property discloses the access restrictions imposed upon context information. The values defined via the *Permission* enumeration show the access restrictions that can be imposed on context information in order to safeguard the privacy of the user. In addition to the properties of the parent metaclass, the *Derived* metaclass includes an *expression* property that is used to define the dependence of context information on other context information via an OCL constraint.

Additionally each context source is associated via the *constraint* relationship to the abstract *Constraint* metaclass, which defines a temporal constraint for the specific context source. This constraint ensures the validity of context information by establishing a valid expiration time for the context source or by designating an absolute time interval indicating both the starting and expiration time. The time constraints imposed onto the context source provide the capability to denote when this information was acquired and for how long information is valid.

Context information is defined as an instance of the *Context* abstract metaclass and can be either *Atomic* or *Composite*, in accordance to the inheritance relationship. The *Atomic* context contains simple properties, which represent the lowest level of context information. The *Composite* context contains both simple properties and atomic context. Moreover, the *Atomic* context is also defined as an abstract metaclass since it is extended by the context categories. Properties can be defined as primitive datatypes or even complex datatypes via the *complextype* association. Additionally a property can be associated to an enumeration (instead of a context datatype) as depicted by the *enumeration* relationship.

The context metamodel definition provides the abstract syntax of the modelling language but does not restrict the designer from defining invalid context models; metamodel instances. In order to guarantee the correctness of the context models certain domain rules need to be identified and mapped to OCL constraints**.** These constraints are subsequently imposed onto the context metamodel definition using the GMF mapping editor and the OCL capabilities provided by the generic service creation environment. This is performed in accordance to step 2 of the model-driven development process; Figure1. Following we present some example OCL constraints imposed onto the metamodel to signify their importance for the context modelling language.

Domain Element Target: Entity::EClass

- *Entity.allInstances()->forAll(e1, e2 | e1 <> e2 implies e1.type <> e2.type)*
- *Entity.allInstances()->forAll(e1, e2 | e1<>e2 implies e1.ERsource <> e2.ERsource)*

Domain Element Target: Context::EClass

- *self.conproperties->forAll(p: Property | p.type = 'char' or p.type = 'String' or p.type = 'boolean' or p.type = 'Integer' or p.type = 'double' or p.type = 'float' or p.type = 'long' or p.type = 'short' or p.type = p.complextype.name or p.type = p.enumeration.name)*
- *self.conproperties->forAll(p1: Property, p2: Property | p1 <> p2 implies p1.name <> p2.name)*

The two OCL constraints defined for the *Entity* metaclass restrict the definition of entities so that duplicate entity instances cannot be defined and cyclic relationships between entities are not permitted. The first constraint defines precisely that for all instances of the *Entity* metaclass the *type* property cannot be identical. Moreover the second constraint defines that for all instances of the *Entity* metaclass, no two instances of the *ERSource* association can be identical. The second group of OCL expressions targets the *Context* metaclass and ensures that the definition of context properties is restricted to valid *Property* instances. The first constraint restricts the context model definition and ensures that the *type* property of the *Property* metaclass is set to one of the following: (i) primitive datatype, (ii) complex datatype, (iii) enumeration. Moreover, the second rule complements the first since it prevents the definition of the same *name* property for distinct context properties.

The definition of the context modelling language and the imposition of OCL constraints provides a coherent abstract syntax for the definition of context models. Following step 3 of the

model-driven development process (Figure 1) the metamodel abstract syntax is mapped to the corresponding concrete syntax. Figure 7 illustrates the automatic interpretation of the metamodel and the extraction of the graphical and tooling context metamodels; concrete syntax. The graphical metamodel defines the graphical components that will be used to define visually the context model within the modelling editor. Likewise, the tooling metamodel defines the palette components that enable the drag-and-drop functionality of the modelling editor. Both metamodels can be customised using the graphical and tooling editors to optimise the appearance of the CMF.

Figure 7 illustrates how the three distinct metamodels are merged automatically into a mapping metamodel that enables the generation of the CMF (step 4 – Figure 1). The framework is generated as an Eclipse plug-in using the capability provided by the existing Java Emitter Templates (JET) generators of the EMF component. These template-based generators define the mapping of the combined metamodel to the Java implementation and drive the generation of the framework code. The CMF comprises of the Context Modelling Language as its core constituent and a context modelling editor with drag and drop capabilities for the definition and validation of context models. Figure 9 illustrates the architecture of the Pervasive Service Creation Environment, which comprises of the generated CMF component integrated with the core components of the generic framework. The software tools of each component that support the pervasive service creation process are also illustrated in the figure.

