Model-Driven Petri Net based Framework for Pervasive Service Creation

Achilleas Achilleos

A thesis submitted for the degree of
Doctor of Philosophy
at the
School of Computer Science and Electronic Engineering
University of Essex
October 2009

Abstract

Mobile devices are becoming increasingly small, providing sophisticated computing and wireless communication capabilities that can be utilised to develop pervasive services. Pervasive services are software applications that have the capability to run anytime, anywhere and on any device with minimal or no user attention. These advancements and diversity in technologies and the dynamic nature of pervasive services increase though the complexity of the creation process. Furthermore, the fast-changing user requirements and the need to create services rapidly steers the computing world to seek innovative and cost-effective software engineering methodologies.

The study of existing work revealed the lack of a conceptual framework that provides a high-degree of automation in software generation for the benefit of service creation. This thesis proposes and defines such a framework with particular focus on pervasive service creation. The framework comprises a model-driven development methodology and a generic supporting environment, which facilitate the automatic generation of domain-specific Service Creation Environments (SCEs) and support the phases of the service creation process.

The Model-Driven Petri Net based Framework (MDPNF) is then defined that introduces pervasiveness into the conceptual framework. It defines a well-formulated process and a Pervasive SCE (PSCE). During the service analysis phase the PSCE, which consists of the Context, Presentation and Petri Net modelling frameworks, is defined and generated. The modelling frameworks support respectively the definition and validation of the context-awareness characteristic, the graphical user interfaces and the dynamic behaviour

of pervasive services. Furthermore, the PSCE supports the implementation of pervasive services mainly by means of code generation.

The MDPNF is evaluated via an example pervasive service scenario, by performing a quantitative analysis using selected software metrics and a parametric software cost estimation model. From the evaluation the efficiency and the cost-effectiveness of the framework is demonstrated for the creation of the pervasive service prototype.

Declaration

I declare that this thesis was authored by myself and describes my own research work; unless otherwise stated. This work has not been submitted for any other degree or professional qualification.

Achilleas Achilleos
School of Computer Science and Electronic Engineering
University of Essex
Colchester, CO4 3SQ, United Kingdom

To my loving and late grandfather, Yiannis.

Sorry for not being there for you during the final years of your life.
I will always live and stand by the values you taught me in life.

Acknowledgements

I am writing this section to show my appreciation to the people who stood by me and helped me during the course of this research. First of all I would like to express my gratitude to Dr. Kun Yang, Reader at the School of Computer Science and Electronic Engineering, University of Essex, who has been my supervisor during my research study. He provided me continuous support, important advice and helpful suggestions during the course of this research work. Also, I would like to thank my co-supervisor Dr. Nektarios Georgalas, Principal Researcher at the Centre of Information and Security Systems Research, British Telecom Innovate for his useful suggestions, constructive advice and continuous support throughout my work.

I also wish to express my sincere gratitude to John Wittgreffe for his help and support. Special thanks are due to Richard Wiseman for being a supportive friend and helping me by proofreading my thesis. I am also thankful for the help and advice provided from my board members Dr. John Woods, Dr. Antonio Liotta and Dr. Martin Colley.

I am also lucky and more than grateful to be doing my doctorate study alongside my good friend Andreas Konstantinides, who was also doing his doctorate during the same years. Without his help and encouragement, this study would not have been completed.

My special appreciation and love go to my parents, Androulla and Panayiotis Achilleos, my sister Margarita Achilleos and my small brother Marios Achilleos whose love and support gave me the strength to overcome any difficulties and complete my research. My thanks go also to relatives and friends for never letting me feel away from home.

Finally, I would like to express my special thanks and unbound love to my other half Andria Hadjigeorgiou. She helped me concentrate on completing this doctorate

dissertation, always encouraged me, motivated me and supported me mentally during the course of this work.

Table of Contents

Table of Figures

List of Tables

List of Abbreviations/Acronyms

API – Application Programming Interface

ATL – Atlas Transformation Language

BPEL – Business Process Execution Language

CASE – Computer-Aided Software Engineering

CDF – Cummulative Distribution Function

CGI – Common Gateway Interface

CMF – Context Modelling Framework

CML – Context Modelling Language

COCOMO II – COnstructive COst MOdel II

CPL – Call Processing Language

DSL – Domain Specific Language

DSM – Domain Specific Modelling

DSMF – Domain Specific Modelling Framework

DTD – Data Type Definition

EMF – Eclipse Modelling Framework

EOS – Elementary Object Systems

GMF – Graphical Modelling Framework

GPL – General Purpose Language

GUI – Graphical User Interfaces

HTML – HyperText Markup Language

HTTP – HyperText Transfer Protocol

IDE – Integrated Development Environment

IETF – Internet Engineering Task Force

IMDE – Integrated Model Driven Environment

INS – Intelligent Network Services

J2EE – Java 2 Platform Enterprise Edition

J2ME – Java 2 Platform Micro Edition

JET – Java Emmiter Templates

LoC – Lines of Code

MDA – Model Driven Architecture

MDD – Model Driven Development

MDPNF – Model-Driven Petri Net based Framework

MMF – Midlet Modelling Framework

MOF – Meta-Object Facility

oAW – openArchitectureWare

OCL – Object Constraint Language

OMG – Object Management Group

OO – Object-Oriented

PIM – Platform Independent Model

PMF – Presentation Modelling Framework

PML – Presentation Modelling Language

PN – Petri Net

PNML – Petri Net Markup Language

PNO – Petri Net with Objects

PN-PMF – Petri Net Process Modelling Framework

PN-PML – Petri Net Process Modelling Language

PSCE – Pervasive Service Creation Environment

PSM – Platform Specific Model

QVT – Query/View/Transform

SCE – Service Creation Environment

SCML – Service Creation Markup Language

SIP – Session Initiation Protocol

SLEE – Service Logic Execution Environment

SMF – Survey Modelling Framework

SOA – Service Oriented Architecture

SOAP – Simple Object Access Protocol

UDDI – Universal Description, Discovery and Integration

UML – Unified Modelling Language

W2S – Workflow Web Services

WSDL – Web Service Description Language

XMI – XML Metadata Interchange

XML – Extensible Markup Language

Chapter 1 Introduction

The motivation behind the definition of the Model-Driven Petri Net based Framework (MDPNF) is the increasing requirements of users for running pervasive services on their mobile devices and the necessity to simplify and expedite the creation of these types of services. Foremost, this chapter presents background information on pervasive computing and introduces the research challenges arising from this current trend. These research challenges steer the definition of the model-driven Petri Net based process and the generation of the supporting Pervasive Service Creation Environment that compose the MDPNF. Finally, the MDPNF is conceptually introduced, the major contributions of the thesis are summarised and an outline of the thesis is presented.

1.1 Background and Motivation

Pervasive Computing, also referred sometimes as Ubiquitous Computing, introduces a new paradigm that aims to incorporate technology into people's everyday life [1], [2]. Starting from the mid-1970s, computing paradigms have evolved remarkably following the early developments of personal computers (PCs), from distributed computing, through mobile computing to pervasive computing [2]. Mark Weiser, the father of Ubiquitous Computing [3], described his vision with the following statement: *"The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it"* [4].

This statement was describing the foreseeable future, because at that point the hardware and software technologies were not mature enough to support the pervasive computing vision [5]. Since that early statement computing has moved largely away from traditional PCs to mobile devices, which aid the advancement and fulfilment of the pervasive computing vision. Furthermore, tremendous research developments have been made in terms of mobile hardware technologies, wireless mobile communications and worldwide networking. These technological advancements have provided more sophisticated computing environments where users are able to interact with services, so as to communicate and easily accomplish everyday tasks.

The future success of pervasive computing is based fundamentally on the capability to provide users with services that can run anywhere, anytime and on any particular device (mobile or stationary) with limited or no user attention [6]. In particular, pervasive services refer to software applications that offer users a specialised and personalised behaviour that allows dynamic computing tasks to be performed effectively. This

prescribes a pervasive environment saturated with computing, communication, cognition and collaboration capabilities (the 4four Cs of Pervasive Computing) [1], enabling users to interact seamlessly with pervasive services, turning them into technologies that actually disappear [5].

Despite the aforementioned benefits, the advancements and diversity in technologies currently present make the creation of pervasive services an increasingly complex task. The versatile requirements [7], [8] associated with pervasive services such as adaptability and the capability to run on different devices with minimal or no user attention complicate further the service creation process. Moreover, the necessity to create pervasive services rapidly and cost-effectively steers the computing world to seek new, effective software engineering methodologies. Consequently, an abstract and more sophisticated service creation framework is required to provide a high-degree of automation in software generation and support pervasive service creation at the static compile time.

Service creation is one of the most important phases in the service engineering lifecycle since it establishes the effectiveness and agility with which services are being developed. More specifically, service creation is a complex process that entails a diversity of development aspects, namely the analysis, design, implementation and validation (i.e. testing) of services [9], [10]. The predicament with this approach is that the service is validated at the late stage of testing. Hence, if any amendments are necessary respective modifications should be carried out iteratively in design and implementation, until eventually a valid service is developed. In some cases, in the interest of expediting delivery, modifications are only performed in the implementation, introducing thus a

discrepancy between the design, documentation and implementation. This introduces additional overheads and significantly affects the efficiency of the process.

Furthermore, the service creation process is commonly supported by a service creation environment (SCE) that aims to simplify and expedite the development of services; refer to Chapters 3 and 6 for examples of SCEs. Many technology-specific SCEs have been developed during the course of research, which aim to satisfy the aforementioned objective. The focus on a platform-specific implementation and the consequent technology-specific complexities imposed by these SCEs, increase further the complexity of the process and do not provide a clear-cut solution. Hence, these technology-specific SCEs aid expert users but certainly do not support novice users in the development of advanced services [9].

The development of pervasive services augments further the complexity of the service creation process. In particular, the diversity of sources from which input information is acquired, the seamless interaction of the user with the service and the dynamic behaviour of the service compose the main requirements that complicate the process. Basically, the service capability to implicitly or explicitly acquire input information heavily influences the interaction of the user with the service and enforces the adaptation of service behaviour. This is commonly described as context-awareness, which is a defining characteristic of pervasive services[1].

Several approaches to context-aware service creation have been proposed that function either at the infrastructure or the application level. The former research initiative deals with the development of an underlying infrastructure that supports low-level tasks such as

[1] The terms pervasive and context-aware are used interchangeably in this thesis since context-awareness is considered as the prime feature of pervasive services.

gathering and processing raw data (e.g. sensors, repositories) to obtain a high-level description and distribute this information to services that require it. In this thesis the application-level research efforts, also referred to as context modelling techniques, are considered. These assume that this low-level functionality exists and deal principally with the representation of context information in a format (i.e. the context model) that can be realised and utilised by pervasive services. In particular, their focus is on context management tasks such as the representation, administration and distribution of context information to pervasive services to achieve their adaptation.

Consequently, existing context modelling techniques address the key characteristic of pervasive services, which is context-awareness. However, they do not consider additional important requirements of pervasive services such as the dynamic behaviour of the service and the graphical user interfaces (GUIs) through which the user interacts with the service. These supplementary requirements are closely related to context-awareness since they are directly affected by alterations in context information. Therefore, it is imperative to provide complementary software modelling capabilities that facilitate the definition of the entire set of requirements that compose the pervasive service.

This thesis proposes the use of the Model Driven Architecture (MDA) [11], [12], [13], [14] paradigm of the Object Management Group (OMG) to support pervasive service creation. MDA's many advantageous features, such as high-level abstraction and platform independence not only support but also simplify the definition of a conceptual framework that facilitates pervasive service creation. More specifically, the MDA-based technique provides the following benefits: (i) generating domain-specific modelling frameworks, (ii) producing rigorous design specifications, (iii) validating static structure

and syntax of models using Object Constraint Language (OCL) [15], [16] constraints and (iv) generating diverse implementations from the pervasive service designs. The absence though of MDA-based software implementations has motivated the selection of Eclipse model-driven software tools, which conform to the MDA standards, as the enabling components that steer the definition of the conceptual framework.

The proposed conceptual framework defines actually a Model Driven Development (MDD) methodology and a supporting generic environment, which facilitate the automatic generation of domain-specific SCEs and support service creation. In particular, the framework overcomes technology-specific complexities and introduces service validation straight after service design so as to preserve the consistency between the design, documentation and implementation. Furthermore, the framework automates tasks such as the development of SCEs and expedites as a result the creation of domain-specific services (e.g. pervasive services, Web Services). Consequently, with the aid of the conceptual framework, this thesis embarks upon the formulation of the MDPNF, which defines a model-driven Petri Net based process and a generated PSCE to support pervasive service creation.

Foremost, the PSCE is developed by integrating the generated Context, Presentation and Petri Net modelling frameworks into the generic environment. Consequently, using the capabilities of the PSCE the design, validation and implementation phases of the process are performed. In particular, the modelling frameworks of the PSCE support the design of context, presentation and Petri Net models, which define the pervasive service specification. Moreover, the modelling frameworks support the validation of the static structure and syntax of the models through the use of OCL constraints. Finally, the PSCE

supports the implementation of pervasive services mainly by means of model-to-code generation and manual implementation of complex computational tasks.

Apart from the MDA paradigm, the Petri Net formalism is also fundamental in the MDPNF since it allows complementing the static OCL validation capabilities. The Petri Net theory [17] comprises of a formal technique suitable for modelling the dynamic pervasive service behaviour, due to its simple and graphical representation and its concurrent and asynchronous nature. Most importantly, though, the definition of the service behaviour using a Petri Net model allows the qualitative analysis and validation of the operational semantics of the service by means of model execution[2]. This ensures consistency of the models that compose the pervasive service and guarantees the generation of non-erroneous implementations. Consequently, the integration of the MDA paradigm and the Petri Net formalism under the unified MDPNF fully satisfies the requirements of pervasive services, so as to simplify and expedite their creation.

1.2 Research Challenges

Prior to the introduction of the MDPNF, the research challenges addressed in this work are explicitly presented. These challenges refer to open issues on high-level pervasive service creation that have not been comprehensively addressed by existing research work; to the best knowledge of the author. Chapter 2 presents a more detailed review on these challenges via comparison with existing research efforts.

Service creation is a complex process that necessitates a ***conceptual model-driven framework*** that provides a high-degree of automation in software generation so as to simplify and expedite the creation of pervasive services. Current research efforts

[2] Two complementary validation techniques are provided in the proposed approach for analysing both the static and dynamic structure of pervasive services.

concentrate on service creation using high-level SCEs that are confined to platform specific implementation technologies [9]. These approaches do not fully exploit the potential of software tools to formulate a generic model-driven development environment that smoothly supports the phases of the service creation process. Moreover, an abstract model-driven development methodology is required, which should depend on the software capabilities of the environment to aid the generation of SCEs for different application domains and support domain-specific service creation.

Pervasive service creation increases the complexity of the process due to the diversity of requirements that must be considered for the development of pervasive services. ***Context-awareness*** refers to the most important characteristic of pervasive services that allows utilising context information in order to achieve the ***adaptability of the service***. This requirement is captured in the form of an advanced information model, on the basis of a context modelling language and its modelling editor. This allows incorporating context-awareness into pervasive services at the static compile time. Consequently, these context-awareness mechanisms built-in at the service creation phase will be triggered during the service execution phase to provide inherent and therefore much enhanced pervasive service adaptability. This approach complements the main-stream service adaptation methodology that is largely based on a complex middleware infrastructure.

Existing modelling approaches that deal with context-awareness at compile time do not consider the prerequisite to have a common and widely-accepted specification, which facilitates the definition and eases the comprehension of context models. Furthermore, supplementary modelling techniques (i.e. modelling frameworks) are required to support the definition of the ***graphical user interfaces*** *and* ***the dynamic behaviour of the***

pervasive service. Also, the lack of a conceptual framework complicates the development of the modelling frameworks that facilitate the definition and validation of the overall pervasive service requirements. This thesis proposes the ***MDPNF*** that defines a model-driven Petri Net based process, which is based on three modelling languages that support the specification of the static and dynamic structure of pervasive services. In addition it defines a PSCE that provides the necessary software tools that support the modelling, validation and implementation tasks associated with the aforesaid process.

The key objective of the MDPNF is to ***minimise the effort, time and cost*** required to create pervasive services at the static compile time. Foremost, the platform-independent nature of the approach facilitates the generation of the service implementation from the pervasive service models. An additional factor that contributes towards this objective is the capability to generate different service implementations from the models, ensuring as a result the ***service portability requirement***. In overall, the MDPNF automates to a large extend the tasks of the pervasive service creation process.

1.3 Model-Driven Petri Net based Framework for Pervasive Service Creation

The conceptual framework proposed in this work comprises a MDD methodology and a supporting generic environment, which facilitate the generation of the PSCE and support the pervasive service creation process. Initially, during the analysis phase the complementary modelling languages are defined, which are the Context Modelling Language, the Presentation Modelling Language and the Petri Net Process Modelling Language. From the abstract and concrete syntax definition of the modelling languages

the corresponding modelling frameworks are effectively generated and integrated into the generic environment to compose the overall PSCE.

The modelling frameworks of the PSCE support the design phase via the definition of context, presentation and process models, which compose the pervasive service. Context models capture the context-awareness characteristic, while presentation models represent the graphical user interfaces with which the user interacts with the service. Furthermore, Petri Net based process models define the dynamic behaviour that guides the execution of the pervasive service. Since the discrete models define the overall pervasive service, the necessity arises to integrate the different modelling domains under a unified process that essentially steers the remaining phases of pervasive service creation.

This is accomplished via the definition of the model-driven Petri Net based process, which is based principally on the MDD methodology of the conceptual framework and the Petri Net formalism. In particular, the adopted Petri Net formalism allows integrating the distinct modelling domains and supports the validation of the dynamic operational semantics of the pervasive service. This complements the validation capabilities of the OCL language that allow checking the static structure and syntax of the models. Hence, following the validation phase, which ensures the correctness of the pervasive service models, the implementation phase can be undertaken. The integrated PSCE provides the necessary software capabilities that permit the execution of the implementation phase mainly by means of model-to-code generation.

1.4 Major Contributions

The technical contributions are summarised as follows:

- The thesis proposes and defines a model-driven conceptual framework to support pervasive service creation. Foremost, the conceptual framework specifies a MDD methodology that facilitates the phases of the pervasive service creation process. Furthermore, an Integrated Model Driven Environment (IMDE – i.e. generic envoronment) is designed and implemented that supports the MDD methodology, automates the development of domain-specific service creation environments and automates to a large extend the service implementation phase.
- A Model-Driven Petri Net based Framework (MDPNF) is defined that introduces pervasiveness into the conceptual framework. In particular, the combination of the Petri Net formalism with the MDA paradigm in the MDPNF is imperative since it allows defining the required modelling languages and integrating as a result these discrete pervasive service modelling domains. Moreover, it complements the static validation capabilities of the conceptual framework by introducing a formalism that allows validating the operational semantics of the pervasive service. The MDPNF is comprised by the generated Pervasive Service Creation Environment and the defined model-driven Petri Net based process that allow designing, validating and implementing pervasive services rapidly at the static compile time.

1.5 Thesis outline

The rest of the thesis is organised as follows:

Chapter 2 introduces service engineering and presents related work on generic service creation. Subsequently, the context-awareness characteristic is introduced and research work on pervasive service creation is presented that considers merely this characteristic. Concluding, related work on Petri Nets is reviewed to reveal its importance in this work.

Chapter 3 first provides an overview on domain-specific MDD. It then proposes essential MDD requirements and compares existing model-driven environments that claim to support MDD. In accordance to these requirements the conceptual framework is proposed and used for the generation of an example domain-specific SCE. This chapter also presents the proposed Model-Driven Petri Net based Framework.

Chapter 4 presents the definition of the Context Modelling Language and the generation of the Context Modelling Framework. In addition the chapter presents the definition of the Presentation Modelling Language and the generation of its supporting Presentation Modelling Framework.

Chapter 5 introduces the concepts of the Petri Net formalism and presents methodically the proposed model-driven Petri Net based process. It also presents the definition of the Petri Net Process Modelling Language and the generation of its supporting Petri Net Process Modelling Framework.

Chapter 6 presents the creation of a pervasive service prototype using the MDPNF. On the basis of the case study a quantitative evaluation of the MDPNF is performed.

Chapter 7 presents the summary and conclusions of the thesis and proposes potential future directions of this work.

Chapter 2 Literature Review

This chapter presents an initial overview of service engineering and introduces the most profound service engineering paradigms. Particular focus is given on the key phase of the service engineering lifecycle that refers to service creation. Following that, related literature is reviewed with emphasis on two very important dimensions of this thesis work. Firstly, enabling methods on generic service creation are presented that use abstract approaches and high-level service creation environments. Furthermore, research efforts on pervasive service creation are introduced that consider merely the context-awareness characteristic for creating pervasive services. Concluding, related work on Petri Nets is introduced that discloses the importance of a formal technique for pervasive service creation.

2.1 Overview of Service Engineering

The escalating dependency of people on computer-aided support to perform job related, personal and miscellaneous everyday-life tasks shifted most of the application load from software programming experts to service engineering experts or even to end users. More specifically, the fast-changing users' requirements and the daily advancements of mobile hardware technologies, wireless mobile communications and worldwide networking moved the maintenance, adaptation and modifications of software services to the hands of service engineering experts. Therefore, service providers demand new service engineering paradigms, technologies and processes that facilitate the rapid introduction of validated services in a cost-effective manner [18].

Service engineering is a very broad term that describes the discipline that defines the processes, supporting software tools and technologies necessary to engineer services that meet the rapidly changing users' requirements. The term *service* in the context of this thesis denotes a software application that can be deployed onto a particular device and platform and can be utilised by the user to perform specific computing task(s). Moreover, the idiom "*engineer services"* describes the processes required to create, deploy, manage and maintain software services. At first, service engineering inherited most of its principles from the software engineering discipline. Nowadays though, there is a clear distinction between the two disciplines since service engineering alters the conventional way that applications were developed using programming languages.

Marie-Pierre Gervals defines service engineering as follows: *"Service engineering is a new discipline in which the telecommunication sector addresses the technologies and processes required for service creation"* [18]. In this thesis the author differentiates from

the above statement and aligns with the more acceptable definition provided in [10], which considers service creation merely as the most important stage in the overall service engineering lifecycle. In particular, service creation is considered the key stage in the service engineering lifecycle due the fact that it defines the agility and effectiveness with which services are being developed. This thesis research work focuses on the service creation aspect of the service engineering lifecycle, which examines the creation of pervasive services during the static compile time.

Recently, various computing paradigms have emerged that attempt to provide solutions to service engineering by providing high-level processes, technologies and software development environments. These paradigms aim to support the creation, deployment, management and maintenance of value-added services, engaging predominantly on the key stage of the service engineering lifecycle; i.e. service creation. The most profound and widely-used state-of-the-art service engineering paradigms are Web Services, Service Oriented Architecture (SOA) and Pervasive Computing.

2.1.1 Web Services

Web Services is an emerging paradigm that allows reuse of software applications as services over the Internet. The Web Services paradigm attempts to exploit the valuable characteristics of the Internet, which are global connectivity, decentralization and openness in order to allow users to consume online services. Web Services are considered reusable software components that are distributed over the Internet and support the execution of a particular computing task or sets of tasks. Furthermore, complex web services can be composed from other more basic Web Services so as to support the

execution of large scale business transactions; e.g. order handling, billing, trouble ticketing.

The paradigm provides solutions for every phase of the service engineering lifecycle, in order to support the creation, deployment, management and maintenance of Web Services. In particular, new software development processes and standards are introduced that allow creating Web Services rapidly and efficiently. For example, the Web Services Description Language (WSDL) is a *de facto* language based on the Extensible Markup Language (XML), which allows a Web Service to be described and its interface defined in a machine processable format that specifies how the service can be accessed and used. Moreover, supplementary open web standards are defined such as the Simple Object Access Protocol (SOAP) and the Universal Description, Discovery and Integration (UDDI) specification. SOAP is a lightweight protocol that allows information messages to be exchanged in a decentralised and distributed network, while UDDI enables businesses to share information in a global business registry, discover services on the registry and define how they interact over the Internet.

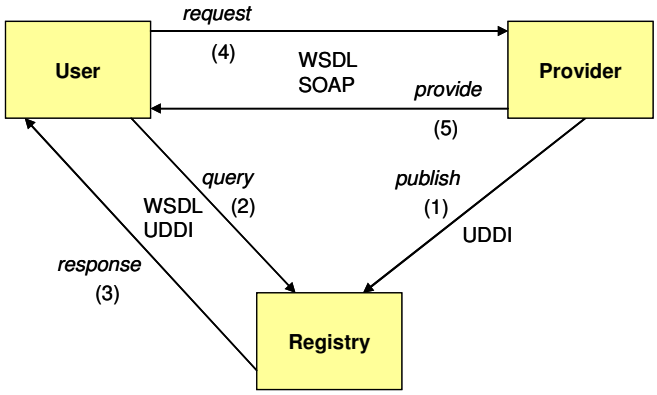


Figure 2.1: Web Services communication scheme.

Figure 2.1 illustrates the communication scheme between a user that requires access to a Web Service, a provider that offers the Web Service and the registry that stores

information about it. Initially the provider is responsible for publishing into the registry the necessary information about the Web Service (i.e. what it offers) using the WSDL, so that users can locate that service and determine how to communicate with it (1). Subsequently, the user issues a query message in order to identify the Web Service that allows him to perform particular computing tasks (2). The user retrieves with the help of a response message (3) the necessary information and then requests access to the service (4). The provider then responds by granting access to the Web Service (5). This communication scheme provides a transparent and straightforward method that allows providers to offer easily their Web Services and users to consume them according to their requirements.

The rapid growth of Web Services motivated the development of additional technologies and standards, such as the Web Services Technologies in Java 2 Platform Enterprise Edition (J2EE) and the Web Services Security (WS-Security) standard. Furthermore, various high-level (i.e. advanced) Web Service engineering tools have been developed, which conform to the defined technologies and standards so as to assist Web Service developers. In particular, these graphical development tools are dedicated principally to Web Service creation and support developers in rapidly and efficiently creating Web Services.

2.1.2 Service Oriented Architecture

Service Oriented Architecture (SOA) is an architectural pattern that emerged principally from the concept of Web Services and the need for reuse of business services. SOA is considered in fact an architectural technique for developing applications that use services (e.g. Web Services) available over the network. In particular, applications are developed

on the basis of services that describe software components with well-defined interfaces. Figure 2.2 illustrates how computational units can be loosely coupled using a service bus through their service interfaces in order to deliver the necessary functionality [19]. Reusability is the keyword in the context of SOA that enables the loose coupling of software components to develop complex business applications. The SOA pattern is commonly applied throughout the architecture of an enterprise for the development of applications or it can be applied over a single system of the overall architecture.

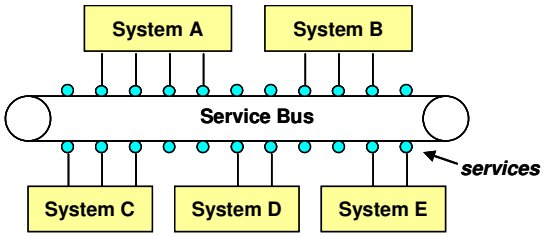


Figure 2.2: Loose coupling of applications over a service bus using services.

SOA provides definite benefits to services engineering since it allows expediting and simplifying the creation of applications by combining existing software components. Moreover, the loose coupling of services in SOA ensures that systems can be easily adapted in accordance to the users' changing requirements and easily maintained in the case of faulty conditions. Apart from the benefits, the loose coupling of services also brings some shortcomings, which are lower performance in contrast to tightly coupled software components and higher incurring hardware costs. Despite the initial costs introduced when adopting SOA, the paradigm provides the analogous return on investement since it offers great flexibility in implementing wide-enterprise changes and allows smooth modification of information systems [19].

SOA is predominantly realised via Web Services but other types of services can be also considered in order to deliver the architectural pattern. Therefore, as in the case of the

Web Services paradigm the necessity arises to develop high-level development tools for SOA support. These tools provide the necessary support across the engineering lifecycle, providing the software capabilities for modelling, assembling, deploying, and managing SOA systems. In particular, a fully integrated development environment is formulated that integrates programming, graphically-oriented and modelling software tools. The integration of a SOA-enabled development environment is imperative since it supports, simplifies and speeds up the activities undertaken by analysts, architects, designers and developers when building SOA applications.

2.1.3 Pervasive Services

Pervasive Computing is a paradigm that differentiates slightly from the above since it introduces an additional aspect to service engineering. In particular, it examines the development of pervasive services that offer a dynamic, personalised and customisable computing experience to end-users. The concept put forward is to provide computing capabilities for users anywhere, anytime and on any mobile device and platform. Therefore, in contrast to the aforementioned paradigms there is no established or preferred method for developing pervasive services, mainly due to their dynamic nature and the augmented service adaptability requirement.

The most significant aspect that differentiates pervasive services is the heterogeneity of sources from which input information originates. This information, termed “context”, affects both the behaviour and the interaction of the user with the service in a variety of different ways. In contrast, when developing any other type of services (e.g. Web Services) input information is considered to be obtained principally from the user. Consequently, the service defines an explicit functionality captured in the implementation

that does not consider and is not subsequently affected by implicit context information. In the case of pervasive services though, the requirement arises to capture in the implementation the conditions that influence the service functionality. This enables the service to adapt its functionality according to implicit and/or explicit context information.

Ongoing research work proposes various approaches for pervasive service creation that are categorised in this thesis as: (i) infrastructure-based methods, (ii) context adaptation methods and (iii) context modelling methods. Firstly, infrastructure-based methods are concerned with the development of an underlying infrastructure that supports sensing, gathering and processing raw data obtained mainly from sensors to obtain high-level context information required by the pervasive service. The service developers then implement the necessary functionality taking into consideration how to utilise information obtained from the underlying infrastructure.

Secondly, context adaptation methods deal with the development of generic context platforms, middlewares, or architectures that facilitate and expedite the development, deployment and provisioning of pervasive services. Context adaptation methods aim to separate the functionality of the pervasive service from the context management (i.e. context adaptation) tasks, so as to simplify the creation of pervasive services. Therefore, context adaptation tasks are solely provided by the platform that is responsible to manage and deliver context information to pervasive services for adapting their functionality.

Finally, context modelling techniques deal with the representation of context in the form of an advanced information model. The context model is then transformed to the corresponding implementation, which is responsible for executing context management tasks that facilitate the adaptation of the service behaviour. Consequently, context

modelling techniques regard context management tasks as an integral part of the service functionality, which needs to be captured in the service design and implementation. Note that application level methods consider that the infrastructure for sensing, gathering and processing raw data to obtain an abstract context description is already implemented.

As already mentioned this thesis work examines pervasive service creation at the static compile time. The proposed Model-Driven Petri Net based Framework follows the research direction of context modelling, which describes the pervasive service using models and automatically generates the required implementation. The proposed approach considers the use of model-driven service creation environments, which facilitate the analysis, design, validation and implementation of pervasive services at an abstract level and provide a high degree of automation in software generation.

In the following section related research efforts on service creation are introduced, which utilise high-level service creation environments to accomplish the service creation process. Initially, generic service creation approaches are presented that aim to undertake service development at a platform-independent level using XML-based scripting methods. Static service creation methods are then introduced that use graphically-oriented service creation environments to develop telephony services at an abstract level, distinct from platform-specific implementations. Concluding, model-driven development approaches are presented that attempt to define services in the form of models and automate the generation of the service implementation.

2.2 Generic Service Creation Enabling Methods

Research work on service creation and particularly in the domain of telecommunication services has revealed that a coherent methodology and a high-level service creation

environment are required to enable rapid development of advanced services [10]. The drift torwards platform-independent service creation approaches emerged starting from the use of the XML language down to the use of model-driven approaches. The following subsections introduce successively XML-based scripting methods, graphically-oriented methods and model-driven methods that provide platform-independent solutions for generic service creation at the static compile time.

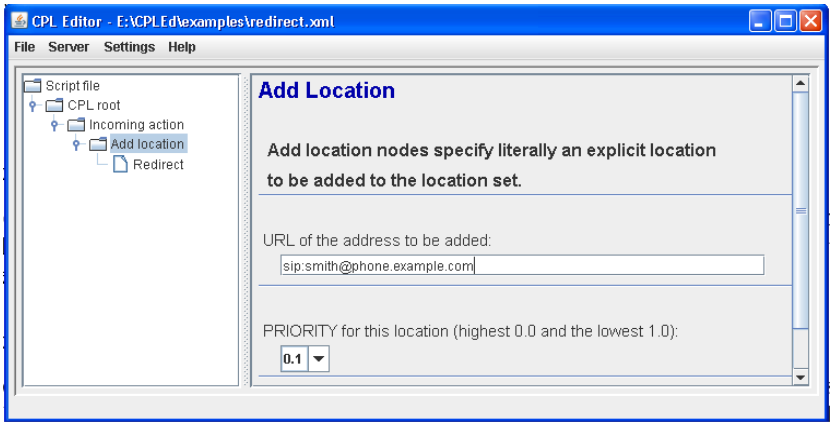
2.2.1 XML-based Scripting Methods

One of the initial XML-based methods is proposed by the Internet Engineering Task Force (IETF) [20], [21], which defines a Common Gateway Interface (GCI) for the Session Initiation Protocol (SIP). The technique is based on HyperText Transfer Protocol (HTTP) CGI and provides the capabilities for creating new telephony services effectively and deploying those services rapidly. Web Service creation environments (CGI in particular) are used as guidelines to facilitate the development of SIP service creation environments. Value-added telephony services are then created via the developed SCEs using a comprehensive service creation process.

Complementing the above, the IETF proposed an additional service creation environment for the development of telephony services, which is based on the Call Processing Language (CPL) [22]. The approach describes an architectural framework (referred by the authors as the CPL), which provides a standardised method to describe signalling operations. The CPL allows users to write programs that define the signalling operations with which network telephony devices respond to signalling events. This facilitates the rapid creation of telephony services and provides an effective approach for their implementation and deployment.

The authors in their approach describe and envision different creation styles for writing and editing a CPL script. Although the basis of the scripts is the Extensible Markup Language, the authors state that as in the case of the HyperText Markup Language (HTML) the following creation styles can be applied to CPL scripts.

- Hand authoring – The CPL scripts are created by hand from users that are knowledgeable with the technology involved.
- Automated scripts – CPL scripts can be created by automated means; perhaps using a web application (e.g. web interface).
- Graphical User Interface Tools – Potentially graphical tools can be provided that allow users to create and edit CPL scripts in an abstract visual manner.




```
<?xml version="1.0" encoding="UTF-8" ?>
<cpl xmlns="urn:ietf:params:xml:ns:cpl"
        xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
        xsi:schemaLocation="urn:ietf:params:xml:ns:cpl cpl.xsd">
<incoming>
        <location priority="0.1" url="sip:smith@phone.example.com">
                <redirect permanent="yes" />
        </location>
</incoming>
</cpl>
```

Figure 2.3: Example script: Call redirect.

While hand authoring is the most commonly used method for writing CPL scripts, the approach provides also partial support for the use of automated means or graphical tools for creating CPL scripts. An example is the open-source CPL Editor (CPLEd)

implemented in Java, which is used for editing scripts in a graphical and straightforward way. Furthermore, the CPL concepts are also used in [23] to define a visual domain-specific modelling language and use the MetaEdit+ modelling tool to design telephony services. Hence, the transition from manual editing that requires specialised expertise to high-level graphical editing or modelling allows novice users to easily create telephony services. Figure 2.3 illustrates the use of the CPLEd tool to define in a graphical manner and parse directly into an XML document the resulting CPL script, which denotes a redirect action for all calls to the specified location.

An approach that differs slightly from the above is the Service Creation Markup Language (SCML), which is actually a standard scripting language [24]. The SCML is developed by the Service Creation Environment Expert Group of the JAIN industry forum. According to the authors, the SCML operates at a higher level of abstraction than the CPL. Furthermore, it provides richer functionality and modular and extensible structure, so as to facilitate further development of the service creation mark-up language standards. The definition of the SCML is provided in the form of an XML schema, while CPL is defined using XML Data Type Definitions (DTD). This is basically the reason for the modularity and extensibility of the approach compared to its CPL counterpart.

Despite the stated advantages of the SCML over the CPL, both refer mainly to using XML-based scripting languages to create telephony services. In particular, the SCML approach allows service creation using different capabilities, such as using XML editor tools or by manually editing scripts using simple text-based editors. Apart from editing scripts the capability is provided to automatically create scripts by translating Java (or JavaBeans) programs implemented in a SCE. Moreover, the modular and the extensible

base of the SCML allow the SCE to load existing SCML scripts or JavaBeans from a repository of services to compose more advanced services. Figure 2.4 illustrates the upper layer of the SCML architecture that refers to the service creation process using the proposed SCE. The figure illustrates the software capabilities of the SCE, which allow the manual definition of SCML scripts or JavaBeans to develop a service or load existing services from the repository to define more complex services.

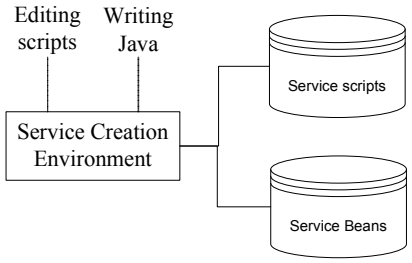


Figure 2.4: Service Creation Layer.

The aforementioned approaches aim at raising the abstraction level and contributing platform independent approaches that simplify and expedite the creation of telephony services. Primarily the XML-based techniques utilise the expressive power and the flexibility of the XML standard for defining service specifications. However, there are some outstanding issues though with the use of the XML for the purpose of service creation. Firstly, XML-based specification languages are rather data-centric and cannot express conveniently the required service behaviour. Hence, the execution of the behaviour relevant to the service is largely based on the application, which is responsible to process the XML-data describing a corresponding behaviour; i.e. function. Consequently, the developers must have the required knowledge about how the scripts are actually executed using the underlying application technologies; e.g. Parlay, JAIN, servlet APIs.

Furthermore, the XML representation of the languages can be difficult to comprehend in order to realise effectively different service aspects. The development of preliminary tools such as the CPLEd introduces an additional abstraction layer to the process that eases the service specification. Specifically, the use of GUI tools simplifies the definition of service scripts and enables better understanding of the service functionality. Another predicament though is the necessity to manually implement these software tools, which act as the front-end of the scripting languages. This is because the development of these GUI tools introduces a time consuming and costly stage to the process. Despite these limitations the use of XML-based scripting languages provides an initial step towards the fabrication of a platform-independent process and the utilisation of a supporting high-level SCE for accomplishing the rapid creation of advanced services.

2.2.2 Graphically-Oriented Methods

Bernhard Steffen et al. [25] proposed, designed and implemented a constraint-oriented service creation environment for the development of Intelligent Network Services (INS). The environment is used for the reliable, aspect-driven creation of advanced telephony services by developing prototypes that are successively modified until every component satisfies the existing requirements. In specific the service creation process is facilitated by thematic views that allow the designer to select particular aspects of the service to investigate, so as to create the service. The implementation of the service creation environment is based on METAFRAME, which is an environment that supports the flexible management of large repositories of complex components.

The approach requires developers with advanced programming skills to be involved in the implementation of distinct service components (i.e. libraries). The components are

developed using the SCE and subsequently designers are able to load and utilise them in a graphically-oriented manner. Figure 2.5 illustrates a fragment of the service components taxonomy, which allows the designers to compose services using the implemented components [25]. Moreover, the service models that are composed using elementary design modules can be validated by enforcing various initial constraints. If a violation is detected, necessary diagnostic information is presented signifying the exact problem and pinpointing the constraint responsible for the violation. Consequently, contstraints validation guarantees the service executability and other consistency conditions via model checking.

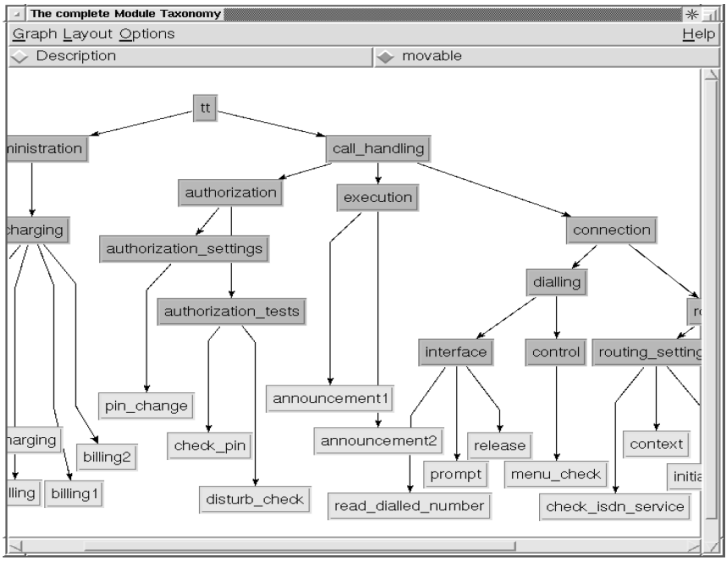


Figure 2.5: Fragment of the service components taxonomy.

In the research work developed by Roch H. Glitho et al. [9] a high-level service creation framework is presented that attempts to steer clear of technology specific complexities. The proposed framework comprises a high-level Service Creation Environment and a Service Logic Execution Environment (SLEE) that respectively support the creation and execution of Internet Telephony services. Consequently, these two environments and

their associated concepts are used to expedite and simplify the development and deployment of services. The authors state the following: *"A SCE is generally a graphical user-interface for developing services using predefined components, also called building blocks"* [9]. This reasoning provides a clear assertion of the research direction towards a graphically-oriented platform independent approach that facilitates service creation.

Figure 2.6 illustrates a snapshot of the SCE that represents a (kind of) model-driven environment, which facilitates the abstract definition of telephony services using a set of predefined Java components. Consequently, via the SCE the designer is able to utilise the drag-and-drop functionality of the editor to select components and model the service. Note that the palette of the SCE is populated with the set of currently implemented components, but the library of components can be enriched with supplementary building blocks that enhance the capabilities of the SCE and facilitate the creation of advanced services. Subsequently the formulated service definition can be transformed using the code generation capabilities of the SCE to corresponding Java classes that represent the actual service implementation. Hence, when the library of components is expanded then additional code generation capabilities should be embedded to the SCE.

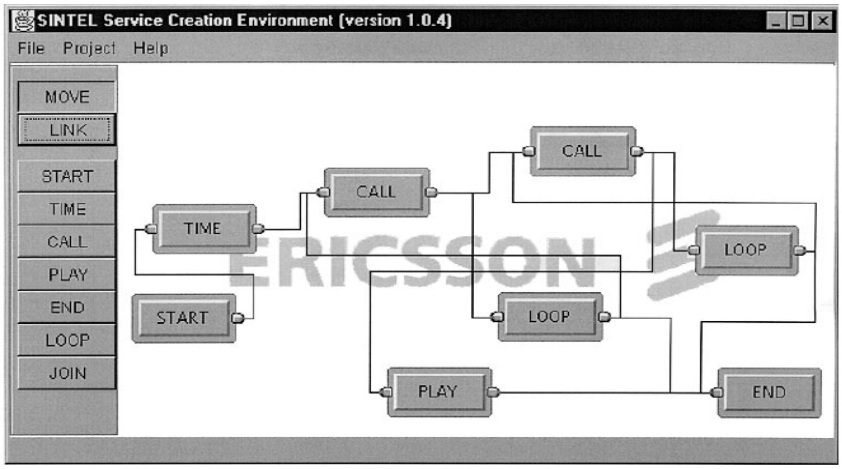


Figure 2.6: Service Creation Environment Snapshot.

The aforementioned approaches focus primarily on the development of a suitable service creation environment, which includes GUIs for creating services using predefined components. This simplifies the service definition and provides a graphical representation that is highly appropriate in terms of human structuring purposes [26]. The manual implementation of the SCE is still a significant issue that needs to be examined in order to propose new solutions. Furthermore, the requirement to re-implement the GUI tools when developing new software components and enhance the code generators complicates further the development of the SCE. Finally, a clear-cut methodology should be defined in the approach so as to aid and guide the developers during the service creation process.

2.2.3 Model-Driven Development Methods

In an attempt that progresses further the concept of high-level service creation Farias [27] proposes a methodology for the architectural design of service-based systems using the Unified Modelling Language. In the approach the main concept for service creation is the functional block that represents an entity (e.g. a system, component, or person) capable of executing behaviour in the real world. On the basis of the entity concept a methodology is defined that comprises different perspectives and views, which facilitate the design of the service at various abstraction levels. Figure 2.7 illustrates the different perspectives and views defined and the relationships between them.