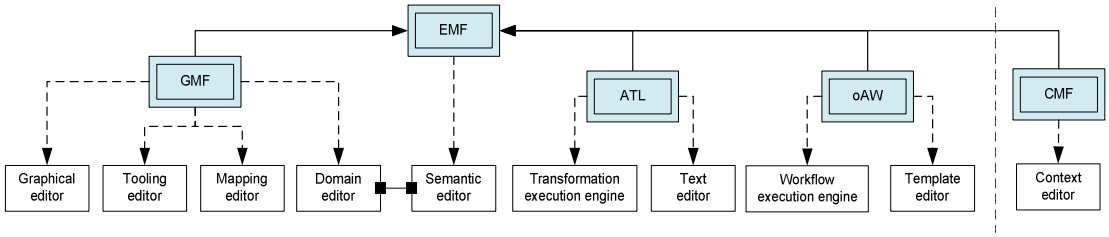

Figure 9: PSCE software components and tools.

Context Model Definition and Validation

The definition and validation of context models is imperative for the correct execution of the remaining steps of the pervasive service creation process. Consequently, the designer must be assisted in the definition of precise and coherent context models. Figure 10 presents the context model as the key input for the model-to-model transformation and model-to-code generation phases. The context (modelling) editor facilitates the model definition (step 5 – Figure 2) and prohibits implicitly the designer from defining an invalid context model. Moreover, the editor supports the validation of the context model (step 6 – Figure 2), according to the OCL constraints imposed during the CML definition phase. The validation exposes any model definition inconsistencies and informs the designer using descriptive information messages. Subsequently, the designer undertakes the necessary steps to rectify the errors before proceeding to the subsequent phases of the process.

Context Model-to-Model Transformation

The coherent definition of the context models enables the model-to-model transformation phase of the process (step 7 – Figure 3). This phase is performed using the ATL component of the PSCE, as shown in Figure 10, which comprises as its core constituent the Atlas Transformation Language. The language provides the capability of writing transformations in the form of a mapping. The ATL editor facilitates the textual definition of the mapping between the semantics

of the CML and the semantics of the target modeling language. Subsequently, the context model and the mapping are accepted as the inputs of the transformation execution engine that drives the translation of the context model, e.g. to a relational model. In many cases the transformation phase is performed in order to aid the model-to-code generation by transforming a platform independent model (e.g. context model) to a platform specific model. The PSM includes implementation specific details that ease the generation of the required implementation code. Most commonly, transformations are used to support the translation (extension) of existing PIMs to improved PIMs versions, rather than aiding the code generation process. This is performed by defining a mapping between the current version of the metamodel and an extended version of the metamodel. The procedure is known as software (model) configuration management and supports the service evolution.

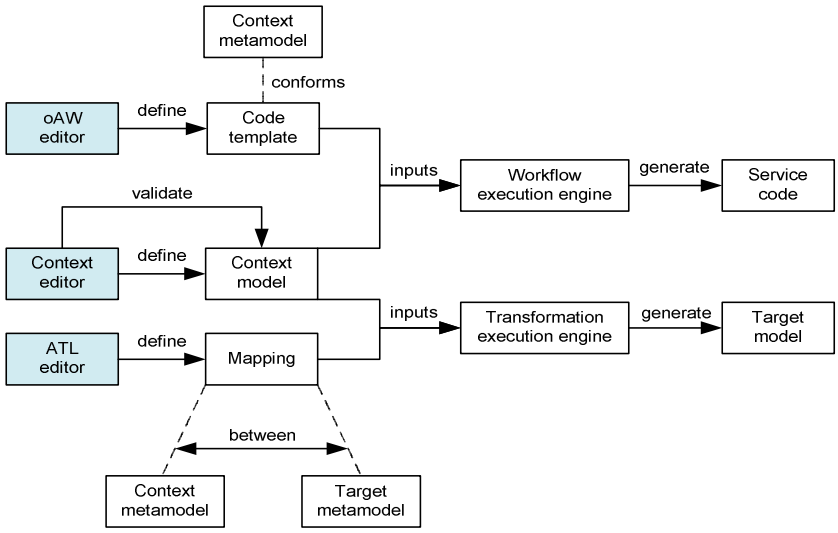

Figure 10: Context model definition, validation, transformation and code generation.

Context Model-to-Code Generation

The model-to-code generation phase facilitates the transformation of PIM or PSM models (e.g. context models) to implementation code (step 8 – Figure 4). In the case that PIM models are interpreted directly to code, implementation specific details are hard-coded within the code generator. The proposed model-driven development process currently supports this approach where the implementation specific details are included in the generator. Therefore the mapping of the context models is defined using a template-based approach to the corresponding programming language. Although the implementation is not generated fully, a considerable portion of the pervasive service implementation is obtained from the context models. This includes information classes, context management classes and contextual situations classes. In this work we assume that low-level architectural components (e.g. widget, interpreters) for acquiring, processing and distributing raw data from sensors either exist or require to be implemented manually. Moreover, graphical user interfaces and complex computations must be also implemented manually.

The oAW component of the PSCE is utilised to accomplish the model-to-code generation phase. Primarily the code generators are defined in the form of templates, using the oAW component's editor. These advanced code generators are defined on the basis of the context metamodel, using the xPand scripting language, to facilitate the transformation of context models to any implementation technology. This means in particular that the artefacts of the CML are

mapped to the operational semantics of the corresponding programming language. Figure 10 illustrates the context model and the code templates, which are accepted as inputs of the workflow execution engine. The engine is responsible to execute the workflow and drive the transformation of models to service implementation code.

PERVASIVE MUSEUM SERVICE

Museum tourist guiding service

In this section we present a case study for the creation of pervasive museum service. We utilise the PSCE to carry out the steps 5-8 (Figures 2-4) of the model-driven development process. The pervasive service aids the visitors touring experience by providing historic and other helpful information on the museum sites and facilities. This information is delivered dynamically to the user of the service due to the occurrence of a context event or because the user has explicitly requested the information. For instance, when a user enters a museum virtual zone a proximity sensor detects his presence. The occurrence of this contextual situation denotes that the user should be presented with information on historic sites available within this virtual zone. The primary aim is to provide to the user context information of the best quality possible (e.g. information is not erroneous) and ensure that this information delivered to the user is also valid (e.g. information is not outdated).

The CMF component of the PSCE is used initially to design the context model illustrated in Figure 11. The model defines both the functional and non-functional properties required for the creation of the pervasive museum service. In the context model the main element is the *Person* entity that signifies the user of the service. Each person is associated to relevant context information via the context source elements, which are defined as instances of the *ContextAssociation* metaclass. For instance each user is associated via the *identity* association to the *Identity* context information, which defines a simple user profile.

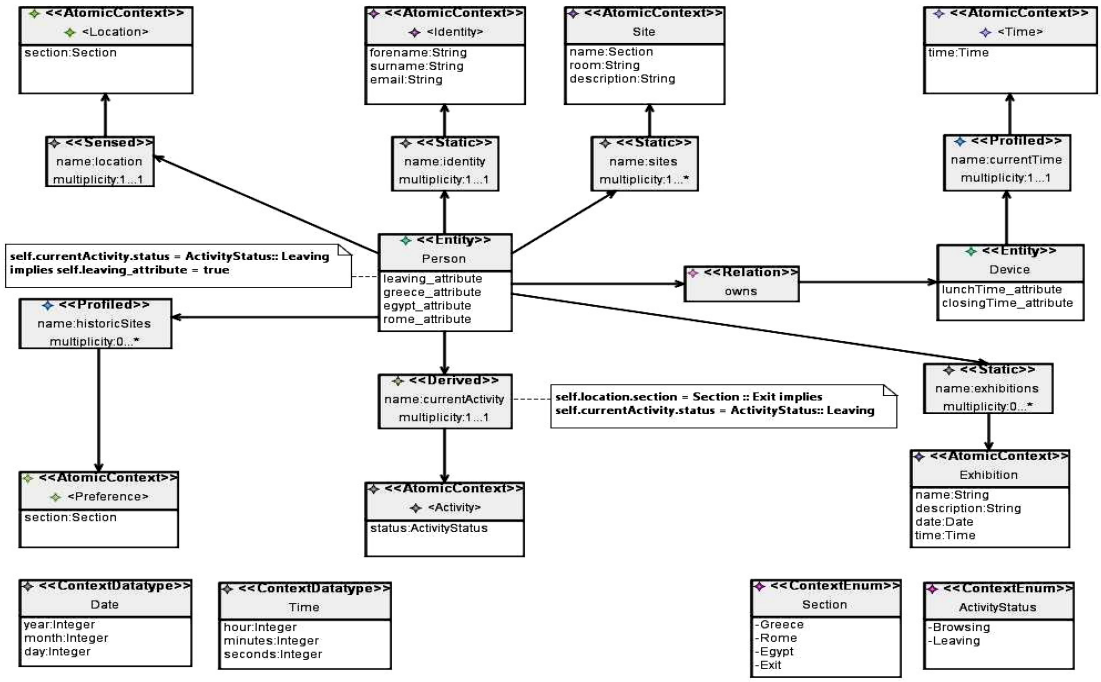

Figure 11: Pervasive museum service context model.