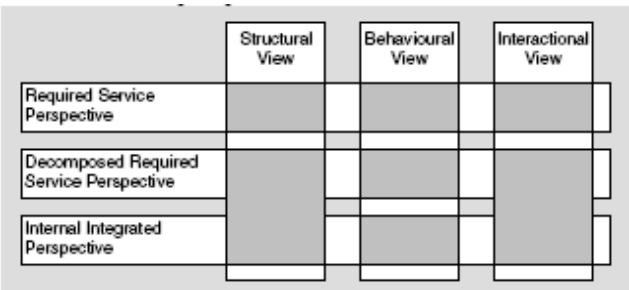


Figure 2.7: Views and perspectives for service specification.

The *Required Service Perspective* describes the service provided by the service provider to the service users. The *Decomposed Required Service Perspective* is concerned both with the description of the service and the relationship of the service with other providers' services. Finally, the *Integrated Internal Perspective* describes the internal behaviour of an entity and its relationship to the decomposed required service of this entity. Apart from perspectives the approach defines also three views, namely, the *structural view, behavioural view* and *interactional view*. The structural view describes the static structure of the corresponding behaviour, the behaviour view is concerned with describing the behaviour of an entity and the interactional view describes the cooperative behaviour of two or more entities.

The aforementioned design concepts are represented effectively using the standardised and widely accepted notation of the generic UML. In particular, various diagrammatic conventions are used such as use case diagrams, activity diagrams, interface diagrams, statechart diagrams and interaction diagrams, which provide the capability to produce rigorous service design specifications. Furthermore, the approach proposes service design refinement methods and a corresponding strategy for the assessment of the refinements performed. Concluding, the author states that although the generic UML is not adequate for designing the complete set of information needed for the design of services, it still serves as a strong basis for service specification. In addition the author notes that on the basis of the produced models, transformation capabilities can be defined that allow miscellaneous service representations (i.e. implementations) to be derived.

A similar approach is proposed by Dionisis X. Adamopoulos et al. [10], which analyses existing issues and proposes solutions that simplify the creation of telecommunication

services. The authors state explicitly that apart from a high-level SCE, a systematic service creation methodology is required so as to guide and support the creation of telecommunication services. In particular, the authors state that a service creation methodology should guide service developers in a methodical and structured way throughout the entire process of service creation. Furthermore, the authors note that a SCE should comprise a bundle of software tools, which must be used according to the defined service creation process. The objective is to assist and facilitate the service developer(s) applying the methodology to automate and simplify many of the phases of the service creation process.

In their proposed approach a service development methodology is presented that comprises of four main phases, which are: (i) service analysis, (ii) service design, (iii) service implementation and (iv) service validation/testing. The objective of the analysis phase is to determine the necessary functionality so as to satisfy the service requirements, by identifying the artefacts required to compose a service (e.g. elements, relationships) and describe a conceptual model for telecommunication services. Then during the design phase the developer needs to specify (i.e. model) the static and dynamic structure of the service. In the proposed approach the service specification is defined using different UML diagrams (e.g. class diagrams, sequence diagrams).

The completion of the service specification phase leads to the service implementation phase, where primarily a particular implementation technology needs to be selected. Hence, the implementation of the service is automatically generated from the service specification with the aid of code generators that are responsible for the translation of the service models. Finally, the methodology defines the service validation and testing phase,

during which the service specification is manually compared against the service implementation. This allows detection of any inconsistencies between the generated service code and the service models, so as to resolve those issues prior to the deployment of the service. Moreover, the software service is evaluated in order to corroborate that it meets the service requirements. The final step of the validation phase defines that software testing should be executed using conventional testing practices.

In addition to the methodology, a SCE is proposed that is composed of a collection of software tools. The majority are actually UML modelling tools that provide the capability to define different service models such as UML class diagrams, interaction diagrams, and use case diagrams. Apart from modelling tools additional software tools exist, such as code generation tools that facilitate the transformation of the service specification to the corresponding service implementation. In particular, each phase of the methodology is supported by a respective set of software tools, so as to assist the developer and simplify the development tasks associated with that phase.

From the aforementioned approaches, the method proposed by Adamopoulos et al. best illustrates the concept of model-driven development since it utilises the UML for service specification. Furthermore, automatic generation capabilities are provided that allow transforming the service specification to the desired implementation. Consequently, the developer works at the modelling level thus reducing the interaction with platform-specific implementation complexities. Apart from the model-driven SCE, a service creation methodology is defined that guides the developer during each step of the development process.

The only limitations of the approach are the use of the generic UML language and the complexity of the methodology. Foremost, the use of the general-purpose UML for executing the analysis and design phases makes it difficult to represent coherently the domain-specific concepts in the service models, since UML was inherently designed to address different problem domains at a higher-abstraction level [28], [29]. Furthermore, although the proposed methodology is well-formed, it relies on multiple sub-processes something that renders the approach increasingly complex.

The work introduced in the thesis builds upon knowledge gained from aforementioned research work to formulate an abstract methodology that supports the service creation process via the use of a model-driven service creation environment. Consequently, the methodology should guide the developer through the phases of service creation, providing also the capability to automate the phases of the process as much as possible. In particular, the methodology and the supporting environment should automate the development of software tools (i.e. modelling frameworks) and simplify the definition of code generators, in order to facilitate the service creation process. This requires defining and designing an MDA-based environment that enables the above tasks and allows the produced tools to be instantaneously integrated as new software capabilities of the environment.

The methodology should be applicable across different service domains (e.g. pervasive services, Web Services) and for that reason the nature of the modelling approach must be domain-specific rather than general-purpose; i.e. UML. A domain-specific approach provides the well-acknowledged advantage of capturing the semantics of the problem domain with a higher-degree of precision [28]. Moreover, a methodology that supports

service creation at the modelling level should allow validating the models prior to the service implementation. In the aforementioned approaches service validation was rather limited or was applied after the service implementation using conventional software testing techniques. The author argues that the validation of service models is essential and should complement conventional testing techniques.

In this thesis the focus is to utilise the proposed MDD methodology and the developed generic environment in the pervasive service domain, so as to reduce the time, effort and costs required for the creation of pervasive services. The following subsection provides an overview of context-awareness, which is considered the key requirement for pervasive service creation. Subsequently, related research work on context-aware service creation is critically evaluated, to identify any shortcomings and propose solutions.

2.3 Pervasive Service Creation

In this section research work on pervasive service creation is introduced that proposes different solutions to facilitate and expedite the creation and deployment of pervasive mobile services. An overview of the most important characteristic of context-awareness is provided, followed by an introduction to distinct research paths that aim though torwards the common objective of pervasive service creation.

2.3.1 Overview of Context-Awareness

As aforesaid, service creation is indeed a complicated process that refers to a set of activities for the rapid analysis, design, implementation and validation/testing of services. In the thesis the term service refers to a software application that implements a set of computing tasks and can be deployed on a specific device (e.g. PC, mobile) and platform that support the execution of these tasks. By constrast, pervasive services denote

applications that can operate in a dynamic environment and have the capability to run anytime, anywhere and on any device with minimal user attention [6]. This indicates that these services are able to process and utilise information relevant to the users (e.g. profile) providing a personalised and specialised behaviour. Therefore, pervasive services are able to perceive information about the current context of the users and utilise it to assist the interaction with the service in a non-intrusive manner.

Context refers to the information that governs the interaction of the user with the service that can be either implicitly or explicitly perceived. During the course of research context has acquired a variety of different meanings [30], [31] depending on the way different researchers perceive the notion of context. Some researchers define context by example restricting the information to explicit categories, while others provide synonyms for context such as the environment or situation [30].

In this work a generic definition is provided that defines context as: *"Any information relevant to the interaction of the user with the service, where both the user's and application's environment are of particular interest"*. The proposed definition is aligned with the generic nature of the term and does not attempt to restrict context to explicit categories. However in order to guarantee the applicability of the approach proposed in this thesis a categorisation of the significant aspects of context is performed. Furthermore, the extensibility to define additional (i.e. secondary) aspects of context, not included in the categorisation, is also provided in the proposed approach.

The capability to perceive, acquire and utilise context information describes the prime characteristic feature of pervasive services; that is context-awareness. In specific context-awareness describes the necessity to react in accordance with certain predefined rules

and/or on the basis of intelligent stimulus, so as to dynamically adapt the pervasive service behaviour. Consequently, it is realised that the complexity of the service creation process is increased when dealing with pervasive services since the functionality of the service must be adapted in accordance to the information obtained from diverse context sources.

In conventional services the interaction of the user with the service is performed in an explicit manner, via the user manually inputting information to the service. Hence, information is obtained from a single context source and subsequently the management of this information is simpler. In the case of pervasive services the interaction of the user with the service is performed in an implicit and/or explicit manner. This means that information can be obtained explicitly or implicitly from different context sources; e.g. users, repositories, sensors. Therefore, the necessity arises to realise from which source the information is obtained in order to react and manage context information accordingly so as to achieve essentially the necessary service adaptability.

Context-aware service creation has been studied during the course of research following three equally important research directions. The first research direction involves context adaptation methods that commonly separate the context management tasks from the functionality of the service when creating and deploying pervasive services. More specifically, platforms, architectures or middlewares are proposed, designed and implemented that facilitate the development and deployment of pervasive services by offering context adaptation mechanisms. The limitation of these methods is the generic nature of the platform that aims to support context adaptation on behalf of pervasive services. In particular, due to the diversity and complexity of pervasive services it is not

easy to utilise a generic platform that fits all possible context-aware scenarios. Therefore, in many occasions the modification of context adaptation tasks provided by the platform is required in order to facilitate the pervasive service under development.

The second research initiative involves several techniques [32], [33], [34] that have been proposed in the literature, which follow a low-level infrastructure-based solution for pervasive service creation. These techniques mainly define an underlying infrastructure that supports low-level tasks such as sensing, gathering and processing context information required by the pervasive service. Although these tasks simplify the service creation process, the requirement arises to tailor the service implementation on the basis of the implementation of the supporting infrastructure [35].

The third research direction complements the above by introducing techniques working at the application level rather than the infrastructure level [26]. These techniques are not concerned with sensing, gathering and processing raw data to obtain an abstract context description. Their only requirements are the representation of context in a suitable format and the management of context so that it can be distributed, realised and utilised by context-aware services. In principle these approaches are termed as context modelling techniques and deal with management tasks such as the representation, administration and dissemination of context to pervasive services to achieve their adaptation.

The primary objective of context modelling techniques is the representation of context in the form of an advanced information model. Hence, the context model can be defined via the use of a context modelling framework that comprises a modelling language and a supporting editor with drag and drop capabilities. Furthermore, the mapping between the language and an implementation technology is defined, so as to facilitate the

transformation of context models to platform specific code. Both the mapping and the context models steer the automatic generation of the corresponding implementation. The implementation acts in fact as the bridging point – similar to an API (Application Programming Interface) – that allows context to be utilised by services [36]. It performs tasks for managing a context repository such as querying, administrating and distributing information to pervasive services. The following subsection presents research work on context-aware service creation introducing context adaptation methods, infrastructure-based approaches and context modelling techniques.

2.3.2 Context Adaptation Platforms, Architectures and Middlewares

The primary research direction investigates approaches that propose intermediary context adaptation platforms, architectures or middlewares that support context management tasks. In particular, the context adaptation tasks are implemented separately from the implementation of the logic of the context-aware service. Stanislav Pokraev et al. [37] define such an approach, which proposes a services platform for the rapid development and deployment of context-aware mobile applications. The platform is based on Web Services, handles different types of context information and offers also sophisticated personalisation mechanisms for pervasive services. Therefore, the context-aware service is implemented based upon the context management capabilities of the platform, making use of those management capabilities to provide a personalised, customisable and dynamic user experience.

The authors denote that the proposed Web Service platform provides a dynamic way to develop pervasive services and is tailored torwards reuse of software components that are exposed as Web Services to the network. Figure 2.8 illustrates the architecture of the

Web Services platform and the demonstration application. At the lower level of the architecture third party services are provided that include 3G network services, context services and business services. The network services comprise network access capabilities provided via accessible web service interfaces and offered by third party mobile network operators. In addition context services provide information about the user's context (i.e. profile) obtained sometimes from the 3G network via Web Services and information relevant to the user when using a business service; e.g. weather information. Finally, business services provide information and services that are required by applications built on the platform.

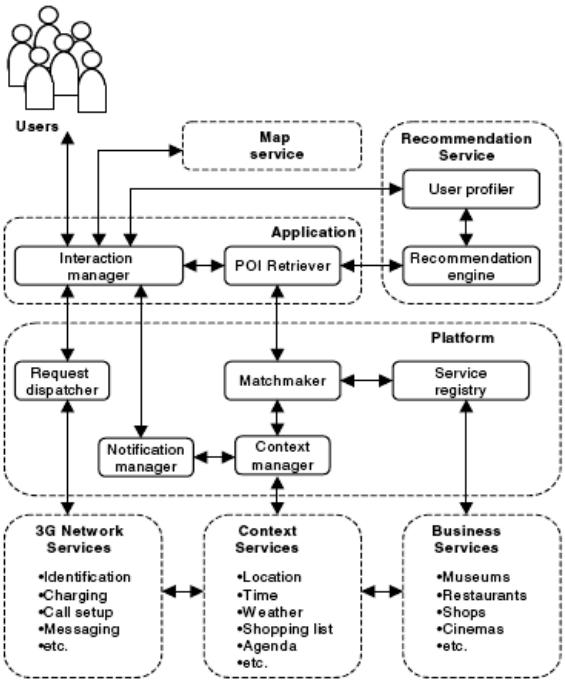


Figure 2.8: Architecture of the Web Services platform.

The middle layer illustrated on the figure shows the platform components that are namely the request dispatcher, the notification manager, the context manager, the service registry and the matchmaker. Each component serves a particular context management or other computing task facilitating the development of the context-aware service. For instance,

the context manager component retrieves user context information by interacting with the appropriate context services and aggregates context or derives new context in accordance with specified rules. The context manager is responsible for updating the notification manager when context information changes.

The COntext-Aware Mobile Personal ASSistant (COMPASS) demonstration application is developed on top of the platform, which provides tourists visiting the city with information and services that aids their touring experience. Firstly, the server-side of the application is composed of the interaction manager and the Point Of Interest (POI) retriever components that serve the client-side part of the application. Furthermore, the server-side uses also an external map service; i.e. Microsoft MapPoint.

The interaction manager component serves requests delegated by the user and assists the interaction of the user with the client-side of the service. Subsequently, the interaction manager forwards the user's request to the POI retriever that creates a search request and delivers it to the matchmaker component. The matchmaker component returns the list of POIs, which the retriever then forwards to the recommendation service, which assigns scores to each POI. The retriever then sends the list to the client side application, which displays POIs with their associated scores.

The components of the platform and additional external components such as the recommendation service are imperative for the implementation of the context-aware service. Therefore, since most of the service tasks are provided by the platform the necessity arises to tailor the implementation of the service closely to the implementation of the platform components. The authors stated that problems were encountered that have

to do with the usage of Web Services technology and that some implementation issues also arose.

For example, a primary problem arose from the incompatibility of the platform-specific implementations since the Microsoft MapPoint service uses NET technology while the authors' implementation uses Apache Axis, which is Java-based. In order to overcome this problem the authors had to change the implementation of the default Axis transport layer authentication mechanism, so that it interoperates with the .NET service. Finally, in terms of the implementation of the COMPASS application the development process started with device independence in mind but ended up having to change part of the Java code that does not generalise to miscellaneous devices.

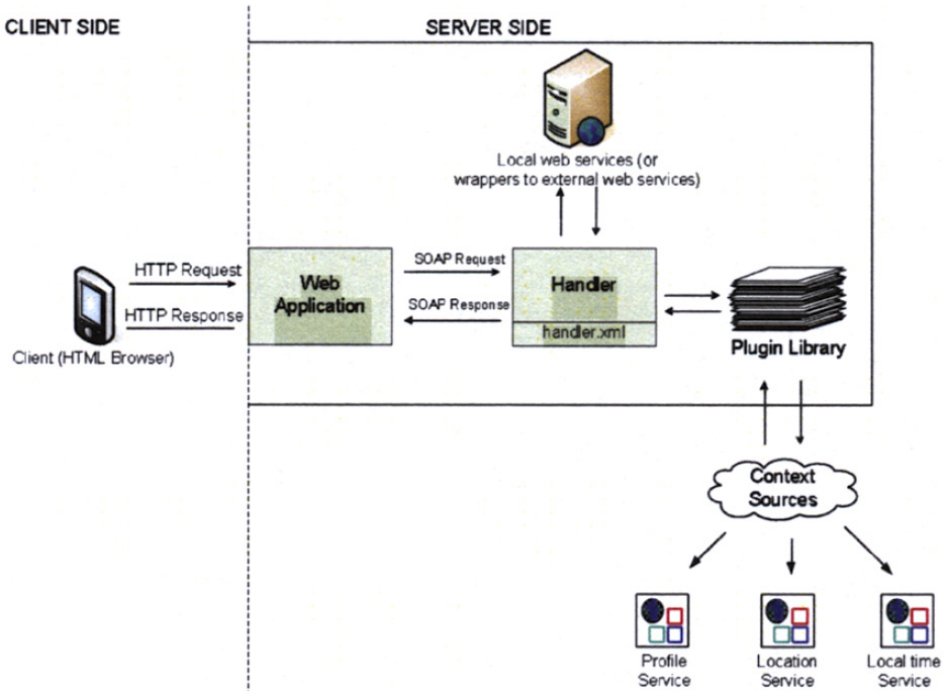


Figure 2.9: Framework architecture and context adaptation process.

In a similar attempt Georgia M. Kapitsaki et al. [38] proposed an architecture based on Web Services that supports the development and provision of pervasive services. The architecture is responsible for executing context management tasks that can be exploited

for the development of pervasive services on the basis of Web Services technology. Web services are actually software components that offer business services to end-users, which the developer can utilise to produce a context-aware Web Service integrating the independent software components. Figure 2.9 illustrates the framework (or platform) architecture and shows the context adaptation process that is served through the use of Web Service technologies and the plug-ins that manage context.

Figure 2.9 illustrates that context management tasks are performed by intercepting the service requests sent over HTTP by the browser application of the thin-client. The requests are handled initially by the web application controller that allows describing in the subsequent SOAP message the analogous Web Services to invoke. In particular, the handler receives the SOAP request and associates the invoked Web Service operation with the appropriate context adaptation plugins. The plugins are then loaded by the handler and adapt the SOAP message body on the basis of the context information obtained from the context sources; e.g. sensors or the user profile database management system. Finally the response message is propagated back to the client application using an analogous communication pattern.

The proposed approach keeps the business functionality provided by the logic of the context-aware service discrete from the context adaptation mechanisms that can be independendly plugged in. In this approach, the first phase of the context-aware service creation involves defining the associations between the business services and the context sources. This entails specifying the mapping in the XML file that indicates the context adaptation that must be performed for each Web Service using the handler component. Subsequently, the context adaptation plugins are developed that perform the required

context management tasks. Finally, the descriptions of the Web Services and their interfaces are defined using the WSDL standard.

According to the authors' claims the main benefit of the approach lies in the complete separation of the context adaptation mechanisms from both the business logic of the application (i.e. combined Web Services) and the application client (i.e. browser). Therefore, the maintenance or modification of the context adaptation logic is completely transparent from the client application, the Web Services and the user. Although it is reasonable and attainable to separate the adaptation logic from the functionality of the context-aware service, this distinction is valid only up to a certain point. In this thesis the argument is made that the strong correlation of context management tasks with the pervasive service behaviour and the interaction of the user with the service does not allow separating them completely and transparently.

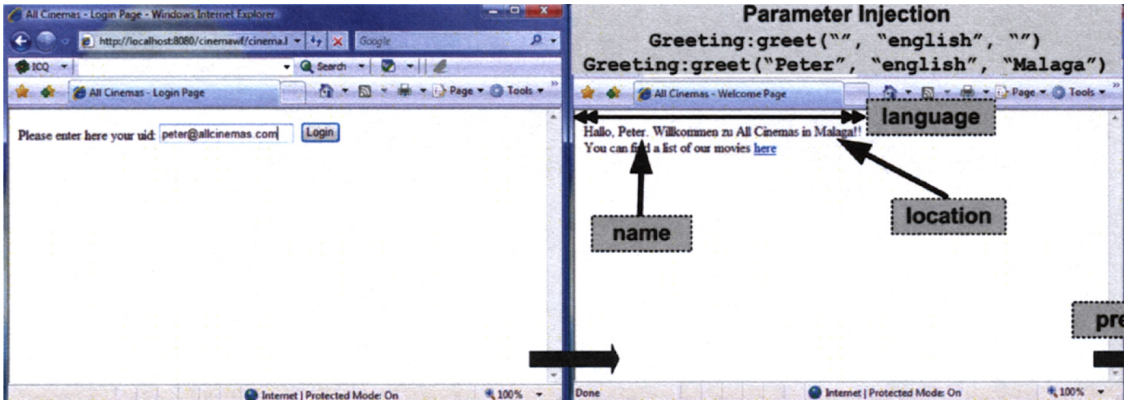


Figure 2.10: Snaphot of the pervasive Web Service cinema example.

Figure 2.10 illustrates part of the cinema example presented by the authors, in which the transparent separation of the context logic and the service might be achievable due to the simplicity and the rather static nature of the context adaptation tasks. The above screenshots indicate that the user is able to login and accordingly the text displayed is adapted according to his name, location and language. In dynamic cases though where

the user-service interaction and the service behaviour need to be adapted in accordance to the context, it is difficult to achieve a clear separation. Consequently, in this thesis work the proposed approach considers as a key aspect, the strong correlation between the context information, the service behaviour and user-service interaction.

Additional adaptation methods are also proposed in the literature [39], [40], which follow an analogous approach to the aforesaid methods by providing middlewares that perform context adaptation tasks to facilitate the development and deployment of context-aware services. These approaches attempt as well to provide a clear separation of the pervasive service functionality from context adaptation, by proposing generic middlewares that support context management tasks.

2.3.3 Infrastructure-based Methods

One of the initial attempts on context-aware service creation was performed by Schmidt et al. [33], who presented an infrastructure-based solution to the problem. The approach proposes a layered-architecture that consists of sensors, cues, contexts and an application layer. At the lowest layer physical and logical sensors exist that are involved with the sensing and gathering of context information in the form of raw data. These raw data are then processed and transformed into abstract context information, which is stored in cues prior to undergoing further processing to form a higher-level context description. In this approach context descriptions are sets of values that describe situations and can be used as a result by pervasive services.

Dey and Abowd [32] proposed an analogous approach that defines an architecture and a Java-based Context toolkit for the benefit of context-aware service creation. In particular, the toolkit provides three architectural components namely widgets, interpreters and

aggregators, which are responsible for the acquisition of context from sensors in the form of raw data. These raw data are then processed to obtain a high-level representation of context information, which can be realised and utilised by pervasive services. In another similar infrastructure-based approach, Chen and Kotz [34] describe a graph abstraction that allows context information to be acquired as raw data. The data obtained are then processed accordingly to obtain a suitable abstract representation, which enables the distribution of context information to services that require it.

The aforementioned initiatives are important since they facilitate low level tasks such as gathering and processing context information from the surrounding environment, which are important to application level approaches. In this thesis, though, the critical assumption is made that this infrastructure-based functionality exists and the focus is principally on context modelling [26]. Hence, an application level approach is proposed that facilitates the representation of information as an advanced context model and supports the generation of the pervasive service implementation for executing context management tasks. In the following section, related research work on context modelling is introduced to identify advantages and possible limitations of the existing techniques.

2.3.4 Context Modelling Techniques

Various techniques [41] have been proposed that utilise ontologies to provide solutions to the requirements of context modelling and reasoning. Wang et al. [42] introduce the CONON modelling approach, which along with the CoBra and SOUPA are considered the most profound ontology-based context modelling techniques [41]. The CONON approach defines a context model that is based on the ontology capabilities of knowledge sharing, logic inferencing and knowledge reuse. In particular an upper ontology is

developed that defines basic context entities and provides the extensibility for defining additional domain-specific ontology concepts in a hierarchical manner. The ontology-based context model tackles issues such as semantic context representation, classification and context reasoning using inference engines. In particular, logic reasoning allows abstract context to be derived from low-level context information and consistency checks to be performed.

The aforementioned research provides many advantages in terms of modelling, reasoning and handling context at a semantic level. However, ontology-based practices provide context models that are increasingly complex and difficult to understand. In particular, ontology-based models are rather unsuitable for human structuring purposes and do not aid the developer in understanding the models and implementing the necessary service functionality. Subsequently, this shortcoming increases the time and effort required by the developer to implement the pervasive service. Moreover, the nature of the models restricts the communication and collaboration between the different roles involved in the design and development of the service.

The strengths of graphically-oriented approaches, such as the one proposed in this work, are certainly on the structure level [26]. Bauer in [43] deals with the identification and modelling of Air Traffic Control context information, which is required by the pilot during the flight. This enables preliminary filtering of required information, precise information support to the pilot and characterisation of constraints and rules [43]. The approach proposes an extension to the UML that allows modelling an information system, combining heterogeneous information sources and providing current context information to the pilot. The Traffic Information System (TIS) is developed as part of the

TALIS[3] project and utilises context information to improve the pilot situational awareness.

The author states that context has two constituent parts: the context relevant information and the actual context. The context relevant information defines details of the domain-specific problem using natural language; i.e. textual description. This information cannot be described using UML diagrams and is useful for the developers, so as to understand and eventually implement the information system. Apart from the detailed description that characterises the contextual situation, different types of UML diagrams are used to define the specific context information. These diagrams may also contain annotations that provide an elementary description of context described in the diagrams and assist the developer in realising the diagrams and implementing the information system.

Model composition is also used in order to avoid the definition of huge and complex UML diagrams that become difficult to comprehend. Hence, the detailed diagrams (e.g. class diagrams, use case diagrams) are abstracted into high-level diagrams, so as to provide a more abstract description of the system. Figure 2.11 [43] illustrates a high-level description of the two contexts that affect the aircraft and describe important information of which both the pilot and the section controller should be aware. The figure describes an entity, in this case the aircraft, which is associated with two pieces of context that can be expressed in the form of attributes.

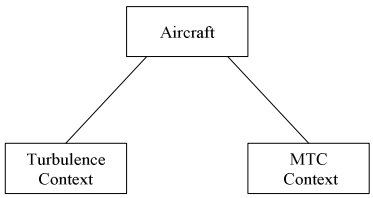


Figure 2.11: Aircraft entity and its associated Turbulence and MTC[4] context.

[3] TotAL Information Sharing for pilot situational awareness

The associated abstract context information can now be described using detailed UML diagrams, such as the one presented in Figure 2.12 for the turbulence context [43]. In the figure the turbulence context is defined using a UML class diagram, which describes primitive pieces of information that are relevant to the turbulence context. It must be noted that these simple pieces of context information defined as attributes of the UML classes, describe details (i.e. readings) which are essential for the pilot navigating the aircraft. The pilot needs to be aware of this context information, so as to know how to react in different circumstances; e.g. strong turbulence.

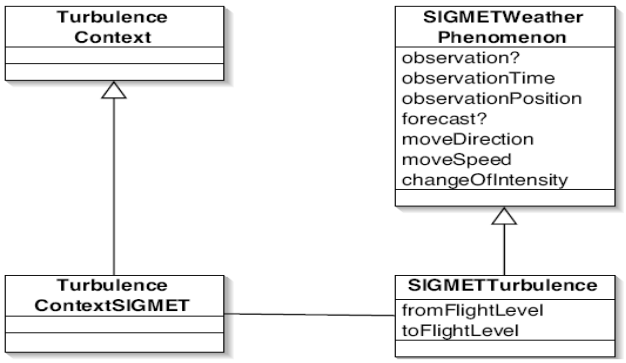


Figure 2.12: Turbulence context information.

The strength of the approach lies in the use of graphical models, defined using the widely accepted UML, which makes it straightforward to represent context information in the form of context models. Moreover, the UML definition of context models makes it easy to comprehend and transform context models to the required implementation. As the author states though the use of the UML provides some constraints in terms of its expressiveness. These constraints are successfully tackled by developing a number of extensions to improve the accuracy of the diagrams. In addition, the context models lack formal basis and consequently they can only be partially validated [26].

[4] Medium Term Conflict

In [44] Simons and Wirtz propose a UML-based Context Modelling Profile (CMP) that allows handling context-awareness and enabling the adaptation of context-aware software applications. The developed UML stereotypes define a domain-specific visual modelling language that provides the capability to model context for mobile distributed systems. More specifically, the models capture context meta-information such as the source, validity and privacy restrictions of context information. In the approach the Object Constraint Language is used to define constraints and enforce well-formedness rules, so as to guarantee the preciseness of context models.

The approach specifies requirements for defining context models in mobile distributed systems, which are fundamental for achieving the objective of context-aware computing. First, the distributed storage requirement specifies that the storage of context information should be achieved by collaboration of distributed network nodes. The second requirement refers to the individual context of the user that must be stored on the mobile device of the user and always be accessible. The third requirement comes as an outcome of the aforesaid requirements since the distributed storage and the individual user context requirements prescribe that a lightweight information exchange scheme is required. This is because of the diversity of network nodes and the limitations of some mobile devices and sensors.

Apart from the technical requirements the approach sets down some characteristics of context, which are imperative for describing information in the form of a context model. These characteristic requirements refer to the validity, quality and privacy of context information. The validity of context can be described in the model using associations, which indicate that information in a context-aware environment are time variant and

subsequently not always valid. Moreover, the different types of sources from which context originates denote the soundness of the information delivered to the context-aware service. Hence, the type of the context source can be regarded as a relative measure of information quality. Finally, information privacy is considered, on the basis of diverse context sources, so as to achieve the acceptance of context-aware services by users.

In order to satisfy the above requirements and define a sound context modelling language the authors utilise the concept of UML profiling. The definition of the UML profile introduces a domain-specific terminology that describes context-awareness. In particular, it provides the capability to customise and introduce additional semantics to the general-purpose UML for the benefit of context-aware application development. As mentioned, OCL constraints are imposed onto the UML profile to restrict the definition of context models and ensure the proper usage of the language.

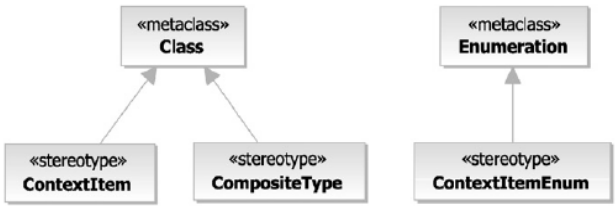


Figure 2.13: UML profile stereotypes for context items.

Figure 2.13 shows examples of the context stereotypes defined for the CMP. The *ContextItem* stereotype extends the abstract *UML Class* and supports the definition of different types of context items. Consequently, instances of the *ContextItem* (meta-) class represent valid context information and the properties of those instances are defined in the form of primitive datatypes; *e.g. String or Integer*. Similarly, the stereotype *CompositeType* extends also the *UML class* and represents actually complex datatypes. Therefore, instances of the *CompositeType* (meta-) class defined in the context model

depict complex properties of context items; *e.g. user profile*. Finally the *ContextItemEnum* stereotype extends the *UML Enumeration metaclass*. This stereotype allows defining a fixed and predefined set of values for a particular context item; *i.e. busy, available*.

The context modelling technique benefits from the use of the UML, which is widely-used throughout the software community. Moreover, the proposed CMP can be deployed and used in many existing UML modelling tools. Despite the clear advantages in using the UML and its associated tools, there are also some deficiencies in terms of the approach. Firstly, the UML tools do not provide a standard way to access model stereotypes in order to enforce constraints and generate the implementation from context models. Hence, OCL constraints are imposed in the approach using the software capabilities of the Eclipse Modelling Framework [45]. Furthermore, the transformation of context models to the corresponding implementation that enforces context-awareness is considered future work. This is mainly because of inadequate access to model steretypes for effective definition of the necessary code generators.

An approach that is highly related to the MDA paradigm is presented by Farias et al. [46], who propose a Meta-Object Facility (MOF) [47] metamodel for the development of pervasive applications. The technique defines a context metamodel using the MOF standard specification that is comprised of concepts restricted to the context modelling domain. The platform-independent nature of the metamodel specification allows both the applications and the supporting infrastructure to share a common understanding of context information. In essence the metamodel defines a context modelling language that facilitates and supports the creation of context-aware applications.

Put simply, the approach considers the diversity of context sources and the different types of context information as the most important challenges faced in such dynamic and highly distributed environments. Moreover, the lack of a formal representation that allows defining context models using a common syntax and semantics is another important aspect considered in this approach. Hence, a MOF context metamodel is defined that provides a common formal representation of the context modelling domain and provides precise abstract syntax for the design of context models. The concepts of the context domain are captured in the MOF metamodel using five different views; *i.e. five MOF packages*. In particular, the *Core View* and the *Service View* are considered as the primary packages that capture the essential aspects of the context domain and support the creation of context-aware applications.

The *Core View* defines the main concepts of the context modelling domain and is closely related to the applications rather than to the supporting platform. It defines basically entities that describe physical or conceptual objects from which context information can be obtained. Context information is defined in the metamodel as an attribute metaclass, which is associated with a content metaclass that defines the value and the temporal property (i.e. timestamp) of the information.

An abstract association metaclass is also used to define the relationships between entities and attributes and the relationships among entities themselves. The association metaclass defines the different types of context sources from which information are acquired. Moreover, an additional metaclass is defined in the metamodel, which allows specifying multiplicity and temporal information related to the context sources. Finally the *Service View* of the context metamodel defines the concepts related to the interoperability

between supporting platforms, applications and context providers. This enables the definition of the standard mechanisms for the interaction between the service platform and pervasive applications.

The approach benefits from the use of the MOF language, which allows defining a metamodel that guarantees precise semantics tailored to the context-awareness domain. In addition the MOF specification of the metamodel defines a precise abstract syntax and facilitates a common representation for the designed context models. This approach tackles the deficiency of the general-purpose UML, whose semantics is too generic to represent effectively a particular application domain (unless UML profiles are used).

The disadvantage of the technique lies in the use of UML tools to accommodate the requirement for a concrete syntax and support the design of context models. A UML-Context-Aware (UML-CW) profile was devised, as in the case of the CMP, to facilitate the definition of context-aware applications. Consequently, the same issues that are valid for the CMP apply also for the UML-CW technique. The problem is that UML tools do not provide a standard way to access model stereotypes, in order to enforce constraints and generate the implementation from the context models. Moreover, a clear one-to-one mapping does not exist between the elements of the UML and the MOF specification.

The most profound graphical modelling approach is defined by Henricksen and Indulska [35], [48], [49], who propose an infrastructure and a context modelling framework for gathering, managing and disseminating context information to services to achieve their adaptation. Principally, the authors have formulated the Context Modelling Language (CML) as a new information modelling technique. This allows concepts of the context

modelling domain to be expressed precisely and flexibly, without any restrictions being imposed.

The diversity of context sources motivated the classification defined by the authors and the definition of the appropriate concepts in the CML. The modelling concepts were then reformed as extensions to Object-Role Modelling (ORM) [50]. ORM refers to a fact-oriented method for defining and querying information models at the conceptual level, where the application is described using concepts that are comprehensible to non-expert users. The ORM method was adopted for performing the extension since it aligns well with the CML.

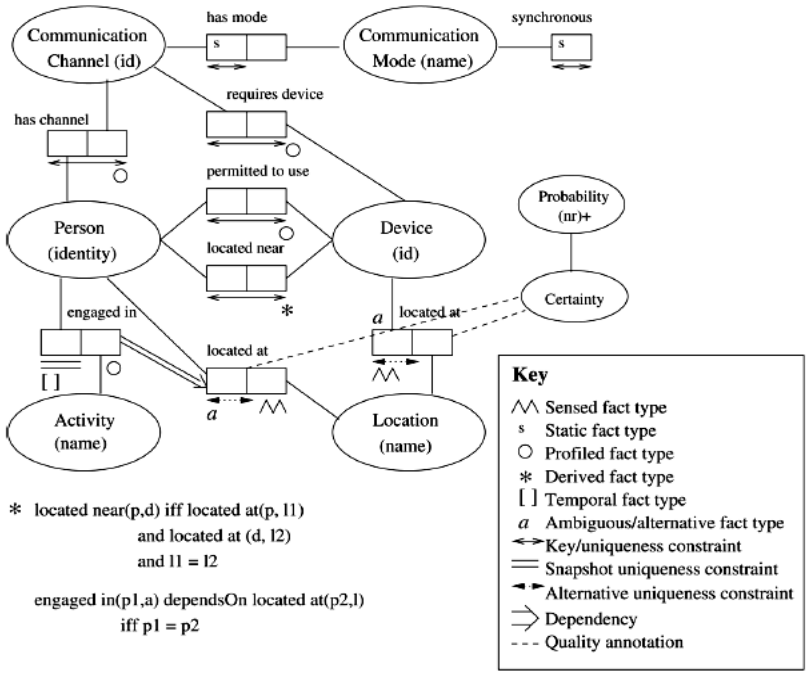


Figure 2.14: An example CML model.

Figure 2.14 presents an example model that introduces the notation used by the CML and can be used for a variety of context-aware communication applications. The model defines different entities such as the user, device and communication channel, which are associated with each other. User activities are also defined as temporal fact types that

cover the past, present and future. The notation explicitly defines the types of context source from which context information are obtained; *e.g. sensed fact type*. In addition to user activities the model defines the location of users and devices and the communication mode of the channel.

Apart from the well-defined CML, the approach utilises a procedure for mapping from ORM to a relational model [50]. The mapping aids the transformation of CML fact types to a relational representation that allows performing context management tasks; *e.g. database storing, querying by applications*. More precisely, the relational mapping leads to a representation of context information that consists of a set of facts expressed in the form of database tuples [35]. The approach also defines a situation abstraction that serves as a natural programming abstraction and is required in order to describe the conditions that govern the pervasive service behaviour. Using the abstraction, situations can be combined allowing reuse and enabling the programmer to define incrementally complex situations. The situation abstraction uses predicate logic and is conceptually similar to but more expressive than the one proposed by Dey and Abowd [51].

Finally, the approach introduces a preference model, two programming models and a software infrastructure to provide support for the implementation of context-aware applications. The preference model attempts to provide solutions to the decision-making process since in context-aware applications the requirements are different for each person. Hence, preferences are defined that comprise a scope and a scoring expression, where the scope defines the preference context and the scoring expression determines if the preference is applicable within the given context.

The branching and triggering programming models are also defined in order to aid the developers and limit the complexity and effort involved in implementing context-aware applications. In order to accommodate low-level and high-level tasks the proposed approach introduces also a layered software infrastructure that integrates a programming toolkit and provides support for various context-related tasks. The software infrastructure comprises the following six layers: context gathering, context reception, context management, query, adaptation and application.

The context modelling technique provides a set of modelling techniques and a set of developed software tools that comprise an overall framework for the benefit of pervasive service creation. In particular, context modelling techniques facilitate the definition of context models and support the generation of the necessary implementation for executing context-aware application tasks; *i.e. context management tasks*.

Despite the benefits, the lack of a widely-accepted representation (e.g. UML, MOF) for the definition of context modelling concepts limits the understanding of the models by designers and developers that are not familiar with ORM. Moreover, the approach defines various models and abstractions with which the developer requires to familiarise in order to apply the technique in practice. An additional predicament is the necessity to develop manually the modelling and implementation tools that formulate the context-aware SCE. Furthermore, the approach is hindered by the absence of a context modelling editor, which is essential for a graphical modelling approach and the low degree of automation provided for the generation of context-aware applications.

The aforesaid and other related approaches [52], [53], [54] tackle the context-awareness characteristic of pervasive services and provide modelling techniques that allow

expressing this requirement. Aside from context-awareness, additional characteristics exist though that must be captured in order to provide an unambiguous pervasive service specification. This thesis proposes the definition of two complementary modelling languages and the generation of their supporting modelling frameworks, which allow defining the dynamic behaviour and designing the graphical user interfaces associated to the pervasive service. Firstly, the definition of the dynamic behaviour is essential since it describes the service execution logic and facilitates the generation of its implementation. Secondly, GUI models represent the user interfaces with which the user is able to interact seamlessly with the pervasive service. Accordingly, from the designed GUI models the corresponding implementation is successfully generated.

Consequently, a conceptual framework is proposed that defines a MDD methodology and provides a supporting MDA-based generic environment to facilitate service creation. The framework should support the generation of the necessary modelling components (e.g. context modelling, presentation modelling), which can be integrated into the generic environment to compose the overall domain-specific SCE; i.e. the PSCE. The generated PSCE should support as a result the development process, so as to simplify and expedite the analysis, design, validation and implementation of pervasive services.

Furthermore, in this thesis the validation phase is applied prior to the implementation of the pervasive service. Therefore, any inconsistencies are detected and rectified at the design level and subsequently the service code is automatically generated from consistent design specifications. In contrast to the aforementioned modelling techniques, the approach proposed in this thesis utilises two complementary validation techniques in order to validate both the static and the dynamic structure of context models. Firstly, the

use of the OCL language provides the capability to enforce constraints and validate the static structure of the defined pervasive service models. Secondly, the Petri Net formalism allows describing the pervasive service behaviour using a Petri Net model and validating the dynamic behaviour via model execution. The following section introduces related work that utilises Petri Nets theory for the validation of service specifications.

2.4 Related Work on Petri Nets

Several research efforts have been devoted to the formulation of a systematic service creation methodology. In an example of the conventional approach, Adamopoulos et al. [10] described service creation as the most abstract and important stage of the service life-cycle. These kinds of approaches, although well-formed, defer validation after the implementation phase where technology-specific complexities are introduced and they do not consider the requirement to generate different service implementations from the design specifications. The proposed approach follows a different research direction where validation is imposed at the modelling level prior to generating the implementation.

This research direction prescribes that it is imperative to apply a formal technique to analyse and validate the operational semantics of the pervasive service models. The formal method aims to complement the powerful MDA-based analysis capabilities (i.e. OCL validation), which facilitate the validation of the static structure and syntax of models. In particular, the method addresses the dynamic semantic analysis of service models, that is, the execution and reachability analysis [55]. Note, though, that the research work introduced next refers exclusively to service composition [56] rather than service creation; the challenge is to disclose the importance of applying a formal dynamic validation method prior to the service implementation.

The necessity to apply mathematical analysis techniques directly onto composite Web Service models was expressed by Nait-Sidi-Noh et al. in [57]. In their approach a UML extension is used, originally defined in [58], which allows modelling Web Services as well as their interactions. Since the extension is defined using the UML, which is a semi-formal specification language, it does not permit verification, validation and performance analysis of the UML for Services (UML-S) models. Hence, the authors state the necessity to transform UML-S models into high-level Petri Nets in an attempt to formalise UML. This allows effective translation of the composite Web Service models into Propositional Petri Nets (PPN) to perform tasks such as verification, validation and performance evaluation.

In a similar attempt Yang et al. [59] propose the use of the well-known Business Process Execution Language (BPEL) to describe the composition of Web Services in the form of business processes. In particular, BPEL allows the logic of a composite Web Service to be expressed using an abstract process specification language but lacks formal and sound semantics. Therefore, a transformation mapping is defined in order to enable the translation of BPEL composite Web Services into an extension of Petri nets, namely Coloured Petri Nets (CPNs). This allows Web Services to be represented in the form of CPNs that provide a formal model, which facilitates the analysis and verification of the BPEL specifications using mathematical techniques and a variety of CPN software tools. Hence, Petri Nets provide once again the apposite formalism that allows verifying and validating the dynamic Web Service composition.

The use of the BPEL language for specifying Web Service compositions is also proposed by Dun et al. [60], [61], whose approach utilises the BPEL specification language to

describe the composition of Web Services in the form of business processes. The authors state that BPEL, being an XML based language, can experience inconsistency, ambiguity and incompleteness when specifying composite Web Services. Consequently, a formal tool is required to analyse and verify business processes specified in BPEL. The approach proposes the formalisation and analysis of BPEL Web Services using Synchronisation nets, an extension of Petri Nets. Transformation rules are defined for translating BPEL business processes into Synchronisation Nets, with the help of which the correctness of composite Web Services can be analysed and verified.

Andonoff et al. [62], in their approach, utilise the Petri Net with Objects (PNO) formalism to model Workflow Web Services (W2S). The PNO formalism combines the Petri Nets theory with the Object-Oriented (OO) approach. While PNs are highly suitable for modelling the dynamic behaviour of a Web Service, the OO approach expresses the flow of active and passive entities (i.e. objects) in the Web Service net. Subsequently, from the definition of W2S models the approach facilitates the automatic generation of OWL-S specifications, which provide a web-based accessible format that enables W2S publication.