The *identity ContextAssociation* describes the necessary properties of the context source from which profile information is acquired. The type of the context source is defined as *Static*, something that denotes that profile information is stored within a context repository. Moreover, the type of the context source depicts that the acquired context quality is very high. In addition the persistence property, which is defined as *fixed*, ensures that the validity of the context information is also very high. This is due to the fact that information stored within a database remains the same for large periods of time. In addition the multiplicity property designates that only a single profile exists for each user. Furthermore, the multiplicityType property is defined as *unique* and the permission property is set as *private*. These properties determine correspondingly that a distinct profile exists for each user and only the user can access this context information. The profile properties are defined as *String* primitive datatypes and are named accordingly as *forename, surname and email*.

The *sites* and *exhibitions* associations denote two additional static sources defined in the context model. These designate once again static repositories that contain correspondingly information on historic sites of the museum and exhibitions taking place at the museum within the current calendar month. Each historic site is defined as a *Site* secondary context, which comprises of the *name, room* and *description* properties. Both the room and description properties are defined as *String* primitive datatypes. In contrast the name property is defined as an enumeration and can be assigned the literal values defined by the *Section* enumeration (ContextEnum). Furthermore, the *Exhibition* secondary context comprises the *name* and *description String* primitive datatypes and the *date* and *time* complex context datatypes (*ContextDatatype*).

Apart from the static context sources, a sensed source is defined in the context model. The *location* source defines the association to the *Location* context, which describes the current position of the user within the museum. For this case study we have separated the museum premises into four virtual zones, which are defined by the *Section* context enumeration. These virtual zones define the possible locations of the user within the museum, which are obtained and processed via the use of proximity sensors. The type of the source is defined as sensed.

Moreover, the *historicSites* profiled association determines a context source that obtains input information directly from the users. This input context information denotes preferences on historic sites of the museum, which are of particular interest to the user. These preferences are defined via the *section* property of the *Preference* context, which derives its values also on the basis of the *Section* enumeration.

The example model includes also the *currentActivity* context source, which determines the derived *Activity* context information on the basis of the *Location* context. This denotes in particular the current activity of the user in accordance to his current location. It is expressed in the context model in the form of an OCL constraint defined using the *expression* property, which describes the derivation rule for this conceptual fact. The OCL expression[1] shown next illustrates the following derived fact: “If a user is located at the exit of the museum this denotes that he is currently leaving the museum premises”. Consequently, if the user is located at any other virtual zone (e.g. Greece, Rome) apart from the exit zone, it means that he is still browsing the museum historic sites.

self.location.section = Section :: Exit implies
self.currentActivity.status = ActivityStatus :: Leaving

[1] The OCL expressions defined for the case study are in a simplistic form to aid the understanding of the proactive behaviour.

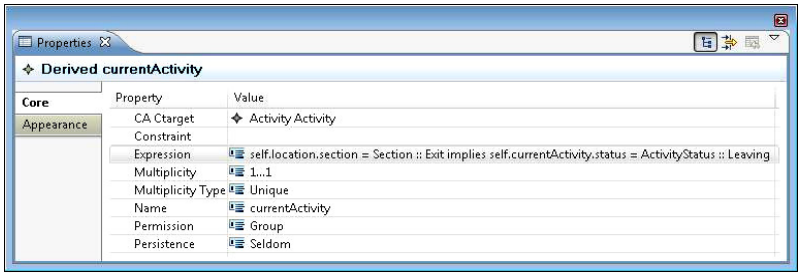

Figure 12: Activity context association properties view.

Figure 12 illustrates the properties view of the *currentActivity* derived source, which comprises of the defined properties that guide the generation of the required functionality (implementation). The properties view defines the target of the context source, which is the *Activity* context information. Accordingly the multiplicity is set to *1...1* and the multiplicityType is set as *Unique* to denote that a person can be engaged in one distinct activity at any given time. Moreover the permission property is set to *Group,* something that defines that only a group of people can access this context information. For instance the security personnel of the museum might require having access and being able to identify the current activity (or location) of the user. The persistence of the source is defined as *Seldom* to depict the infrequent change of the activity context information; user is either *Browsing* or *Leaving*. Finally the expression property defines the textual derivation rule of the conceptual fact, which can be validated for consistency using the Interactive OCL console of the PSCE.