The authors state initially that although the OWL-S can be used to model W2S it has two main drawbacks: (i) the absence of a graphical modelling tool that allows unambiguous definition of Web Services and (ii) the lack of formal operational semantics that permit the analysis, simulation and validation of Web Services. Consequently, the approach compensates for these disadvantages by utilising a formal, graphical modelling technique and providing the transformation rules that guide the translation of PNO Web Services into OWL-S specifications.

Although the aforementioned approaches introduce service validation straight after the design phase, the necessity arises to transform miscellaneous specifications (i.e. BPEL) to PN extensions, in order to validate the service designs. Even in cases where the service is defined directly using a PN formalism (e.g. PNO) the requirement to generate diverse implementations is not satisfied. This thesis proposes for the first time, to the best of the author's knowledge, the integration of the MDA paradigm with the Petri Net formalism. This provides a suitable modelling language and automatically generates the supporting modelling framework, which in turn allows the dynamic behaviour of a pervasive service to be defined as a PN model. Therefore, the capability is provided to utilise the PN formalism to model the service process and analyse, simulate and validate its operational semantics. The combination of MDA and PNs enables the abstract definition of rigorous service design specifications and facilitates the validation and generation of diverse implementations from the service designs.

2.5 Summary

The disclipline of service engineering imposes heavy challenges on analysts, architects, designers and developers. Service creation is a key stage for the viability of the service engineering lifecycle since nowadays the fast changing and demanding user requirements necessitate creating and deploying services rapidly and effectively. In this chapter service creation enabling methods are presented that attempt to utilise high-level SCEs to create services efficiently. The review of related work reveals a trend towards an abstract, platform-independent approach. This thesis examines explicitly the domain of pervasive services in order to provide an efficient model-driven approach for pervasive service creation. Existing research efforts on pervasive service creation and Petri Nets have been

also introduced in this chapter in order to classify and exemplify the importance of the proposed approach.

Chapter 3 A Conceptual Framework for Pervasive Service Creation

In this chapter a conceptual framework is presented that supports the creation of domain-specific (i.e. pervasive) services. Foremost, the framework defines a well-formulated MDD methodology that maps to and facilitates the phases of service creation. The methodology is supported by a generic environment that is designed and implemented using Eclipse-based implementations of the MDA standards. The applicability of the framework is demonstrated via the generation of an example domain-specific SCE that supports the creation of online services for contacting surveys. Concluding, on the basis of the conceptual framework the architecture of the Pervasive Service Creation Environment is devised and the proposed Model-driven Petri Net based process for pervasive service creation is presented.

3.1 Motivation and Approach

The creation of advanced and customised services is an extremely complicated task that necessitates a well-defined framework. Especially when dealing with pervasive services the complexity of the service creation process is drastically augmented. This is mainly because of the dynamic nature of pervasive services and the heterogeneity of information sources, which increase the complexity of the conventional implementation phase. Hence, a platform independent service creation methodology is required in order to enable the efficient and rapid creation of pervasive services at the static compile time.

Furthermore, the current literature has yet shown the presence of an up and running model-driven software environment that smoothly integrates the phases composing the service creation process. Current state-of-the-art model-driven environments envision and attempt to realize model-driven development but they still lack in terms of essential MDD requirements. In this work a coherent list of MDD requirements was defined and a comparison of existing environments was contacted so as to identify their conformance to those requirements. The comparative study exposed the lack of support for the complete set of requirements and subsequently for the proposed MDD methodology.

Consequently, in order to tackle the aforementioned complexity issues and provide full support to the requirements a conceptual framework is proposed, which comprises of an abstract MDD methodology and a generic model-driven environment. The basis of the framework is the extensible architecture of the generic environment that allows generating automatically domain-specific modelling frameworks and integrating the frameworks into a domain-specific SCE. Hence, the generated SCE can effectively support the model-driven development of domain-specific services. The main objective

of this thesis is to utilise the conceptual framework to define a Model-Driven Petri Net based Framework, which supports the creation of pervasive services. Note that, the conceptual framework can be applied across different domains to generate the appropriate SCE that supports the domain-specific service creation process; i.e. web service creation.

The following section introduces the combination of model-driven development with domain-specific modelling, which allows formulating a domain-specific model-driven development methodology. The methodology is defined in accordance to key MDD requirements derived and proposed in this work, which facilitate its application in practice. Subsequently, a comparative study of existing MDD environments is performed to determine their compliance to those requirements. The conceptual framework is then introduced and an example domain-specific SCE is developed to demonstrate the applicability of the framework. Finally, the Model-Driven Petri Net based Framework that supports pervasive service creation is also introduced.

3.2 Domain-specific Model-Driven Development

The concept of Model-Driven Development kicked off essentially with the use of the Unified Modelling Language (UML) [63], which defines a standardised *General Purpose Language (GPL)* used for the development of software systems. Basically the UML is a semi-formal specification that provides a graphical notation for the definition of abstract models of software systems. More specifically, the UML introduces different diagrammatic conventions, such as class diagrams, activity diagrams and sequence diagrams, which facilitate the definition of the static structure and the dynamic behaviour of a software system.

Although the UML 2.0 specification [64], [65] is widely-accepted throughout the industry and the academia, its general-purpose nature comprises of two main issues. Primarily, the UML is largely complex [66] and subsequently best suited to describe different problem domains at a higher-abstraction level [28], [29]. The second issue comes as an outcome of the first, since the UML aims to be applied across different problem domains and consequently the information described using the language are of limited precision; in terms of a specific domain [28].

In contrast to general purpose languages such as the UML, problem-specific modelling languages exist that are considered more powerful when addressing a particular domain. These kind of modelling languages are conventionally called *Domain Specific Languages (DSLs)* [66], [28], [67] and generally capture information with a higher degree of precision. Consequently, the semantics of these languages can be interpreted precisely to platform-specific code, as they leave no room for miscellaneous interpretations [68], [69]. The process undertaken to develop and utilise graphical DSLs is referred in the literature as *Domain Specific Modelling (DSM)* [68], [70]. Domain-specific modelling denotes in particular a software engineering paradigm that raises the level of abstraction by introducing models as the prime entities of the software development process [66], [71], [72]. The models are defined using the domain-specificmodelling language, which includes the artefacts and the graphical notation applicable to the specific problem domain. In most cases domain-specific modelling is highly interlinked with the concept of code generation, which describes the automatic production of executable code directly from models.

Domain-specific modelling differs though from the approaches proposed by early Computer-Aided Software Engineering (CASE) tools or UML tools, where the modelling language and the tooling were developed by software vendors. This imposed restrictions since the modelling language was difficult to master and the generated code did not serve explicitly the needs of the developers. In contrast domain-specific modelling defines that both the modelling language and the tooling are built from designers within the organisation. Consequently, the modelling language becomes easily comprehensible since it uses concepts that designers of the organisation are acquainted with. Furthermore, code generation is tailored to the requirements of the problem domain tackled within the organisation and subsequently can be easily realised and implemented by developers.

Despite the aforementioned benefits, the major problem with domain-specific modelling is the difficulty faced with the design, development and application of DSLs [28], [73]. The definition of a modelling language and the implementation of its supporting domain-specific modelling framework from scratch involve a time consuming and costly process. Consequently, an efficient and systematic approach is required that facilitates the definition of the modelling language using a common language representation and the generation of its supporting modelling framework.

The Object Management Group proposes two key foundations for its Model Driven Architecture, namely, the UML and the Meta-Object Facility specification. In this work the MOF specification, and not the UML, is considered since it can effectively drive a domain-specific modelling approach for the development of software services. The MDA paradigm relies on the MOF specification, which is a DSM language designed for the definition of domain-specific modelling languages. In particular the MOF specification is

considered as the meta-metalanguage that supports the definition of the abstract syntax of the modelling language in the form of a metamodel.

Metamodelling [74] is the process that guides the definition of a metamodel, which describes the elements of a certain domain with their corresponding properties and relationships. Hence, the metamodel is placed one abstraction layer higher than the domain models designed on the basis of its definition. In this way, an indefinite hierarchy of abstraction layers can be built, where models at layer *n* are specified using the precise semantics of the modelling language defined as a metamodel at layer *n+1*. Consistent with this setting, models situated at layer n are instances of metamodels at layer n+1.

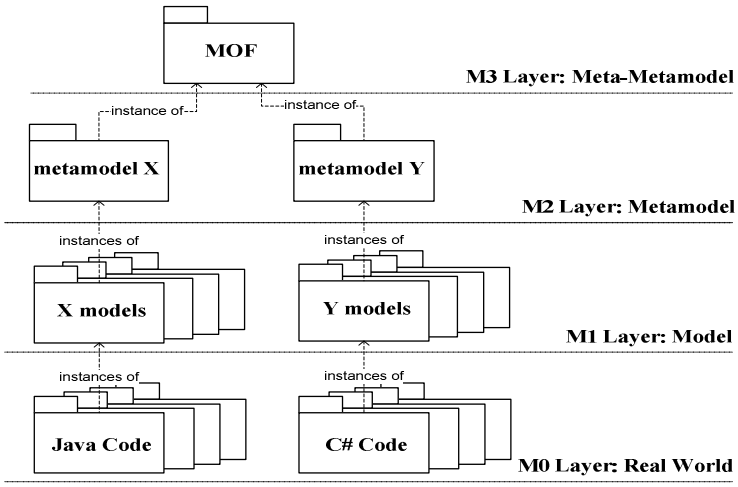


Figure 3.1: MDA four-layer metadata architecture.

MDA provides such a layered architecture limiting the number of abstraction layers to four as illustrated in Figure 3.1. At the top level, M3, the meta-metamodel of the MOF specification is situated. Layer M2 is populated by metamodels that represent MOF-based domain-specific languages. Moreover, Layer M1 hosts domain models written in M2-defined DSLs. These models can be either *Platform Independent Models (PIMs)* or *Platform Specific Models (PSMs)*. In essence, PIMs comprise the abstract information necessary in order to effectively represent software applications separately from the

technology specific code that implements them. On the contrary, *Platform Specific Models (PSMs)* include implementation specific details and consequently describe a software system with full knowledge of the implementation platform [12].

PIMs can be effectively translated to PSMs by means of automatic model-to-model transformation [75]. The transformation is essentially driven by a mapping definition, which describes a set of transformation rules. These rules define in specific how each artefact of the input source model (i.e. PIM) can be transformed to an artefact of the output source model (i.e. PSM). Subsequently, the PSMs can be transformed to runtime domain objects (e.g. Java code), which are considered as instances of M1 domain entities and reside at layer M0 of the architecture. Note also that PIMs can be transformed directly to the necessary implementation, by hard-coding platform specific details within the code generator.

Consequently, the integration of the MDA paradigm with the DSM paradigm defines conceptually the proper guidelines to apply model-driven development. Nevertheless, the MDA paradigm does not specify a well-defined methodology for applying MDD for the benefit of service creation. More importantly though it does not contribute any MDD tools that enable its application in practice. The significance of software tools for enabling such an abstract model-driven approach is of outmost importance and cannot be omitted [66], [76], [77]. Therefore, a systematic MDD methodology is required, which should be steered by model-driven development tools that conform essentially to the MDA specifications; e.g. MOF, OCL, Query/View/Transform (QVT).

In this thesis a conceptual framework is proposed that compises a systematic MDD methodology and a generic model-driven environment for service creation. Basically, the

generic environment and the methodology can be applied accordingly to facilitate the creation of services of any particular domain. In this thesis the main objective though is to utilise the generic environment and apply the MDD methodology for the definition of a Model-Driven Petri Net based Framework, which supports the creation of pervasive services. The following section proposes a set of essential requirements for applying MDD and presents a comparative study of existing metamodelling environments that attempt to support MDD.

3.3 Requirement Analysis for Model-Driven Development

3.3.1 Formulation of Requirements for Model-Driven Development

In order to render practical the use of a domain-specific MDD methodology the significance of model-driven tools should be taken into consideration. As aforementioned the MDA standards do not provide any implementations to support the paradigm in practice. This predicament drives the identification and definition of the necessary requirements that steer effectively the composition of the generic environment and the definition of the abstract MDD methodology. These principles are shaped into a set of concrete requirements, which are explained in detail as follows:

[R1] *Abstract syntax:* The primary requirement is the utilisation of a high-level meta-metalanguage that provides the capability to define the concepts of a domain-specific modelling language in the form of an abstract syntax. In particular the abstract syntax must be preferably defined using a graphical notation so as to improve the understanding of the modelling language [78]. Consequently, every modelling language should be specified as a metamodel with aid of a meta-metalanguage that fundamentally conforms to the MOF specification.

[R2] *Metamodel level constraints:* Apart from the definition of the abstract concepts of the language, the imposition of metamodel level constraints onto the metamodel is of prime importance. This is because solely the abstract syntax definition allows designing erroneous domain models [68]. Therefore, the capability to impose the necessary rules that enforce the proper usage of the language should be effectively provided. A constraint language that conforms to the formal OCL specification is essentially required to satisfy this requirement.

[R3] *Concrete syntax:* Alongside the abstract syntax of the language, a concrete syntax definition is essential in order to represent the abstract concepts using a model design in a consistent manner. Consequently, the abstract elements, relationships and properties of the language should be mapped accordingly to a corresponding graphical notation. The visual representation enables better understanding of the modelling language and subsequently simplifies the design of domain models.

[R4] *Domain-specific modelling framework generation:* The main predicaments that hinder model-driven development approaches are the excessive time and costs introduced when defining, implementing and maintaining a DSL [29], [67], [73]. In particular the development and maintenance of the modelling framework that supports the DSL denotes a time consuming process that increases the DSL development overheads [29]. Therefore, the goal is to satisfy this requirement by mapping the abstract and concrete syntax of the modelling language into a unified representation that will eventually drive the automatic generation of the domain-specific modelling framework.

[R5] *Model level constraints:* In addition to the specification of metamodel-level constraints the capability to impose model-level constraints should be also provided. The

difference between metamodel and model level constraints is that the former denote well-formedness rules of the metamodel, while the latter restrict the values assigned to domain models. For instance a metamodel-level constraint defines that an attribute of a specific metaclass should be of datatype *Integer; i.e. self.balance.oclIsTypeOf(Integer)*. On the contrary a model-level constraint defines that the account balance of a customer must not be negative; i.e. *self.balance > 0*. Consequently, model-level constraints restrict essentially the service implementation.

[R6] *Model Transformations:* The definition of model-to-model transformations is an important capability of the generic environment and the MDD methodology. It facilitates primarily the code generation phase by transforming PIMs to PSMs. Moreover it supports the transformation of PIMs that conform to a source metamodel to other PIMs that conform to a target metamodel. For instance the transformation of an Entity-Relationship (E/R) model to a Relational model is an example of a PIM-to-PIM transformation. Finally the maintenance (or evolution) of the language can be effectively accomplished by transforming outdated PIMs, designed using a previous version of the language, to new PIMs that conform to the latest version of the language.

[R7] *Text-based generation:* Text-based generation refers to the final requirement that is directly related to the software capabilities of the generic environment. In essence it refers to model-to-code generation, which is the principal objective of MDD since it aids the rapid development of software applications. Furthermore, the capability to transform domain models to miscellaneous text-based representations (e.g. XML, SCML) should be gratified. Therefore, a generic model-driven environment should comprise the necessary software component(s) so as to support this core requirement.

[R8] *Standards Conformance:* The alignment to the OMG MDA standards (e.g. MOF, XMI, QVT and OCL) provides clear-cut benefits since it facilitates the definition of a conceptual framework that follows strictly the guidelines of the well-established MDA paradigm. Consequently, the software components of a generic environment should encompass non-proprietary implementations of the standards proposed by the paradigm.

[R9] *Accelerated adoption:* Both the generic environment and the generated domain-specific modelling frameworks should be fairly easy to use by designers and developers. In this thesis the accelerated adoption requirement depicts the utilisation of a multi-language and widely-used software development platform, which provides the capability to operate expediently both at the modelling and implementation level.

The aforementioned requirements, also grounded on previous work [79] and MDA case-studies [80], [81], describe general principles that depict from a conceptual perspective a certain flow of steps that allows to define a coherent domain-specific MDD methodology. Moreover, from a practical perspective the above principles drive the composition of the generic model-driven environment that aims to support the methodology. Note that, the generic environment forms the backbone of the developed PSCE. The following subsection introduces a wide-ranging comparative study of model-driven development environments. This comparison is performed in order to assess the conformance of the environments to the MDD requirements proposed in this work.

3.3.2 Comparative Study of Model-Driven Environments

In this subsection, existing model-driven environments are cautiously examined so as to determine any deficiencies that hinder the application of the methodology. Table 3.1 introduces and evaluates these environments as per their ability to best meet the proposed

requirements. Hence, the assessment allows determining the practical aspects that lack essential support. The prime factor considered is essentially the compliance of these environments to the OMG standards. Essentially, the comparative study reveals the necessity to formulate an appropriate MDD environment that supports the complete set of requirements for applying the methodology for the benefit of service creation.

Table 3.1: Metamodelling environments requirements conformance.

	MDD Requirements	***GME***	***DOME***	***AndroMDA***	***XMF-Mosaic***	***MetaEdit+***	***Microsoft DSL Tools***	***Borland Together***
R1	*Abstract syntax*	Pr (√)	Pr (√)	NPr (√)	NPr (√)	Pr (√)	Pr (√)	NPr (√)
R2	*Metamodel level constraints*	NPr (√)	Pr (√)	NPr (√)	NPr (√)	Pr (P)	Pr (P)	NPr (√)
R3	*Concrete syntax*	(√)	Pr (P)	(×)	(√)	(√)	(√)	(√)
R4	*DSM framework generation*	(√)	(√)	(×)	(√)	(√)	(√)	(√)
R5	*Model level constraints*	NPr	(×)	(×)	NPr (√)	Pr (P)	Pr (P)	NPr (√)
R6	*Model transformations*	(×)	(×)	Pr (P)	NPr (√)	Pr (P)	Pr (P)	NPr (√)
R7	*Text-based generation*	Pr (P)	Pr (√)	Pr (√)	NPr (√)	Pr (√)	Pr (P)	NPr (√)
R8	*MOF compliant*	(×)	(×)	(√)	(√)	(×)	(×)	(√)
	XMI compliant	(×)	(×)	(√)	(×)	(×)	(×)	(√)
	QVT compliant	(×)	(×)	(×)	(√)	(×)	(×)	(√)
	OCL compliant	(√)	(×)	(√)	(√)	(×)	(×)	(√)
R9	*Accelerated adoption*	(×)	(×)	(×)	(√)	(×)	(√)	(√)

Pr (√): Proprietary-Full Support Pr (P): Proprietary-Partial Support NPr (√): Non-Proprietary-Full Support
NPr (P): Non-Proprietary-Partial Support (√) : Full Support (×): No Support

Table 3.1 presents the environments used in this survey, which are considered the most dominant ones for contacting the survey due to the fact that they facilitate MDD, with extensive support for DSM [82]. Related work [82], [83] acknowledges that DSM in conjunction with the MDA paradigm increase significantly software productivity. The major predicament expressed though by the aforementioned research efforts is the necessity to develop rapidly and maintain efficiently the modelling frameworks of the

DSLs. Currently the model-driven environments have improved significantly and provide many of the required software capabilities that aid in resolving the above problem.

Initially, two research model-driven environments are examined, namely GME and DOME, which address merely half of the proposed requirements. Moreover, the prerequisites satisfied by the research environments introduce proprietary software solutions to MDD. For instance, the definition of the abstract syntax is performed using proprietary languages, such as the GME's UML-based MetaGME and the DOME tool specification language. These metamodelling languages are proprietary since they do not conform to the MOF specification.

Furthermore, the imperative model-to-model transformations requirement is not gratified by the environments. Also the accelerated adoption aspect is not accounted since both metamodelling environments are not deployed onto a well-established development platform. Consequently, designers and developers require to become acquainted with the environment in order to effectively utilise and adopt its software capabilities. Essentially, this prerequisite along with the common interest on metamodelling motivated the GME research group to bridge with the Eclipse community into a joined modelling initiative.

In contrast to the above research projects, AndroMDA is an open source, extensible environment that adheres to most of the MDA specifications. AndroMDA provides software capabilities that satisfy half of the proposed MDD requirements. The key deficit is the absence of the necessary software capabilities that facilitate the generation of DSM frameworks. Furthermore, support for model-level constraint validation is not provided and model-to-model transformations are defined using Java, which is not desirable for a platform-independent approach. Finally, the manual configuration of the building blocks

of the environment can be tedious and troublesome. Consequently, an Eclipse-based implementation (i.e. plug-in) of AndroMDA was developed at a later stage to satisfy the accelerated adoption requirement and avoid configuration complexities.

The XMF-Mosaic started initially as a commercial product but ended up in the process as a freely available distributable under the Eclipse project licence. It is a metamodelling environment that satisfies the majority of the requirements of MDD; aside from XMI-compliance. Moreover, the implementations of the model-driven software capabilities of the environment are aligned with the specifications defined by the OMG. Although the XMF-Mosaic is a powerful environment built on the Eclipse platform, its development was terminated and the source code was disclosed through the web. In the latest version, the tool interoperates closely with the software capabilities of the Eclipse modelling projects. This is basically due to the wide-acceptance and utilisation of these projects by the larger modelling community. Moreover, the consideration is for the environment to become part of the Eclipse Generative Modelling Technologies (GMT) project, which sole scope is to produce a set of prototypes in the area of Model Driven Engineering.

Furthermore, Table 3.1 presents three commercial products that are not freely available and subsequently a corresponding licence is required in order to use them for modelling and development. Primarily the MetaEdit+ environment introduces software solutions to a large set of the requirements. However, the environment induces strong proprietary support and does not conform as a result to the MDA standards. For instance, metamodel level constraints are defined in the form of plain data defined using tabular editors and imposed onto the metamodel. In addition, accelerated adoption is not gratified, since the environment is not deployed onto a widely-used development platform.

Microsoft DSL Tools is a very powerful model-driven environment that supports MDD with specific focus on domain-specific modelling. The DSL Tools software factory comprises essentially of a bundle of proprietary tools, which are developed on the basis of the Visual Studio development platform. Apart from the non-conformance to the OMG specifications, MS/DSL Tools lack also in terms of an explicit technique for supporting model-to-model transformations. Merely, partial proprietary support is provided for model transformations and code generation with the use of the T4 template-based toolkit. Microsoft Corporation recently joined the OMG in an attempt to meet the standards defined by the group so as to fulfil their strategy and assist in taking modelling into mainstream industry use.

Borland Together is a commercial visual metamodelling environment that facilitates modelling using domain-specific languages and drives the implementation of software applications. The environment is composed by modelling projects that comply with the specifications proposed by the OMG and satisfy the entire set of requirements defined in this work. The software capabilities integrated into the overall environment are actually Eclipse modelling projects, which have been tailored and supplied to designers and developers with a user-friendly frontend. The sole predicament with Borland Together is the fact that being a commercial product does not provide the capability to use it freely and does not allow extending the environment in accordance to the users' needs.

The majority of the modelling projects included in Borland Together are equivalent to the ones assembled for constructing the generic environment proposed in the context of this thesis. To the author's best knowledge when the generic environment was proposed the existing literature and documentation on Borland Together [84] did not disclose such

software capabilities. This does not abolish the fact that analogous attempts were made by Borland during that period to develop and deliver a unified model-driven environment with analogous software capabilities.

The comparative study revealed the deficiencies that the existing environments impose in terms of applying effectively the MDA paradigm with strong DSM support. This lead to the formulation of the conceptual framework, which provides an abstract MDD methodology and a generic environment that facilitate domain-specific model-driven development. The key objectives of the framework are to provide the capability to automatically generate domain-specific service creation environments and support the phases composing the service creation process. Consequently, the framework is utilised accordingly to gratify the key objective, which is the definition of a suitable MDPNF that supports the creation of pervasive services.

3.4 The Proposed Conceptual Framework

The conceptual framework is defined on the basis of the formulated requirements and due to the significance of software tools it proposes a MDD methodology that is rigorously associated to the architecture and the software capabilities of its supporting model-driven environment. The following section begins by describing the architecture of the generic environment, which captures the motivation behind the formulation of the conceptual framework. Subsequently, the methodology that supports the phases of service creation is presented and the capabilities of the developed environment are effectively introduced.

3.4.1 Generic Model-Driven Environment Architecture

The architectural design of the environment takes into consideration the imperative requirement, which is the necessity to develop modelling frameworks for each DSL with

the minimal time, effort and cost. In principle the generic environment comprises core software components (i.e. modelling frameworks) that facilitate the generation of offspring domain-specific modelling frameworks. These offspring modelling frameworks are subsequently integrated into the generic environment, formulating as a result the overall domain-specific service creation environment.

The aforementioned concept is realised on the basis of the architecture of the Eclipse platform. Eclipse is a software development platform designed for building Integrated Development Environments (IDEs), and arbitrary tools [85]. This is possible due to the plug-in architecture of the platform, which provides the capability to integrate software components that are developed in the form of Eclipse plug-ins. In addition the open-source nature of the Eclipse platform allows extending or modifying as required the developed IDE, in order to best suit the designers and developers requirements.

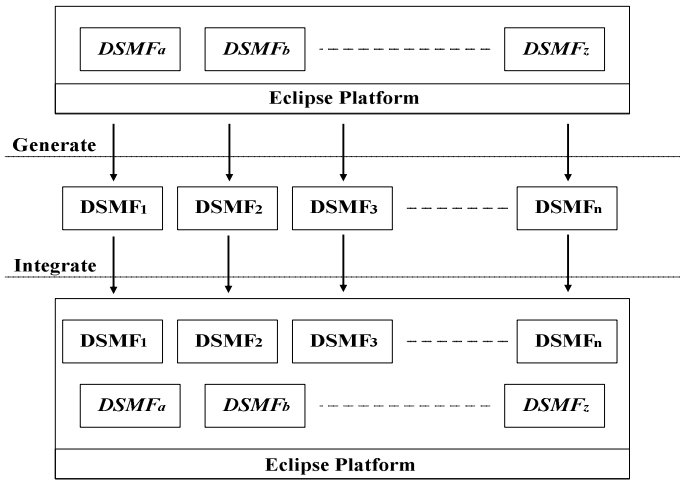


Figure 3.2: Generic model-driven environment architecture.

Figure 3.2 illustrates the architecture of the environment that steers the development of SCEs and supports the service creation process. The environment is composed essentially by core components, which are referred also as domain-specific modelling frameworks (DSMFs). These DSMFs illustrated at the top of the architecture are integrated onto the

Eclipse platform in the form of software plug-ins. Basically, the core components should comprise of the necessary software capabilities, in order to facilitate the generation of one or more secondary DSMFs. Moreover, the core components should support fundamental model-driven capabilities, such as model-to-model transformation and model-to-code generation.

Consequently, the integration of the generated DSMFs with the core components composes the domain-specific SCE as illustrated at the base of Figure 3.2. This assembled environment contains essentially the entire set of model-driven capabilities required to support the development of services of a particular domain. In the context of this thesis the PSCE is developed on the basis of the aforementioned generalised model-driven environment architecture. Apart from the necessity to fabricate such a dynamic architecture, it is also of outmost importance to define a systematic methodology to steer the developers and guide effectively the domain-specific service creation process.

3.4.2 Domain-Specific Model-Driven Development Methodology

In this subsection the MDD methodology is introduced that facilitates the generation of SCEs and supports domain-specific service creation. The methodology is based on the previously formulated MDD requirements and the specified generic architecture of the environment. Table 3.2 introduces the mapping between the proposed methodology and the phases of the service creation process. The mapping reveals how the methodology can successfully realize each phase of the service creation process starting from service analysis to service implementation. Basically, the aim is to provide an agile high-level development process that reduces the time, effort and the costs of the service creation process.

Table 3.2: Mapping the methodology to the service creation process.

Model-driven development methodology	***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

The service analysis phase aggregates the service concepts and their associated relationships, in accordance to the requirements of a services domain [10]. In terms of the methodology the domain-specific language definition phase reflects precisely the service analysis phase. Principally, the language definition involves the identification of the different domains (e.g. context modelling, process modelling), in order to effectively represent the functionality of the services. Moreover, it is necessary to define the structure of the generated service implementation. This is performed so as to steer accordingly the definition of the modelling language and the subsequent development of the code generators that aim to produce the implementation from the service specification. Consequently, from the language definition the corresponding DSMF is generated and integrated into a unified service creation environment that guides the subsequent phases of the methodology.

Figure 3.3 illustrates the *Steps 1-4* required for the definition of each DSL and the generation of its respective DSMF. Initially, the semantics of the domain are identified and mapped to the abstract syntax of the language *(Step 1)*. In essence the abstract syntax is defined in the form of a metamodel, which comprises the elements, the associations and the properties of each individual modelling domain. Furthermore, well-formedness rules are identified for the domain, mapped to constraints and imposed onto the language definition *(Step 2)*. Although constraints are most commonly defined explicitly and

separately from the metamodel definition, some rules are implicitly stated in the definition. This denotes that the definition of the metamodel could capture implicit rules that inherently restrict the abstract syntax of the modelling language.

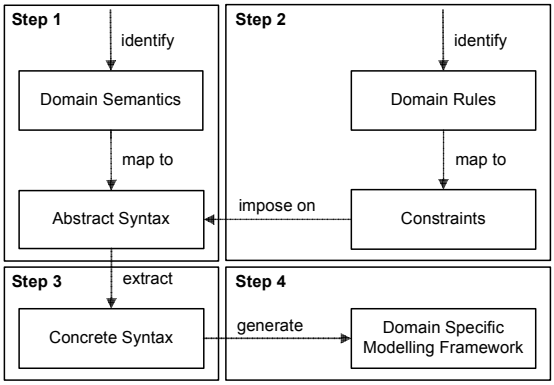


Figure 3.3: Domain-specific language definition; service analysis.

The metamodel definition and the successful imposition of constraints provide the capability to control the modelling language and ensure that merely coherent domain models can be designed. In order to design though domain models, it is compulsory to develop a graphical modelling framework. The implementation of a framework from scratch is often the most cumbersome and costly stage of the domain-specific language definition phase. Especially its maintenance introduces quite an overhead on the process and subsequently increases the complexity, the development effort and cost.

The methodology should allow overcoming the above predicaments by utilising the capabilities of the generic environment to simplify the creation and the maintenance of the modelling framework. This could be performed by extracting the concrete syntax of the language, which represents the graphical notation, from the abstract syntax *(Step 3)*. The graphical notation designates essentially the graphical figures used to design domain models onto the drawing canvas and the palette buttons that enable the drag-and-drop functionality of the modelling editor. Subsequently, the abstract and concrete syntax of

the language define the entire set of artefacts necessary to facilitate the generation of the domain-specific modelling framework *(Step 4)*.

The generated DSMF conforms to the definition of the modelling language, which acts as the backbone and supports the design of domain models *(Step 5)*; see Figure 3.4. Moreover, the DSMF provides the capability to enforce the constraints imposed during the language definition phase. In this way domain models can be validated to ensure their coherency, by evaluating their static structure and syntax *(Step 6)*. Consequently, non erroneous implementations can be automatically generated from the domain models [86]. The domain model definition and validation phases reflect correspondingly the service design and validation phases.

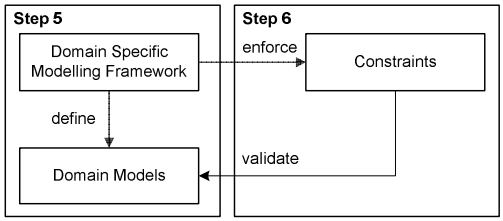


Figure 3.4: Domain model definition and validation; service design and validation.

The methodology entails also a model-to-model transformation phase *(Step 7)*, which aids either the service implementation or the configuration management of services. Figure 3.5 illustrates at the top-half that via the use of a transformation language the PIM-to-PIM mapping is defined on the basis of the source and target metamodels. In fact the mapping describes the transformation rules that guide the translation of the input domain model to an output domain model. Respectively, each domain model conforms to the abstract syntax of the modelling language that is used to design the model. The PIM-to-PIM transformation supports effectively the service configuration management since outdated models, build using previous versions of a language, can be transformed to new

models that conform to an enhanced version of the language. Apart from the service configuration PIM-to-PIM transformations support the translation of domain models to diverse domain models.

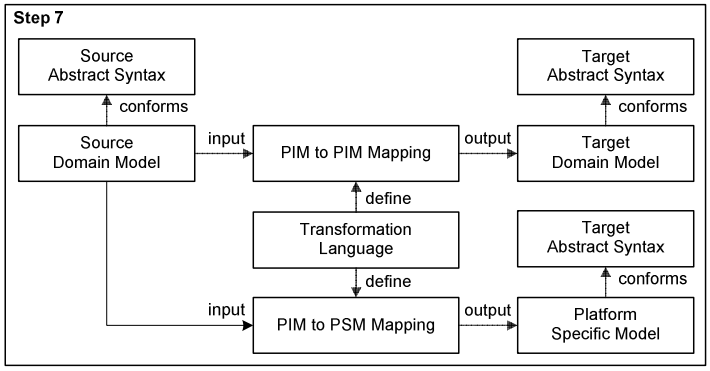


Figure 3.5: Model-to-model transformation; service implementation/management.

Moreover, as depicted at the lower-half of Figure 3.5, the mapping defined with the help of the transformation language directs the translation of domain models to platform specific models. The obtained PSMs include implementation specific details and conform essentially to the target abstract syntax that describes the operational semantics of a programming language. Consequently, the PIM-to-PSM transformation aids the code generation process and supports the overall service implementation phase.

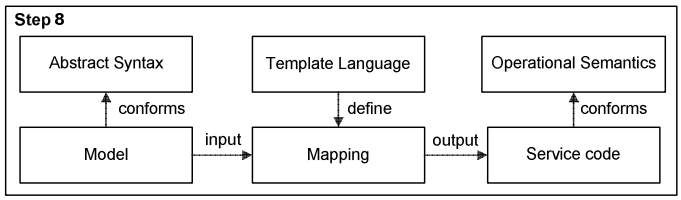


Figure 3.6: Model-to-code generation; service implementation.

The transformation phase enables the model-to-code generation phase since it affects the mapping definition of the code generator. In particular, if the intermediary PIM-to-PSM transformation is performed the mapping requires capturing merely the rules that guide the code generation. On the contrary if the intermediary phase is omitted, the platform

specific details must be hard-coded within the mapping of the code generator, in order to generate the service implementation from the abstract domain models. Figure 3.6 illustrates the definition of the code generator in the form of mapping, which accepts as input a domain model or a PSM and generates respectively the corresponding service code *(Step 8)*. Although a significant part of the service implementation is automatically generated, it is assumed that the service implementation phase is successfully completed via the manual implementation of the complex functionality of the service. However, code generation reduces the time, effort and cost associated with the service implementation.

3.4.3 The Proposed Integrated Model Driven Environment

The formulation of the IMDE is driven by the necessity to compose a generic environment that satisfies the proposed requirements and supports the methodology. In particular the environment should gratify the main prerequisite, which is the capability to generate DSMFs and combine them into an integrated service creation environment. Therefore, each generated domain-specific SCE includes as a result the entire set of software capabilities necessary to support the service creation process. This subsection introduces the core components of the IMDE and their corresponding software tools, which disclose the essential model-driven capabilities.

Predominantly, the extensible architecture of the Eclipse was the primary feature that steered the selection of the platform as the basis of the generic environment architecture. Furthermore, the comparative study of existing environments stimulated the methodical investigation of several Eclipse modelling initiatives, which captivate the interest of the modelling community due to the rapid growth of their model-driven capabilities. Hence,

the examination of the Eclipse platform and its related modelling projects disclosed various advantages and confirmed the potential of the platform in formulating an appropriate model-driven environment.

Primarily, the extensible architecture of the Eclipse provides the capability to generate and integrate DSMFs as plug-ins of the platform. Furthermore, the implementation of the majority of the Eclipse modelling projects is largely driven by the MDA specifications *(R8)*. Also the wide acceptance of the Eclipse platform and its adoption in numerous model-driven environments satisfies the accelerated adoption requirement *(R9)*. An additional benefit granted by the platform, is the capability to modify or extend the DSMFs so as to tailor them to the developers specialised needs. Finally the possibility to incorporate language specific IDEs, such as Java and C#, into the generic environment allows improving further software productivity.

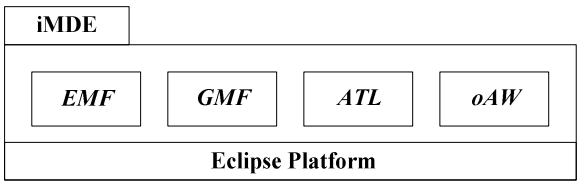


Figure 3.7: Integrated Model Driven Environment.

Several Eclipse modelling projects were carefully studied and evaluated in practice as per their ability to best meet the conceptual framework requirements. Figure 3.7 presents the Eclipse platform as the foundation and the container of the chosen core components of the generic environment. The components are namely the *Eclipse Modelling Framework (EMF) [45], the Graphical Modelling Framework (GMF) [87], the Atlas Transformation Language (ATL) [88] and openArchitectureWare (oAW) [89]*, which are introduced in detail next explaining their individual model-driven software capabilities.

The root component of the environment is the Eclipse Modelling Framework, which originally started as an implementation of the MOF specification. Both EMF and MOF are conceptually very similar and express in reality analogous metamodelling concepts [90], [91]. In effect the EMF devotes more emphasis on the implementation of the necessary tooling to express metadata and subsequently uses a simplified metamodel. Conversely, the MOF specification puts emphasis on the formulation of a metamodel that provides a richer set of semantics and facilitates consequently the expressive definition of DSLs in the form of metamodels.

More specifically, the current version of the specification, that is MOF 2.0, introduces a subset of the MOF that is called the Essential MOF (EMOF). The EMOF metamodel is identical to the Ecore metamodel of the EMF, whereas the differences are predominantly on naming rather than conceptual. Therefore, the EMF can effectively read and write serialisations of the EMOF metamodel. The EMF influenced heavily the MOF 2.0 specification, especially towards the critical direction of modelling tools integration, in order to achieve the overall objective of model-driven development.

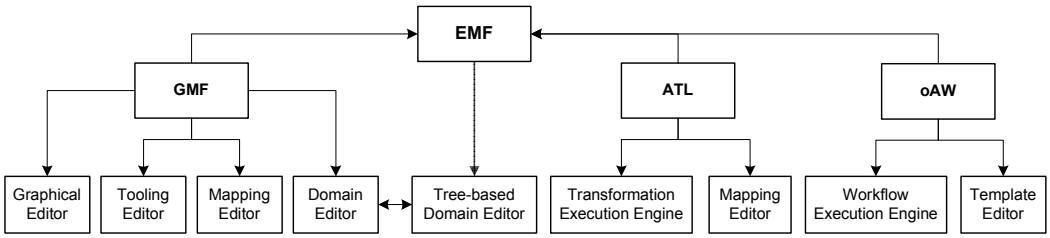


Figure 3.8: Generic environment components' software tools.

Figure 3.8 illustrates the software tools of each individual component and showcases that the EMF component is the heart of the environment. This denotes that the metamodels defined using the constructs of the Ecore metamodel, are the foundations that enable the functionalities of the rest of the modelling components. The EMF is fundamentally a

metamodelling framework and code generation facility that comprises of the *Ecore meta-metalanguage*. The meta-metalanguage provides the capability to define the abstract syntax of the DSLs using the *tree-based domain editor* illustrated in Figure 3.8 *(R1)*. Subsequently, the metamodel definition of each DSL can be automatically transformed to an EMF-based Java implementation using the code generation capabilities of the EMF.

The implementation is delivered in the form of individual Eclipse plug-ins and comprises the required software capabilities of the modelling language. Primarily, the model plug-in provides Java interfaces and implementation classes of the elements, associations and properties defined in the metamodel. In addition the adapter plug-in provides implementation classes that adapt the aforementioned metamodel classes for editing and display. Finally, the editor plug-in comprises of implementation classes that define an appropriately structured editor that complies with the tree-based representation of EMF editors. The generated tree-based modelling editor can be used consequently to define domain models that conform to the DSL.

Complementary to the EMF is the Graphical Modelling Framework that comprises of the software tools and capabilities required to derive the concrete syntax of the DSL *(R3)*. As can be realised from Figure 3.8 the capabilities of the *GMF domain editor* are highly correlated with the capabilities of the EMF editor. This denotes in particular that the abstract syntax of the DSL can be defined using either editor. Therefore, any changes performed with one editor will be reflected dynamically to the other editor. In practice the GMF editor is preferred for the metamodel definition since it provides a graphical representation, which is more comprehensible than the tree-based view.

The key capability of the GMF though is the automatic transformation of the abstract metamodel definition to the corresponding *graphical and tooling metamodel definitions*. Both the graphical and the tooling metamodels comprise the concrete syntax of the modelling language. The graphical metamodel definition specifies the graphical figures (e.g. rectangles, arrows) that map to the concepts described in the abstract metamodel. For instance a specific metaclass can be mapped to a corresponding visual shape (i.e. rectangle), which will be used to design an instance of the metaclass in the drawing canvas of the modelling editor. Moreover, the tooling metamodel defines configuration components of the modelling editor, which are essentially the palette buttons that enable the drag-and-drop functionality of the editor.

Although the concrete syntax is automatically extracted from the abstract metamodel, the corresponding *graphical and tooling editors* of the GMF (Figure 3.8) provide the capability to further customise the metamodels. This is performed in order to improve the appearance of the visual modelling editor to be generated. Subsequently, once both the abstract and the concrete metamodels are created, they can be automatically combined into a common *mapping metamodel*. The *mapping editor* allows manually inspecting the metamodel and validating it using the capabilities of the GMF. Furthermore, it provides the capability to define audit rules expressed as OCL constraints, in order to restrict the abstract syntax of the modelling language and permit merely the definition of valid domain models *(R2), (R5)*.

Following the imposition of constraints, the mapping metamodel contains the necessary constructs that drive the generation of the visual editor using the GMF capabilities. In particular the generated visual editor plug-in provides the implementation classes that

contribute the required functionality of the structured GMF-based editor. Consequently, the generated model, edit, editor and visual editor plug-ins compose the DSMF that can be integrated into the IMDE to formulate the domain-specific SCE *(R4)*.

Furthermore, the ATL component of the environment is responsible for the definition and execution of model-to-model transformations *(R6)*. More specifically, the component comprises of the *Atlas Transformation Language* for defining transformations, which conforms to the QVT standard. Transformations are defined using the *mapping editor* of the ATL language, on the basis of the source and target Ecore metamodels. Moreover, the transformation engine of the component is responsible for executing the transformation and producing the output model from a given input model.

Figure 3.8 illustrates also the oAW component of the IMDE, which comprises of the *Xpand template language, a template editor and the workflow execution engine*. Moreover, the component includes supplementary languages, namely *Check* and *Xtend*, with their corresponding textual editors. Foremost the Xpand language supports the definition of advanced generators in the form of templates, which control the output document (e.g. XML, Java) generation *(R7)*. These templates are defined using the Xpand-specific textual editor and include also references to extension functions defined using the Xtend language. These extension functions are actually utility functions that provide the capability to define well-formulated generators and improve the output obtained during text generation. Furthermore, the Check language supports the definition of metamodel and model level constraints using a proprietary language. Finally the workflow execution engine steers the generation in accordance to the template definition and converts the input model to the corresponding text-based output.

Table 3.3 presents the features of the environment and illustrates the fulfilment of each of the proposed MDD requirements. In this work it was considered imperative to pursue an approach that follows the guidelines described by the MDA standards. Consequently, these core modelling components were selected mainly because they provide non-proprietary implementations of the standards. Moreover, the core components provide the necessary software capabilities that facilitate the application of the abstract MDD methodology. In the following section the applicability of the conceptual framework is showcased via an example scenario, which presents the generation of a domain-specific SCE that supports the creation of online services for contacting surveys [92].

Table 3.3: Integrated Model-Driven Environment requirements compliance.

	MDD Requirements	***IMDE Features***
R1	Abstract syntax	EMF - NPr (√)
R2	Metamodel level constraints	GMF - NPr (√)
R3	Concrete syntax	GMF - NPr (√)
R4	DSM framework generation	EMF + GMF - NPr (√)
R5	Model level constraints	GMF - NPr (√)
R6	Transformations	ATL - NPr (√)
R7	Code generation	oAW - NPr (√)
R8	MOF compliant	(√)
	XMI compliant	(√)
	QVT compliant	(√)
	OCL compliant	(√)
R9	Accelerated adoption	Eclipse platform - (√)
Pr (√): Proprietary-Full Support NPr (√): Non-Proprietary-Full Support (√) : Full Support		

3.5 A Domain-Specific Service Creation Environment

The conceptual framework is applied in this section for the definition and generation of a domain-specific SCE that supports the analysis, desing, validation and implementation of online surveys. More specifically, this example scenario indicates the applicability of the methodology and the IMDE in miscellaneous service domains. Primarily, the *domain-specific language definition phase* is performed using the *Ecore meta-metalanguage*. The

Ecore language steers the definition of the modelling language as an abstract EMF-based metamodel. Subsequently, the GMF meta-metalanguages facilitate the generation and customisation of the concrete syntax of the modelling language. Finally, the EMF and GMF support the generation of the Survey Modelling Framework (SMF).

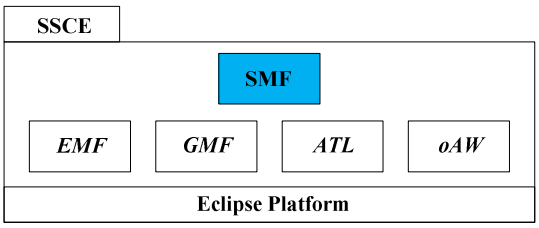


Figure 3.9: Architecture of the domain-specific service creation environment.

Figure 3.9 illustrates the generated SMF integrated with the core modelling components of the IMDE to compose the Survey-specific Service Creation Environment (SSCE). The SMF supports correspondingly the design and validation of survey models; *i.e. model definition and validation phase*. Furthermore, the *model-to-model transformation phase* is performed via the ATL component, which supports the translation of online survey models to corresponding PIM or PSM models. Finally, the oAW component is utilised to carry out the *model-to-code generation phase*, which supports the transformation of survey models to platform-specific implementation technologies.

3.5.1 Domain-Specific Language Definition

Foremost, the abstract syntax of the modelling language is defined in the form of a survey metamodel using the Ecore meta-metalanguage. Figure 3.10 illustrates the Survey Modelling Language (SML) that defines the necessary concepts for the design of domain models, which represent online surveys. The metamodel definition includes four metaclasses *(i.e. SurveysMetamodel, Page, Question, Answer)* defined using the *EClass* construct displayed on the palette of the GMF domain editor. Moreover, each metaclass

contains the required properties defined using the *EAttribute* construct. Apart from metaclasses and properties the metamodel definition includes aggregations and associations defined correspondingly using the *Aggregation* and *Association* constructs.

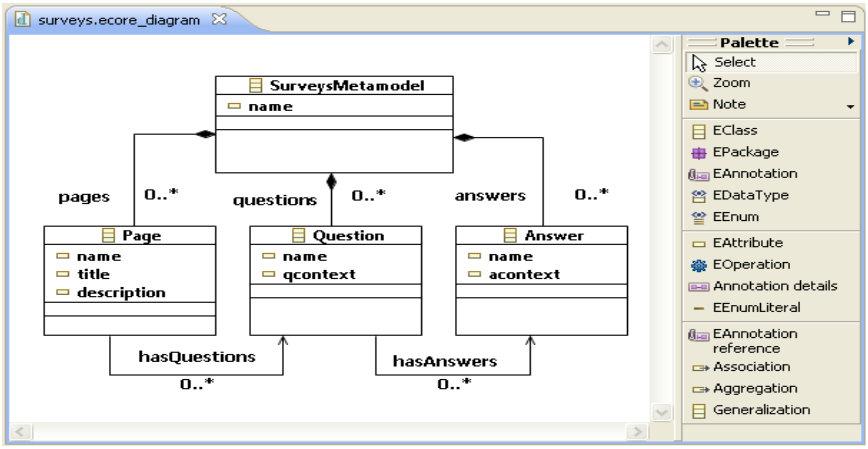


Figure 3.10: Survey metamodel definition.

In terms of the semantic meaning the metamodel definition denotes that each survey model contains several pages, questions and answers. This is depicted via the aggregation relationships *(i.e. pages, questions, answers)*, which describe the containment associations to the root metaclass. Complementary to aggregations, are the association relationships that reveal relations between model elements. The *Page* metaclass is related via the *hasQuestions* association to the *Question* metaclass, which denotes that each page of the survey is associated with a specific set of questions. Furthermore, the *Question* metaclass is related via the *hasAnswers* association to the *Answer* metaclass. This rationally denotes that each particular question has a specific set of answers logically associated to it. As aforementioned each metaclass contains also distinct attributes, which are enumerated in Table 3.4 providing their precise semantic meaning.

The definition of semantics is a widely argued and in many occasions misconceived topic that is interpreted in a variety of erroneous ways [93]. A common misconception is that

semantics are the metamodel but in reality the metamodel is purely the description of the language's abstract syntax. Semantics commonly refer to the meaning of the modelling language constructs in terms of real world entities. Typically the semantics of a modelling language are described by providing a mapping of the abstract syntax of the language to an already known language; conventionally to a mathematical formalism.

Table 3.4: Semantic meaning of the metaclasses properties.

Metaclass	***Property***	***Semantic meaning***
Page	name	*The name of the page of the survey.*
	title	*The title of the survey.*
	description	*Describes the topic of the survey.*
Question	name	*The name of the question*
	qcontext	*The question content.*
	index	*The sequence onto the page.*
Answer	name	*The name of the answer.*
	acontext	*The answer to the question specified.*
	index	*The sequence in terms of the question.*

Undoubtedly a rigid set of syntactic rules, as in the case of the Ecore metamodel and its associated OCL rules, is crucial so that software tools have a common understanding of the language semantics [93]. In order to realise though the semantics of the metamodel we need to provide the *Mapping (M)* of its *Abstract Syntax (AS)* to a *Semantic Domain (SD)*, which is considered an abstraction of the real world. The SD refers to a specific set of meanings or a language that holds the set of meanings for that particular domain. Consequently the definition of the mapping from the metamodel to a target language that has well established semantics provides the necessary semantics for the metamodel [93]. *In this thesis the mapping of the metamodel to the operational semantics of a programming language conveys the semantics of the newly defined modelling language;* $M : AS -> SD$. This mapping is defined actually in the model-to-code generation phase and denotes the precise semantics of the modelling language.

Apart from the abstract syntax, the metamodel definition encapsulates additional implicit information concerning the concrete syntax of the language. The capabilities of the GMF provide the potential to interpret the domain metamodel and extract automatically the corresponding concrete syntax. Figure 3.11 presents the generation wizard that showcases the *SurveysMetamodel* as the top level container and the metamodel elements mapped automatically to respective graphical representations. The figure illustrates the three metaclasses, *namely Page, Question and Answer*, which are translated as nodes (i.e. rectangle figures). Moreover, *the hasQuestions and hasAnswers* associations are interpreted as connection links (i.e. arrow figures) and the attributes of each metaclass are translated as properties (i.e. label figures).

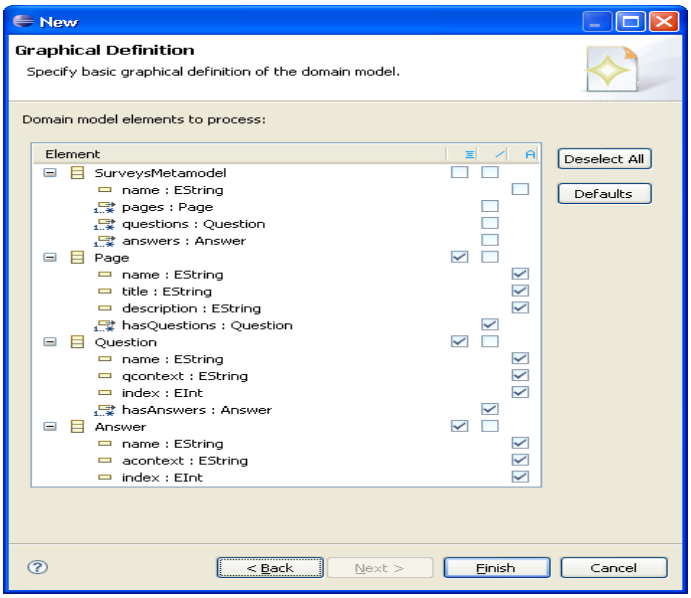


Figure 3.11: Abstract to concrete syntax transformation.

The wizard executes actually an abstract-to-concrete syntax transformation and generates the corresponding graphical metamodel that consists of the necessary figures. Further customisation of the graphical metamodel is possible using the GMF graphical editor, on the basis of the graphical meta-metalanguage. The customisation provides the capability

to improve the appearance of the graphical representation and increase the understanding of the domain models. For instance, a common customisation refers to the addition of a read-only stereotype label to the graphical figure of each metaclass. This provides the capability to distinguish between instances of different metaclasses defined in the survey model. Using an equivalent procedure the transformation of the domain metamodel to a corresponding tooling metamodel is accomplished. Accordingly, metaclasses and associations are transformed to palette buttons that enable the drag-and-drop actions of the modelling editor. Further customisation of the tooling metamodel is permitted using the GMF tooling editor, in accordance to the tooling meta-metalanguage.

The graphical and tooling metamodels provide the concrete syntax of the modelling language. Subsequently, the combination of the concrete metamodels and the abstract metamodel into a common mapping metamodel provides a unified representation of the modelling language. The mapping metamodel facilitates the automatic generation of the corresponding EMF-based Java implementation of the DSMF. Prior to generating the implementation it is essential though to impose some rules that restrict the abstract syntax of the language. The GMF mapping editor supports the definition of domain rules in the form of OCL constraints imposed onto the metaclasses of the abstract metamodel.

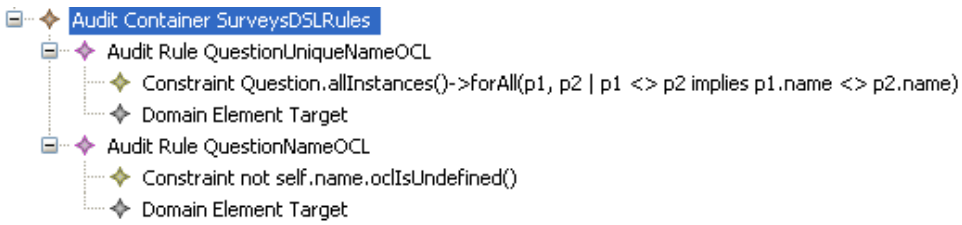


Figure 3.12: Definition of OCL constraints for the language.

Figure 3.12 illustrates two constraints defined for the Survey Modelling Language. The primary constraint, namely *QuestionUniqueNameOCL*, denotes that the entire set of

Question instances defined in the domain model should have a distinct name. In the case that two or more instances of the *Question* metaclass comprise the same name then a constraint violation is raised when the domain model is validated. Moreover, the second OCL constraint, *namely QuestionNameOCL*, forbids the existence of *Question* instances in the domain model that do not have a specific name defined. Apart from the definition of constraints, the mapping editor provides the capability to validate the defined constraints before generating the DSMF implementation.

The definition of constraints successfully completes the domain-specific language definition phase. Subsequently, the respective SMF is generated and integrated into a unified Survey-specific SCE. The SCE is utilised next to illustrate the definition and validation of an example survey model. Moreover, the transformation and code generation phases are showcased on the basis of the defined survey model.

3.5.2 Domain Model Definition and Validation

The generated *Survey Modelling Framework* comprises of a modelling editor with drag-and-drop capabilities that supports the design of domain models; i.e. online surveys. The backbone of the modelling editor is the abstract and concrete syntax definition of the SML. Figure 3.13 illustrates a section of a designed *domain model that represents an online survey for the "University of Essex library facilities"*. Furthermore, the figure exemplifies the look-and-feel of any GMF-specific modelling framework, such as the SMF.

The example survey model is composed by a single page that is associated with seven questions. Each corresponding question is respectively associated with four answers. Apart from the palette that allows adding new elements onto the drawing canvas the

properties view of the modelling editor permits editing the corresponding attributes of each element. Moreover, the SMF editor provides the capability to enforce the constraints imposed during the language definition phase. This is performed to ensure the validity of the models prior to executing the transformation and/or code generation phases. Figure 3.13 showcases that two question instances of the survey model have the same name definition. Subsequently, a constraint violation is raised, which is depicted via the validation decorators and the tootip diagnostic information presented by the editor. This allows the designer to identify and rectify the problems ensuring the correctness and consistency of the survey model.

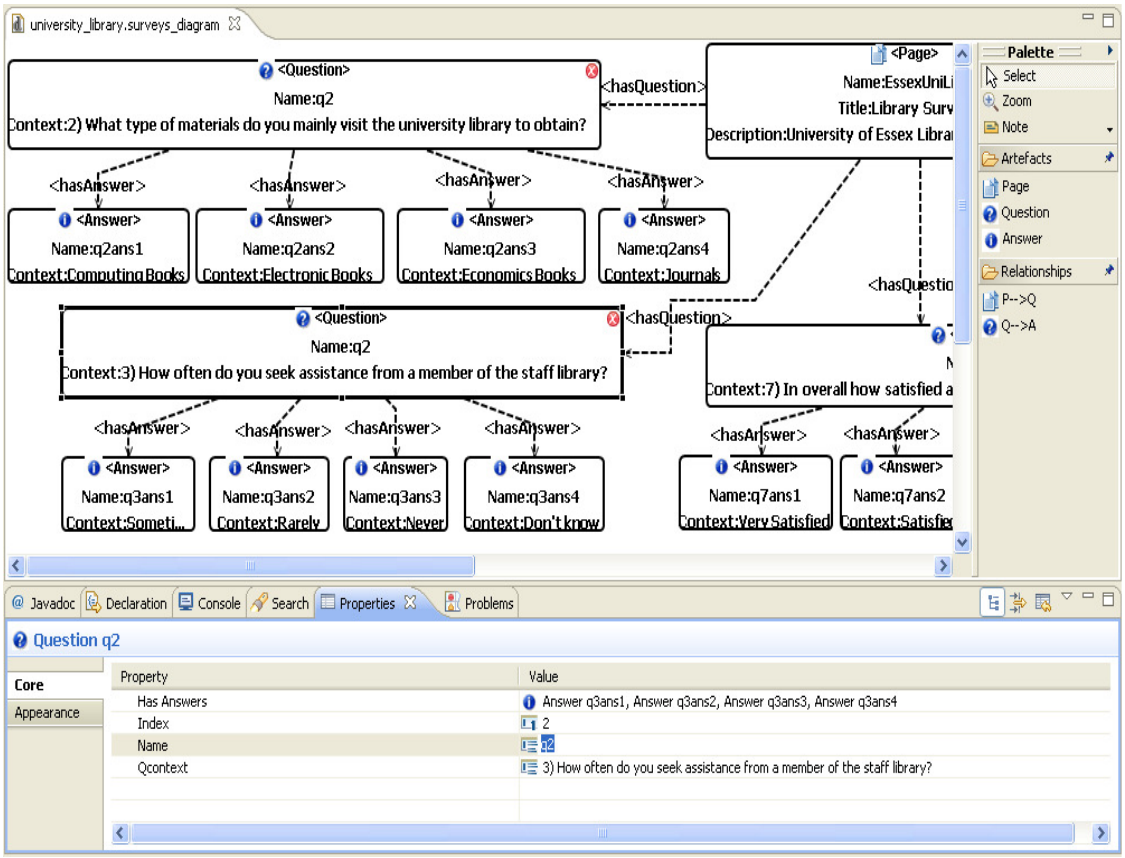


Figure 3.13: Definition and validation of an online survey model.

3.5.3 Domain Model-to-Model Transformation

The validity of the survey model is essential for the transformation phase since the input model affects crucially the output model. In the case that the source model is erroneously defined, the target model produced will also have an erroneous definition. Therefore, it is imperative to have a consistent input model prior to the definition and execution of the transformation. Apart from the source model, the definition of the transformation requires a source and a target metamodel, on the basis of which the mapping is defined.

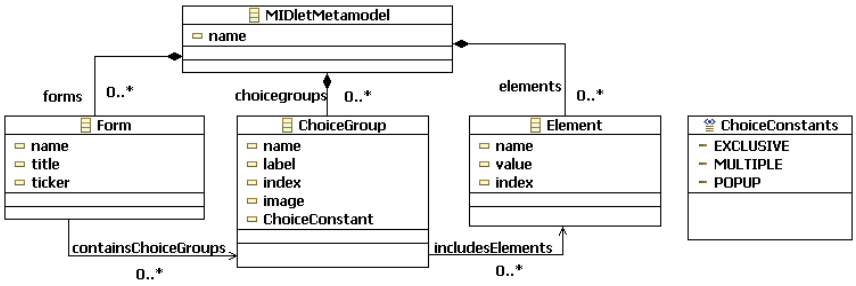


Figure 3.14: Simplified J2ME MIDlet metamodel.

Figure 3.14 illustrates the target metamodel that defines a simplified version of a GUI modelling language. In particular, the modelling language is considered as technology specific since it includes artefacts that convey features bound to the J2ME Mobile Information Device Profile (MIDP). The MIDP is an application implementation specification that allows developing enhanced mobile user interfaces. Therefore, each MIDlet model includes properties confined to the J2ME platform and subsequently it can be easily translated to platform-specific implementation code.

The metamodel definition includes the root metaclass, namely *MIDletMetamodel*, which represents the MIDlet model that contains several Forms, ChoiceGroups and String elements. This is depicted via the aggregation relationships *(i.e. forms, choicegroups, elements)* that describe the containment associations to the root metaclass. The *Form*

metaclass represents a mobile display that is associated to choice group items via the *containsChoiceGroups* association. In addition to the association, the metaclass includes its individual properties, which are the *name, title and the ticker* of the Form graphical component. The name and title properties are apparent as to their meaning, while the ticker property denotes a piece of text that runs continuously across the display.

Furthermore, the *ChoiceGroup* metaclass represents a group of selectable elements that are contained within the Form component. The ChoiceGroup component includes distinct properties, which are the *name, label, ChoiceConstant, index and image* that characterise the component. Foremost, the *name* property depicts the name of the component while the *label* property depicts the text displayed on this component. In addition the *index* property designates the sequence for placing the choice group within the display and the *image* property determines the path to the image file loaded for the component. Finally, the *ChoiceConstant* property designates the type of the choice group, which is defined either as a single choice, as multiple choices or as a popup window with choices. This property is defined in the model using the *ChoiceConstants* enumeration literals.

Apart from the properties of the *ChoiceGroup* metaclass the association *includesElements* defines the actual choices of each choice group, which are described in the model via the *Element* metaclass. The *Element* metaclass defines through each instance a corresponding option included in the choice group component, which is represented in fact as a *String* primitive datatype. Therefore, each primitive element includes a name, a text value and an index number that denotes the actual positioning of the element within the choice group. These features are described in the metaclass via the corresponding *name, value and index* properties.

The midlet metamodel provides the third component required for the definition of the transformation mapping. Figure 3.15 presents an extract of the mapping shown in Appendix A that drives the transformation of survey models to midlet models. The mapping is defined using the ATL mapping editor in accordance to the syntax of the ATL language. This specific part of the mapping steers the translation of instances of the *Page* metaclass to instances of the *Form* metaclass. Hence, the *name, title and description* properties of the *Page* metaclass are transformed to the corresponding *name, title and ticker* properties of the *Form* metaclass. Moreover, the association of pages to questions is transformed to the analogous association of forms to choice groups. In essence this designates that each survey page is represented by a form component and each question is represented as a choice group component on the mobile device display.

```
1.  rule Page2Form{
2.      from
3.          page: surveys!Page
4.      to
5.          form: midlets!Form (
6.              name <- page.name,
7.              title <- page.title,
8.              ticker <- page.description,
9.              containsChoiceGroups <- page.hasQuestions
10.         )
11. }
```

Figure 3.15: An extract of the ATL language Survey to MIDlet mapping.

The transformation process relies on the ATL execution engine, which accepts as inputs the survey model and the mapping and transforms the artefacts of the survey model to the corresponding artefacts of the MIDlet model. Figure 3.16 illustrates the produced output model that conforms to the MIDlet modelling language and subsequently can be loaded within the generated MIDlet Modelling Framework (MMF). The MMF provides the capability to define additional platform specific properties for the model. For instance,

the blue highlighted combo-box shown in the properties view of the MMF facilitates the definition of the choice group type to one of the enumeration values permitted by the modelling language. Consequently, the enrichment of the MIDlet model with platform specific details completes the transformation phase and aids the following model-to-code generation phase.

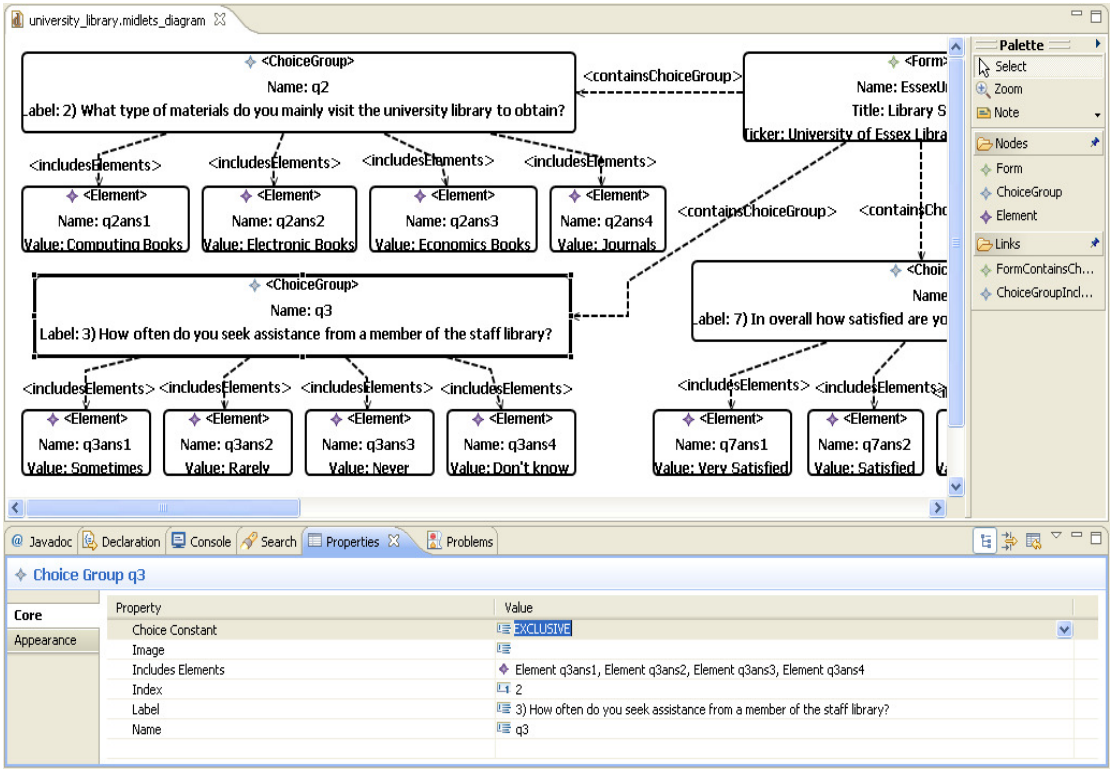


Figure 3.16: The output MIDlet model obtained from the transformation.

3.5.4 Domain Model-to-Code Generation

The generation process is highly dependent on the template definition, which describes the rules that guide the translation of the elements and properties of the domain model (i.e. survey model). These rules are defined in the form of *Xpand template code* that conforms to the metamodel definition and follows the non-functional programming paradigm. Moreover, the template is defined in accordance to the syntax of the output

document that will be produced; *e.g. Java code, XML, HTML.* This means that the structure of the template definition resembles accurately the structure of the output document, excluding the values derived and generated from the domain model.

Apart from the template(s) definition, the validity of the PIM or PSM models is critical for the implementation phase, since the code generation process relies heavily on these models. In particular, the PIM or PSM model is the primary input while the template definition is the secondary input of the code generator. Figure 3.17 illustrates the generator component that accepts the survey model and the template as the inputs that steer effectively the generation process.

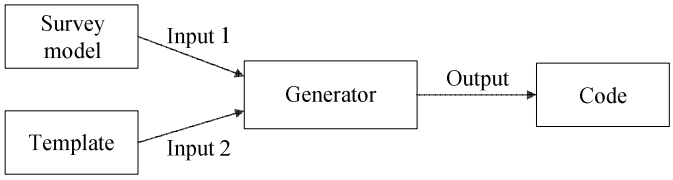


Figure 3.17: Model-to-code generation process.

The generator is defined in the form of an executable script and provides the capability to delegate calls to the Java implementation classes of the oAW component. Furthermore, the executable script includes references to the domain model and the corresponding defined template(s). The generator delegates initially a call to the required *org.openarchitectureware.emf.XmiReader* class, which is actually an *XmiParser* object that parses and loads the domain model elements and properties into memory. This can be achieved essentially since the designed model is stored in the form of XML-Metadata Interchange (XMI) format. Hence, the elements and properties described in the survey model become accessible at runtime.

Following, the generator calls the necessary *org.openarchitectureware.xpand2.Generator* class, which is responsible to execute the transformation on the basis of the defined

template and generate the corresponding equivalent text-based output; i.e. J2ME code. In this thesis the code generation phase is performed by direct transformation of the abstract domain models to the corresponding service code. Hence, the intermediate PIM-to-PSM transformation phase presented in the previous subsection is excluded and the platform specific details are hard-coded as a result within the defined template(s). This is performed in order to simplify the overall service creation process, since the definition of platform specific metamodels and the generation of their modelling frameworks can be subsequently omitted.

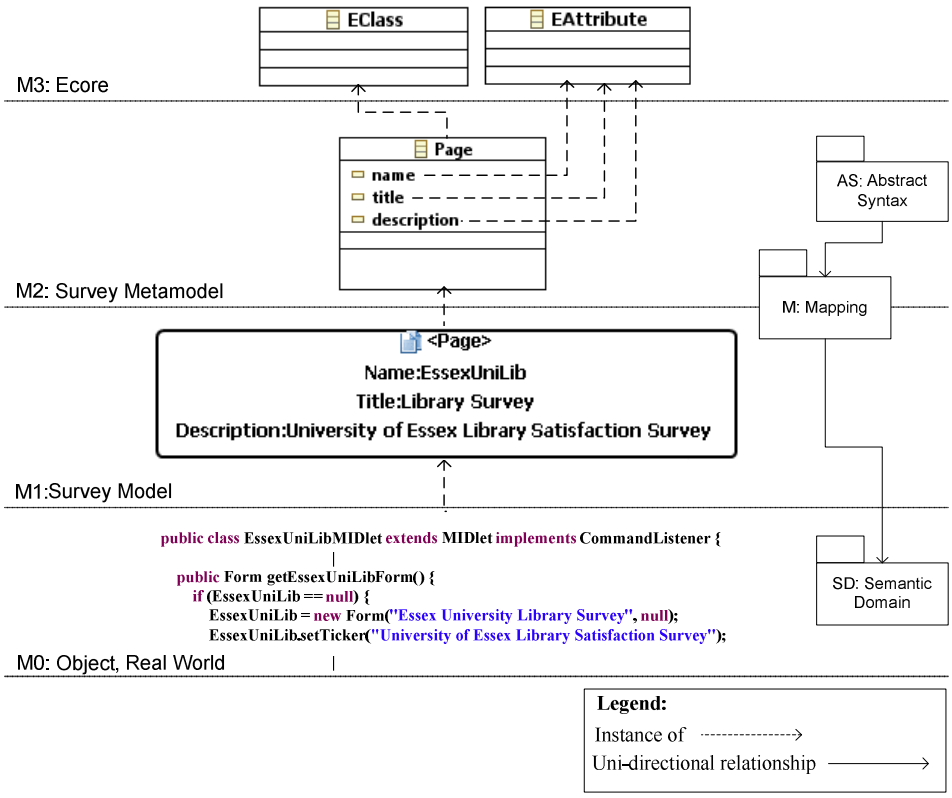


Figure 3.18: Mapping the abstract syntax to a semantic domain.

Figure 3.18 presents at *Level M3* the Ecore constructs *EClass* and *EAttribute*, which are used to define the *Page* metaclass residing at *level M2*. The metaclass is actually an instance of the *EClass* construct and contains the *name, title and description* properties,

which are instances of the *EAttribute* construct. Accordingly the *Page* entity at *level M1* designates an instance of the *Page* metaclass with the respective properties being assigned sample values. Both the metaclass and the entity represent abstractions of the *Form* component, which is to be displayed onto the actual mobile device. Although the J2ME code presented at *level M0* is also an instance of the *Page* entity and an abstraction of the runtime object, it is considered as the object itself since it contains all the necessary information for constructing the real-world object.

```
1.      public «this.name»getForm() {
2.              if («this.name» == null) {
3.              «this.name» = new Form («this.title», null );
4.              «this.name».setTicker(«this.description»);
5.  «FOREACH hasQuestions AS question»
6.              «this.name».insert(«question.index», getChoiceGroup«question.name»());
7.  «ENDFOREACH»
8.              «this.name».addCommand(getExitCommand());
9.              «this.name».addCommand(getOkCommand());
10.             «this.name».setCommandListener(this);
11.             }
12.         return «this.name»;
13.     }
```

Figure 3.19: Extract of the mapping to the J2ME operational semantics.

Figure 3.19 illustrates an extract of the mapping presented in Appendix B, which describes the transformation of the page element to the corresponding J2ME service code[5]. The mapping defines the transformation of each *Page* element to a corresponding J2ME function that returns a *Form* component. The name of the function is defined in accordance to the name of the *Page* element and the individual features of the *Form* component are set to the corresponding values obtained from the properties of the *Page* element. Moreover, lines 5-7 define a loop that iterates the set of questions associated to the *Page* and transforms them to choice groups, which are subsequently inserted within

[5] Appendix C presents the extension functions used by the J2ME template to generate common functions.

the *Form*. The final statements add commands and associate a command listener to the *Form* so as to respond to actions generated by the user of the mobile service.

Apart from the J2ME-specific template definition the mapping of the survey modelling language to the operational semantics of the Java programming language is also defined; see Appendix D. This demonstrates the capability to transform the same abstract survey model to different implementation technologies. Subsequently, either template mapping can be accepted along with the survey model as inputs of the code generator, which executes the transformation of the survey model to the corresponding implementation technology. Figure 3.20 illustrates the generated services running on diverse execution platforms. The primary service runs on a J2ME mobile device emulator and the second service on a Java enabled laptop device. Both service implementations are generated from the same survey model illustrated in Figure 3.13.

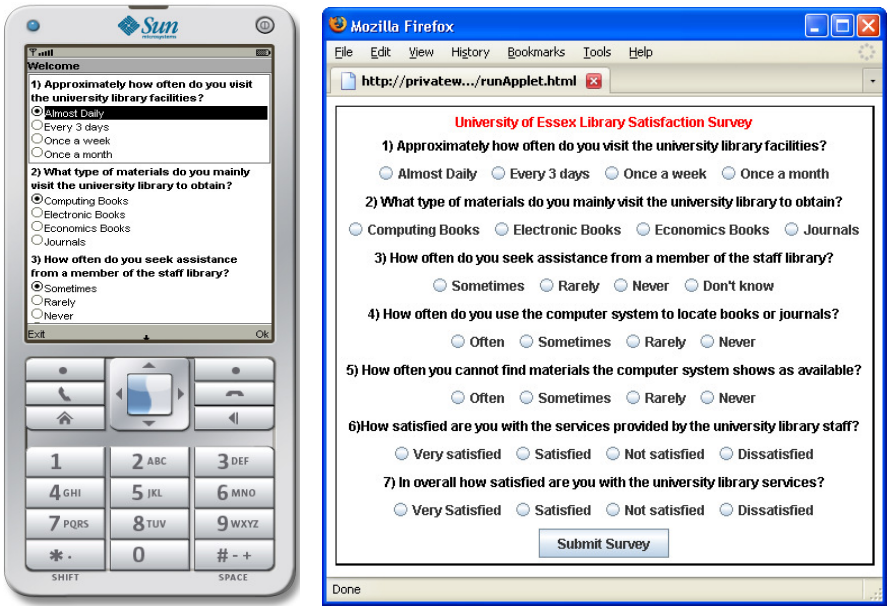


Figure 3.20: Generated service running on diverse execution platforms.

The aforementioned scenario demonstrates the applicability of the conceptual framework via the automatic generation of a domain-specific SCE for the development of online

surveys. Apart from the example scenario the framework was applied in the following industrial projects: (i) Product Modelling for Service Delivery Frameworks and (ii) Managing Assurance, Security and Trust for Services (MASTER) [94]. These projects showcase the merits of the conceptual framework in generating diverse service creation environments and applying the approach across different domains for the development of domain-specific services. Subsequently, the proposed framework is utilised in this thesis to generate the Pervasive Service Creation Environment of the MDPNF, which supports the creation of pervasive services.

3.6 Model-Driven Petri Net based Framework

This section presents the architecture of the Pervasive Service Creation Environment and the proposed Model-driven Petri Net based process, which compose the MDPNF used for pervasive service creation. The prime characteristics of pervasive services identified in this work steer the definition of the corresponding modelling languages that facilitate the design of pervasive services. These requirements are namely service adaptability, seamless user-service interaction and the dynamic behaviour of pervasive services. While service adaptability is examined by existing context modelling approaches, characteristics such as user-service interaction and the dynamic behaviour of the service are also fundamental when defining pervasive services.

The fulfilment of these requirements necessitates the definition of three complementary modelling languages and the generation of their supporting modelling frameworks. These DSMFs facilitate the definition and validation of pervasive services and interact with the core components to deliver the service implementation using an automated software process. In specific the generated modelling frameworks are integrated into a unified

PSCE that supports the phases of the pervasive service creation process. Figure 3.21 presents the architecture of the PSCE, which includes the developed modelling components integrated with the core components on top of the Eclipse platform.

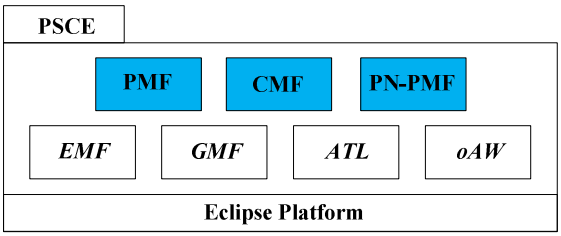


Figure 3.21: Architecture of the Pervasive Service Creation Environment.

The core components are necessary in the overall PSCE since they are the enablers of the functionalities of the generated components. The developed modelling components are namely, (i) the Presentation Modelling Framework (PMF), (ii) the Context Modelling Framework (CMF) and the (iii) Petri-Net based Process Modelling Framework (PN-PMF). Foremost, the PMF addresses the user-service interaction requirement, while the CMF and the PN-PMF satisfy correspondingly the service adaptability and the dynamic service behaviour. Moreover, the platform independent nature of the methodology and the IMDE gratify non-functional requirements such as service portability and the reduction of the time, effort and costs to create and deploy pervasive services.

On the basis of the architecture of the PSCE the Model-driven Petri Net based process is defined that combines the MDA paradigm with the Petri Net formalism to support the pervasive service creation process. The MDA paradigm facilitates the definition of the PN-PML and the generation of its supporting modelling framework. Hence, using the generated Petri net-based Process Modelling Framework the definition and static validation of Petri Net process models can be undertaken. On the other hand, Petri Nets

theory serves as a formal specification of the pervasive service functionality, i.e. the service tasks, and facilitates the validation of the service operational semantics.

Figure 5.22 presents the proposed process that utilises the developed PMF and CMF for the definition of the presentation and context models. Additionally the generated PN-PMF supports the design of Petri net Process Models, which define the pervasive service behaviour. Subsequently, via the use of an XML-based generator the Petri Net Markup Language (PNML) standard representation of the process model is generated. This provides the capability to import the model into the Renew software tool to validate its operational semantics by means of model simulation. In case any inconsistencies are detected in the model, the execution halts, presenting the erroneous semantics to the user. Therefore, the designer must undertake the necessary steps to refine the model. This iterative process continues until the designer rectifies the problems, so as to guarantee the correctness of the models prior to the service implementation phase.

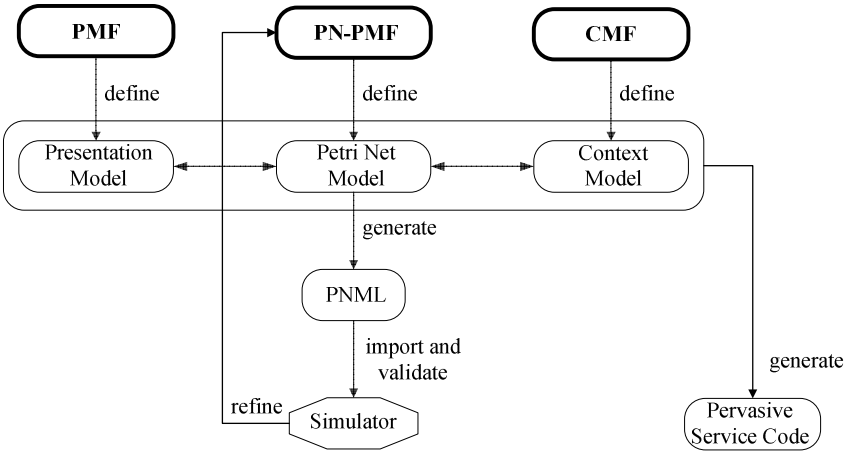


Figure 5.22: Model-driven Petri Net based process.

Furthermore, the corresponding J2ME and Java code generators are defined that drive the transformation of the defined presentation, process and context models to the analogous implementation technology. Consequently, via the exploitation of the set of generators

developed for the different modelling frameworks the pervasive service implementation can be effectively generated. Note that the aim is to generate a substantial part of the implementation from the models. Thus, the developer still necessitates implementing manually complex computational tasks to obtain the complete service implementation. The generated pervasive service code reduces though the manual coding effort and aids also the developer in the implementation of complex functionality.

The proposed process benefits from the utilisation of the *Pervasive Service Creation Environment*, *which is composed by the PMF, the CMF and the PN-PMF integrated on top of the generic IMDE*. In fact, both the process and the PSCE compose a Model-Driven Petri Net based Framework, which supports effectively the analysis, design, validation and implementation of services for the pervasive domain.

3.7 Summary

This chapter focuses on the definition of a conceptual framework that guides the model-driven development of domain-specific software services. The framework defines the generic architecture of a model-driven environment and describes an abstract MDD methodology, which aim to support the service creation process. In accordance to the generic architecture the Integrated Model-Driven Environment is formulated that facilitates the generation of domain-specific service creation environments. Moreover, the IMDE allows applying the methodology in practice for the benefit of domain-specific service creation. The applicability of the framework is showcased via an example scenario that facilitates the generation of a domain-specific SCE for an online survey system. Concluding, the MDPNF is introduced that compises of the PSCE and the Model-driven Petri Net based process that support pervasive service creation.

Chapter 4 Context Modelling and Presentation Modelling Frameworks

The Context Modelling Framework and the Presentation Modelling Framework presented in this chapter are two of the modelling frameworks of the overall Pervasive Service Creation Environment. The Context Modelling Framework provides the capability to introduce context-awareness at service creation to facilitate the adaptation of pervasive services. In particular, it allows defining and validating context models. These models are effectively transformed to the required implementation, which enforces at run-time the context-awareness requirement. The Presentation Modelling Framework supports mainly the definition and validation of models that describe graphical user interfaces of the pervasive service. In essence, the modelling framework supports the model-driven development of advanced graphical user interfaces and satisfies essentially the user-service interaction and service portability requirements.

4.1 Motivation and Approach

The prerequisite to gratify the service adaptability, service portability and user-service interaction characteristics of pervasive services steers the definition and generation of the context and presentation modelling frameworks. Via these frameworks many of the advantageous features of MDA can be utilised, such as high-level abstraction, increased automation of software generation and platform independence. This permits not only to facilitate but also to simplify the process of context and presentation modelling and the eventual transformation of the models to the corresponding service implementation; that conforms to a particular programming language [95].

More specifically, the modelling frameworks developed in this work support the definition and the validation of context and presentation models, which are considered as abstractions of dynamic context information and graphical user interfaces. While context modelling is highly imperative for achieving the adaptation of the pervasive service at runtime, presentation modelling is also important in terms of providing the necessary GUIs that facilitate the interaction of the user with the service.

Primarily, the Context Modelling Framework facilitates the definition of context models, which describe service and user related information in a format that can be recognised and utilised by pervasive services. Consequently, these context models are validated using the CMF and then transformed to platform specific code via the capabilities of the overall PSCE. In this way the generated implementation provides the capability to enforce at runtime context management tasks, such as the administration and distribution of context information to services to achieve their adaptation.

Furthermore the generated Presentation Modelling Framework facilitates and simplifies the definition and validation of presentation models that represent abstract notions of GUIs of the pervasive service. These presentation models are designed using the main constituent of the PMF, which is the Presentation Modelling Language defined in the form of an EMF-based metamodel. Hence, the validation of the models is directly accomplished via the OCL constraints imposed onto the PML and the transformation of the models to different implementations is achieved via the capabilities of the PSCE. The generated implementations can be therefore deployed onto diverse platforms, representing in fact GUIs with which the user is able to interact with the service.

4.2 Context Modelling Framework

4.2.1 Context Modelling Domain Requirement Analysis

Context-awareness denotes the capability of devices to detect changes in context information and react accordingly to adapt the service behaviour. The primary context categories the author acknowledges and builds upon in this work are explicitly stated by Dey and Abowd in [30]. These are namely the *Identity, Time, Location and Activity* category types. Apart from these the *Preference* category type is introduced, which is of particular importance since it depicts different service behaviours in accordance to individual user preferences. For example a specific user might prefer to contact a particular friend by sending a simple SMS message and another friend via an email message. Therefore, a distinct behaviour should be realised by the service for each of the two recorded cases. Furthermore, recording users' preferences and providing a rating to individual preferences enables the service to undertake the appropriate actions on behalf of the user; with increased probability of correctness.

In addition to the primary category types, secondary context categories are introduced that denote explicit information of the context-aware service. For instance a restaurant's details, *e.g. name, address*, describe secondary context categories that are explicit to a particular service. Furthermore, primary and secondary categories facilitate the derivation of simple pieces of context information [30]. For instance, in the case that the *Identity* of the person is known primitive context information can be derived about that individual, such as his *name, email, gender etc*.

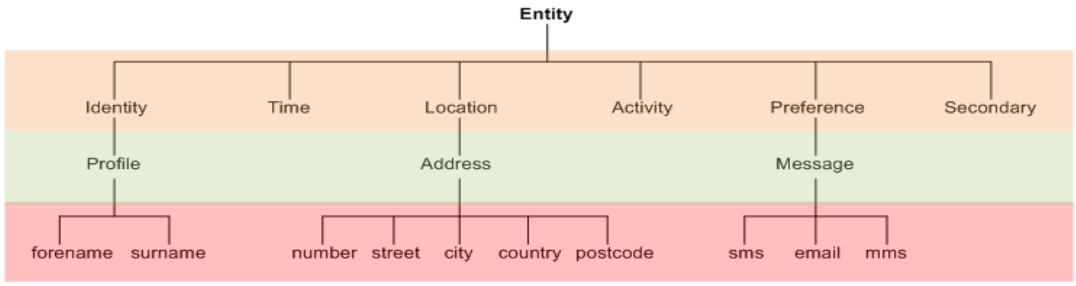


Figure 4.1: Context categories abstraction levels.

Figure 4.1 illustrates the different abstraction levels of context categories. Each particular entity *(e.g. Person, Device)* is described by different context categories. At the top level context represents information objects *(e.g. Identity, Location)*, which are composed by complex datatypes such as *Profile, Address and Message*; i.e. at the second level. These complex datatypes consist of primitive datatypes such as *Integer, String, and Boolean*, which reside at the third abstraction level. For instance the *Identity* of a person is defined in the form of a *Profile* complex datatype, which is composed by the forename and the surname *String* primitive datatypes. Furthermore, context information can defined using enumeration types rather than complex and/or primitive datatypes. For example, the *Preference* context information object illustrated in Figure 4.1 is defined using the *Message* enumeration, which contains three literals namely *"sms", "email" and "mms"*.

In conventional services information can be managed in a common way since it is mainly acquired from a single input source; i.e. profiled by the user. On the contrary, in context-aware services information needs to be managed differently since it is obtained implicitly or explicitly from diverse input context sources (e.g. sensors, repositories). Consequently, a classification of context sources is essential in order to distinguish and manage context information accordingly. In the context of this thesis the classification defined in [35] forms the basis that allows distinguishing between different context sources and realising the context management and distribution tasks. The classification segregates context sources as follows and allows determining context validity, context quality and context privacy on the basis of the classification.

- ***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.

Context validity can be determined by the persistence of context information. The persistence property defines the frequency with which context information are subject to change. Conventionally, static context sources disclose a permanent correlation between the entity and its associated context information. Profiled sources reveal infrequent (i.e. 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 (i.e. frequent or volatile). This is due to the fact that context information obtained from sensors or derived from other information is highly unpredictable.