The final context source depicted in the model associates the user's *Device* to the corresponding *Time* context information. This information is profiled by the user on its own device (e.g. mobile, laptop) and can be acquired and managed in accordance to the properties defined for the *currentTime* context source. The set of context associations introduced in the model and their individual properties, realize the main prerequisite to distinguish between diverse classes of context information and manage differently this information.

In addition to entities, sources and context information the model comprises of contextual situations, which are defined for each entity. One of the contextual situations defined in the context model determines the following: "As soon as the user leaves the museum premises details of forthcoming exhibition tours within the current calendar month should be presented to the user". This proactive behaviour is determined by the OCL constraint illustrated next, which depicts a contextual situation defined for the *Person* entity.

self.currentActivity.status = ActivityStatus :: Leaving
implies self.leaving_attribute = true

The expression defines that if the person activity status changes from *Browsing* to *Leaving* this means that the user is currently leaving the museum. Therefore, in accordance to the evaluated logical expression the value of the *Boolean* variable *leaving_attribute* becomes true. Consequently, monitoring the state (true or false) of the attribute using different instances of the contextual situation provides the capability to detect the context event and react accordingly. Likewise, each attribute defined in the context model (*e.g. greece_attribute, closingTime_attribute*) is accompanied by an OCL expression, which describes every contextual situation that must be depicted in the model.

Following the model definition phase, the validation of the museum context model is performed using the CMF capabilities in accordance to the imposed metamodel level constraints. This prevents the developer from attempting to transform or generate implementations out of erroneous context model definitions. In the case that an inconsistency is detected in the context

model a corresponding error message is displayed suggesting possible resolutions using an informative description. Subsequently, the designer can undertake the necessary actions to rectify the problems discovered in the model definition.

Pervasive museum service evaluation

The implementation phase is carried out primarily by means of context model-to-code generation. For this purpose we have defined the mapping between the context metamodel and the operational semantics of two programming languages. The mapping was defined in the form of code templates to facilitate the transformation of context models (e.g. museum context model) to the Java and J2ME implementation technologies.

Appendix A illustrates an extract of the template mapping defined for transforming context associations to J2ME implementation code. From the mapping (lines 6-8), we can observe that for each context source defined in the context model a corresponding class is being generated. Furthermore, the conditional statements defined at lines 15 and 17 drive the generation of the required implementation in accordance to the multiplicity property depicted in the context source. Another important section of the mapping is presented in lines 10-13 where each property of the context source is mapped accordingly to a corresponding class variable. Accordingly, helper functions are defined that allow accessing these variables from other classes to facilitate and simplify the implementation of the required functionality.

Appendix B illustrates the helper functions defined for the association properties, using the extension language of the oAW component. The extension language facilitates the definition of rich libraries of functions, which can be accessed within code templates to aid the code generation process. These helper functions are imported in the code template definition via the Extensions statement defined at line 1 and accessed via the «ecas.SourceHelperFunctions()» statement defined at line 19 of Appendix A.

The implementation generated from the context model eradicates the requirement to manually implement repetitious code such as information and context management classes (Henricksen & Indulska, 2006). This simplifies and enables the rapid creation of context-aware services. Despite that fact, complex computations and graphical user interfaces require to be manually implemented by extending or modifying the generated service code. Table 2 demonstrates an evaluation of the developed pervasive museum service according to selected software metrics. The analysis results are obtained via the use of the CCCC analysis tool (Littlefair, 2001). The selected metrics for the evaluation are: (i) Lines of Code (LOC) and (ii) McCabe's cyclomatic complexity (Bhansali, 2005). The analysis shows the effectiveness of the approach in minimising the cost, time and effort required to implement the pervasive service.

Table 2. Pervasive museum service evaluation.

Implementation	**Metric**	**Generated service code**		**Overall service code**	
		Overall	*Per Module*	*Overall*	*Per Module*
J2ME	Number of modules	40	-	53	-
	Lines of Code	1768	44.200	2330	43.962
	Cyclomatic Number	128	3.200	181	3.415
	Lines of Comment	560	14.000	648	12.226
Java	Number of modules	33	-	53	-
	Lines of Code	1282	38.848	1859	35.075
	Cyclomatic Number	49	1.485	101	1.906
	Lines of Comment	543	16.455	564	10.642

Table 2 illustrates the analysis results for both the J2ME and Java implementation of the museum context-aware service. Primarily, the LOC metric shows the number of non-comment and non-blank lines of code. From the table it is calculated that 75.879% of the J2ME service code and 68.96% of the Java service code is generated from the context model. The percentages calculated in this case study provide the capability to derive the necessary conclusions but do not serve in any case as an explicit baseline for future case studies. These analysis results indicate simply that the effort required for the implementation of the pervasive museum guiding service has been significantly reduced.