Moreover, context validity can be determined via the use of temporal constraints. These are defined either as comparative or absolute time constraints. The comparative constraint establishes a valid expiration time with respect to the exact time the context information was obtained. Therefore, it depicts a relative time interval from the moment in time this information was obtained until the time the information becomes outdated. Alternatively, absolute time constraints provide the capability to define the precise time interval that determines the validity of the context information. This is depicted via the definition of the starting and expiring times, which denote the absolute time frame during which this context information is valid. Temporal constraints are of prime importance since they complement the persistence characteristic and allow determining the *validity of context information*.

The *context quality* characteristic is also considered on the basis of the classification of context sources. Static information contained within a data repository is generally of superior quality than profiled information inputted by the user; assuming correct information is defined in the repository. The aforementioned statement is true since a user might overlook or forget to update the necessary context information when required. Correspondingly, sensed context is of inferior quality than the static and profiled context information. This is due to the fact that context acquired from sensors can be typically erroneous or inaccurate and consequently unreliable. Furthermore, context can be derived on the basis of other information. For instance, potential restaurants for dining can be derived from the food preference of the person. Therefore, the quality of derived information relies both on the quality of other context information (e.g. sensed) and the accuracy of the derivation rule.

Context privacy is another major characteristic that needs to be addressed in order to achieve the acceptance of context-aware services by users. Different context information necessitates to be treated differently in terms of privacy requirements. For instance a user might want to keep his profile information accessible to all users of the service. Opposed to this, some context information such as the credit card details of the person must not be accessible to other users. Hence, different permissions should be imposed on each context source to restrict accordingly the access to sensitive context information in accordance to the users' privacy requirements.

Apart from context categories and input context sources, contextual situations are fundamental for modelling and creating context-aware services. Context situations represent the conditions that drive the execution of explicit behaviours of the context-aware service in the case a context event occurs. For instance the change of context information related to a person (*e.g. location*) eventuates in the alteration of his contextual state (*e.g. in office or at a meeting*). Another example of a context situation denotes that in the case a person is present at a meeting then his mobile phone must be automatically set to silent mode. Therefore, it is necessary to realise the change in the person's contextual state in order to be able to drive accordingly the corresponding service behaviour. Context situations are imperative for modelling the conditions that drive the execution of explicit context-aware behaviours. These context-based conditions are also essential for steering the interaction between the user and the service.

The proposed *taxonomy of context categories, the classification of context sources, temporal constraints and contextual situations* depict the core requirements, which describe the context modelling domain. Therefore, on the basis of these requirements the

Context Modelling Language is defined to facilitate the design of context models and the mapping of models to the corresponding implementation technology. The CML supports the definition of context categories from which information objects are generated (e.g. Java objects). Furthermore, the CML facilitates the modelling of context sources that drive the generation of context management objects for handling diverse context information as required in a distributed environment.

The CML supports also the definition of temporal constraints from which temporal objects are generated, so as to provide the capability to determine the validity of context information. In addition, the CML enables the definition of contextual situations and assists the generation of distinct situation objects, which determine the conditions for undertaking explicit service behaviours. Consequently, the generated management objects allow querying, obtaining and distributing context information to the pervasive service to achieve its adaptation.

4.2.2 Context Modelling Language Metamodel Definition

The aforementioned requirement analysis facilitates the identification of the essential artefacts of the context modelling domain, which support the definition of the *Context Modelling Language*. On the basis of the defined CML the generation of the *Context Modelling Framework (CMF)* is possible via the execution of the Steps 1-4 of the MDD methodology; refer to Chapter 3. Figure 4.2 illustrates the utilisation of the EMF and GMF software capabilities provided by the IMDE for the definition of the abstract syntax, the concrete syntax and the domain rules of the CML. The figure demonstrates also the mapping of the different artefacts of the modelling language into a common metamodel that facilitates the generation of the CMF.

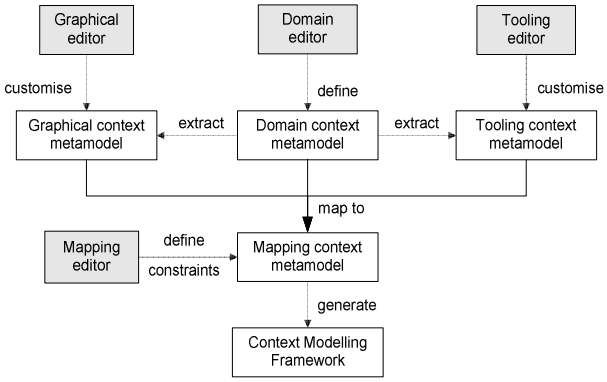


Figure 4.2: Definition and generation of the Context Modelling Framework.

In accordance to Step 1 of the MDD methodology the domain semantics identified during the requirements analysis phase must be mapped to language elements. This denotes in particular the definition of the CML in the form of a context metamodel, using the domain editor of the EMF ECore meta-metalanguage. The context metamodel defines the abstract syntax of the modelling language in a machine readable format that provides the capability to process models and perform tasks such as model definition, transformation and code generation.

Figure 4.3 illustrates the *Context metamodel* that describes the elements (*e.g. Entity*), properties (*e.g. multiplicity*) and relationships (*e.g. ECAsource*) of the context modelling domain. It comprises of the required artefacts that provide the capability to define context models, which are considered metamodel instances. The *DocumentRoot* metaclass is the root of the metamodel and the container of the elements of the context model. In essence the root metaclass represents the context model that includes artefacts such as entities, context information and context sources. These containment relationships are depicted via the aggregation associations of the *DocumentRoot* metaclass (*e.g. entities, contexts*) with the rest of the artefacts. The root metaclass apart from aggregations contains also a single property that denotes the name of the context model.

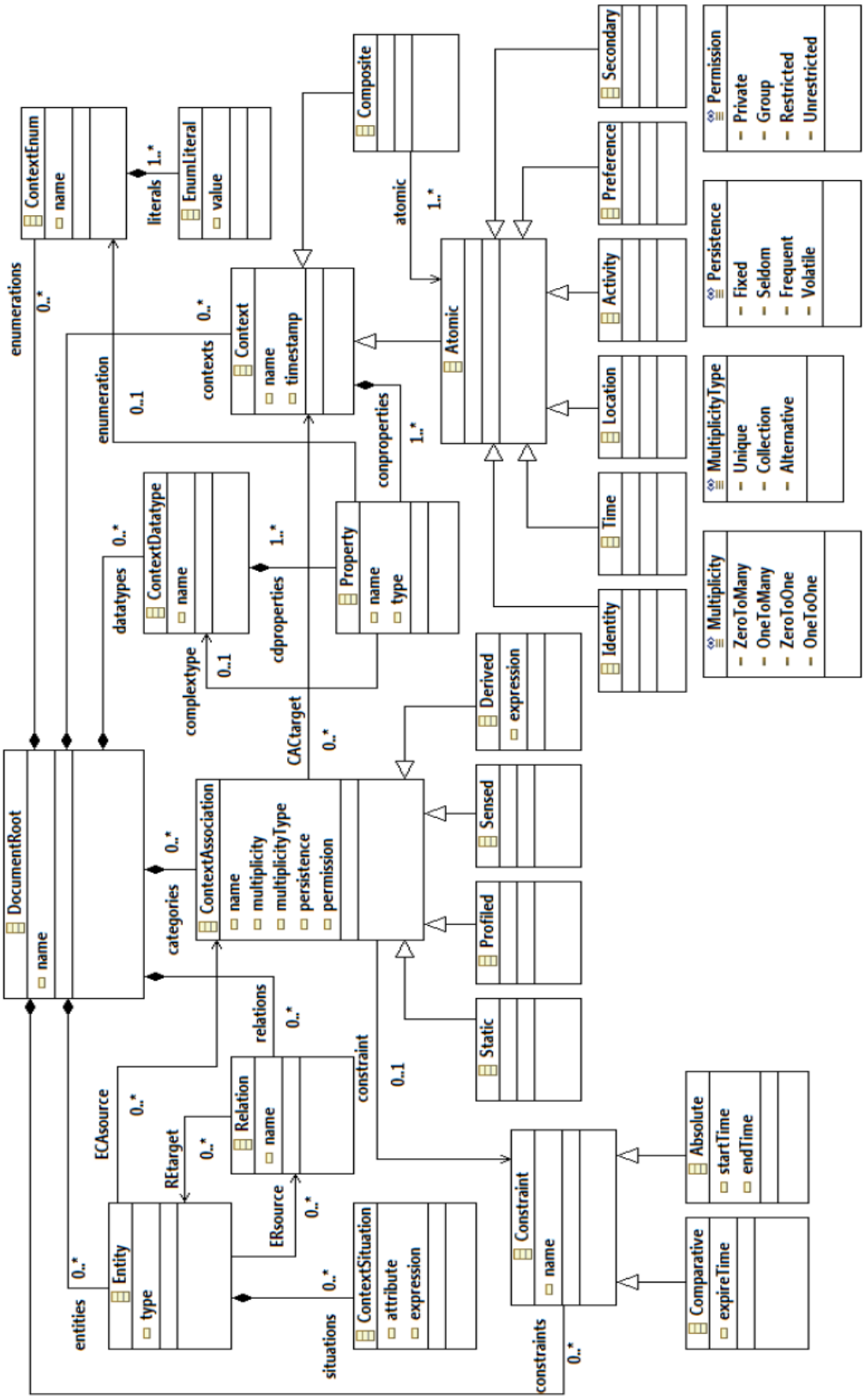


Figure 4.3: Context Modelling Language metamodel.

Primarily, the *entities* aggregation determines that the context model may include (0...*) instances of real-world entities; *i.e. objects*. The *Entity* metaclass is an abstract representation of real-world entities and contains the *type* property that designates the category of the entity; *e.g. Person, or Device*. Furthermore each 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 translated as a specific behaviour bound to the two entities. For instance if a person owns a specific device, this ownership relationship (i.e. Java function) is defined via the *Relation* metaclass; *e.g. Person → owns → Device.*

Furthermore, entities contain one or more situations as depicted by the situations aggregation. Contextual situations are defined via the *attribute* and *expression* properties of the *ContextSituation* metaclass. The *attribute* property defines a *Boolean* variable that allows reasoning about the occurrence or not of the contextual situation. In particular, the reasoning rule is defined in the form of an OCL constraint using the *expression* property. Hence, on the basis of the specified constraint the *Boolean* variable is logically evaluated to either true or false, which designates the validity of the contextual situation.

Apart from the relationships, each entity is associated through the *ContextAssociation* metaclass and the *ECAsource* and CAC*target* associations to a variety of context information. *ContextAssociation* is defined as an abstract parent metaclass from which the *Static, Profiled, Sensed and Derived* metaclasses inherit their properties. The primary property of the abstract metaclass denotes the name of the context source. In addition the *multiplicity* property designates a collection of context information in accordance to the multiplicity value obtained from the *Multiplicity* enumeration. This denotes in particular

the number of copies of context information that can be obtained by the context source. Moreover, the *multiplicityType* property determines the number of simultaneous valid occurrences of context information and obtains its value from the *MultiplicityType* enumeration. For instance, in the case that the *Unique* enumeration literal is defined this denotes that merely a single instance of this information is valid at any given time. In contrast if the *Alternative* enumeration literal is defined, then a choice between different instances of context information is applicable.

The *persistence* property is bound to the *Persistence* enumeration, which describes the frequency with which context is subject to change. Therefore the value of the persistence property describes the volatility of context information associated with a particular context source. For example, if the persistence property of the context source is set to *Fixed*, the context information associated with the source is invariable during its lifetime. Typically, information contained within a context repository has a fixed persistence since this information remains unaltered for long periods of time.

The *permission* property discloses the access restrictions imposed upon context information. In fact, the values defined via the *Permission* enumeration show the different access restrictions that can be imposed upon context information in order to safeguard the privacy of the user. Consequently, the definition of a context source that associates a person to sensitive information, such as emails related to his work, necessitates imposing the *private* value for the property. Hence, an authentication method is required to be implemented in order to validate the credentials of the person that requests access to confidential information. Correspondingly, the *Group* enumeration literal designates that context information can be accessed by a team of individuals,

which are granted specific access permissions. Moreover, the *Restricted* value determines that permissions are given explicitly to designated persons in order to restrict the access to the information. Finally the *Unrestricted* value determines that any person can access the associated context information without any restriction being imposed.

Besides the common context source properties, the *Derived* metaclass includes an *expression* property that is used to define the dependence of context information on other context information. The derivation rule is defined in the form of an OCL constraint and determines the relation between different context information. More specifically, the expression describes that the change in particular context information causes the alteration of interrelated context information. Consequently, the derivation rule allows to reason, on the basis of the context-based event, as to the proactive service behaviour that must be triggered.

Moreover, each context source is associated via the *constraint* relationship to the abstract *Constraint* metaclass that defines a time constraint for the specific context source. The temporal constraint allows reasoning about the validity of context information obtained from the source. Essentially the *Absolute* and *Comparative* metaclasses facilitate modelling time constraints and provide the capability to determine the soundness of information on the basis of time. The absolute temporal constraint supports the definition of a fixed time interval, designating both the starting and expiration time in which context information are valid. Conversely, the comparative constraint depicts that context information are valid from the time it was obtained from the context source until the defined expiration time.

The diverse sources defined in the context model aim primarily to associate entities with context information and describe how this information can be managed effectively. In terms of the context model, information is defined as an instance of the *Context* abstract metaclass and can be either *Atomic* or *Composite* in accordance to the generalisation relationship. The *Atomic* context is also defined as an abstract metaclass in the metamodel since it is extended by the *Identity, Time, Location, Activity, Preference and Secondary* context metaclasses.

Each atomic context represents an information object that is characterised by the *name* property and contains basic properties *(i.e. conproperties)* defined via the *Property* metaclass. Properties can be modelled as primitive datatypes *(e.g. String, Boolean)* using the *name and type* attributes of the metaclass. In addition properties can be modelled as complex datatypes via the *ContextDatatype* metaclass, which is related to the *Property* metaclass through the *complextype* association. As can be realised from the metamodel a cyclic relationship exists since each complex *ContextDatatype* is composed by various primitive properties as denoted via the *cdproperties* aggregation. Furthermore, a property can be associated to an enumeration rather than a complex datatype, as depicted by the *enumeration* relationship. The *ContextEnum* metaclass allows defining enumerations in the context model using its name property and the aggregated *EnumLiteral* metaclasses; *i.e. value property*. Finally the *Composite* context element extends the *Context* metaclass and facilitates the definition of complex information using both simple properties and atomic context information.

The definition of the domain context metamodel concludes the primary step of the MDD methodology since the abstract syntax of the CML is defined. Although the metamodel

definition includes the elements, relationships and properties required to define unambiguously a context model, invalid metamodel instances can be still defined by the designer. Consequently, in accordance to Step 2 of the methodology domain rules need to be identified and imposed onto the CML.

4.2.3 Context Modelling Language Constraints Definition

The metamodel-level constraints imposed onto the context metamodel definition provide the capability to validate the context models prior to undertaking the corresponding implementation specific phases. These constraints are defined with the aid of the GMF mapping editor and facilitate the validation of context models using the modelling editor of the CMF. The following constraints comprise examples obtained from the complete set of OCL constraints imposed onto the CML and presented in Appendix E.

context Entity

inv: if not self.type.oclIsUndefined() **then self**.type.size() > 0 **else self**.type.oclIsInvalid() **endif**

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

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

inv: not Entity.allInstances()->exists(e1: Entity | e1.ERsource.REtarget->exists(e2:Entity | e2 = e1))

The primary group of OCL expressions targets the *Entity* metaclass and provides the capability to enforce the correct definition of context entities. Foremost, the initial invariant OCL constraint provides the capability to validate that the *type* property of the *Entity* metaclass is properly defined. The constraint ensures via the conditional statement that when the *type* property is defined, the size of its *String* literal should be greater than zero. In the case the *type* property is *null*, rather than an empty *String*, the *else* statement ensures that the erroneous definition is detected and a constraint violation is raised.

Moreover, the second constraint allows querying the entire set of entity instances and validating that each entity *type* property is uniquely defined in the model. This restricts the context model and prohibits the definition of duplicate entity instances. The third and the fourth invariant constraints are complementary to each other and aim to restrict the definition of cyclic relationships between entities. The leading constraint allows querying the entire set of entity instances and checking if cyclic relationships exist between different entities. In addition, the latter constraint prohibits the definition of the same entity as the source and the target of the relation association.

```
context Context
    inv: if not self.name.oclIsUndefined() then self.name.size() > 0 else self.name.oclIsInvalid() endif
    inv: Context.allInstances()->forAll(c1: Context, c2: Context | c1 <> c2 implies c1.name <> c2.name )
    inv: self.conproperties->forAll( p1: Property, p2: Property | p1 <> p2 implies p1.name <> p2.name )
    inv: 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 )
```

Following, the second group of OCL expressions targets the *Context* metaclass. The primary rule facilitates the definition of valid names for context information instances, similarly to the aforementioned case of the entity constraint. Respectively, the second constraint provides the capability to guarantee that two or more context instances cannot have the same *name* property definition. Furthermore, the validity of instances of the *Property* metaclass is verified through the third and the fourth constraint. The third rule is applied onto the *Context* metaclass, rather than the *Property* metaclass, since properties of different context instances are permitted to have the same *name* attribute definition. In contrast properties included in the same context instance cannot have the same *name*. The final constraint restricts further the context model definition since it ensures that the *type* attribute of the *Property* metaclass is set to one of the following: (i) primitive datatype,

(ii) complex datatype, (iii) enumeration. Hence, the definition of context properties is restricted via these two final constraints to valid *Property* instances.

The definition of metamodel-level constraints provides a consistent abstract syntax for the CML and provides the capability to define valid context models. Subsequently, the third step of the MDD process is performed that allows extracting the concrete syntax of the CML from the domain context metamodel; see Figure 4.2. The graphical and tooling editors are then utilised to customise the concrete syntax definition and optimise the visual notation of the CML. Figure 4.2 illustrates also the merging of the distinct metamodels into a common mapping metamodel that enables the automatic generation of the CMF in the form of Eclipse plug-ins. The generation of the CMF is driven by EMF Java Emitter Templates (JET) and GMF Xpand templates, which transform effectively the mapping metamodel to the required implementation technology. The generated CMF comprises of the Context Modelling Language as its core component and a supporting context modelling editor for the definition and validation of context models.

Figure 4.4 illustrates a running instance of the CMF that presents the context modelling perspective. The context model presented in the figure showcases the violation of several of the imposed OCL constraints. Consequently, the designer is presented with diagnostic information on the errors detected in the model definition, using the validation decorators of the modelling editor. The designer can therefore undertake the appropriate steps to rectify the problems, in order to guarantee that the correct implementation is generated from the context model. More specifically, the context-awareness characteristic defined in the context model at the static compile time is transformed to the corresponding implementation that supports the adaptation of the pervasive service during runtime.

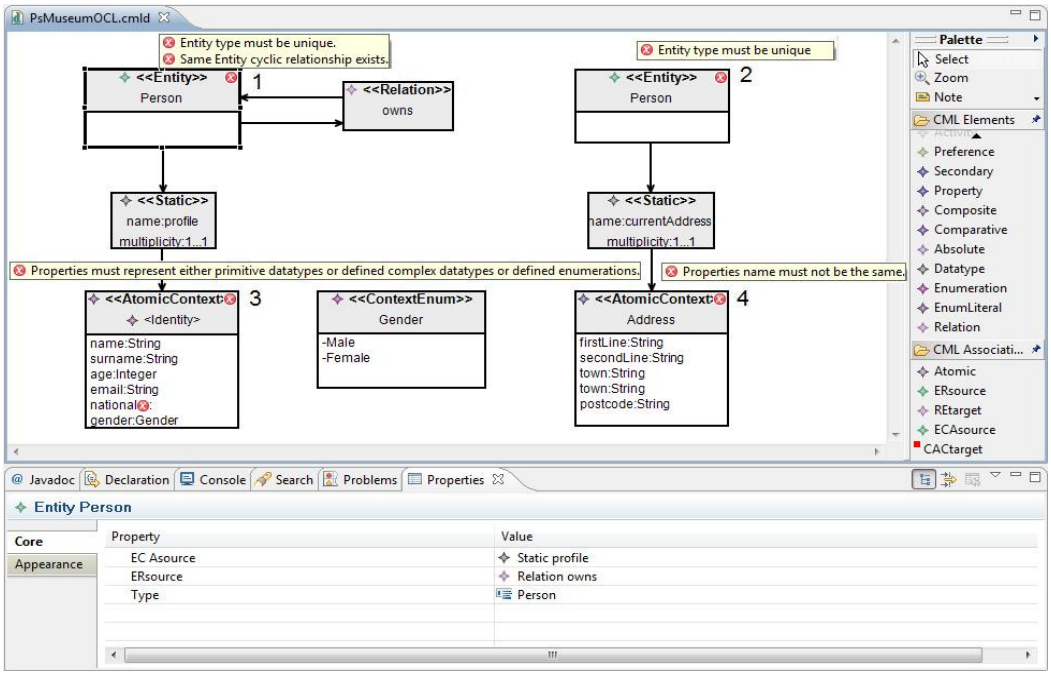


Figure 4.4: Running instance of the Context Modelling Framework.

The example context model of Figure 4.4 showcases initially a constraint violation (1) that indicates the existence of duplicate entities in the model definition. In particular, the second constraint violation (2) designates the actual duplicate entity instance defined in the model. Furthermore, the primary constraint violation (1) indicates also that a cyclic relationship exists for this particular entity instance. The subsequent constraint violation (3) indicates that the *nationality* property of the *Identity* atomic context is not defined appropriately, since it is not specified as a primitive datatype, complex datatype or as an enumeration. On the contrary, the *gender* attribute of the *Identity* atomic context is correctly defined as an enumeration that obtains its values from the *Gender ContextEnum* instance. Concluding, the final constraint violation (4) indicates to the designer that two properties with the same name (*i.e. town*) are defined for the *Address* atomic context.

4.2.4 Platform Specific Code Generators Templates Definition

In this subsection the defined code generators are presented in the form of templates, which are defined using the *Xpand* language of the *oAW* component. The definition is performed using an identical procedure to the one presented in Chapter 3 for the survey code generator. More specifically, the *Java and J2ME template-based code generators* are defined to support the transformation of context models to the respective context-aware service implementation. Figure 4.5 presents the process that guides the generation of the service implementation from the context model, in accordance to the specific set of templates (*i.e. Java or J2ME*) loaded by the generator component.

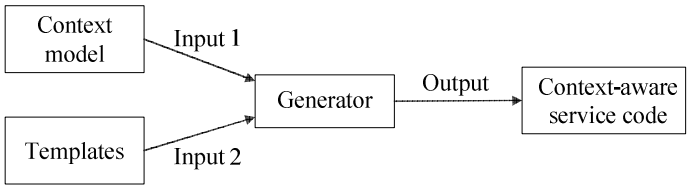


Figure 4.5: Context-aware service implementation generation process.

Each set of templates describes respectively the code structure of the context-aware platform specific implementation to be generated. The J2ME templates are defined in accordance to the CML metamodel definition and describe the code patterns (i.e. code architecture), which are enriched upon transformation with information obtained from the context models. Similarly, the defined Java templates represent the code architecture of the context-aware service implementation in terms of the Java implementation technology.

In order to achieve the interpretation of diverse context models, the templates include generic references to the elements, properties and relationships described in the CML metamodel. More specifically, the templates describe the mapping of the modelling language to the operational semantics of the corresponding implementation technology.

This drives the generation of the implementation, which acts as the bridging point that allows context to be utilised by services [36]. The generated implementation serves common tasks for managing a context repository such as querying, administrating and distributing context information to services. Consequently, these context management mechanisms built in at the service creation stage will be triggered during service execution to provide inherent and therefore much enhanced service adaptability.

In this chapter certain parts of the J2ME templates are selected and explained to illustrate how the transformation of context models to service code is effectively accomplished. Note that, the description of Java templates is omitted since their definition follows an equivalent logic. Figure 4.6 illustrates using pseudocode a part of the *ContextAssociation* template presented in Appendix F, which is responsible for transforming instances of context sources to respective J2ME code.

```
1.  DEFINE Root FOR DocumentRoot
2.  EXPAND Entity FOREACH entities
3.  ENDDEFINE
4.  DEFINE Entity FOR cml::Entity
5.  FOREACH entity context association AS cSource
6.                      |
7.                      |
8.  FOREACH cSource context property AS cProperty
9.  Read properties from model and generate property initialisation statement
10. ENDFOREACH
11.                     |
12.     IF cSource.multiplicity == "0…1" OR cSource.multiplicity == "1…1"
13.         Read properties from model and generate the corresponding method
14.     ELSEIF cSource.multiplicity == "0…*" OR cSource.multiplicity == "1…*"
15.         Read properties from model and generate the corresponding method
16.     ENDIF
17.                     |
18. FOREACH cSource context property AS cProperty
19. Read properties from model and generate property accessor method
20. ENDFOREACH
21.                     |
22.                     |
23. ENDFOREACH
24. ENDDEFINE
```

Figure 4.6: Context Association template pseudocode definition.

The primary section of the pseudocode (i.e. lines 1-4) specifies the *DocumentRoot* metaclass as the root element of the definition, so as to expand each *Entity* element contained in the *entities* collection; *i.e. aggregation*. This provides the capability to bind the context of the template definition to each entity of the metamodel. Furthermore, it allows accessing the elements and properties associated with every instance of the *Entity* metaclass. Subsequently, a loop is defined at lines 5-23, which allows iterating through the context sources associated with each entity instance. Hence, every context association is declared as a variable *cSource*, in order to be able to query, load and read the properties associated to the context source.

For instance, this provides the capability to logically compare the value of the *multiplicity* property defined in the context model, so as to direct the code generation in accordance to the satisfied logical condition; *i.e. lines 12 and 14*. In the case that the *multiplicity* property is defined either as *"0…1" or "1…1"* the corresponding method is generated that provides the functionality required for managing a single copy of the context information. On the contrary, in the case that the multiplicity property is defined either as *"0…*" or "1…*"* the functionality of the generated method allows managing multiple copies of the associated context information.

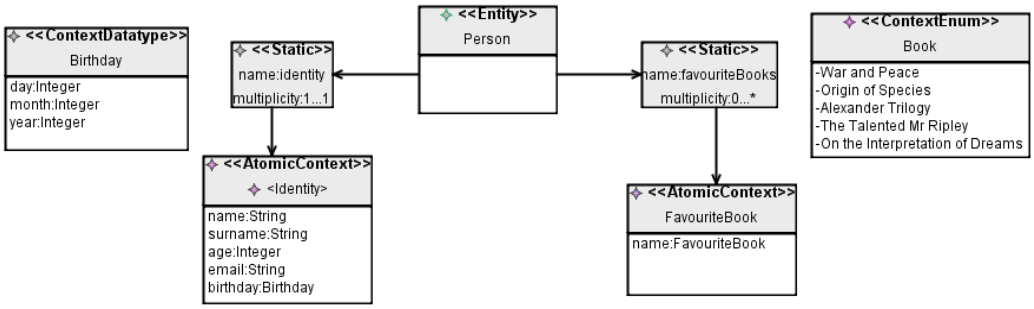


Figure 4.7: Example context model.

Furthermore, the pseudocode definition of Figure 4.6 illustrates the navigation to nested elements of the context association, via the loops defined at lines 8-10 and 18-20. Hence, each context association property is declared via the primary statement as a variable *cProperty*, which denotes the properties of the information associated to the respective context source. Moreover, the variable initialisation statement and the corresponding accessor method are generated for every property included in the respective context information; *i.e. lines 9 and 19*. The generated accessor method provides the capability to access the context property from additional classes of the generated implementation.

```
public PersonIdentity getIdentity() {
        int i = 0;
        int j = 0;
        pi = new PersonIdentity();
        b = new Birthday();
        try {
                if (rsm == null)
                        rsm = new RecordStoreManagement(atomic_context);
                int records = rsm.getSize();
                while (i < records) {
                        pi.setName(rsm.get(name + j));
                        i++;
                        pi.setSurname(rsm.get(surname + j));
                        i++;
                        pi.setAge(rsm.get(age + j));
                        i++;
                        pi.setEmail(rsm.get(email + j));
                        i++;
                        temp = new String[b.size];
                        temp = sr.renderLine(rsm.get(birthday + j), b.size);
                        b.setDay(new Integer(Integer.parseInt(temp[0])));
                        b.setMonth(new Integer(Integer.parseInt(temp[1])));
                        b.setYear(new Integer(Integer.parseInt(temp[2])));
                        pi.setBirthday(b);
                        i++;
                        j++;
                }
        } catch (RecordStoreException rse) {
                rse.printStackTrace();
        }
        return pi;
}
```

Figure 4.8: Generated method for identity context association.

Figure 4.7 presents an example context model that assists the illustration of the code generation process, on the basis of the aforementioned pseudocode definition. The context model includes the *Person* entity instance that is associated with the *Identity and FavouriteBook* atomic context instances. Although both associations are defined as static sources of context information, the *multiplicity* property is different for the two individual context sources. In the case of the *identity* context association the *multiplicity* property is defined as *"1...1"*. Consequently, the primary logical expression of the pseudocode definition is evaluated to true *(i.e. line 12)* and the method illustrated in Figure 4.8 is effectively generated. The method comprises of the necessary functionality that allows managing a single instance of the *Identity* context information.

```
public Vector getFavouriteBooks() {
        int i = 0;
        int j = 0;
        Vector favouriteBooks = new Vector();
        try {
                if (rsm == null)
                        rsm = new RecordStoreManagement(atomic_context);
                        int records = rsm.getSize();
                while (i < records - 1) {
                        String[] temp = null;
                        StringRenderer sr = new StringRenderer();
                        pf = new PersonFavouriteBook();
                        b = new Book();

                        b.setState(rsm.get(name + j));
                        pf.setName(b);
                        i++;
                        favouriteBooks.insertElementAt(pf, j);
                        j++;
                }
        } catch (RecordStoreException rse) {
                rse.printStackTrace();
        }
        return favouriteBooks;
}
```

Figure 4.9: Generated method for favourite books context association.

On the contrary, the multiplicity property of the *favouriteBooks* context association is set accordingly as *"0...*"*, which designates that multiple instances of the *FavouriteBook*

context metaclass can be associated to every person. This denotes in specific that each person can have zero to many favourite books. Therefore, the transformation of the context model generates correspondingly the context management method presented in Figure 4.9, since the second logical condition is evaluated to true. As a result the method returns a list of favourite books contained within a *Vector* object, rather than a single instance of the *FavouriteBook* implementation class.

```
public static String getName() {
        return name;
}
                |
                |
public static String getBirthday() {
        return birthday;
}
```

Figure 4.10: Generated accessor methods for identity context association.

The transformation of the *identity* context source generates also the accessor methods partially illustrated in Figure 4.10. These accessor methods are produced via the translation of the properties of the *Identity* context information; *i.e. lines 18-20*. The accessor methods return *String* literal references for each property, so as to facilitate the generated context management classes in obtaining the appropriate values from the *RecordStore*; *i.e. J2ME context repository*. Apart from the accessor methods the initialisation statements of the variables are also generated for the context association implementation class; *i.e. lines 8-10*.

```
1.  DEFINE Root FOR DocumentRoot
2.  EXPAND Entity FOREACH entities
3.  ENDDEFINE
4.  DEFINE Entity FOR cml::Entity
5.                  |
6.  FOREACH entity association context information AS eacInfo
7.  Read properties from model and generate logical statements for context events
8.  ENDFOREACH
9.                  |
10. ENDDEFINE
```

Figure 4.11: Context Receiver template pseudocode definition.

Figure 4.11 illustrates an additional example of a J2ME template that presents part of the actual *ContextReceiver* template definition presented in Appendix G. In particular, the definition is bound to each entity described in the context model; *i.e. lines 1-4*. This is performed in order to identify and process information related via the context sources association to every entity element. The loop defined at lines 6-8 provides the capability to iterate through the collection of context information associated to a particular entity. Hence, different logical code statements are generated accordingly for different context information, so as to facilitate the detection of context change events. More specifically, the generated logical code statements allow identifying which context information was altered and fire consequently the appropriate action.

```
public class PersonReceiver {
        private Object _context_object;
        private Context _context = null;
        private Vector _listeners = new Vector();
        public synchronized void receivePersonContext(Context _context,
                                        Object _context_object) {
                this._context_object = _context_object;
                if (_context == Context.Person_Identity) {
                        _ this._context = Context.Person_Identity;
                        _fireContextEvent();
                }
                else if (_context == Context.Person_FavouriteBook) {
                        _ this._context = Person_FavouriteBook;
                        _fireContextEvent();
                }
        }
        public synchronized void addContextListener(ContextListener cl) {
                        _listeners.addElement(cl);
        }
        public synchronized void removeContextListener(ContextListener cl) {
                        _listeners.removeElement(cl);
        }
        private synchronized void _fireContextEvent() {
                        ContextEvent context_event = new ContextEvent(this, _context);
                        Enumeration listeners = _listeners.elements();
                        while (listeners.hasMoreElements()) {
                                ((ContextListener) listeners.nextElement()).contextAltered(
                                context_event, _context_object);
                        }
}
```

Figure 4.12: Context Receiver generated object.

Figure 4.12 showcases the code generated from the transformation of the example model of Figure 4.7, on the basis of the context receiver template definition. The generated J2ME code comprises of the *receivePersonContext* method that provides the capability to detect changes in context information. Note that, the method is defined as *synchronized* because it shouldn't be possible for two method invocations to interleave. This means that since different context events can occur at any given time, subsequent invocations of the method are suspended, until the preceding method invocation is complete.

The synchronized method includes conditional statements that permit to detect the change in context information. Foremost, the *Context* object contains static and final variable references (*e.g. Person_Identity*), which facilitate the comparison with the input parameter (*i.e. _context*) of the method. Consequently, the global *_context* variable of the *PersonReceiver* object is set to the input variable reference of the respective context information. The *_fireContextEvent* method is subsequently invoked and the corresponding *ContextEvent* object is created using the *_context* variable of the *PersonReceiver* object. Consequently, the *contextAltered* method of the appropriate *ContextListener* object is invoked and the action associated with this particular context event is performed.

Another important attribute presented in the example model of Figure 4.7, is the *permission* property included in the *ContextAssociation* metaclass definition. Although the property does not appear on the actual graphical model representation, it is apparent and can be defined via the properties view of the CMF. For this particular context model the *permission* property is defined respectively as *private* and *unrestricted* for the *identity* and *favouriteBooks* context sources. Figure 4.13 illustrates a section of the generated

method (*i.e. manageContextInformation*) of the *Person* object, which is responsible for managing information in accordance to the specified context association properties.

```
if (_sourceIdentity.equals(source) && _pi.equals(context)) {
        String multiplicity = Identity.getMultiplicity();
        String multiplicityType = Identity.getMultiplicityType();
        String persistence = Identity.getPersistence();
        String permission = Identity.getPermission();
        // recordstore exists
        if (exists == true) {
                // TODO
                throw new UnsupportedOperationException(
                        "Operation not supported yet. Implementation required");
        }
        // recordstore does not exist
        else {
                // TODO
                throw new UnsupportedOperationException(
                        "Operation not supported yet. Implementation required");
        }
}
```

Figure 4.13: Part of the generated code for the person entity.

The code illustrates that by accessing the variables of the generated *Identity* context association object the values specified in the context model can be queried, obtained and stored into the corresponding local variables (*i.e. persistence, permission*). Therefore, the developer can manually define the appropriate conditional statements, in order to verify the permissions of each individual. In the case that *private* access restrictions are imposed for this particular context information (*i.e. identity of the person*) the user must be authenticated prior to accessing the information. On the contrary, if *unrestricted* access restrictions are imposed then it is not necessary to delegate a call to the method that verifies the credentials of the users.

Note that, the generated service code comprises of a *ResourceManagement* class that allows the developer to impose the appropriate information access behaviour. This is possible since a specific method of the generated *ResourceManagement* class facilitates the authentication of the user by accessing the resource that stores the users' credentials.

In contrast, a supplementary method of the *ResourceManagement* class provides direct access to context information bypassing the authentication stage.

The defined templates and their complementary extensions facilitate the transformation of the complete set of elements, properties and relationships specified in the context models; similarly to the aforementioned cases. Via the transformation of the models a large percentage of the context-aware service implementation is effectively generated. The capability is also provided to the developer to utilise the generated objects and methods in order to extend easily the implementation and deliver the complete service functionality. Therefore, by blending code generation and manual implementation of the complex functionality, the time, effort and cost for delivering the context-aware service implementation are effectively reduced.

4.3 Overview of Graphical User Interface Design

The development of graphical user interfaces is a taunting but essential task in software service creation. In particular, the service creation overheads are largely increased while developing the same software service for miscellaneous platforms with different restrictions and requirements [96]. This applies explicitly to pervasive services, since the complexity of implementing GUIs is increased due to the advanced user-service interaction and the service portability requirements. Hence, the capability should be provided to define GUIs in an abstract manner, which can support the explicit and/or implicit interaction of the user with the service [97], [98]. Furthermore, the generation of diverse implementations from the same GUI designs should be satisfied, so as to enable the deployment of pervasive services onto different execution platforms.

Undoubtedly, GUIs are imperative to the interaction of the user with the pervasive service, in order to successfully execute the required computing tasks. The definition of a modelling framework that facilitates the precise specification of GUIs in the form of models is considered as a prerequisite in the pervasive service creation process. As denoted also in [99], the absence of continuous tool support to guide the design of GUI models and the automatic generation of the respective code from those models is the main disadvantage of MDD with regards to GUI development.

This imperative prerequisite is satisfied in this work via the MDD methodology and the IMDE, which guide and provide unambiguous tool support to facilitate the definition, validation and code generation phases. Subsequently, the Presentation Modelling Framework is generated in order to explicitly support the design and validation of GUI models and achieve their transformation through an automated process to platform specific code. This allows generating a considerable part of the required GUI code for diverse platforms, simplifying as a result the implementation of graphical user interfaces.

4.4 Presentation Modelling Framework

4.4.1 Presentation Modelling Domain Requirement Analysis

Service portability refers to the capability to deploy the same software application on diverse platforms performing minimal changes to the actual service implementation code. The portability requirement is gratified in this work via the specification of the context, presentation and process models and their transformation to implementation code for different platforms. In particular, the generation of the GUI service code from the presentation model is considered as the most essential aspect for achieving the portability requirement. This is because of the highly diverse restrictions and requirements that

different platforms impose, specifically in terms of the implementation of GUIs. Fundamentally, the restrictions and requirements imposed by different technologies when dealing with GUI development restrict the capability to implement and deploy rapidly the same pervasive service on diverse platforms.

Apart from the service portability, the advanced interaction of the user with the pervasive service requires to design and implement GUIs using an abstract approach. This is due to the fact that the user should be capable to utilise GUIs to execute explicitly computing tasks. Moreover though, implicit information (i.e. sensed context) could trigger dynamic actions that require utilising GUIs, to perform tasks such as notifying the user of a context event or allowing the user to respond to a specific context event. Consequently, this seamless interaction of the user with the service describes a fundamental requirement that neseccitates an abstract approach for the development of advanced GUIs.

In order to satisfy these objectives it is imperative to provide an extensible model-driven approach that allows specifying unambiguously GUIs in a platform-independent manner. Consequently, the Presentation Modelling Language should consist of abstract concepts that have a direct mapping to different platform specific implementation technologies. For instance, the top level component for implementing GUIs in Java is provided either via the *javax.swing.JFrame* or the *java.awt.Frame* implementation class. Similarly, the *javax.microedition.lcdui.Display* class represents the top-level GUI component in terms of the J2ME technology. Hence, the prerequisite is to devise an abstract definition for this top-level component, irrespective of the actual implementation platform. Moreover, the abstract component should include properties that can be mapped to corresponding variables of the Java and J2ME implementation.

The second graphical component included in the PML definition is encountered in both platforms and can be described as a container that provides the capability to place other graphical components within it. In the case of the Java technology this component is implemented either via the *javax.swing.JPanel* class or the *java.awt.Panel* class. Equally, the respective container component of the J2ME technology is implemented using the *javax.microedition.lcdui.Form* class. Furthermore, the main properties of the container components are identified in order to satisfy the key requirements in terms of both platforms. Finally, the capability to place containers within top-level abstract components is also considered, so as to provide an abstract and coherent container definition.

In addition the most important and widely used components of the two platforms are identified, so as to facilitate the complete and unambiguous specification of GUIs. Preferably, the components that represent comparable graphical concepts are selected from the distinct implementation technologies. For instance, the *java.awt.Label* and *javax.microedition.lcdui.StringItem* implementation classes serve a similar purpose since both represent a component for displaying a single line of read-only text. Another example is the *TextField* class that has identical naming for both platforms and serves the task of editing a single line of input text. An additional requirement considered is the capability to place these components within the container components.

Apart from the specification of graphical components the potential to define different properties for those components should be essentially provided. Due to the diversity of the graphical properties of each platform-specific component, it is imperative to avoid imposing restrictions on the definition of properties in the presentation model. In this way the flexibility is provided to specify miscellaneous properties in the presentation model,

on the condition that the definition and transformation of these properties is supported by the modelling framework. The flexibility and extensibility of the PML is preserved via the definition of OCL constraints that control the definition of GUI properties.

4.4.2 Presentation Modelling Language Metamodel Definition

The identification of the fundamental graphical requirements aids the specification of a coherent and unambiguous Presentation Modelling Language (PML). The PML is defined in the form of a metamodel, which is considered the foundation for the generation of the Presentation Modelling Framework. Figure 4.14 illustrates the PML metamodel definition that comprises of the essential elements, associations and properties that assist the specification of platform-independent GUIs.

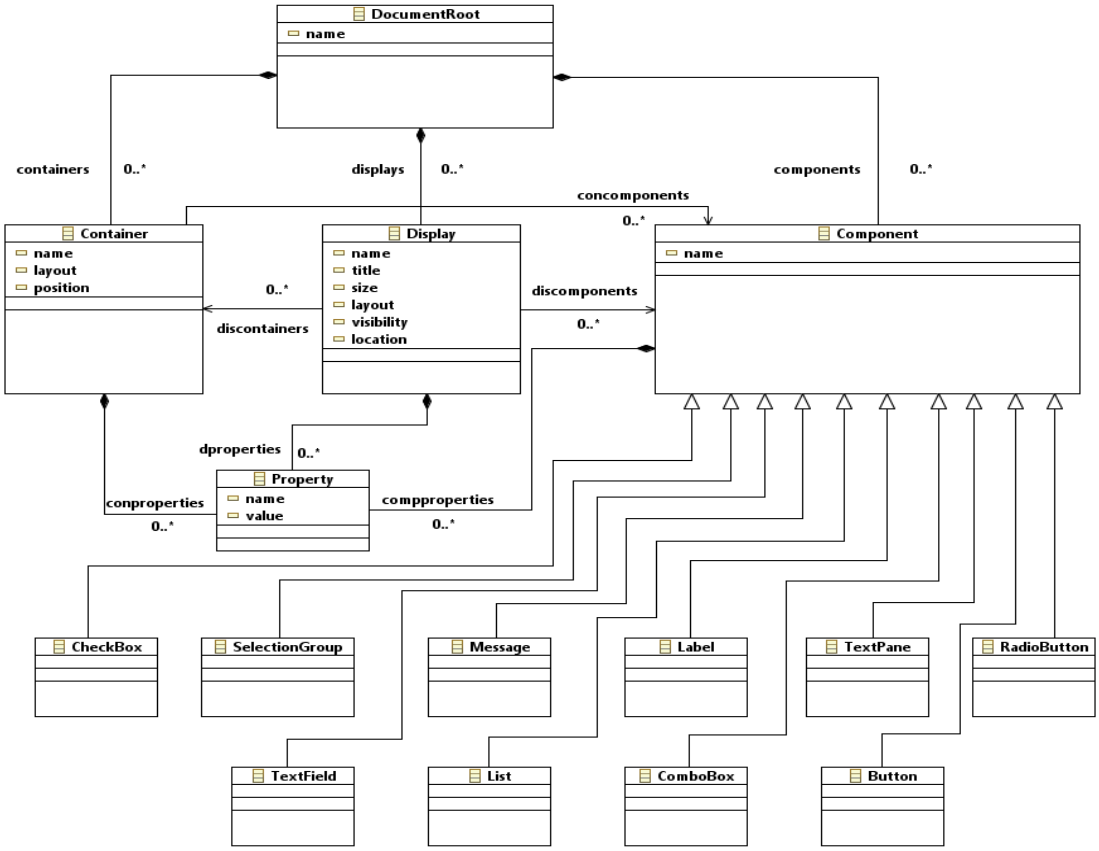


Figure 4.14: Presentation Modelling Language metamodel.