Code complexity is another valuable metric, which denotes the degree of code understandability and indicates the code amenability to modification. Moreover it's a dominant indicator of the code testability (Bhansali, 2005). From the table it is clear that the complexity of the generated code (indicated by the cyclomatic number) is lower than that of the overall service code. This indicates that the generated code can be easily modified and be subjected to testing. Furthermore, it is realised that the mapping can be optimized further to decrease the generated service code complexity. Conversely, it is very difficult to achieve optimization and reduce the code complexity when manual implementation is involved.

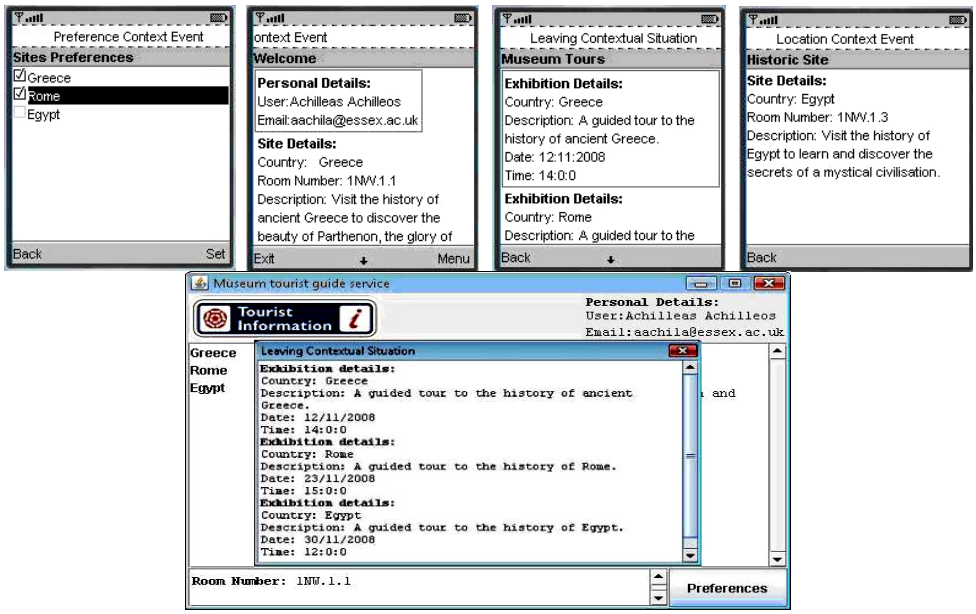

Figure 13: Pervasive museum service running on a J2ME device emulator.

Figure 13 illustrates the pervasive museum service running on a J2ME device emulator and a Java enabled laptop device. The screen capture on the top left of the figure shows the historic sites preference selection list. In accordance to the user's preferences the service retrieves and loads the appropriate information on historic sites, as illustrated onto the second screen capture. Following, onto the next screen capture we can observe the occurrence of the leaving contextual situation, which causes a list of upcoming exhibitions tours to be displayed to the user. Moreover, we have the occurrence of the location contextual situation, which signifies that the user has entered the virtual zone of Egypt. Consequently, the user is presented with information on historic sites located in this virtual zone, as illustrated onto the fourth screen capture. Finally, the screenshot shown at the bottom of the figure presents respectively the occurrence of the leaving contextual situation, while the service is running on the Java enabled laptop device.

CONCLUNSIONS

In this work we propose a model-driven development methodology that supports the service creation process and utilise the process for the creation of pervasive services. The generic service creation environment developed guides the process and provides the capability to define and generate domain specific modelling frameworks. Each generated framework can be effectively integrated into the generic environment to comprise a service creation environment that addresses a particular services domain (e.g. pervasive services, web services). Consequently, via the composed environment the service creation process is accomplished, starting from service modelling to service implementation.

The chapter deploys the methodology for the development of a Context Modelling Framework that addresses the definition of non-functional properties in the form of context models. The CMF comprises a new component of the generic service creation to compose the Pervasive Service Creation Environment, which supports efficiently the pervasive service creation process. In particular the PSCE facilitates the definition of context models that describe mainly the non-functional properties required for the creation of pervasive services. Moreover, via the use of the environment context models can be validated to ensure their correctness and support respectively the model-to-code generation phase. The generation of the pervasive service implementation from the context models provides the capability to enforce the defined non-functional requirements during the service execution stage.