Following the naming convention used throughout this thesis, the top level element is defined as the *DocumentRoot* metaclass and represents the GUI model. The aggregations illustrate the containment relationships of the root metaclass with the graphical elements, which can be included in the presentation model specification. Foremost, the *displays* aggregation illustrates that within each presentation model the definition of various instances of the *Display* metaclass are permitted. Despite that fact, an OCL constraint is specified (see next subsection) that restricts the definition of the model to a single display instance. This is performed in order to reduce the overall complexity of the designed presentation models and improve also their understanding.

Each display element includes common properties that are namely the *name, title, size, layout, visibility and location*. The primary property depicts the name of the display and the secondary property defines the title that will be displayed onto the main frame of the service. Moreover, the *size* property determines the actual size of the frame while the *layout* property designates the actual positioning of the containers onto the frame. In the case of the J2ME technology only the *name* and *title* properties are important since the J2ME API provides a default layout manager, which handles explicitly the outline of the components on the mobile device display. Therefore, properties such as *location* and *size* are merely important for the generation of the Java implementation from the model.

The *containers* aggregation relationship depicts the components that act as the carriers of the secondary components included in the model definition. In terms of the Java technology the container components are represented by the *javax.swing.JPanel* object, while in the case of J2ME they are represented as *javax.microedition.lcdui.Form* objects. Each container element comprises of the *name*, *layout* and *position* properties, which are

fundamental for controlling the naming and the appearance of the component. As in the case of the display element the appearance specific properties (*i.e. layout, position*) are only required for the Java implementation.

Furthermore, the *components* aggregation denotes that each model definition contains various graphical components. Hence, the *Component* element is defined as an abstract metaclass from which the child elements, namely *Message, Label, Button, TextPane, RadioButton, ComboBox, CheckBox, TextField, List and SelectionGroup*, inherit their properties. For instance, every child element inherits a unique property that defines the name of each graphical component. Each child component is considered as an abstract representation of a corresponding real-world component of the J2ME and Java platforms. For example, the *TextPane* metaclass represents text components that allow to display and/or edit large volumes of text. In the case of the Java platform *TextPane* elements correspond to *javax.swing.JTextPane* objects, while in the case of the J2ME platform these elements symbolise *javax.microedition.lcdui.TextBox* objects.

Apart from the metaclasses, the *discontainers, discomponents and concomponents* associations are also defined in the metamodel. These associations designate the relationships between the display, container and component elements. Primarily, the *discontainers* association denotes that each display element may include zero to many containers. This translates, in the case of the Java platform, as the containment of various *javax.swing.JPanel* containers within a particular *javax.swing.JFrame* component. Moreover, the *discomponents* relationship depicts that each display element can contain one or more secondary components. Finally, the *concomponents* association defines the containment relationship between the container and component elements. In the case of

the Java technology this association symbolises the containment of components (*i.e. javax.swing.JLabel*) within container elements (*i.e. javax.swing.JPanel*).

Concluding, the key element of the PML is defined as a *Property* metaclass, which allows specifying miscellaneous properties for each display, container and component element. The aggregations *dproperties, conproperties and compproperties* depicted in the metamodel indicate clearly that each element can contain various properties. In this thesis the specification of GUI model properties is performed using a flexible and extensible approach, which does not restrict by any means the applicability of the PML.

In particular, different properties are defined using the *name* property that allows specifying *keywords*, which are transformed accordingly by the platform-specific code generator. Moreover, it is possible to ensure through OCL constraints that the designer of the model will simply define the keywords that are supported by the current version of the PML. Such a flexible and extensible approach is fundamental for the effective specification of GUI models and the generation of the implementation, due to the numerous and diverse graphical properties of each platform. Moreover, the *value* attribute defines the value of the property and is manually inputted by the designer during the model definition. For instance, the value property may represent the text that will appear on a Label component or the coordinates that depict the location of the Label component.

4.4.3 Presentation Modelling Language Constraints Definition

The definition of the presentation metamodel provides a well-defined PML that allows the specification of graphical user interfaces in the form of EMF-based models. In order to ensure the coherency of the modelling language the second step of the methodology is

undertaken, which comprises the identification and definition of domain-specific rules that govern the PML. These constraints are later enforced using the GUI modelling editor of the PMF, so as to guarantee the correctness of the designed models prior to executing the code generation phase. Appendix H illustrates the complete set of OCL constraints identified and imposed onto the presentation metamodel definition. The following example OCL constraints showcase the importance of applying domain-specific rules, so as to define precise and coherent GUI models. The primary constraint targets the *Display* metaclass and restricts the specification of the container's *position* in accordance to the defined *layout* of the display.

```
context Display
    inv: if self.layout = 'default'
        then self.discontainers->forAll(con: Container| con.position = 'CENTER')
            or self.discontainers->forAll(con: Container | con.position = 'EAST')
            or self.discontainers->forAll(con: Container | con.position = 'WEST')
            or self.discontainers->forAll(con: Container | con.position = 'NORTH')
            or self.discontainers->forAll(con: Container | con.position = 'SOUTH')
        else self.discontainers->forAll(con: Container | con.position.toInteger().oclIsTypeOf(Integer))
        endif
```

In specific the domain rule defines that if the *layout* property is set as *"default"* then the *position* property of the associated containers must be set to one of the following values: *(i) CENTER, (ii) EAST, (iii) WEST, (iv) NORTH and (v) SOUTH*. The *"default"* value represents the *java.awt.BorderLayout* class of the Java platform, which controls the layout of the display element and allows placing the containers to the aforementioned positions; *i.e. CENTER*. Moreover, the conditional statement (*i.e. else*) specifies that when the *layout* property is set as "*grid"*, the *position* property should be defined as an *Integer* indicating the index number for placing the container onto the display element. In the current definition of the PML, only the *BorderLayout* and *GridLayout* managers are supported for the specification and transformation of GUI models.

The second OCL constraint shown next is fundamental to the definition of the PML since it provides the flexibility required by a platform-independent GUI modelling approach. In essence the domain rule defines the permitted *keywords* that can be defined as properties of the *display, container and component* elements. Consequently, the PML is restricted to the list of *keywords (i.e. properties)* included in this particular constraint. This provides though the capability to easily enhance and extend the PML, simply by introducing additional keywords to the OCL constraint. Hence, by executing this simple step the PMF can be extended and regenerated without affecting any of the existing presentation models. In this way the flexibility, extensibility and applicability of the PML is preserved, providing also the capability to cope with the diverse restrictions and requirements imposed when defining GUIs for different implementation technologies.

context Property

inv: Property.allInstances()->forAll(p: Property | p.name = 'text' **or** p.name = 'title' **or** p.name = 'message' **or** p.name = 'rows' **or** p.name = 'columns' **or** p.name = 'lineWrap' **or** p.name = 'stringArray' **or** p.name = 'command')

The specification of OCL constraints concludes the definition of the PML and supports the design of coherent and valid presentation models. Having at hand the abstract syntax of the PML, the concrete syntax can be extracted as in the case of the Context Modelling Language; see Figure 4.2. Consequently, merging the abstract and concrete metamodels into a common mapping representation allows generating the PMF in the form of Eclipse plug-ins. Subsequently, the specification and validation of presentation models is possible, on the basis of the defined Presentation Modelling Language.

The example *CurrencyConverter* GUI model presented in Figure 4.15 illustrates the modelling and validation capabilities of the PMF. In specific, the figure presents the PMF runtime and showcases the violation of two fundamental constraints, which should be

resolved so as to evade the generation of erroneous GUI code. The validation decorator trace shown on the Display element presents using an informative message the constraint violation identified in the model definition. In particular the error message designates that the *position* property of the Container element is not set in accordance to the definition of the *layout* property of the Display element.

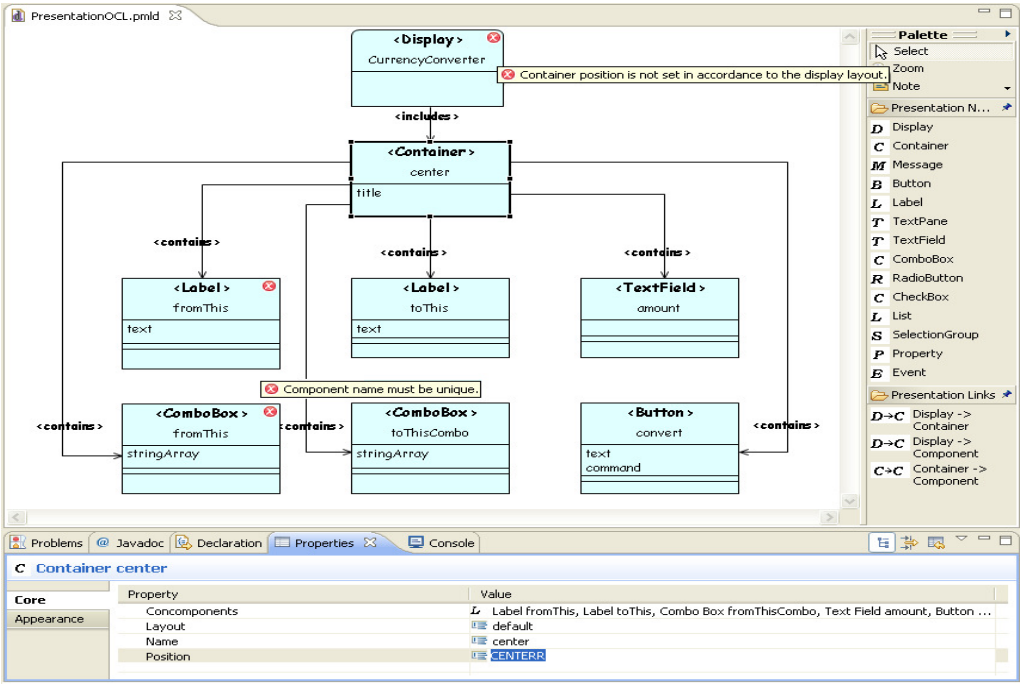


Figure 4.15: Running instance of the Presentation Modelling Framework.

Consequently, the *position* property must be set one of the permitted values since the *layout* property of the *Display* element is defined as "default" (*i.e. BorderLayout*). Moreover, the validation of the example model indicates via the second validation decorator that both the *Label* and the *ComboBox* have an identical *name* property definition. Therefore, different element names should be provided in order to avoid generating the same variable for the two components of the implementation class. The validation phase allows minimising the modelling errors of the designer and ensuring that the operational semantics of the generated implementation are preserved.

4.4.4 Platform Specific Code Generators Templates Definition

The definition of precise template-based code generators is also essential, in order to ensure the correctness of the operational semantics of the generated implementation. The templates introduce the necessary transformation patterns, which facilitate the translation of the abstract information specified in the presentation models. As in the case of the CML, the code generator accepts as inputs the presentation model and the corresponding templates and executes the transformation to obtain the platform-specific implementation code. In particular, J2ME and Java templates have been defined for the transformation of presentation models.

```
1.  DEFINE Root FOR DocumentRoot
2.  EXPAND Display FOREACH displays
3.  ENDDEFINE
4.  DEFINE Display FOR pml::Display
5.                  |
6.  FOREACH container element AS con
7.                  |
8.  FOREACH container component element AS comp
9.          IF comp.metaType == “Label”
10.         Read properties from model and generate the StringItem component
11.         ELSEIF comp.metaType == “Button”
12.         Read properties from model and generate the Command component
13.         ELSEIF comp.metaType == “TextField”
14.         Read properties from model and generate the TextField component
15.         ELSEIF comp.metaType == “ComboBox”
16.         Read properties from model and generate the ChoiceGroup component
17.                 |
18.                 |
19.         ENDIF
20. ENDFOREACH
21. ENDFOREACH
22.                 |
23.                 |
24. ENDDEFINE
```

Figure 4.16: Container element pseudocode definition.

Figure 4.16 illustrates an extract of the J2ME template presented in Appendix I in the form of pseudocode. The initial lines of the pseudocode (lines 1-4) specify the *DocumentRoot* metaclass as the root element of the template definition. This allows

expanding the displays aggregation association and defining the *Display* metaclass as the top-level element of the template definition; *i.e. line 4*. Subsequently, the loop defined at lines 6-21 supports the iteration through the list of containers associated with the display element. Furthemore, it provides the capability to declare each container in the form of a variable; *i.e. con*. The variable declaration provides the capability to query and access the properties and associations of each container element. Subsequently, the containers' aggregation association that refers to the components collection is accessed via the loop definition *(lines 8-20)* and each component is declared as a variable; *i.e. comp*.

Respectively, the component properties and associations can be accessed via the variable reference and the appropriate conditional statements can be defined; *i.e. lines 9-19*. More specifically, each conditional statement provides the capability to check the *object type (i.e. metaType)* of the corresponding component in the iteration and generate accordingly the appropriate implementation code. The properties of every component are defined in the form of *keywords* and drive accordingly the generation of the corresponding operational semantics of the J2ME implementation.

For instance, the logical condition depicted at line 15 verifies that the current component in the iteration is an instance of the *ComboBox* metaclass. Consequently, in terms of the transformation, an equivalent J2ME *ChoiceGroup* component is generated from the abstract *ComboBox* element definition. Moreover, the PML definition and the imposed OCL constraints depict that each *ComboBox* component should include a corresponding *stringArray* property. Consequently, the value of the property defined in the GUI model is also transformed to a respective string array that composes the choices appended to the *ChoiceGroup*.

Table 4.1 illustrates the mapping between the abstract elements of the PML and the specific components of the different technologies. The mapping is unambiguously defined in the J2ME and Java templates and describes how each abstract model element is translated effectively to a corresponding plarform specific component. For example, each model element that represents an instance of the *Button* metaclass is transformed accordingly either to a J2ME *Command* or a Java *JButton* component. Consequently, the appropriate implementation code is generated, which initialises the API object, creates an instance of the object and sets its variables in accordance to the specified abstract properties; *i.e. keywords*.

Table 4.1: The mapping of the abstract elements to J2ME and Java components.

Abstract model element definition	**J2ME MIDP Component API class**	**Java Swing Component API class**
Display	*javax.microedition.lcdui.Display*	*javax.swing.JFrame*
Container	*javax.microedition.lcdui.Form*	*javax.swing.JPanel*
Label	*javax.microedition.lcdui.StringItem*	*javax.swing.JLabel*
Button	*javax.microedition.lcdui.Command*	*javax.swing.JButton*
TextField	*javax.microedition.lcdui.TextField*	*javax.swing.JTextField*
TextPane	*javax.microedition.lcdui.TextBox*	*javax.swing.JTextPane*
RadioButton	*javax.microedition.lcdui.StringItem*	*javax.swing.JRadioButton*
CheckBox	*javax.microedition.lcdui.StringItem*	*javax.swing.JCheckBox*
ComboBox	*javax.microedition.lcdui.ChoiceGroup*	*javax.swing.JComboBox*
Message	*javax.microedition.lcdui.Alert*	*javax.swing.JDialog*
List	*javax.microedition.lcdui.List*	*javax.swing.JList*
SelectionGroup	*javax.microedition.lcdui.ChoiceGroup*	*javax.swing.ButtonGroup*

The generated code comprises roughly the complete *CurrencyConverter* GUI implementation. More specifically, in the case of the J2ME implementation 73 Lines of Code (LoC) are automatically generated from the presentation model and merely 7 LoC have been manually implemented. The manual implementation involves basically placing the components within the container, the invocation of the generated method that returns the container and placing the container onto the handset display. Consequently, the

implementation overheads are significantly reduced since 91.25% of the code is automatically generated from the presentation model. Similarly, for the case of the Java transformation 85.42% of the implementation (41 LoC) is generated from the model while the remaining 14.58% is manually implemented (7 LoC). Figure 4.17 illustrates on the left-hand side the runtime instance of the J2ME interface, while on the right-hand side the Java interface is displayed. At this stage the generated GUIs serve merely as an illustration of the service functionality.

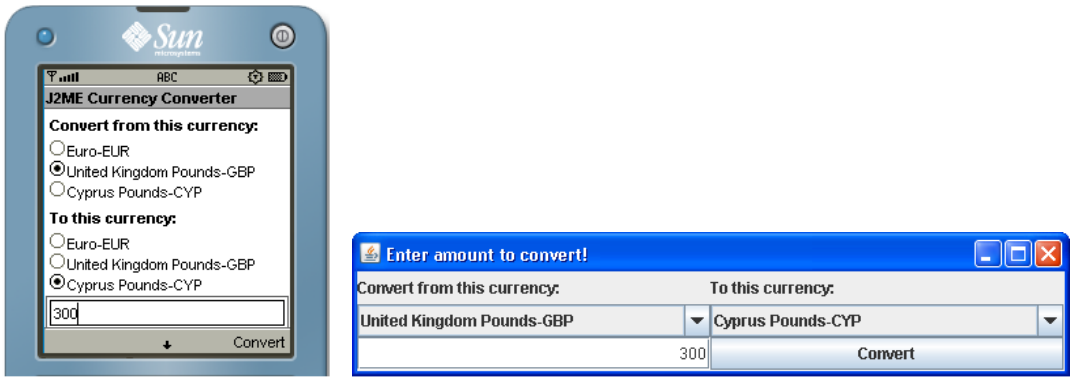


Figure 4.17: Runtime J2ME and Java instances of the *CurrencyConverter* model.

As aforementioned the flexibility and applicability of the PML relies largely on the keywords that define the properties of each abtract model element. Table 4.2 presents the elements of the PML and the properties required to provide a precise definition for each model element. The table provides also an informative description for each property and indicates the technology specific component to which the property applies. For instance, the *rows* and *columns* properties are compulsory for the *Container* element, so as to have a persistent model definition. These properties are only considered in the case of the Java transformation, while in the case of the J2ME transformation they do not affect the generated code. In particular, the properties denote the row and column parameters of the *GridLayout* class, which define the layout of the *JPanel* object.

Table 4.2: The description and application of the abstract elements' properties.

Abstract element	Property (keyword)	Description	Applies to
Display	*rows*	*The number of rows when the Display layout property is set as "grid".*	*JFrame*
	columns	*The number of columns when the Display layout property is set as "grid".*	*JFrame*
Container	*title*	*The title to be displayed on the Container.*	*Form*
	message	*The ticker message scrolling on the top of the Container.*	*Form*
	rows	*The number of rows when the Container layout property is set as "grid".*	*JPanel*
	columns	*The number of columns when the Container layout property is set as "grid".*	*JPanel*
Label	*text*	*The text to be displayed on the Label.*	*StringItem, JLabel*
Button	*text*	*The text to be displayed on the Button.*	*Command, JButton*
	command	*The Command type constants of the Button.*	*Command*
TextField	-	-	-
TextPane	*text*	*The text label of the TextPane.*	*TextBox*
	message	*The initial contents of the TextPane.*	*TextBox*
RadioButton	*text*	*The text to be displayed on the RadioButton.*	*StringItem, JRadioButton*
CheckBox	*text*	*The text to be displayed on the CheckBox.*	*StringItem, JCheckBox*
ComboBox	*stringArray*	*The array of strings to be displayed on the ComboBox.*	*ChoiceGroup, JComboBox*
Message	*title*	*The title to be displayed on the Message.*	*JDialog*
	message	*The actual information message of the Message element.*	*Alert, JDialog*
List	*title*	*The title to be displayed on the List.*	*List*
	message	*The ticker message scrolling on the top of the List.*	*List*
SelectionGroup	*text*	*The text to be displayed on the SelectionGroup.*	*ChoiceGroup*

Figure 4.18 illustrates an additional example of a presentation model that aids further the rationalization of the keywords concept, which is fundamental for the model definition and code generation phases. The figure presents the *Registration* model and gives particular focus on the properties of the *Container* element. Each property is graphically associated to its properties view in order to demonstrate the exact property definition using the PMF. More specifically, the presentation model showcases that the *rows* and

columns properties are defined as *Integer* values, while the *title* and *message* properties are defined as *String* literals.

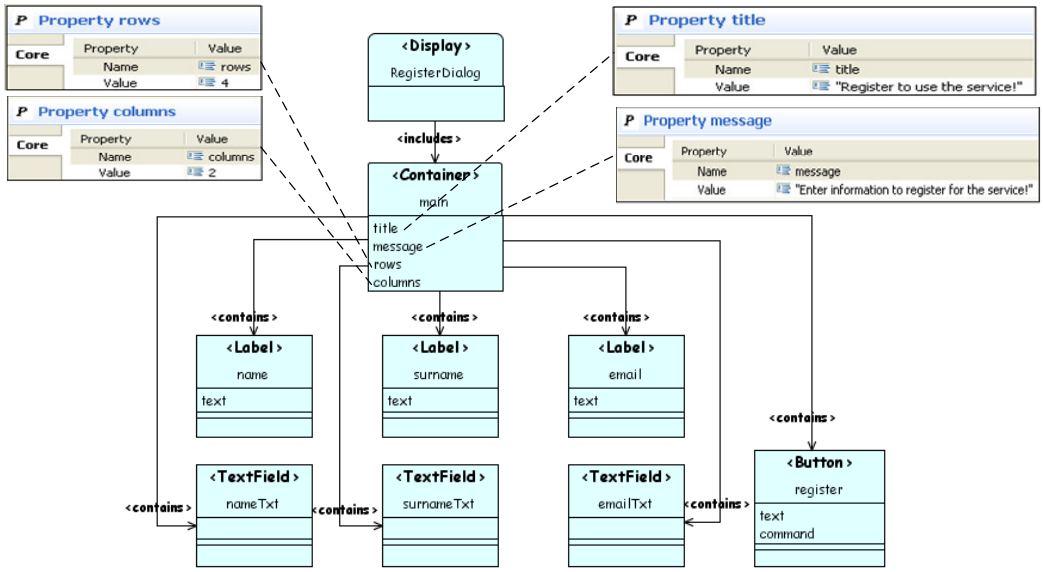


Figure 4.18: The *Registration* presentation model with highlighted properties views.

The properties (i.e. keywords) specified in the model definition steer the code generation. Figure 4.19 illustrates an extract of the J2ME template (Part A) and demonstrates the exploitation of the keywords concept for the generation of the GUI code. Consequently, via the execution of the transformation using the template definition the code illustrated in Figure 4.19 (Part B) is generated, which creates an instance of the *Form* component and sets its title in accordance to the property defined in the model. Furthermore, the message that scrolls over the *Form* component is set via the final code statement.

```
Part A.  «IF discon.conproperties.exists(ele.name.matches("title"))-»
                «discon.name» = new Form(«discon.conproperties.
                                  select(ele.name.contains("title")).value.first()»);
         «ENDIF»
         «IF discon.conproperties.exists(ele.name.matches("message"))-»
                «discon.name».setTicker(new Ticker(«discon.conproperties.
                                  select(ele.name.contains("message")).value.first()»));
         «ENDIF»
Part B.          main = new Form("Register to use the service!");
                 main.setTicker(new Ticker("Enter information to register for the service!"));
```

Figure 4.19: Extract of the J2ME template for transforming the Container element.

In the case of the J2ME transformation merely the title and message properties are considered, since the layout of the GUI and the positioning of components are explicitly controlled by the mobile handset *Display* implementation class. Although it is possible to suggest a layout, in the case that a specific component is pushed out of the displayable screen the handset *Display* class of the J2ME platform is responsible to position the component onto the next line. Hence, the J2ME code generation does not consider the *rows* and *columns* layout properties defined in the model.

```
Part A.     «discon.name» = new JPanel();
        «IF discon.layout.matches("default")-»
            «discon.name».setLayout(new BorderLayout());
        «ELSEIF discon.layout.matches("grid")-»
            «discon.name».setLayout(new GridLayout(
            «discon.conproperties.select(ele.name.contains("rows")).value.first()» ,
            «discon.conproperties.select(ele.name.contains("columns")).value.first()»));
        «ENDIF»
Part B.     main = new JPanel();
            main.setLayout(new GridLayout(4, 2));
```

Figure 4.20: Extract of the Java template for transforming the Container element.

On the contrary, for the Java technology the *BorderLayout* and the *GridLayout* managers, support arranging, resizing and placing components onto the container. Figure 4.20 illustrates an extract of the Java template definition (Part A) that handles the generation of the corresponding code statements, which create an instance of the *JPanel* container and specify the layout of the container. For the example presentation model the second conditional expression of the template definition is fullfiled since the layout property of the Container element is defined as *"grid"*. Consequently, the primary generated code statement (Part B) allows creating an instance of the *JPanel* class. Furthermore, via the subsequent statement an instance of the *GridLayout* manager is created with the specified *rows* and *columns* properties set, so as to control the layout of the container.

Figure 4.21: Runtime J2ME and Java instances of the *Registration* model.

Consequently, by introducing keywords throughout the template definition the effective transformation of presentation models is achieved and the necessary GUI implementation is obtained. The generated code requires merely minor additions and/or modifications, in order to provide the necessary GUIs. Figure 4.21 illustrates respectively the J2ME and the Java graphical user interfaces obtained from the transformation of the second example presentation model, which support the interaction of the user with the service.

4.5 Summary

This chapter presents the Context and Presentation Modelling Frameworks formulated to support the service adaptability, portability and user-service interaction requirements. The generated CMF facilitates the design and validation of context models and the automatic generation of different platform specific implementations, which enforce the context-awareness characteristic at runtime. Correspondingly, the PMF supports the design, validation and transformation of presentation models to diverse implementation technologies, which comprise the GUIs of the pervasive service. Basically, the modelling frameworks support the automatic generation of tedious and repetitive implementation code, which handles distributed context management tasks and the interaction of the user with the pervasive service.

This repititive code requires to be manually implemented for each pervasive service that must be created. For instance, managing and distributing context information stored in a repository typically involves common implementation tasks that depend merely on information management properties such as multiplicity. Hence, in accordance to the multiplicity property defined in the model the necessary context management task can be automatically generated. Moreover, the implementation of GUIs typically involves re-writing the same code and considering the miscellaneous platform-specific requirements and restrictions. Consequently, the definition of GUIs as abstract models provides the capability to automatically generate the repetitious code and expedite as a result the implementation process. In the following chapter the final component of the PSCE is presented, which refers to the Petri Net-Process Modelling Framework. Moreover, the model-driven Petri Net based process of the overall MDPNF is introduced in detail.

Chapter 5 A Petri Net based Process Modelling Framework

In this chapter a modelling framework is introduced that facilitates the definition, validation and implementation of the dynamic behaviour of pervasive services. Foremost, a requirement analysis of modelling techniques is performed on the basis of existing modelling techniques, so as to identify the best suitable technique for process modelling. The analysis motivates the selection of the Petri Nets formalism as the technique, which guides the generation of the Petri Net based Process Modelling Framework. The modelling framework enables the integration of the modelling domains under a unified Model-Driven Petri Net based Framework that supports pervasive service creation. The model-driven nature of the unified MDPNF keeps the approach at a platform independent level, while the Petri Net paradigm supplies a formal method that endorses the modelling and validation of the operational semantics of pervasive services.

5.1 Motivation and Approach

The development of pervasive services using current state-of-the-art abstract modelling techniques embraces merely the context modelling domain. In order to model precisely and implement unambiguously each distinct pervasive service, it is important to consider multiple domains and model both the static and dynamic structure of pervasive services. In this work, the context and presentation languages deal effectively with the modelling of the static structure of pervasive services. The dynamic nature of pervasive services requires though the introduction of a well-established modelling technique to define their dynamic behaviour. Moreover, the integration of the static and dynamic modelling domains is another crucial requirement that must be considered, so as to contribute a unified MDPNF for pervasive service creation.

Apart from the design requirements, the selection of a formal method allows validating the dynamic structure of the models by means of simulation. This complements the validation capabilities provided via the use of the OCL language, which allow validating the static structure and syntax of the models. Therefore, a coherent validation scheme is provided that facilitates also the validation of the dynamic structure of models and guarantees the generation of non-erroneous pervasive service implementations.

This chapter investigates the merging of the meta-modelling concept with the Petri Nets formalism that allows defining the model-driven Petri Net based process and facilitate the definition of the Petri Net-based Proces Modelling Language. The modelling language is defined on the basis of the PNML standard and forms the basis of the proposed process. From the metamodel definition a Petri Net-based Process Modelling Framework is generated that comprises the final component of the PSCE. Hence, the devised model-

driven Petri Net based process and the developed PSCE compose the unified MDPNF that supports in this thesis the creation of pervasive services.

The generated Petri Net-based Process Modelling Framework provides the capability to design Petri Net based process models that combine the distinct pervasive service modelling domains. The modelling framework facilitates also the validation of the static structure and syntax of the models by enforcing OCL constraints. Furthermore, it allows transforming the MDA-specific process models into a corresponding PNML format. This allows importing and validating as well the dynamic structure of the process models using a Petri Net software simulation tool. Concluding, the implementation of the pervasive service behaviour is generated from the Petri Net process model using the software capabilities of the combined PSCE.

5.2 Process Modelling Domain Requirement Analysis

In the context of this thesis a process model is defined as a compilation of structured activities for modelling the dynamic behaviour of pervasive services. The model represents the states (or set of conditions), actions and operations that depict the overall functionality of the pervasive service. More specifically, the model defines the ways in which conditions are fulfilled and actions and/or operations are carried out to realize the intended functionality of the pervasive service.

There are several modelling techniques that can support the definition of the dynamic behaviour of a pervasive service. The modelling techniques presented in Table 5.1, namely Petri Nets, statecharts and activity diagrams, are the most dominant ones for modelling the complex, concurrent and distributed behaviour of pervasive services. Furthermore, these modelling techniques provide the capability to graphically represent

the pervasive service behaviour in the form of a process model. Consequently, the visual representation improves greatly the understanding of the service behaviour. Following, an overview of these modelling techniques is provided and an analysis is performed on the basis of the pervasive service behavioural requirements listed in Table 5.1.

Table 5.1: Requirements for object-oriented behavioural modelling.

	Petri Nets	***State Charts***	***Activity Diagrams***
Concurrency	√	P	√
Hierarchy	√	P	√
Object Behaviour	√	√	×
Semantics	√	√	P
Verification	√	√	×
Validation	√	P	×

√ -- Full support P -- Partial support × -- No support

Petri nets were invented by Carl Adam Petri as part of his doctoral thesis [17]. It is a widely accepted formalism particularly suited for modelling complex systems. Furthermore, Petri nets provide both a graphical and algebraic representation something that makes them easily understandable. A Petri net represents actually a directed bipartite graph that consists of places, transitions and arcs that are used to interconnect them.

Statecharts is a visual formalism used for the specification and design of complex reactive systems. They were initially introduced by Harel [100] and are used specifically when it is required to describe the behaviour of an object within a complex system. A statechart contains basically states and transitions, where transitions indicate mainly the transit from one state of the system to another. Additionally statecharts are usually marked with events and conditions.

Activity diagrams are nowadays considered as an industry standard that is mainly used for business process modelling. The technique comprises the necessary routing constructs that provide the capability to model the process-logic dimension of complex systems

[101]. There are no formal semantics defined for activity diagrams, although an initial attempt is made by the OMG to define the semantics for activity diagrams in the current UML 2.0 specification, on the basis of Petri nets token game. In contrast to Petri Nets and Statecharts, activity diagrams define a larger variety of graphical constructs for the definition of the system's process-logic.

Statecharts support partially the *hierarchy requirement* by including states within other superstates using the hierarchical state decomposition technique. Although this technique is important for modelling reactive systems, the complexity of the diagram increases exponentially when dealing with complex systems with a large number of states. Therefore, it becomes very difficult to interpret the model diagram and most importantly in concurrent systems it becomes merely impossible to represent all states in a single diagram. This is due to the dependency and parallelism of actions present in concurrent systems that largely increases the number of states and transitions in the model [102]. The *concurrency requirement* is also addressed by statecharts since two composite states of a system can transit to a new state independently or simultaneously. Because concurrency is defined in statecharts using the state decomposition technique, it is realised that concurrency is also partially supported.

On the contrary the *hierarchy requirement* is fully addressed by the Petri Nets formalism. Hierarchical Petri Nets are a significant variant that introduces the concept of modelling complex systems by interconnecting smaller nets using various levels of refinement and abstraction. The hierarchy concept provides the capability to handle the refinement of places and transitions even if they are adjacent [103]. A different form of hierarchy is proposed by object-oriented Petri nets, which defines that tokens within a system net

represent object nets; i.e. subnets. This induces a hierarchical structure to the Petri Net model that facilitates the effective design of complex concurrent systems. Furthermore, *concurrency* is also supported in the formalism since two or more transitions that don't share any common conditions can occur independently or simultaneously.

Activity diagrams do not impose any restrictions in terms of *hierarchical modelling*. Primarily each activity defined in a model diagram can be de-composed into several different actions. An activity defines a specific behaviour that represents a state of the system whereas an action is considered as a single task carried out within an activity. Furthermore, swimlanes are used to separate actions within an activity in order to depict the specific actor in the system that undertakes that particular activity. In terms of *concurrency* the use of fork and join elements in activity diagrams allows to model concurrent actions occurring within a particular system.

The *object behaviour requirement* refers to the definition of the dynamic behaviour of objects within a system. In the context of the Petri net formalism object-oriented nets satisfy this prerequisite since tokens represent objects within an executing system. Statecharts introduce objects in a system that are commonly further refined in the form of class diagrams. This designates that an object is defined as a static class diagram that comprises of elements, attributes and operations, whereas the statechart represents the dynamic behaviour of this object. In contrast to the aforementioned modelling techniques activity diagrams do not introduce the notion of an object within a model.

The *semantics* definition is another important requirement that plays a major role in the *verification* of the essential properties of modelled systems. Both Petri Nets [17] and statecharts [100] have formalised semantics and various verification techniques exist that

allow verifying the consistency and correctness of the system. On the other hand there are no well-established semantics for activity diagrams, although various attempts were made to provide formalised semantics to UML activity diagrams. Consequently, in most of the cases the verification of UML diagrams is performed simply by transforming them either to statecharts or Petri net models [55], [104].

The *validation requirement* is frequently misinterpreted with the verification requirement. While verification denotes the process of checking the properties of the system using formalised means, validation refers to the process that allows determining if the system behaves as it is expected [105]. More specifically, validation can be performed using different methods such as plain inspection of the system, analysis of the system model, simulation of the executable models and/or code generation [106]. The process might include one or more of the above methods beginning from the visual observation of the system down to code generation. In this process the operational semantics of the models are of prime importance for simulation and code generation.

Both Petri nets and statecharts have a formal basis and well-defined operational semantics that allow validating models either via simulation and/or code generation. Nonetheless, in this work Petri nets are considered to provide full support for validation in contrast to statecharts which grant only partial support. This is mainly because a larger variety of Petri net based software tools is available, which facilitate essentially the validation phase. UML activity diagrams do not support the validation requirement due to the absence of well-formulated operational semantics.

5.3 Petri Net Formalism

Petri Nets [17], [106], [107] are a well-established mathematical formalism that is particularly suited for modelling and development of parallel and distributed systems, since it supports concurrency. The superiority of Petri Nets for modelling concurrency lies on the principle of locality [108]. Furthermore, Petri Nets serve as a graphical modelling technique that is used effectively in various research fields for the modelling, verification, validation and implementation of systems.

The formalism is exceedingly investigated through the course of research with a variety of extensions being developed to provide solutions in different research areas, such as distributed multimedia systems [108] and software systems [109]. Extensions of the basic Petri Net model add supplementary properties in order to increase the expressiveness of the original model. These models are described as high-level Petri Nets and examples of these powerful extensions are namely: Coloured Petri Nets [110], Timed Petri Nets, Hierarchical Petri Nets [103] and Object Petri Nets [111].

The basic Petri Net model that forms the basis for all Petri Net extensions is represented by the Net structure illustrated in Definition 5.3.1 [106]. The model is composed of *Places* and *Transitions* that are effectively interconnected via *Arcs*. Each place is represented as a circle (or ellipse) and represents a particular state of the behaviour of the modelled system. Transitions are represented as rectangles and denote actions of the system. An *input arc* connects a place to a transition and correspondingly an *output arc* connects a transition to a place; arcs are represented as arrows.

Definition 5.3.1 (Petri Net): The basic Petri Net is a triple $N = (P, T, F)$, where

i. $P = \{p_1, p_2, ..., p_k\}$, *where* $k \geq 0$, *is a finite set of Places.*

ii. $T = \{t_1, t_2, ..., t_l\}$, *where* $l \geq 0$, *is a finite set of Transitions, where the set of places and transitions are disjoint, i.e.* $P \cap T = \emptyset$, *and*

iii. $F \subseteq (P \times T) \cup (T \times P)$, *is a flow relation for the set of Arcs.*

- $I : P \rightarrow T^{\infty}$, *is the Input Arc, a mapping from places to bags of transitions.*
- $O : T \rightarrow P^{\infty}$, *is the Output Arc, a mapping from transitions to bags of places.*

Apart from places, transitions and arcs, each model includes *Tokens* that reside in places and represent the current state of the modelled system. Hence, the holding of a condition is represented by an unstructured black token in the corresponding place. The set of tokens in the Petri Net model is commonly referred as the marking of the system and it is depicted in accordance to the following definition.

Definition 5.3.2 (Petri Net Marking): $m = \{p_1, p_{2,}, ..., p_n\}$, $n \geq 0$, *is a finite set of dynamic markings associated with places.*

The modelled system is governed by a set of Firing Rules that enable the transit from one state to another. Figure 5.1 illustrates an example triple net N that represents the model of a car. Assuming the car is currently found at the *broken* state, this means that the initial marking of the net N is equal to $m = \{1,0\}$. Therefore, transition t_1 is enabled and can fire in a marking m if $\forall p \in {}^{\bullet} t : m(p) \geq 1$, i.e. since there is at least one token in each of its input places. As soon as transition t_1 fires (i.e. repair action occurs), it removes the token from the input place p_1 and deposits it to its corresponding output place p_2, resulting to a new

marking m'. In the example model the new net marking becomes $m' = \{0,1\}$, which denotes that the car is currently found at the *fixed* state; i.e. car is repaired.

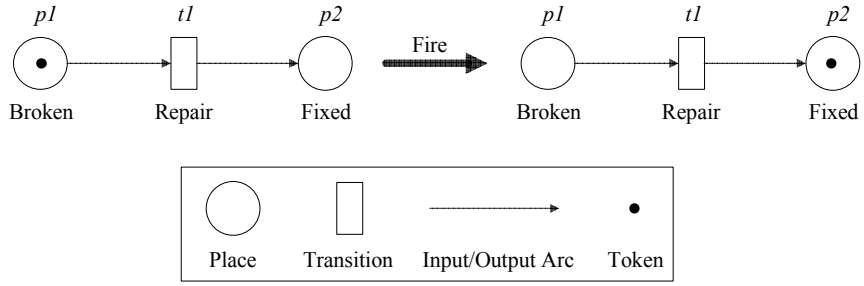


Figure 5.1: Basic Petri Net model representing the car repair process.

5.3.1 Place/Transition Nets

Place/Transition (P/T) nets are the simplest extension of the basic Petri Net model. These types of nets introduce preconditions and postconditions on input and output arcs. Consequently, the Petri Net structure becomes equal to $N = (P, T, \mathrm{Pr}e, Post)$ as depicted in Definition 5.3.1.1. As a general rule the arc inscriptions denote natural numbers that depict the number of tokens required to satisfy the occurrence rule. In particular, an input arc inscription determines the number of tokens required at an input place to enable the firing of a transition. Conversely, an output arc inscription determines the number of tokens generated and deposited to each output place when an enabled transition fires.

Definition 5.3.1.1 (Place/Transition Net): A Place/Transition net is defined by a quadruple $N = (P, T, \mathrm{Pr}e, Post)$, where

i. P *is a finite set of Places of the net* N.

ii. T *is a finite set of Transitions of the net N, disjoint from* P, *i.e.* $P \cap T = \emptyset$.

iii. $\mathrm{Pr}e \in IN^{|P|\times|T|}$ *is the backward incidence matrix of the net* N, *and*

iv. $Post \in IN^{|P|\times|T|}$ *is the forward incidence matrix of the net* N.

For the equivalent graphical representation of the *net N*, there is an incoming arc with weight $n > 0$ going from a place $p \in P$ to a transition $t \in T$ iff $\mathrm{Pre}[p,t] = n$ with $n > 0$. There is also an outcoming arc with weight $n > 0$ going from a transition $t \in T$ to a place $p \in P$ to iff $Post[p,t] = n$ with $n > 0$. Consequently, the flow relation of the *net N*, becomes $F := \{(p,t) \in P \times T \mid \mathrm{Pre}[p,t] > 0\} \cup \{(t,p) \in T \times P \mid Post[p,t] > 0\}$.

Definition 5.3.1.2 (Marking of a Place/Transition Net): A marking of a P/T net $N = (P,T,\mathrm{Pre},Post)$ is a vector $\mathrm{m} \in \mathrm{IN}^{|P|}$. A transition $t \in T$ is enabled in a marking m if $m \geq \mathrm{Pre}[p,t]$.

Definition 5.3.1.3 (Successor Marking of a P/T Net): The successor-marking relation of an enabled transition is defined by: $m \xrightarrow{t} m' \Leftrightarrow m \geq \mathrm{Pre}[p,t] \wedge m' = m + Post[p,t] - \mathrm{Pre}[p,t] = m + C[p,t]$. $C[p,t] = Post - \mathrm{Pre}$ denotes the incidence matrix of the P/T net N. For a transition $t \in T$ the expressions $\mathrm{Pre}[p,t]$ and $Post[p,t]$ depict the column vectors in Pre and $Post$ associated with the transition.

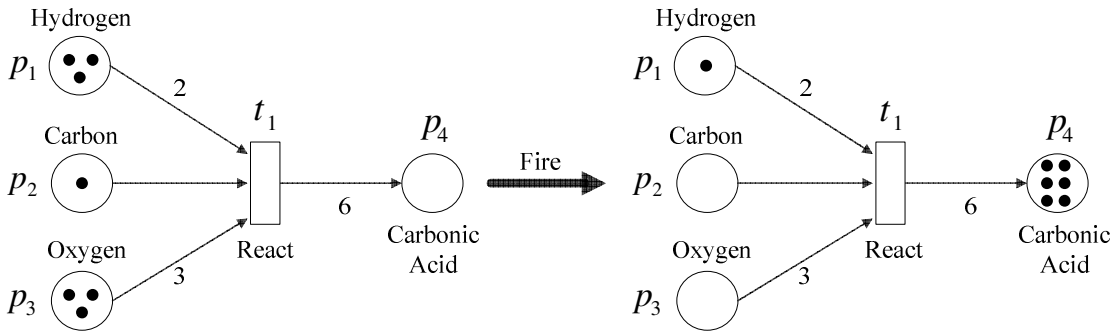


Figure 5.2: A Place/Transition net representing a chemical reaction.

Figure 5.2 illustrates an example P/T net that depicts a chemical reaction for the production of the carbonic acid. The figure demonstrates that the arcs of the P/T net are marked with inscriptions. The transition t_1 is enabled since there are at least two tokens in

place p_1, one token in place p_2 and three tokens in place p_3. Consequently the *React* action of transition t_1 takes place and six tokens are deposited to the place p_4. Table 5.2 illustrates the incidence matrices $\mathrm{Pr}e$, $Post$ and C for the chemical reaction P/T net.

Table 5.2: The incidence matrices of the chemical reaction P/T net.

Pre	t_1
p_1	2
p_2	1
p_3	3
p_4	0

Post	t_1
p_1	0
p_2	0
p_3	0
p_4	6

C	t_1
p_1	-2
p_2	-1
p_3	-3
p_4	6

5.3.2 High-Level Nets

High-level Nets introduce higher expressiveness into the basic Petri Net model. This is performed via the introduction of complex data that represent tokens and replace the unstructured black tokens. These types of high-level nets are also referred in the literature as Coloured Petri Nets [106], [110] due to the colours that represent different tokens. Apart from the coloured tokens, transitions and arcs are marked respectively with expressions and inscriptions that control the execution of the Petri Net model.