The Pervasive museum service developed in this work provides the capability to evaluate the efficiency of the methodology. According to the results obtained from the analysis performed using the Lines of Code and the Code Complexity software metrics the effort required for the implementation of the pervasive service is significantly reduced. Moreover, the development cost is also drastically decreased since it is proportional to the time consumed to undertake the implementation. The results presented on Table 2 illustrate that for 40 modules of the generated J2ME code the Cyclomatic Complexity number computed is 128. In contrast the increase of the Cyclomatic Complexity number with merely 13 manually implemented J2ME modules is 52. This denotes an increase of 8.12% in code complexity when manual implementation is involved.

Consequently, by optimising the context modelling language and the model-to-code generator the complexity of the generated service implementation can be minimised. This is performed as part of an iterative process that aims to refine and enhance both the modelling language and the generator. As part of future work we aim is to address this issue by carrying out further case studies that will provide the capability to detect and rectify any deficiencies in the modelling language and the code generator.

REFERENCES

Achilleos, A., Georgalas, N., &Yang, K. (2007). An Open Source Domain-Specific Tools Framework to Support Model Driven Development of OSS, In ECMDA-FA, *Lecture Notes in Computer Science, Vol. 4530* (pp. 1 – 16).

Achilleos, A., Yang, K., & Georgalas, N. (2008). A Model-driven Approach to Generate Service Creation Environments, In IEEE Globecom, *Global Telecommunications Conference* (pp. 1 – 6).

Adamopoulos, D. X., Pavlou, G., & Papandreou, C. A. (2002). Advanced Service Creation Using Distributed Object Technology, *IEEE Communications Magazine,* 40 (3), 146 - 154.

Atlas Transformation Language (ATL) (2008). from http://www.eclipse.org/m2m/atl.

Azmoodeh, M., Georgalas, N., & Fisher, S. (2005). Model-driven systems development and integration environment, *British Telecom Technical Journal (BTTJ),* Vol. 23 (03), 96-110.

Bauer, J. (2003). *Identification and Modelling of Contexts for Different Information Scenarios in Air Traffic*, Diplomarbeit, Faculty of Electrical Engineering and Computer Sciences, Technische Universitat Berlin.

Bhansali, P. V. (2005). Complexity measurement of data and control flow, *ACM SIGSOFT Software Engineering Notes,* 30 (1) 1 - 2.

Dey, A. K., & Abowd, G. D. (2000). The Context Toolkit: Aiding the Development of Context-Aware Applications, In ICSE, *Workshop on Software Engineering for Wearable and Pervasive Computing*.

Eclipse Modelling Framework (EMF) (2008). from http://www.eclipse.org/modeling/emf/.

Frankel, D. S. (2003). *Model Driven Architecture: Applying MDA to Enterprise Computing*, Indianapolis: Wiley Publishing Inc.

Gerber, A., & Raymond, K. (2003). MOF to EMF: There and Back Again, *In Proc. of the OOPSLA Workshop on Eclipse Technology eXchange* (pp. 60 – 64).

Glitho, R. H., Khendek, F., & De Marco, A. (2003). Creating value added services in Internet Telephony: An overview and a case study on a high-level service creation environment, *IEEE Transactions on System, Man and Cybernetics- Part C: Applications and Reviews,* 33 (4) 446 - 457.

Graphical Modelling Framework (GMF) (2008). from http://www.eclipse.org/gmf/.

Henricksen, K., Indulska, J., & Rakotonirainy, A. (2002). Modeling Context Information in Pervasive Computing Systems, In Pervasive Computing, *Lecture Notes in Computer Science,* Vol. 2414 (pp. 79 – 117).

Henricksen, K., & Indulska, J. (2004). A Software Engineering Framework for Context-Aware Pervasive Computing, *2nd IEEE International Conference on Pervasive Computing and Communications* (pp. 77 – 86).

Henricksen, K., & Indulska, J. (2006). Developing context-aware pervasive computing applications: Models and approach, *Pervasive and Mobile Computing Journal,* 2 (1), 37 - 64.

Kleppe, A., Warmer, J., & Bast, W. (2005). *MDA Explained: The Model Driven Architecture: Practice and Promise*, Boston: Addison-Wesley.

Lennox, J., & Schulzrinne, H. (2000). Call Processing Language Framework and Requirements, *RFC 2824, Internet Engineering Task Force,* from http://www.ietf.org/rfc/rfc2824.txt.

Littlefair, T. (2001). *An investigation into the use of software code metrics in the industrial software development environment,* Ph.D. Thesis, Faculty of Communications, Health and Science, Edith Cowan University.

McFadden, T., Henricksen, K., & Indulska, J. (2004). Automating context-aware application development, In UbiComp, *1st International Workshop on Advanced Context Modelling, Reasoning and Management* (pp. 90 – 95).

Mohamed, M., Romdhani, M., & Ghedira, K. (2007). EMF-MOF Alignment, *3rd International Conference on Autonomic and Autonomous Systems* (pp. 1 – 6).

Object Management Group (OMG) - Meta Object Facility (MOF) Core Specification v2.0 (2005). from http://www.omg.org/docs/formal/06-01-01.pdf.

Object Management Group (OMG) - Model Driven Architecture (MDA) Specification Guide v1.0.1 (2003). from http://www.omg.org/docs/omg/03-06-01.pdf.

Object Management Group (OMG) - Object Constraint Language (OCL) Specification v2.0 (2005). from http://www.omg.org/docs/formal/06-05-01.pdf.

openArchitectureWare (oAW) User Guide v4.3.1 (2008). from http://www.openarchitectureware.org/pub/documentation/4.3.1/openArchitectureWare-4.3.1-Reference.pdf.

Schilit, B., Adams, N., & Want, R. (1994). Context-Aware Computing Applications, *In Proc. IEEE Workshop on Mobile Computing Systems and Applications* (pp. 85 – 90).

Simons, C., & Wirtz, G. (2007). Modelling Context in Mobile Distributed Systems with the UML, *Journal of Visual Languages and Computing,* 18, 420 - 439.

Strang, T., & Linnhoff-Popien, C. (2004). A Context Modelling Survey, In UbiComp, *1st International Workshop on Advanced Context Modelling, Reasoning and Management* (pp. 34 – 41).

Yang, K., Ou, S., Azmoodeh, M., & Georgalas, N. (2005). Policy-based Model-driven Engineering of Pervasive Services and the Associated OSS, *British Telecom Technical Journal (BTTJ),* 23 (3), 162 - 174.

Appendix A

```
1. «EXTENSION extensions::functions»
2. «DEFINE Root FOR cml::DocumentRoot»
3. «EXPAND Entity FOREACH entities»
4. «ENDDEFINE»
5. «DEFINE Entity FOR cml::Entity»
6. «FOREACH this.ECAsource AS ecas-»
7. «FILE ecas.ext1()+".java"»
8. public class «ecas.ext1()»{
9. private static final String atomic_context = "« this.ext2()+""+ecas.CACtarget.first().ext3()»";
10. private static final String multiplicity = "«ecas.multiplicity»";
11. private static final String multiplicityType = "«ecas.multiplicityType»";
12. private static final String persistence = "«ecas.persistence»";
13. private static final String permission = "«ecas.permission»";
14.
15. «IF ecas.multiplicity == "0...1" || ecas.multiplicity == "1...1"»
16.
17. «ELSEIF ecas.multiplicity == "0...*" || ecas.multiplicity == "1...*"»
18.
19. «ecas.SourceHelperFunctions()»
20.
21. «FOREACH ecas.CACtarget.conproperties AS cp-»
22. public static String «cp.get1()»() {
23. return «cp.name»;
```

```
24. }
25. «ENDFOREACH»
26. }
27. «ENDFILE»
28. «ENDFOREACH»
29. «ENDDEFINE»
```

Appendix B

```
1:  String SourceHelperFunctions(ContextAssociation ca) :
2:  'public static String getMultiplicity() {
3:  return multiplicity;
4:  }
5:  public static String getMultiplicityType() {
6:  return multiplicityType;
7:  }
8:  public static String getPersistence() {
9:  return persistence;
10: }
11: public static String getPermission() {
12: return permission;
13: }';
```

Index Terms:

Model-driven development: A software development methodology that focuses on the design and implementation of software applications at an abstract platform-independent level.

Non-functional requirements: Properties that define how applications must behave and evaluate the operation of these applications. Typical non-functional requirements are: quality, validity, security, privacy, etc.

Pervasive computing: Defines a software engineering paradigm that deals with the development of highly adaptive software applications that can run anywhere, anytime and on any particular device (mobile or stationary) with limited or no user attention.

Context-awareness: Describes the key characteristic of pervasive services that defines the capability to obtain, process and utilise context information in order to adapt software applications.

Pervasive service creation: Describes a service creation process that deals with the analysis, design, validation and implementation of pervasive services.

Metamodelling: The process that guides the definition of a metamodel, which describes the elements, properties and relationships of a particular modelling domain.

Context modelling: A modelling technique that deals with the design of an advanced information model that captures the context-awareness characteristic of pervasive services. From this model the implementation is automatically generated that handles representation, administration and distribution of context information to pervasive services to achieve their adaptation.