Definition 5.3.2.1 (Coloured Petri Net): A Coloured Petri Net (CPN) is defined as a tuple $N = (P, T, \mathrm{Pr}e, Post, C, cd)$, where

i. P *is a finite set of Places of the net* N.

ii. T *is a finite set of Transitions of the net* N, *disjoint from* P, *i.e.* $P \cap T = \emptyset$.

iii. C *is a finite set of non-empty types named coloured sets.*

iv. $cd : P \cup T \rightarrow C$ *is the colour domain mapping, and*

v. $\mathrm{Pr}e, Post \in B^{|P|\times|T|}$ *are the backward and forward incidence matrices of the net N, such that* $\mathrm{Pr}e[p,t] : cd(t) \rightarrow Bag(cd(p))$ *and* $Post[p,t] : cd(t) \rightarrow Bag(cd(p))$ *are*

mappings for each pair $(p,t) \in P \times T$. *B can be taken as the set of mappings of the form* $f : cd(t) \rightarrow Bag(cd(p))$ [106].[6]

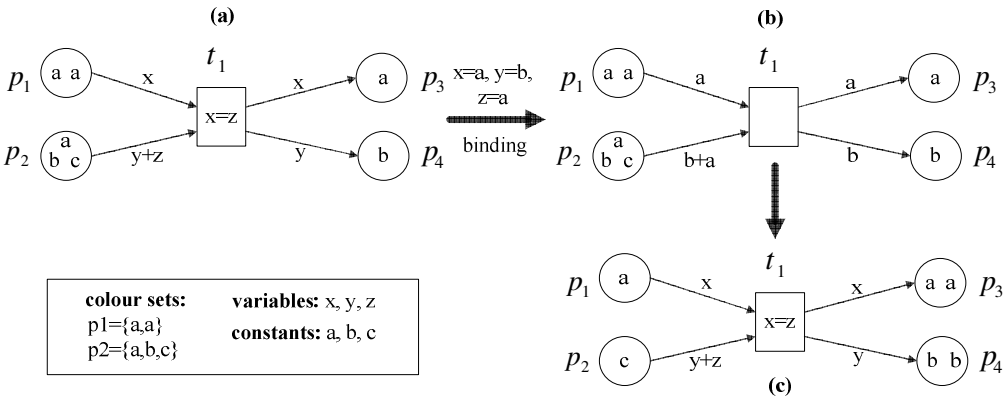


Figure 5.3: Occurrence rule for a Coloured Petri Net.

The model of Figure 5.3 represents an example of a Coloured Petri Net. In order for transition t_1 to be enabled the occurrence rule must be set accordingly. Initially, a binding $\beta_1 = [x = a, y = b, z = a]$ must be defined by assigning particular values to variables; as illustrated in Figure 5.3. Therefore by setting the appropriate binding β_1 the guard holds and each variable is replaced by the corresponding value as shown in (b) of Figure 5.3. Hence transition t_1 is enabled and consequently the occurrence rule is applied resulting in the execution of the net as illustrated in (c) of Figure 5.3.

The binding must be defined appropriately in order to preserve the correct execution of the model. For instance, if the binding is defined as $\beta_2 = [x = a, y = b, z = c]$, the transition t_1 cannot be enabled since the guard expression of t_1 is not satisfied. Moreover, if an improper binding $\beta_3 = [x = a, y = a, z = a]$ is defined, transition t_1 cannot fire because there are insufficient tokens of type a in the input place p_2. Thus the binding definition is of prime importance for the correct execution of the Coloured Petri Net behaviour.

[6] A Bag denotes a multiset in which each member can have more than one membership.

5.3.3 Object-Oriented Petri Nets

The concept of integrating object-orientation with the Petri Net formalism has been investigated by many researchers [62], [112], [113], [114]. This ongoing state-of-the-art research work aims mainly at resolving the predicaments encountered in both of these paradigms. Primarily object-orientation introduces additional structuring facilities into the Petri Net model, so that modelling large distributed systems becomes an easier task to accomplish. Conversely, Petri Nets theory provides a formal method that facilitates the verification and validation of software systems.

In the current literature various types of Object-Oriented (OO) Petri Nets have been defined, focusing mainly on two complementary research directions. The primary initiative deals principally with the mathematical analysis of OO-Petri Nets and does not introduce high-level properties such as inheritance and polymorphism into the basic Petri Net model. On the contrary, the latter initiative introduces object-oriented concepts into the model as rich as possible; i.e. high-level properties are considered.

In this thesis the former class of OO-Petri Nets is considered, which is conventionally called an Elementary Object System (EOS). Each EOS is effectively composed by a System Net and one or more Object Nets [115], which are represented as tokens of the System Net. Reference Nets [116], [117], [118] are powerful EOSs, where any instance of the System Net includes references to instances of Object Nets in the form of tokens. More specifically, Reference Nets form the basis for the definition of the Petri Net-based Process Modelling Language proposed in this chapter.

Figure 5.4 illustrates an example EOS modelled using the software Renew tool [116] of the Reference Net formalism, which presents a specific snapshot taken during the model

execution; i.e. model simulation. The model instance (b) represents the System Net, which includes references to the Object Net instances (a) and (c). Primarily p_1 is marked with an initialisation token "[]" that initiates the execution of the net since it enables transition t_1. Transition t_1 is the creation transition that creates two distinct instances x and y of the *othernet* Object Net and stores the references in places p_2 *and* p_3 in the form of tokens. Consequently, the guard expression $x!= y$ of transition t_2 is satisfied because the two net instances are not identical. The net execution terminates at place p_4, which represents the final state of the system.

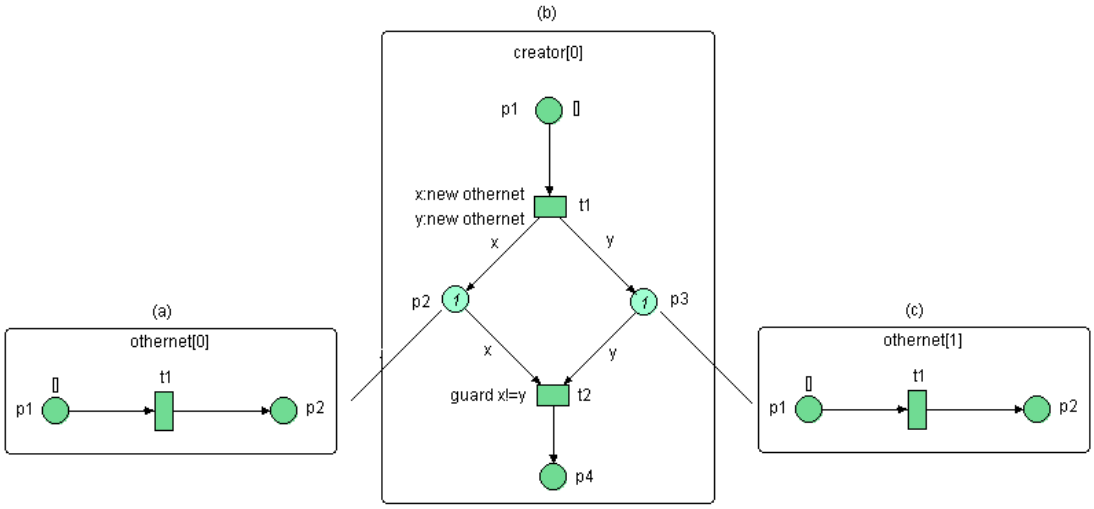


Figure 5.4: An Elementary Object System [118].

The example of Figure 5.4 demonstrates how intuitively object-oriented concepts are introduced into a formal modelling paradigm such as Petri Nets. This provides the capability to obtain a Petri Net formalism with enhanced structuring capabilities, which supports the definition, validation and implementation of the dynamic behaviour of pervasive services.

5.4 Petri Net-based Process Modelling Language

5.4.1 Three-dimensional Model

The Context and Presentation modelling languages introduced in Chapter 4, address the definition of context and presentation models, which represent the static structure of the pervasive service. This section presents the Petri Net-based Process Modelling Language that facilitates the definition and validation of the dynamic structure of the pervasive service. Furthermore, the nature of the PN-PML facilitates the integration of the modelling languages (i.e. domains) in an intuitive manner [119] and steers the definition of the model-driven Petri Net based process for pervasive service creation.

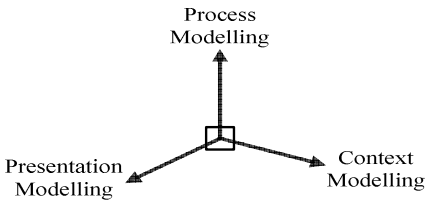


Figure 5.5: The three dimensions for Pervasive Service creation.

The modelling views presented in Figure 5.5 denote the domains investigated in this work for the creation of pervasive services at the static compile time. In the context of this thesis *process modelling* refers to the definition of the distributed dynamic behaviour of the pervasive service in the form of a process model. In particular the process model depicts the different states, actions and operations governing the service execution. Furthermore, the *Context modelling* domain refers to the definition of an advanced information model that captures the context-awareness characteristic, so as to satisfy the service adaptability requirement. Finally, the *Presentation modelling* domain deals with the definition of Graphical User Interfaces in the form of an abstract model. These GUIs serve as the frontend with which the user interacts with the pervasive service.

Figure 5.6 illustrates example pervasive service models and depicts how the integration of the models is achieved. The Petri net model acts as the carrier of the objects defined in the context and presentation models, by representing these objects as tokens of the service net. Consequently, it orchestrates the behaviour of these object nets within the service net and defines the overall pervasive service behaviour. Initially, the places p_1 and p_2 are marked with initialisation tokens "[]". Therefore, transitions t_1 and t_2 are enabled and create correspondingly two instances of the *Display and Person* objects. These reference object instances are stored as tokens in places p_3 and p_4, representing the current state of the service execution. Moreover, arcs include inscriptions and/or expressions that control the flow of the tokens (i.e. objects) within the service net. Since places p_3 and p_4 hold the object instances *d* and *p* the arc inscriptions are satisfied and subsequently transition t_3 is enabled. Hence, the transition fires and the resulting *tuple [d,p]* is deposited at place p_5, which denotes the final state of the service execution.

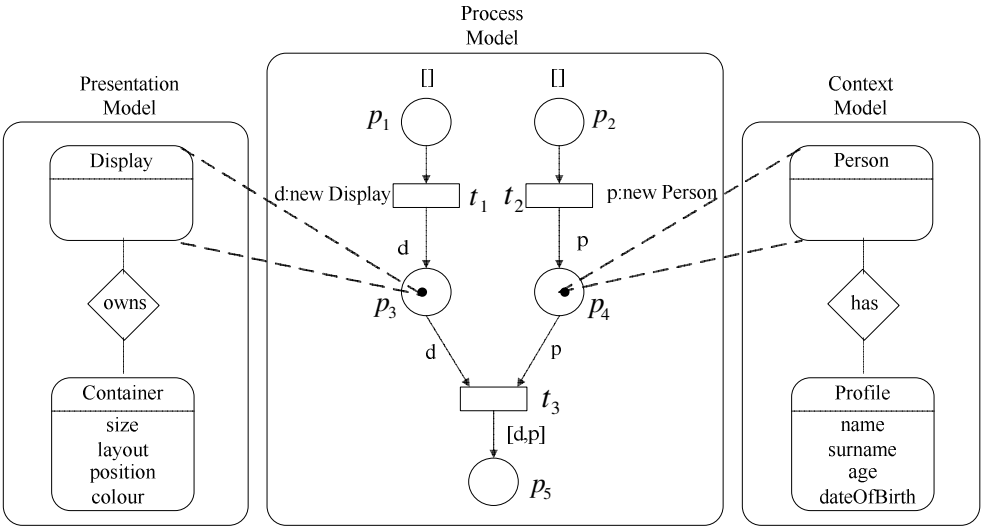


Figure 5.6: Pervasive service models integration.

The integration of the distinct modelling domains allows devising a common process, which facilitates the fulfilment of the pervasive service requirements. Primarily, the utilisation of the MDA paradigm as proposed by the OMG, keeps the approach at an abstract platform independent level. This reduces the complexity of the process since the approach steers clear of platform specific implementation complexities. Subsequently, diverse implementations can be generated from the models addressing also the portability requirement of pervasive services. Furthermore, the introduction of the object-oriented Petri Net paradigm provides a formal technique for the validation of pervasive services. In particular, the technique supports the validation of the dynamic behaviour of the service, which is complementary to the validation of the static structure and syntax of the models using OCL constraints.

5.4.2 Model-driven Petri Net based Process

In this subsection the proposed pervasive service creation process, originally introduced in Chapter 3, is presented in a more comprehensive manner. Figure 5.7 illustrates at the top the use of the generated modelling frameworks that facilitate the definition of the pervasive service models. Aside from the models definition, the modelling frameworks are also used to validate the static syntax of the models. Hence, once the static validation is performed the Petri Net model is translated via the PNML generator to the corresponding PNML format. The Petri Net model is the carrier of the objects defined in the context and presentation models and consequently the operational semantics of the pervasive service can be validated. More specifically, with the use of the Renew Petri Net simulator the generated PNML file is imported into the tool, the Petri Net model is redrawn and the operational semantics of the pervasive service are validated.

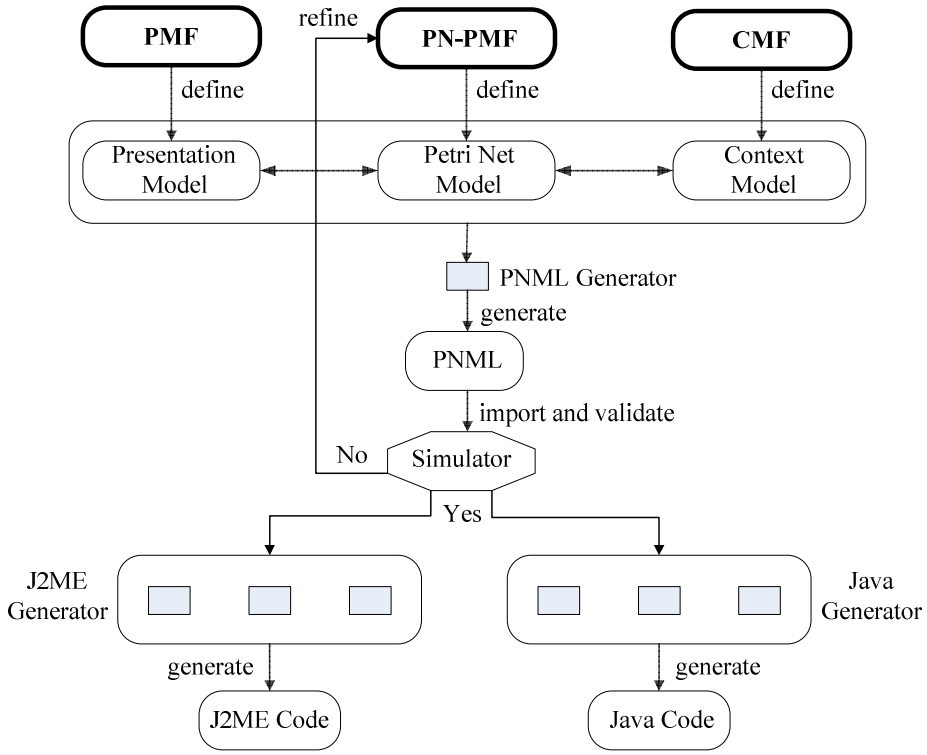


Figure 5.7: Pervasive service creation process.

The execution of the validation yields the corresponding output result, which is negative in the case an erroneous definition is detected during the simulation. Therefore, the user is presented with the appropriate diagnostic information that indicate the errors and subsequently the designer needs to manually refine the model. This is performed in an iterative process until the simulator yields a positive validation result, which initiates correspondingly the generation process.

Figure 5.7 illustrates also the J2ME and Java code generators that are composed respectively by three discrete template definitions, which are defined for handling and transforming each model. Consequently, in accordance to which generator script is invoked by the designer the respective platform specific implementation of the pervasive

service is generated with the aid of the oAW workflow engine. Concluding, the developer can deliver the complete implementation by manually writing the complex service code.

5.4.3 Petri Net Markup Language Core Metamodel

The definition of the PN-PML and the creation of the supporting modelling framework are guided by the Steps 1-4 of the domain-specific MDD methodology; see Chapter 3. Primarily the definition of the *PNML Core Metamodel* is performed in accordance to the ISO/IEC 15909-2 International Standard [120]. This standard defines the universal XML-based transfer syntax for Petri Nets. The main objective of the standard is to provide an exchange format, namely the Petri Net Markup Language, which enables the compatibility and interoperability among heterogeneous Petri Net tools [121].

The PNML syntax is a well recognised and widely accepted standard supported by a variety of Petri Net tools. Hence, the definition of the Core Metamodel is performed in accordance to the specification of the basic PNML [122], which is provided in the format of RELAX NG; i.e. an XML schema language. The basic PNML is manually translated and defined as the EMF-based metamodel illustrated in Figure 5.8.

The PNML Core Metamodel presented in the figure defines the standard features for every type of Petri Net model. These properties denote the aspects related to the graphical representation of models and the corresponding Petri Net tool information. For instance, the metamodel depicts that each *DocumentRoot* metaclass contains zero to many (0...*) *PlaceContent* instances. Each *PlaceContent* instance inherits its properties from the *NodeContent* element, which is defined as an abstract metaclass. The *NodeContent* metaclass defines the *group* and *id* properties that denote correspondingly the group and the unique identifier of each node element.

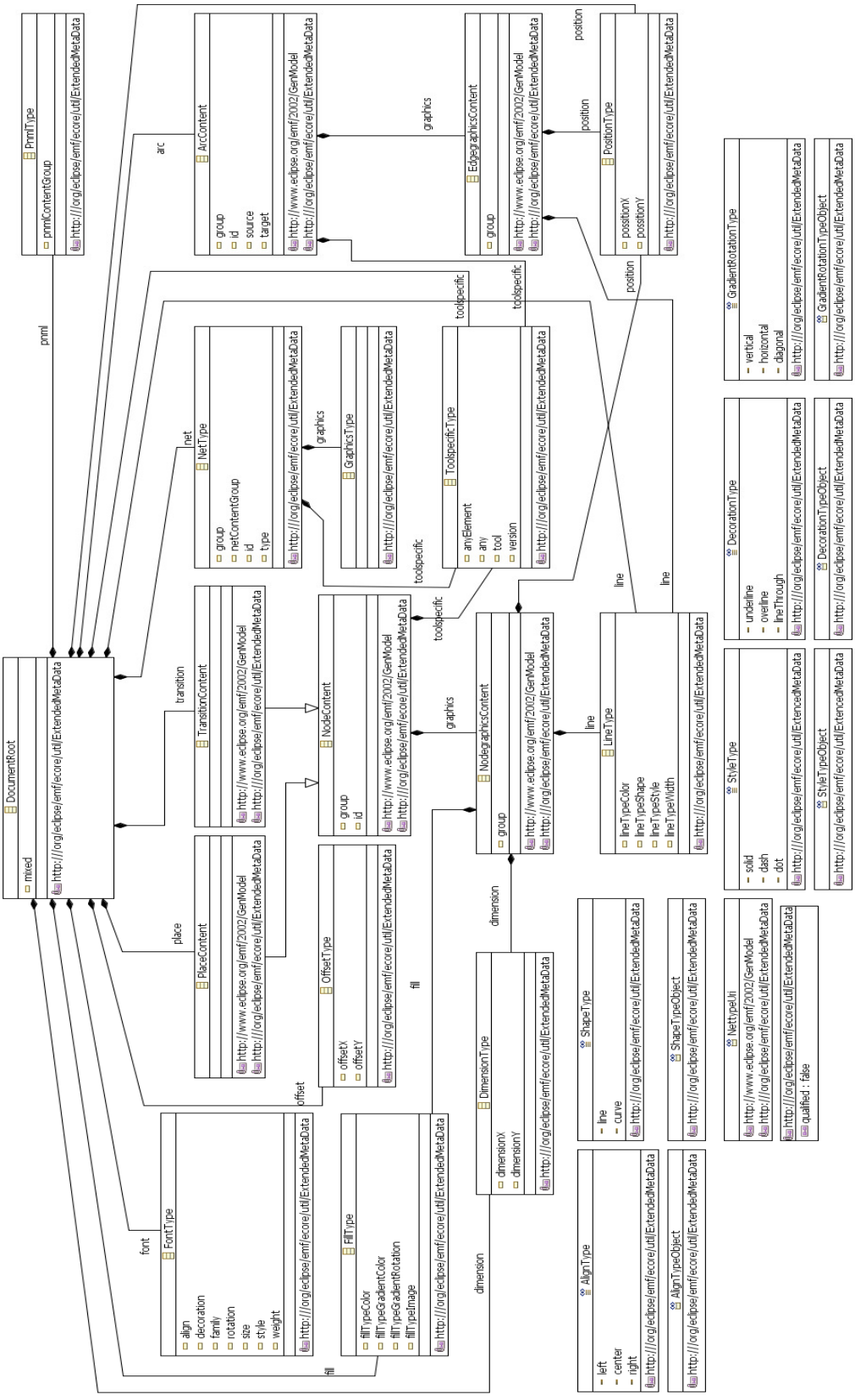


Figure 5.8: Petri Net Markup Language Core Metamodel.

Next, the *NodeContent* metaclass is associated via the *graphics* aggregation to the *NodegraphicContent* metaclass that defines the graphical representation of model elements. The aggregations *dimension, position and lineType* denote the dimensions, the position and the characteristics of the line required to draw effectively the node element; i.e. place or transition. For example, each *PositionType* metaclass includes the properties *possitionX and possitionY* that indicate the specific coordinates of each node element. Finally, the metamodel is enriched with enumerations, such as the *StyleType*, which restrict the values that can be assigned to a particular property. The *lineTypeStyle* feature of the *LineType* metaclass defines such a property, since it is associated to the *StyleType* enumeration. Consequently, it can only be assigned the literal values *solid, dash and dot* that designate the possible styles for a line.

The PNML Core Metamodel proposed in this subsection serves merely as the basis for defining concrete Petri Net extensions. Hence, a modelling framework cannot be generated from the core metamodel and subsequently a concrete Petri Net model cannot be defined. Therefore, a child metamodel that inherits its graphical and tooling properties from the core metamodel must be defined, which is expediently called in the thesis an *extension metamodel*. The naming arises from the fact that each extension metamodel inherits properties from the core metamodel, in order to define a coherent Petri Net extension that conforms to the PNML standard.

Figure 5.9 illustrates the technique proposed in this work, which allows utilising the MDA capabilities of the generic environment to simplify the development of Petri Net based modelling languages. Hence, the definition of Petri Net modelling languages as extension metamodels simplifies also the generation of their supporting modelling

frameworks. Using this extension technique the Petri Net based Process Modelling Language is defined in the form of an extension metamodel that inherits many of its properties from the PNML core metamodel. Correspondingly, from the PN-PML metamodel definition the concrete syntax of the modelling language is derived and domain rules are imposed, which restrict the definition of the static structure and syntax of Petri net process models. Following, the different constructs of the language are mapped into a common metamodel that facilitates the generation of the supporting Petri Net Process Modelling Framework.

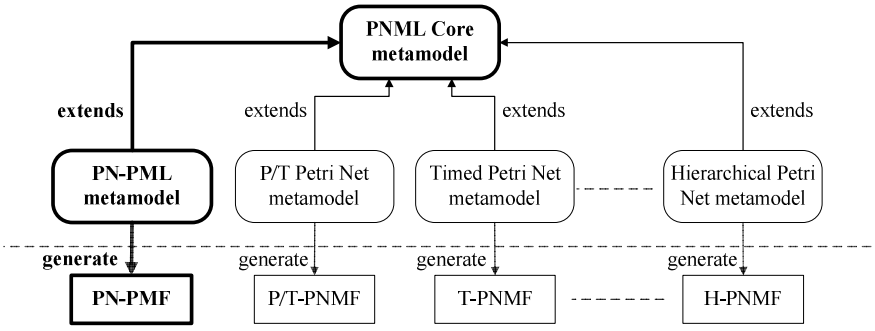


Figure 5.9: Developing Petri Net extensions and modelling frameworks.

5.4.4 Petri Net-based Process Modelling Language Formal Definition

This subsection introduces the proposed PN-PML that facilitates the integration of the modelling domains and provides the capability to define, validate and implement the dynamic behaviour of a pervasive service. The modelling language is defined in the form of a metamodel using the capabilities of the generic environment. For the design of the metamodel the import and inheritance capabilities provided by the EMF are used, to extend the PNML Core Metamodel. The process is performed as an alternative to the UML merge concept, since the EMF does not natively support the merge concept. Prior

to introducing the concepts of the metamodel, the formal definition of the PN-PML is presented, which conforms to the definition of an Elementary Object System [115].

Definition 5.4.4.1 (Elementary Object System): *An Elementary Object System is defined as a sextuple* $N = (SN, \hat{ON}, \rho, A, \hat{M})$, *where*

i. $SN = (P,T,F)$ *is the System Net of the* EOS[7].

   a. P *is a finite set of Places of the* SN.

   b. T *is a finite set of Transitions of the* SN, *disjoint from* P, *i.e.* $P \cap T = \emptyset$.

   c. $F \subseteq (P \times T) \cup (T \times P)$, *is a flow relation for the set of Arcs.*

ii. $\hat{ON} = \{ON_1, ON_2, ..., ON_i\}, i \geq 1$ *is a finite set of Object Nets of the* EOS, *which is defined as* $ON = (B,E,F,m_0)$.

iii. $\rho \subseteq T \times E$ *is the interaction relation between a system net and an object net.*

iv. $A := W \rightarrow 2^{\{1,....,n\}} \cup IN$ *is the arc type function.*

v. $\hat{\mathrm{M}}$ *is the marking of the EOS.*

The simplest model that constitutes object nets within a system net is the so-called Unary Elementary Object System, which consists mainly of a single object net. In accordance to the definition of the Unary Elementary Object System (Definition 5.4.4.2) the occurrence rules for the EOS are essentially defined (Definition 5.4.4.3).

Definition 5.4.4.2 (Elementary Object System): *A unary EOS is defined as a triple,* $N = (SN, ON, \rho)$. *The system net of the EOS is an elementary net system defined as* $SN = (P,T,W,M_0)$ *with* $|M_0| = 1$. *The object net of the EOS is again an elementary net*

[7] The term System Net, expediently refers to a service, since each supernet represents a pervasive service.

system defined as $ON = (B, E, F, m_0)$. *Moreover, the synchronous interaction relation between the system net and the object net is defined in the form of* $\rho \subseteq T \times E$.

Definition 5.4.4.3 (Occurrence Rules for an Elementary Object System): *A bi-marking of unary elementary object system* $N = (SN, ON, \rho)$ *is a pair* (M, m) *where* M *is the marking of the SN and* m *is the marking of the ON.*

i. *System-autonomous occurrence: A transition* $t \in T$ *is enabled in a bi-marking* (M, m) *of an EOS if* $t_p = 0$ *and* t *is enabled in* M. *Then the resulting bi-marking* (M', m') *is defined by* $M \xrightarrow{t'} M'$ *and* $m = m'$. *This is defined as* $(M, m) \xrightarrow{[t, \lambda]'} (M', m')$ *in this case.*

ii. *Object-autonomous occurrence: A transition* $e \in E$ *is enabled in a bi-marking* (M, m) *of an EOS if* $\rho_e = 0$ *and* e *is enabled in* m. *Then the resulting bi-marking* (M', m') *is defined by* $m \xrightarrow{e'} m'$ *and* $M = M'$. *This is defined as* $(M, m) \xrightarrow{[\lambda, e]'} (M', m')$ *in this case.*

iii. *System-Object synchronous interaction: A pair* $[t, e] \in T \times E$ *is enabled in a bi-marking* (M, m) *of an EOS if* $(t, e) \in \rho$ *and* t *and* e *are enabled in* M *and* m, *respectively. Then the resulting bi-marking* (M', m') *is defined by* $M \xrightarrow{t'} M'$ *and* $m \xrightarrow{e'} m'$. *This is defined as* $(M, m) \xrightarrow{[t, e]'} (M', m')$ *in this case.*

Reference nets [116], [117], [118] are Elementary Object Systems with reference semantics that are used as the basis of the PN-PML. The Reference net formalism supports the entire set of requirements necessary to obtain a language that facilitates

object-oriented behavioural modelling. Even so, the tight integration of Petri nets and Java within the formalism opposes to the platform-independent approach proposed in this work. Consequently, a subset of Reference nets is derived and formulated in such a way so that the abstract properties and semantics of the language are preserved. Figure 5.10 presents an example model designed using the constructs of the PN-PML. The model represents a *Bank* service that comprises a *Customer service net*, which contains the *Account and Balance object nets*. The model serves as an example service that allows introducing the abstract syntax of the proposed process modelling language.

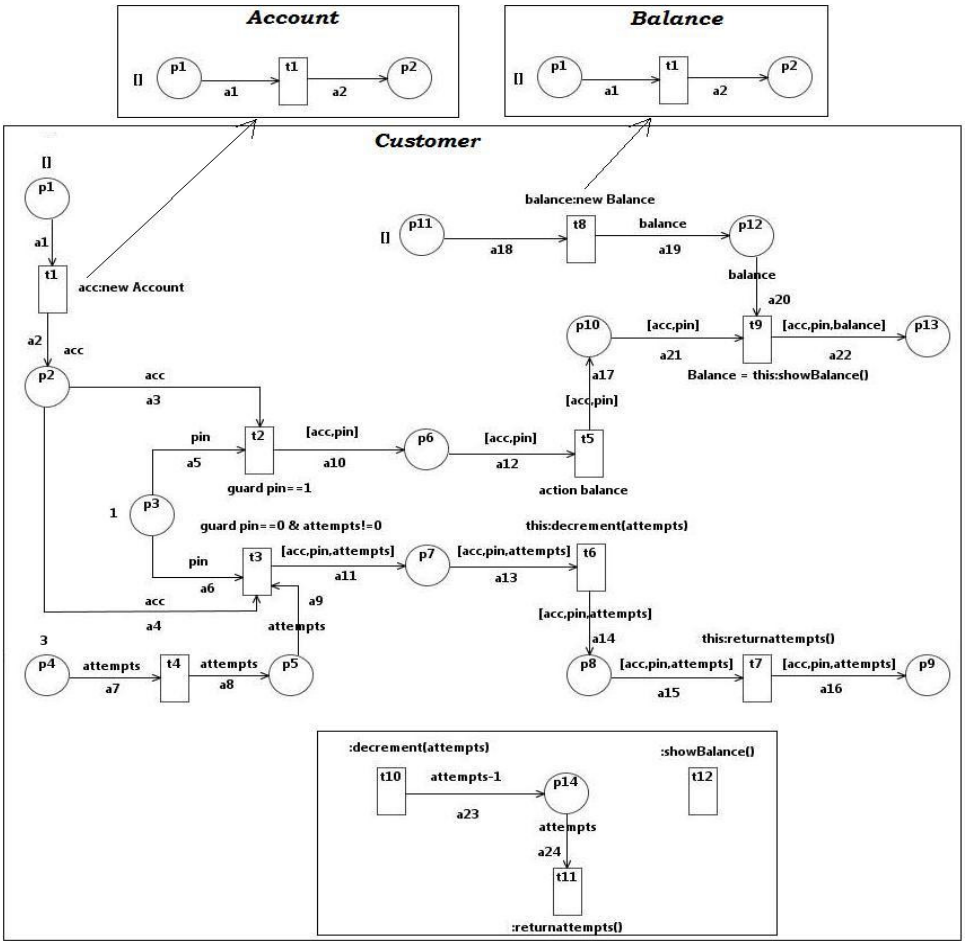


Figure 5.10: Service process model defined using the PN-PML.

The PN-PML consists of *places, transitions and arcs*, as with any other Petri net modelling language. These elements can be assigned different semantic inscriptions and/or expressions. Primarily, each *place* represents a particular state in the execution of the service and that state is depicted by a set of conditions that must hold for this specific place. The place can be marked using different inscriptions, which are restricted to *Initialisation* tokens *"[]"*, primitive datatypes or a tuple of primitive datatypes. For instance, in the example model of Figure 5.10 places p_1 and p_{11} are marked by *Initialisation* tokens and places p_3 and p_4 are marked using *Integer* numbers. These tokens satisfy the conditions required to enable the corresponding transitions and initiate subsequently the service model execution. Note that, for the case of the place construct no subclasses are defined for the modelling language.

On the contrary, the *transition* element of the modelling language contains six distinct subclasses that enhance the dynamic nature of the PN-PML in terms of model hierarchy and concurrency. These diverse subclasses are namely the *basic, object, downlink, uplink, action and guard* transitions. Every subclass serves a particular purpose in the execution of the service net. The *basic* transition is the simplest transition that can be defined in a net, since it does not carry any expression. Subsequently, the transition becomes enabled once the conditions in all of its input places are satisfied. Transition t_4 of the *Customer* service net and transitions t_1 of the *Account and Balance* object nets are examples of basic transitions.

The *object* transition brings in perspective the idea of creating instances of object nets within the service net. This improves the hierarchical structure of the Petri Net model since a complex model can be broken down to a service net with various object nets. In

relation to object-oriented programming this denotes that the main class is aware of other objects and subsequently it can utilise their functionality; i.e. invoke methods, use variables. These object nets are modelled by object transitions and reside in the service net as tokens. Figure 5.10 illustrates two example object transitions within the *Customer* service net. More specifically, an instance of the *Account* object net is created by transition t_1 and an instance of the *Balance* object net is created by transition t_8. These instances become tokens of the service net and are represented respectively via the *acc* and *balance* inscriptions.

Apart from the idea of net instances an additional intuitive concept is captured within the language. This is the concept of synchronous communication that was initially proposed for Coloured Petri nets [123]. Using synchronous channels two transitions can synchronise and fire atomically, provided that they initially agree on the name of the channel and the parameters involved in the synchronisation. The use of the *downlink* and *uplink* transitions provides the capability to define synchronous communication channels as originally implemented in [116]. Note that, in the context of this work synchronous channels are considered only within a service net and consequently synchronous communication is not accounted between different net instances.

The *downlink* transition is considered as the initiator of the communication channel. This is analogous to the invocation of a class method that may carry an argument or a list of arguments. Each downlink transition is marked by an expression that denotes the current service net, the name of the channel and the set of arguments used in the synchronisation. Figure 5.10 illustrates t_6 that defines a downlink transition marked with the expression *"this:decrement(attempts)"*. The expression part *"this"* refers to the current service net,

while the *"decrement"* expression part denotes the name of the channel and the *"attempts"* expression part defines an argument enclosed within the parenthesis. In the case that many arguments are defined these are separated by commas.

Accordingly the *uplink* transition is considered as the terminating element in the synchronous communication, which serves requests delegated by downlink transitions. In association to object-oriented programming this is considered as the method within a class that implements a specific functionality. Figure 5.10 shows t_{10} that defines the uplink transition that serves the request delegated by transition t_6. The transition carries the expression *":decrement(attempts)"*, which denotes the name of the channel (i.e. *"decrement"*) and the argument (i.e. *"attempts"*) passed as a parameter.

Furthermore, *action* transitions defined within the model designate the occurrence of an action generated by the user. This denotes in particular that an action may have specific side effects and can influence the state of the objects within the service net. For instance, when a user clicks a button associated to a specific graphical user interface a corresponding action is being generated. The service model of Figure 5.10 defines transition t_5, which designates an action called *"balance"* that is executed when the user wants to see his current account balance.

The final subclass is the *guard transition* that depicts a basic transition extended with a logical expression. The expression must be effectively evaluated to *true* for the transition to become enabled and fire. Figure 5.10 presents the guard transitions t_2 and t_3 that carry respectively the expressions *"guard pin==1"* and *"guard pin==0 & attempts!=0"*. The former denotes that if a user credit card number has been entered correctly (i.e. pin == 1) the subsequent action, which displays the current account balance to the user can be

effectively undertaken. Furthermore, the latter checks if the credit card number entered is not valid (i.e. pin == 0) and that there are remaining attempts (i.e. attempts != 0) in order to decrement the number of possible retries. Otherwise, the guard expression is evaluated to *false* and subsequently that particular service execution path cannot be followed.

The modelling language includes arc elements as well, which provide complementary control to the execution flow of the service model. These arcs may carry inscriptions that define either primitive datatypes, tuples of primitive datatypes, object nets or tuples of object nets. Arcs inscriptions are used to evaluate the tokens consumed or generated by transitions. Therefore, the inscription of an input arc should be equivalent to the type of the input place associated with the connecting transition. Moreover, the inscription of an output arc must be comparable to the input token of the transition; in that case the firing of the transition produces no side effects (i.e. no tokens).

Conversely, if the transition has side effects then the output arc inscription must be comparable to the token(s) produced by the transition. Figure 5.10 presents the transitions t_1 and t_8 that generate respectively the object tokens *acc* and *balance*. Consequently, the output arcs carry inscriptions that are typed with the analogous token values. Examples of transitions that generate no side-effects are transitions t_2, t_3 and t_4. In terms of the modelling language inscriptions and expressions are written using the operators defined by Reference nets [116], which are binary, logical and assignment operators.

5.4.5 Petri Net-based Process Modelling Language Metamodel Definition

On the basis of the concepts introduced the abstract syntax of the PN-PML is defined in the form of an extension metamodel. The modelling language utilises and extends the PNML Core metamodel using the import and inheritance mechanism provided by the

EMF component. Figure 5.11 presents the *PN-PML metamodel* that defines the abstract syntax of the modelling language and drives the generation of the concrete syntax of the language. Accordingly the mapping of the abstract and the concrete syntax facilitates the generation of the Petri net-based Process Modelling Framework. The PN-PMF is the third modelling component of the PSCE that provides the following software capabilities: (i) design and static validation of process models, (ii) transformation of process models to PNML syntax and (iii) transformation of process models to executable code.

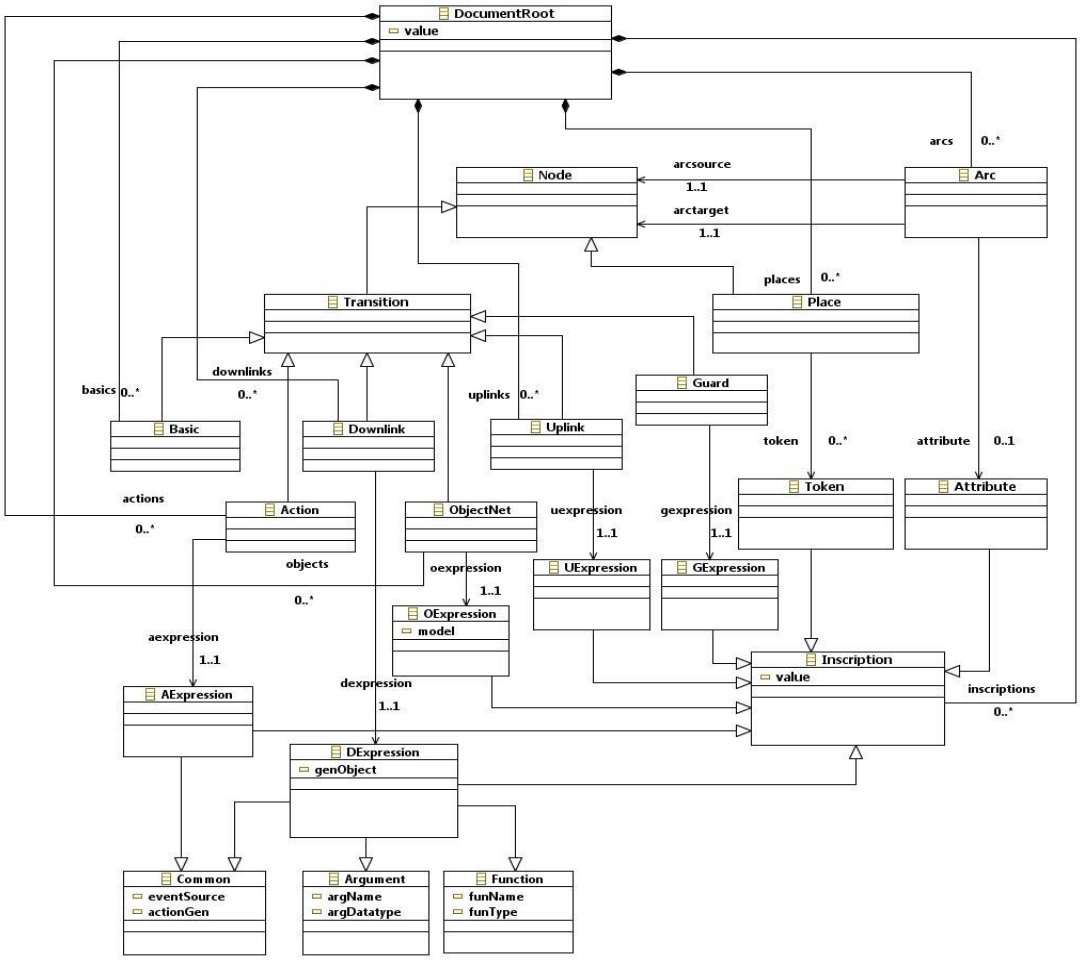


Figure 5.11: Petri net-based Process Modelling Language metamodel.

The *Process* metamodel definition follows the same pattern as the one used for the Context and Presentation metamodels. This denotes specifically that the root element is

the *DocumentRoot* metaclass that acts as the container of the rest of the elements. The root metaclass owns a set of aggregations that illustrate the containment relationships to the corresponding metaclasses. Furthermore, the *DocumentRoot* extends the *NetType* metaclass of the core metamodel and inherits its *id and type* properties. Consequently, the metaclass represents essentially the service net that contains places, transitions and arcs. Apart from the properties inherited from the *NetType* metaclass an additional *value* property is defined, which designates the name of the service net.

Primarily the *places* aggregation defines that each service net may include zero to many (0...*) instances of the *Place* metaclass. The *Node* element is considered the abstract metaclass of the *Place* element from which places inherit their properties. Furthermore, the *Node* metaclass extends accordingly the *NodeContent, PositionType, LineType and FillType* elements of the core metamodel. Conceptually the *Node* element should extend merely the *NodeContent* metaclass, which provides access to the rest of the elements (*PositionType, LineType and FillType*) via the aggregations defined in the core metamodel. The constraints imposed though by the requirement to provide a concrete syntax for the modelling language (i.e. Petri-net based graphical representation), steer the definition of the inheritance relationship using this approach. As a result, each place element must be represented in the model using a single visual construct that carries the entire set of properties inherited from its parents.

Apart from the graphical properties, each place contains zero to many (0...*) structured tokens as depicted by the *token* association. These tokens are defined in the form of inscriptions, since the *Token* element extends the *Inscription* metaclass. More precisely, the *value* property of the *Inscription* element allows defining syntactically structured

tokens. The *Inscription* artefact extends correspondingly the *OffsetType* metaclass of the core metamodel and inherits its properties; i.e. *offsetX* and *offsetY*. These properties indicate the offset placement of inscriptions with respect to the position of the place.

The *Node* artefact is defined also in the metamodel as the parent of the *Transition* metaclass. Therefore, transitions contain the equivalent set of properties as for the case of places. Despite that fact, the description of transitions in the metamodel differs slightly, since the abstract *Transition* metaclass is defined as the parent of six distinct subclasses. The *DocumentRoot* aggregations, namely *basics, downlinks, uplinks, actions, objects and guards*, are associated to the corresponding subclasses. This provides the capability to distinguish between the different types of transitions and their respective discrete expressions. Furthermore, it allows querying effectively the metamodel structure through the aggregations and associations that lead directly to the required properties. This is extremely beneficial for the imminent development of the text-based generators that are used for the transformation of Petri Net-based process models.

Every transition subclass, excluding the basic transition, is associated to a specific expression metaclass that carries a related inscription. For instance, the *Uplink* transition is related via the *uexpression* association to the *UExpression* metaclass. The association indicates that all uplink transitions must contain at least one expression $(1...1)$. Expressions are defined using the *value* property of the *Inscription* metaclass, which comprises the parent of the *UExpression* element. Using an identical approach the definition of the rest of the transitions is effectively accomplished.

The connections between places and transitions are defined via the use of arcs, which are described in the metamodel using the *Arc* metaclass. Each instance of the *Arc* element is

linked to a particular source node via the *arcsource* association and to a specific target node via the *arctarget* association. Moreover, the *Arc* element extends both the *ArcContent and LineType* metaclasses defined in the core metamodel. The *ArcContent* element includes the *id, source and target* properties, which denote the unique identifier and the exclusive source and target nodes for any specific arc. Furthermore, the *LineType* metaclass contains properties that are concerned with the graphical representation of the arc line; i.e. *style, shape, width* and *colour*. Finally, each arc is related via the *attribute* association to a specific attribute (i.e. arc weight), which is defined via the use of the *Attribute* metaclass. In particular, the *value* property is used for the definition of the arc weight since the *Attribute* metaclass extends the *Inscription* element.

Additional utility metaclasses and properties are defined in the metamodel with the aim to simplify the development of the generators, which facilitate the transformation of Petri net models to the corresponding output document. For instance, the *OExpression* metaclass includes the *object* property that designates the model that contains this object instance; i.e. the context or presentation model. Another example is depicted by the *Function* metaclass that is extended by the downlink expression and includes the properties *funName* and *funType*. These properties are defined in accordance to the downlink expression and depict the name and type of the method invoked. Similarly, the rest of the artefacts (*i.e. Common, Argument*) are defined in the metamodel in order to simplify the transformation of process models.

5.4.6 Petri Net-based Process Modelling Language Constraints Definition

The proposed model-driven Petri net based process supports the validation of Petri Net models at two distinct levels. At the first level, the PN-PML metamodel illustrated in

Figure 5.11 is enriched with OCL constraints. The imposed constraints provide the capability to validate the static structure and syntax of the process models. Furthermore, at the second level the transformation of process models to the equivalent PNML syntax provides the capability to import the models into a Petri net tool (i.e. software simulator) and validate accordingly their operational semantics.

Appendix J illustrates the complete list of constraints imposed onto the process modelling language using the OCL. A number of descriptive example constraints are presented next to demonstrate their significance in the definition of a coherent modelling language. The primary constraint is imposed in the context of the *Arc* metaclass, so as to be able to query all instances of the metaclass. Subsequently, the collection of arcs present in the model definition is compared in order to determine any cyclic relationships between dissimilar nodes. More specifically, the constraint indicates that if the target node of the arc instance *a1* is equal to the source node of the arc instance *a2*, then the corresponding target node of *a2* cannot be identical to the source node of *a1*.

context Arc

inv: Arc.allInstances()->forAll(a1:Arc, a2:Arc | a1 <> a2 **and** a1.arctarget = a2.arcsource **implies** a2.arctarget <> a1.arcsource)

The Petri net theory does not permit under any circumstances to have direct associations between places or transitions. The definition of the following two constraints ensures that this specific rule is preserved in the definition of the process modelling language and that the designer is not permitted to associate elements of the same metatype. Using the *oclIsTypeOf* operation the source and target nodes of every arc are queried and evaluated, in order to ensure that the source node definition is not identical to the target node definition. This means that in the case the source node is defined as an instance of the

Place metaclass the target node cannot be defined as an instance of the same metaclass. Furthermore, if the source node is defined as an instance of the *Transition* metaclass then the target node should not be defined as an instance of the same metaclass.

context Arc

inv: self.arcsource.oclIsTypeOf(Place) <> **self**.arctarget.oclIsTypeOf(Place)

inv: self.arcsource.oclIsTypeOf(Transition) <> **self**.arctarget.oclIsTypeOf(Transition)

The fourth constraint supports the evaluation of synchronous channels present in the Petri-net based model definition. The *Downlink* metaclass is defined as the metamodel element target for the specification of this particular constraint. Consequently, every downlink transition is evaluated against all instances of uplink transitions to identify if there is a matching uplink transition. This is accomplished by comparing the syntax of every uplink expression property to the syntax of the downlink expression property. For instance, a constraint violation is raised in the case that the expression of a downlink transition is defined as "*this:execute(a,b)*" and none of the uplink transitions expressions is not defined as "*:execute(a,b)*".

context Downlink

inv: Uplink.allInstances()->forAll(u: Uplink | u.uexpression->exists(uexpr: UExpression |
uexpr.value = **self**.dexpression.value.substring(5, **self**.dexpression.value.size())))

The specification of the appropriate constraints onto the PN-PML metamodel concludes the definition of the abstract syntax of the modelling language. Furthermore, the concrete syntax of the modelling language is generated by automatic transformation of the abstract elements of the metamodel to graphical components. This allows producing the mapping metamodel that supports the generation of the Petri net-based Process Modelling Framework. The PN-PMF comprises of the modelling language as its backbone and a

suitable modelling editor at the front-end, which supports the design and validation of process models.

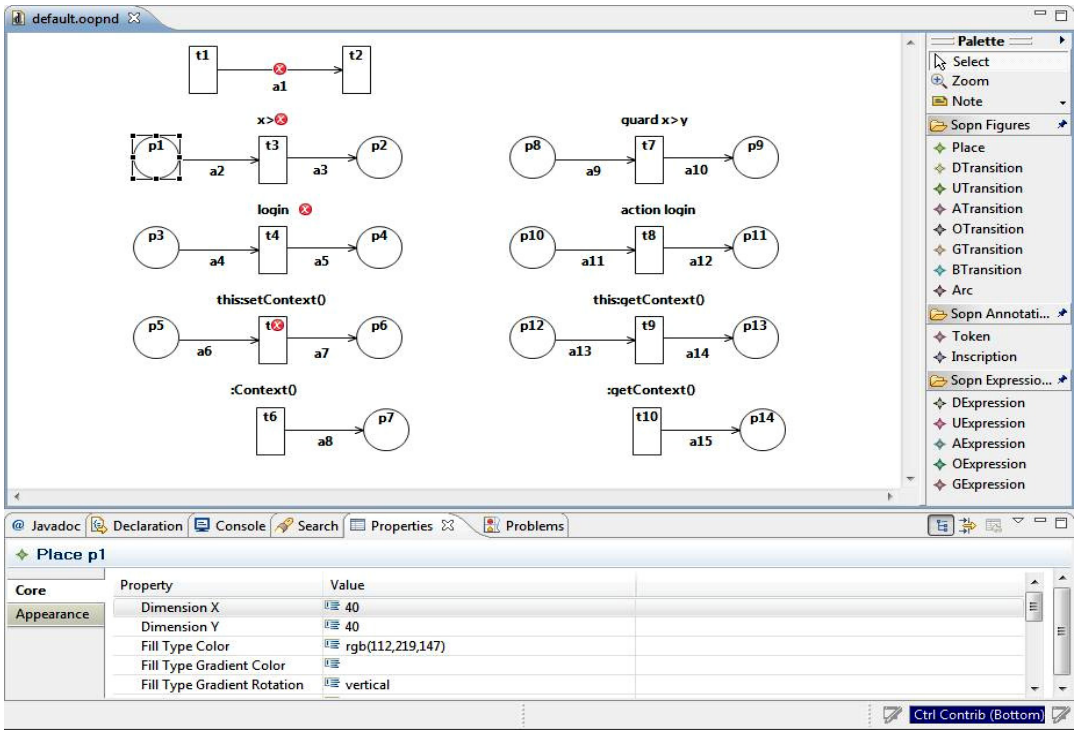


Figure 5.12: Running instance of the Petri net Process Modelling Framework.

Figure 5.12 illustrates a running instance of the PN-PMF that presents the modelling editor and the associated properties view used for the definition of process models. The example process model definitions presented onto the figure demonstrate the capability of the modelling framework to enforce the imposed constraints and validate the static structure and syntax of the model. Initially, the *arc instance a1* is marked with a validation decorator, which describes the error in the definition since transitions cannot be associated to transitions. The guard transition t_3 violates an additional constraint, since the definition of the expression does not start with the "*guard*" statement. Correspondingly, transition t_7 illustrates an appropriate definition for a guard transition. Furthermore, transitions t_4 and t_8 describe an erroneous and a correct definition for an

action transition. The downlink transition t_5 is also marked with an error decorator since the uplink transition t_6 is not appropriately defined so as to setup the synchronous communication channel. In contrast, the expressions of transitions t_9 and t_{10} are properly defined, in order to setup the synchronous channel formed between the two transitions.

5.5 Template-based Generators Development

5.5.1 Petri Net Markup Language Document Generator

This subsection presents the definition of the template-based generator using the oAW component of the PSCE. The generator facilitates the transformation of process models to the equivalent PNML syntax. Consequently, the PNML representation provides the capability to import the models into the selected Renew Petri net tool [116]. The Renew simulator supports the complementary validation phase, since it provides the capability to validate the operational semantics of the process models via model execution.

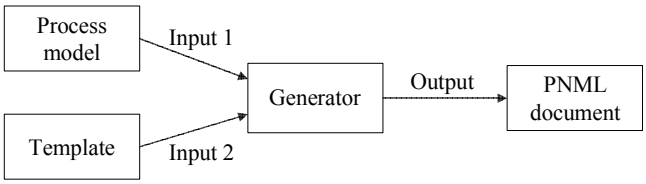


Figure 5.13: PNML format generation process.

Figure 5.13 illustrates the process for the generation of the PNML document from the model. Both the process model and the template definition are accepted as inputs by the PNML document generator. The definition of the PNML template (i.e. Appendix K) is explained in this subsection using the pseudocode definition provided for the place and downlink transition elements of the process model. In specific the pseudocode definition

allows explaining how the transformation of process models to the corresponding PNML document is accomplished.

```
1.  FOREACH places AS place
2.      Read and generate place id from model
3.      IF place.token != null
4.          Read and generate the place initial marking from model
5.      ENDIF
6.      Read and generate the place graphic properties from model
7.  ENDFOREACH
```

Figure 5.14: Pseudocode for the transformation of places.

Figure 5.14 illustrates the pseusocode definition that guides the transformation of place elements to the respective XML representation. The definition starts by defining a loop that allows iterating and querying the collection of places; i.e. places aggregation. Hence, each place and its properties are parsed and loaded into memory and assigned the variable name *place*. This provides the capability to read primarily the *id* property of the place and generate the corresponding XML tag in accordance to the PNML document structure. The following logical expression (i.e. line 3) allows evaluating if the place is associated with a token, so as to read its value and generate accordingly the required XML output. Furthermore, the graphical properties defined for that particular place included in the collection are read from the model and generated in the form of XML tags.

```
1.  FOREACH downlinks AS downlink
2.      Read and generate downlink id from model
3.      IF downlink.expression != null
4.          Read and generate the expression graphic properties from model
5.          Read and generate the expression value from model
6.      ENDIF
7.      Read and generate the downlink graphic properties from model
8.  ENDFOREACH
```

Figure 5.15: Pseudocode for the transformation of downlink transitions.

The pseudocode presented in Figure 5.15 defines an additional part of the transformation that supports the translation of downlink transitions to PNML syntax. Primarily a loop is defined that facilitates iterating and querying the collection of downlink transitions; i.e.

downlinks aggregation. The loop allows querying each downlink transition, reading and parsing its individual properties. Therefore, via the *downlink* variable assignment the *id* property of the transition can be read and generated in the form of an XML tag. In addition, the evaluation of the logical statement determines if a corresponding expression is associated to the transition, so as to generate its graphical properties and semantics. Finally, the graphical properties of the downlink transition are read from the model and generated also in the form of XML tags.

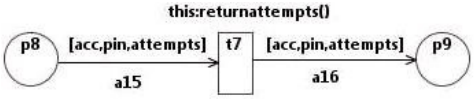


Figure 5.16: Extract of the example Bank service process model.

Apart from the above definitions, the PNML template includes analogous definitions for the transformation of the rest of the elements of the modelling language; see Appendix K. Figure 5.16 presents an extract of the process model illustrated in Figure 5.10, which serves as an illustration of the process model to PNML syntax transformation process. In particular, emphasis is given on the downlink transition t_7 presented in the model and how it is transformed to an equivalent PNML representation.

```
<transition id="t7">
    <downlink>
        <graphics>
            <offset x="0" y="-30" />
         </graphics>
        <text>this:returnattempts()</text>
     </downlink>
    <graphics>
        <position x="696" y="516" />
        <dimension x="25" y="45" />
        <fill color="rgb(112,219,147)" />
        <line color="rgb(0,0,0)" />
    </graphics>
</transition>
```

Figure 5.17: The PNML representation of a downlink transition.

Figure 5.17 presents the XML-based representation of the downlink transition t_7, following the execution of the transformation. The generated output shows the mapping of the properties of the downlink transition to equivalent properties defined in the form of XML tags. Consequently, the operational semantics and the graphical properties of the process model are unquestionably preserved. This allows to import the process model defined using the PN-PML modelling framework into the Petri Net simulator in order to perform the model validation.

Figure 5.18 illustrates the simulation of the process model illustrated in Figure 5.10 using the Renew Petri Net tool, which allows validating the model to guarantee its correctness. More specifically, the validation is performed by checking the abstract syntax and the operational semantics of the model by means of simulation. The validation is supported by the simulator since the model conforms to the PNML standard, which forms the basis of the PN-PML.

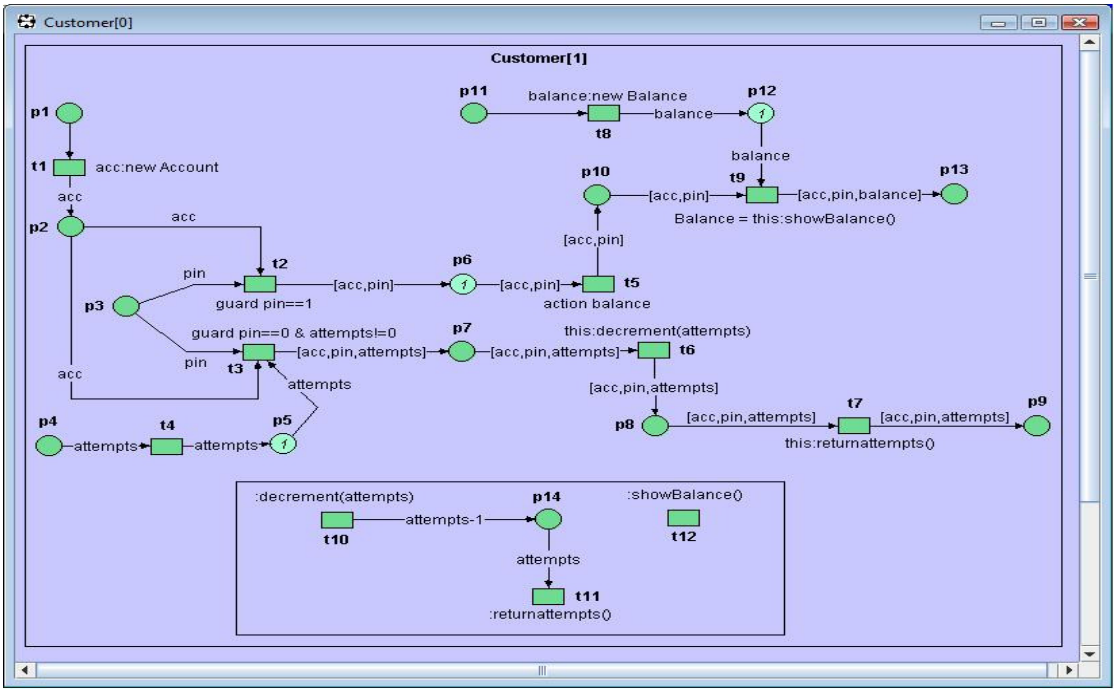


Figure 5.18: Bank service process model execution.

In addition to the transformation of process models to the required format for validation specific purposes, the capability to transform process models to different implementation technologies is also provided. The objective of the two complementary levels of validation applied to process models is to fully guarantee their consistency prior to the service implementation phase. This provides the capability to generate non-erroneous service code directly from the process models. The following subsection describes the definition of the templates that steer the platform specific code generators and facilitate the generation of the service implementation from the process models.

5.5.2 Platform Specific Code Generators Templates Definition

The model-to-code generation process follows an analogous procedure to the PNML generation process illustrated in Figure 5.13. In particular, the definition of the J2ME and Java templates is required to guide the transformation of process models to the respective implementation technology. The code generator accepts as inputs the process model and the templates definition and executes accordingly the transformation. Subsequently, the code generation process produces either J2ME or Java code depending on the input templates invoked by the code generator. Note that, the executable script of the generator includes references to both the model and the corresponding template.

Figure 5.19 presents parts of the J2ME template definition using pseudocode, which aids the rationalization of the model-to-code generation process. The primary branch of the pseudocode (i.e. lines 1-7) guides the generation of the necessary statements that import the J2ME presentation or context implementation classes. Foremost, a loop is defined that allows iterating and querying object transitions and declares the variable name *objectnet* to each transition in the collection. Hence, the expression *model* property can be queried,

read and subsequently the logical condition can be evaluated. In the case the property value is equal to *"presentation"* the corresponding statement is generated that imports the J2ME implementation classes of the presentation model. On the contrary, if the property value is equal to *"context"* the alternative code statement is generated that imports the J2ME implementation classes of the context model.

```
1.  FOREACH objects AS objectnet
2.      IF objectnet.expression.model == "presentation"
3.          Read properties from model and generate the import statement
4.      ELSEIF objectnet.expression.model == "context"
5.          Read properties from model and generate the import statement
6.      ENDIF
7.  ENDFOREACH
```

```
1.  FOREACH action expression AS aexpr
2.      Read properties from model and generate action command
3.      FOREACH downlink expression AS dexpr
4.          IF aexpr.eventSource == dexpr.eventSource && dexpr.funType != "void"
5.              Read model properties and generate the analogous method call
6.               IF dexpr.eventSource == aexpr.eventSource &&  dexpr.argDatatype != null
7.                  Read model properties and generate the arguments
8.               ENDIF
9.           ELSE IF aexpr.eventSource == dexpr.eventSource && dexpr.funType == "void"
10.             Read model properties and generate the analogous method call
11.              IF dexpr.eventSource == dexpr.eventSource && dexpr.argDatatype != null
12.                 Read model properties and generate the arguments
13.              ENDIF
14.         ENDIF
15.     ENDFOREACH
16. ENDFOREACH
```

```
1.  FOREACH downlink expression AS dexpr
2.      IF dexpr.funType == "void"
3.          Read model properties and generate the void method
4.          IF dexpr.argDatatype != null
5.              Read model properties and generate the method's arguments
6.          ENDIF
7.      ELSE
8.          Read properties from model and generate the return method
9.          IF dexpr.argDatatype != null
10.             Read model properties and generate the method's arguments
11.         ENDIF
12.         IF dexpr.funType != "void"
13.             Read model properties and generate the return statement
14.         ENDIF
15.     ENDIF
16. ENDFOREACH
```

Figure 5.19: J2ME template pseudocode definition.

The second branch of the pseudocode (i.e. lines 1-16) describes the generation of the *commandAction* method. This method is called when using the pervasive service to handle commands invoked by the user on a particular displayable; i.e. J2ME GUI component. The pseudocode defines two iteration loops that allow querying each action and downlink transition expression by assigning the necessary variable names. Hence, the properties of these expressions steer the code generation since they determine the validity of the logical statements. For instance, the first statement (i.e. line 4) describes the logical condition that guides the generation of a corresponding method call that returns an object. The second statement (i.e. line 6) describes a complementary logical condition that guides the generation of the arguments associated with the method call. In terms of the secondary part of the conditional statement (i.e. lines 9-16), the same logical reasoning applies but the generated method call invokes a method of type void.

Finally, the third branch of the pseudocode (i.e. lines 1-16) guides the generation of the code stubs for the methods of the process implementation class. First, a loop is defined that allows querying the expressions of the downlink transitions defined in the model. The first part of the statement (i.e. line 2) defines the logical condition that guides the generation of a void method. Furthermore, a complementary logical condition (i.e. line 4) is defined that guides the generation of the arguments associated with the void method. Concluding, the second part of the statement (i.e. lines 7-15) follows the same reasoning but the generated method returns an object; i.e. not a void method.

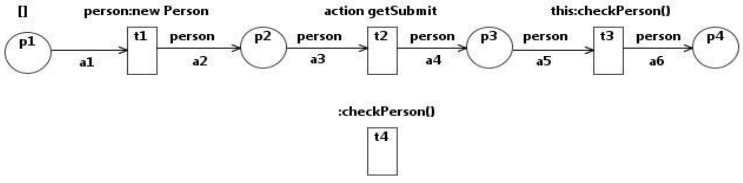


Figure 5.20: Example process model for explaining code generation.

Figure 5.20 illustrates an extract of the process model defined in Figure 5.10, which provides the capability to exemplify the model-to-code generation process. The object transition t_1 is transformed to the equivalent import statement (i.e. line 1) illustrated in Figure 5.21, since the expression *model* property is defined in the form of a String literal "*context*". Moreover, the expression of the action transition t_2 is transformed to the conditional statement shown in the generated code; i.e. line 5. Respectively, the expressions of the downlink transition t_3 and the uplink transition t_4 drive the generation of the method call (i.e. line 6) and its corresponding method stub (i.e. lines 18-20). In this case the execution of the code generation produces a method of type void that essentially carries no arguments.

```
1.  import j2me.context.entities.Person;
2.                      |
3.                      |
4.  public void commandAction(Command command, Displayable displayable) {
5.                  if (command == this.getSubmit) {
6.                          checkPerson();
7.                  }
8.                  if (command == this.setSubmit) {
9.                          setPerson(Person person);
10.                 }
11.                 if (command == this.getHspref) {
12.                         preferenceForm  = getPreferenceForm();
13.                 }
14.                 if (command == this.getPreferences) {
15.                         sitePreferences  =  setSitesPreferences(Vector sites);
16.                 }
17. }
18.  public void checkPerson() {
19.         //TODO Auto-generated method stub
20. }
```

Figure 5.21: Example generated J2ME code.

The generated code illustrated in Figure 5.21 presents three additional cases of model-to-code transformation, which are representative of the conditional statements defined in the second branch of the J2ME pseudocode; see Figure 5.19. More specifically, the supplementary code presented in Figure 5.21 is not generated from a particular process

model but it aims to demonstrate the miscellaneous cases of generating method calls with and without arguments. The generated statement at line 9 represents a call to a void method that incorporates also an argument. Following, the generated statement at line 12 represents a method call that carries no arguments but returns an object; i.e. J2ME Form object. In the final case, the generated method call (i.e. line 15) carries related arguments and returns in addition the object associated with the method.

5.6 Summary

In this chapter a Petri Net Process Modelling Framework is introduced, which is generated from the PN-PML metamodel definition and supports the design, validation and implementation of the dynamic behaviour of pervasive services. Moreover, the model-driven Petri net based process is introduced in detail, which facilitates the integration of the modelling domains (i.e. context, presentation) via the definition of the Petri Net-based Process modelling language. This integration allows combining the static structure with the dynamic behaviour of pervasive services, so as to provide an unambiguous service specification. The PN-PMF comprises the third component of the PSCE, which along with the proposed process formulate the overall MDPNF that supports pervasive service creation at the static compile time. The next chapter performs a quantitative evaluation of the defined MDPNF via an example scenario that presents the development of a pervasive service prototype.

Chapter 6 Evaluation of the Model-Driven Petri Net based Framework

In the previous chapter the Petri Nets formalism was combined with the MDA paradigm to provide a unified model-driven approach for pervasive service creation. This chapter serves as an evaluation of the Petri Net based model-driven development process and it's supporting Pervasive Service Creation Environment. The evaluation provides evidence as to the benefits the approach delivers for the efficient creation of pervasive services at the static compile time. This is performed by undertaking an exemplary case study that involves the development of an interactive pervasive service prototype. The approach is evaluated with the aid of the Petri Net formalism and using well-known software metrics, which provide evidence as to the effectiveness of the approach in reducing the overall pervasive service creation overheads.

6.1 Introduction

The pervasive service case study presents the execution of the Steps 5-8 of the domain-specific MDD methodology introduced in Chapter 3. This concludes effectively the pervasive service creation lifecycle, which engages from the definition of the languages and the generation of the modelling frameworks that compose the overall PSCE. First, the pervasive service models (Step 5) are defined that represent the graphical user interfaces, the context management system and the dynamic behaviour of the pervasive service. The validation of the models (Step 6) is then performed via the enforcement of OCL constraints and the execution of the pervasive service behaviour using the Renew Petri Net simulator tool. Concluding, the transformation of the pervasive service models to platform specific implementation code (Steps 7-8) is undertaken with the aid of the defined template-based code generators.

6.2 Case Study: Pervasive Museum Interactive Service Overview

For the purpose of the case study a pervasive service is developed that represents actually an interactive and dynamic museum software guide. The pervasive service aims to ease the visitors touring experience by providing useful information regarding the historic sites, the facilities and the forthcoming exhibition tours available at the museum. This information describes the set of circumstances that surround a particular individual such as situations and events, which guide the interaction of the user with the service.

For instance, the occurrence of a contextual situation such as the entry of a user within a historic museum site results in conveying historical information about this site to the user. Moreover, the pervasive service should allow realising in which historic site the user is currently located, the device that he is using and the power level of his device. Therefore,

in accordance to the location, device type and power level the analogous historical information must be displayed to the user using an appropriate format; i.e. video, or text. In this case study the museum is divided into four virtual zones that depict the different historic sites of the museum. Hence, the proximity sensors placed in each virtual zone provide the capability to detect the presence of the user and subsequently perform the appropriate tasks to deliver the analogous information. Note that, the development of pervasive services at the application level relies on the critical assumption that the implementation of low-level infrastructure-based functionality exists.

An additional example that could change the contextual situation of the user is a time-specific context event. For instance, in the case that the time is fifteen minutes prior to the museum closedown, the necessary context event is generated that acts as a notification to the user. Consequently, the user is informed via a message displayed onto his mobile device, to proceed shortly to the nearest exit. In addition to implicit contextual situations, the pervasive service allows the user to explicitly request information concerning historic sites that are in exhibit within the museum. More specifically, the example pervasive service provides to the user the functionality that allows selecting the preferred sites for displaying relevant historical information.

Therefore, the context, presentation and Petri Net process models should describe precisely the functionality of the pervasive museum service, which involves details such as context information, contextual situations, graphical user interfaces and the behaviour of the service. These abstract concepts captured in the models would drive eventually the generation of the implementation, which allows enforcing at runtime the functional requirements of service adaptability, user-service interaction and the dynamic service

behaviour during runtime [7]. Furthermore, the time, effort and cost required to develop the pervasive service and the service portability are two fundamental non-functional properties that are also considered [7]. The following subsections present the definition of the pervasive service models.

6.2.1 Presentation Model

The primary model defined for the pervasive museum service represents the components, properties and associations that specify the graphical user interfaces of the service. These GUIs are the front-end of the pervasive service and are considered as the visual representation of the service functionality. Hence, the user exploits the GUIs to interact with the pervasive service and perform various computing tasks. The presentation model does not define though the actual functionality that the pervasive service supports but provides rather an abstract notion of the computing tasks involved.

Figure 6.1 illustrates the presentation model that represents the graphical components properties and associations of the service. At the heart of the model definition is the *PsMuseum Display* element, which represents the main displayable that comprises the rest of the graphical containers. Apart from the *name* property of the *Display* instance, which is apparent in the figure, the concealed element properties are set accordingly using the properties view of the PMF. The display element includes four container components that represent either Java *JPanel* components or J2ME *Form* components.

Furthermore, the display element is directly associated with the *preference List* and the *setPreferences Button* graphical components. The restriction imposed by J2ME denotes that any *List* component must be placed within the actual display of the mobile device and not within a container. This does not apply in the case of Java since a List can be

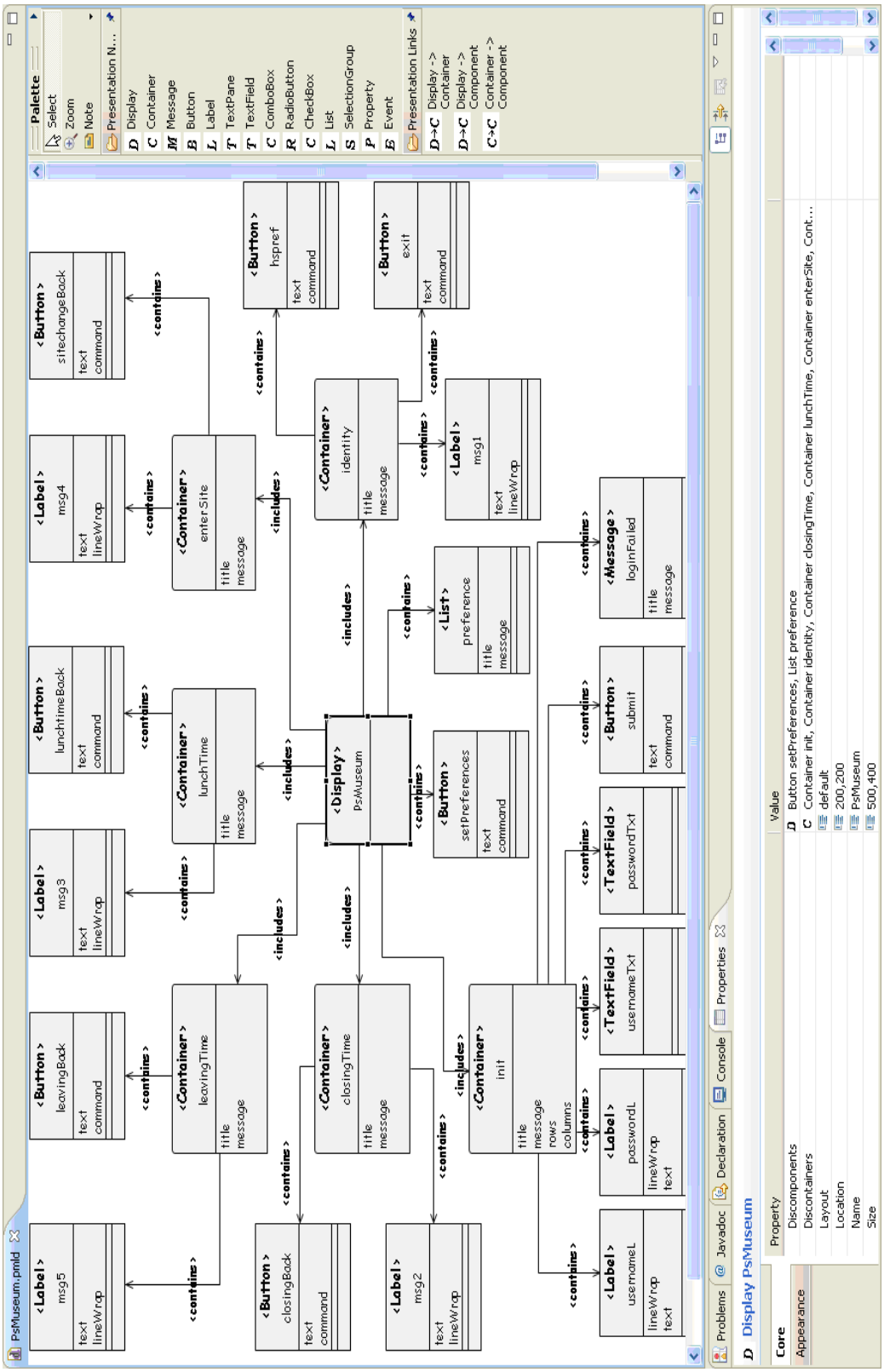


Figure 6.1: Pervasive Museum Service Presentation Model.

associated to either a frame or a container. In order to preserve though the abstract nature of the model and its applicability to both technologies the *List* is associated to the main display. Moreover, the *Button* element depicts the component directly associated to the *List* that allows executing an action; i.e. set the preferences of the user.

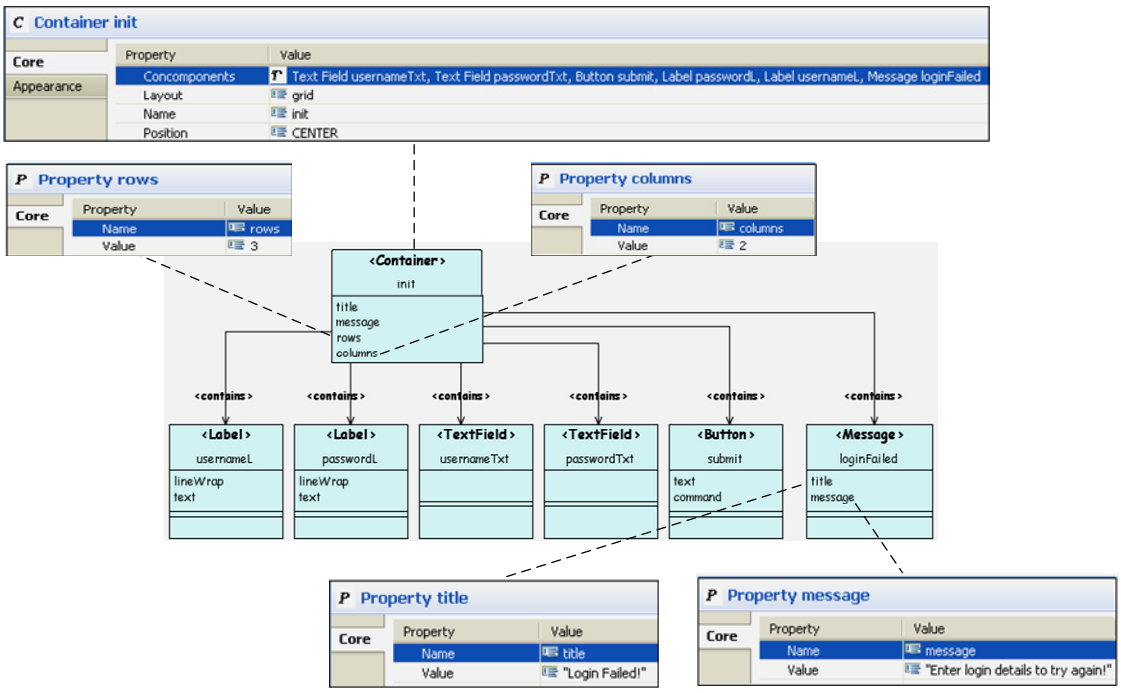


Figure 6.2: Presentation model Container element and properties.

Due to the detailed nature of the model design the *init Container* is selected to illustrate the definition of the element, its properties and relationships in order to clarify how these are utilised for source code generation. The remaining containers are roughly described in order to disclose their meaning in the overall model specification. Figure 6.2 presents the selected *Container* component and illustrates in detail some of the abstract properties of the GUI model definition. The container composes actually the initial dialog that accepts the user credentials and performs the authentication prior to using the pervasive service. Foremost, the common properties of the *init Container* instance are displayed on the top

of the figure via the container's properties view. The attributes represent respectively the associated components, the layout, the name and the position of the container.

In specific the collection of components is updated automatically following the definition of the associations shown visually via the containment arrows illustrated in the figure. Furthermore, the layout of the container is defined as *GridLayout* and subsequently the supplementary *rows* and *columns* properties are respectively defined. These properties specify the layout of the container for the Java implementation, which is defined as a grid with three rows and two columns. The property views of the rows and columns secondary attributes are also illustrated in Figure 6.2.

Moreover, the name of the container is defined as *"init"* and its position in terms of the *Display* element is specified as *"CENTER"*. This designates that the container is placed at the centre of the display's layout, which is defined as *BorderLayout*. Once again, it is denoted that the layout properties are essential in terms of the Java implementation, whereas in the case of J2ME the mobile device default layout manager arranges the placement of the components.

An additional element demonstrated in detail in the definition is the *Message* component, which comprises of the *name* primary property (i.e. "loginFailed") and the *title* and *message* secondary properties. These properties are mandatory for the component definition in order to satisfy the restrictions and the requirements of the two technologies considered in this work. The primary property designates the name of the message dialog (or alert message in J2ME), while the secondary properties determine the title and the message that will appear on the information dialog. Note that this GUI component is displayed only in the case the user enters the wrong authentication credentials.

Prior to the transformation of the presentation model to the respective implementations the validation of the model is performed by enforcing the imposed OCL constraints. The model validation was executed using the modelling editor of the PMF to ensure that the designed model does not comprise any inconsistencies. For this particular model, the validation did not raise any problems and subsequently it was not required to perform any modifications to the model. In the case though that errors are detected in the definition these have to be resolved before undertaking the model-to-code generation phase.

```
public Form getInitForm() {
        init = new Form("Connecting...");
        init.setTicker(new Ticker("Enter info to connect"));

        usernameL = new StringItem("Username:", null);
        passwordL = new StringItem("Password:", null);

        usernameTxt = new TextField("", null, 20, TextField.ANY);
        passwordTxt = new TextField("", null, 20, TextField.ANY);

        submit = new Command("Submit", Command.OK, 0);
        loginFailed = new Alert("Login Failed!",
                "Enter login details to try again!", null, AlertType.ERROR);
        /** TODO starts
         */
        init.append(usernameL);
        init.append(getUsernameTxt());

        init.append(passwordL);
        init.append(getPasswordTxt());
        passwordTxt.setConstraints(TextField.PASSWORD);
        /** TODO ends
         */
        init.addCommand(submit);
        return init;
}
```

Figure 6.3: The J2ME implementation of the example container.

Figure 6.3 presents the generated source code obtained from the transformation of the *init Container* element to the operational semantics of the J2ME implementation technology. The source code showcases a method that returns a *Form* component, which is composed by two *StringItem* components and two *TextField* components. Furthermore, the *Form* comprises of a *Command* component that encapsulates the semantic information of a user

command that activates correspondingly the required action. The generated code includes two comment statements that denote that the code required for appending objects to the form is to be manually implemented by the developer. In addition the password *TextField* necessitates defining constraints that restrict the behaviour of the text entry and display. This is performed via the "*setConstraints()*"statement that indicates via the constraint parameter that the user text entry should be obscured. Finally the command is added to the form and the concluding statement returns the created component.

The statement that creates an instance of the *Alert* object is intentionally omitted from the above description since this component is not essentially appended to the form. In fact the alert message is displayed to the user only in the case that the authentication information provided are erroneous. The generated code declares the *Alert* object as a global variable. Hence, the developer is allowed to use accordingly the generated component and display the information message when required. This practically means that the developer must implement manually this graphical event in accordance to the dynamic behaviour of the pervasive service.

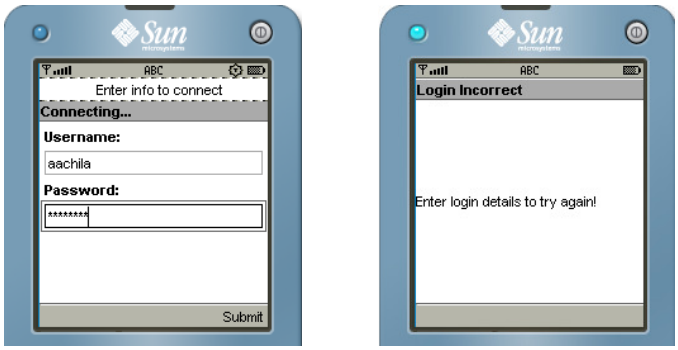


Figure 6.4: The J2ME authentication Form and Alert message.

Figure 6.4 presents the runtime instances of the source code generated from the extract of the presentation model; see Figure 6.2. The displayable shown on the left represents the

login *Form* that allows the user to enter the required information and sign in to use the service. Correspondingly, when invalid authentication details are inputted by the user, the runtime *Alert* object shown on the right is displayed to notify accordingly the user of the authentication error. The message is displayed to the user for a few seconds and then returns to the previous state providing the capability to re-enter the login credentials. The expected behaviour of the service does not enforce any restrictions as to the number of incorrect attempts that a user is allowed.

Similarly, to the transformation of the example container and its associated components, the complete model presented in Figure 6.1 is translated to the respective source code. The template-based code generation process provides the capability to transform the model to the required implementation technology. Apart from the case study model, the generic templates definition (i.e. J2ME or Java) facilitates the transformation of different presentation models to platform specific implementations. This is feasible since the mapping is defined on the basis of the PML metamodel definition and the operational semantics of the corresponding output programming language.

6.2.2 Context Model

In the previous subsection the presentation model was introduced, which represents the GUIs of the pervasive service. Furthermore, the translation of a selected part of the model was described in order to demonstrate the transformation capabilities of the PMF. The context model is introduced in this subsection, which represents any information relevant to the interaction of the user with the pervasive service. In particular the context model defines: conceptual entities, context information, heterogeneous input context sources, temporal constraints, contextual situations, complex datatypes and enumerations.

Figure 6.5 presents the context model designed for the pervasive museum service using the CMF. The core element of the model definition is the *Person* entity that designates any particular user of the pervasive service. In the model the *Person* entity is associated with relevant context information via the context source elements, which are defined as instances of the *ContextAssociation* metaclass. For instance, every user is associated via the *identity* context association to the *Identity* context information. This denotes precisely that every user is associated with a respective personal profile.

The *identity* context association depicts the necessary properties that characterise the input source from which profile information can be acquired. Foremost, the *type* of the source is defined as *Static*, something that denotes that the user's profile information is stored within a context repository. In addition the *multiplicity* property of the source is set as *1...1*, which depicts that a single profile exists for each potential user of the service.

Apart from the properties shown visually for the *identity* source, there are also important attributes that can be defined and observed from the properties view of the CMF editor. For example the *persistence* property of the context source is defined as *fixed*, which signifies that profile information is permanently stored in a context repository without changing for large periods of time. Moreover, the *multiplicityType* property is defined as *unique*, which determines that a distinct profile exists for every user. Also the *permission* property is defined as *private*, which denotes that merely the user can access this context information. Finally, the associated profile is defined as an instance of the *Identity* context metaclass and comprises of the *forename, surname and email* properties, which are defined as *String* primitive datatypes.

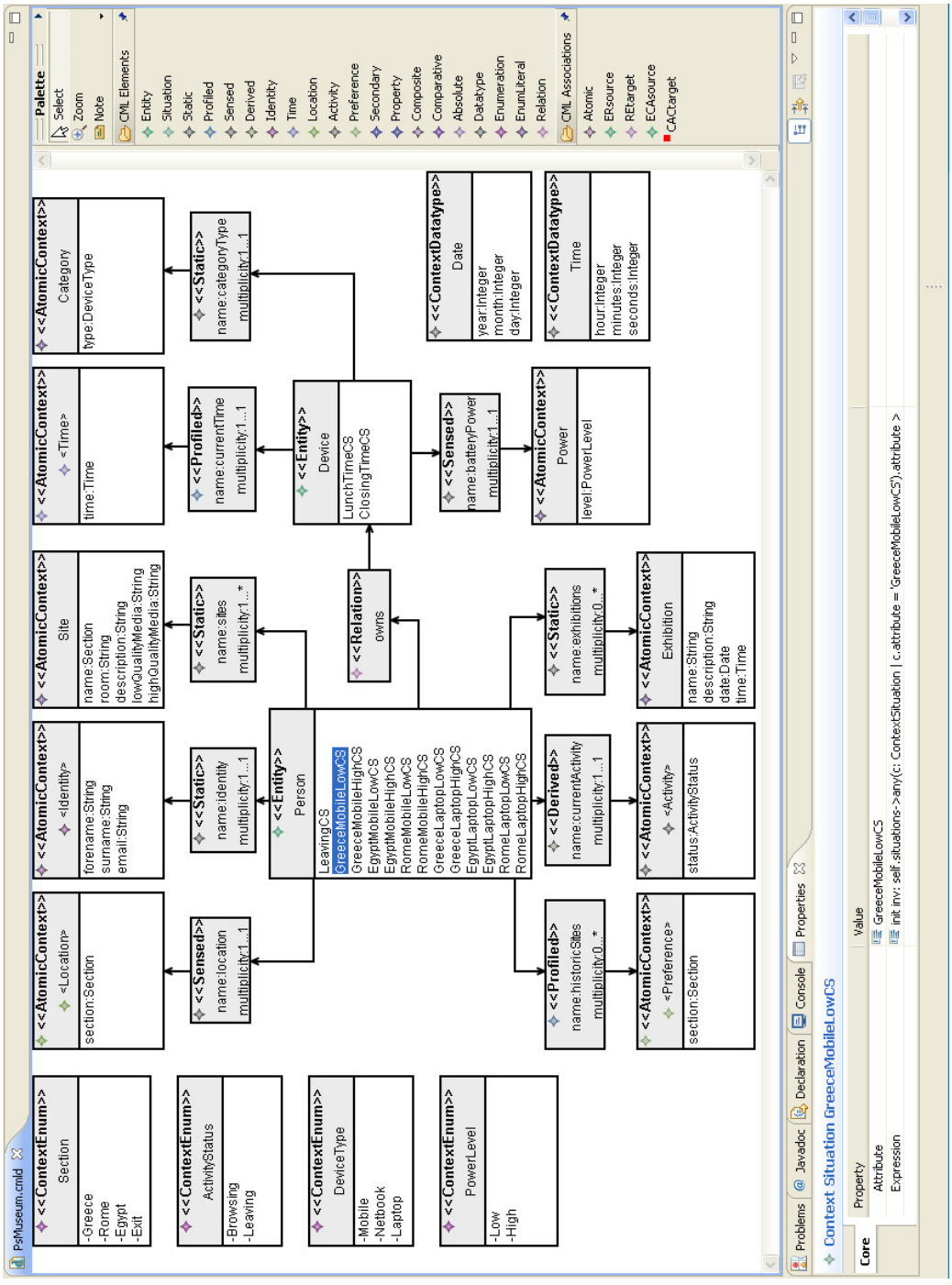


Figure 6.5: Pervasive Museum Service Context Model.

In addition to the identity context source, two additional static sources are associated to the *Person* entity defined in the context model. The *sites* static source defines a context repository that contains historical information for each of the historic sites that are in exhibit at the museum. Furthermore, the *exhibitions* context source designates also a static repository that includes records of the exhibitions (i.e. tours) taking place at the museum within the upcoming calendar month. The *sites* context source associates the entity to the *Site* atomic context while the *exhibitions* context source associates the entity to the *Exhibition* atomic context.

More specifically, every historic site is defined as an instance of the *Site* secondary atomic context metaclass and includes the *name, room and description* properties. The room and the description properties are defined as *String* primitive datatypes, whereas the name property is defined as an enumeration. This enumeration can be assigned the literal values defined by the *Section* enumeration; i.e. *ContextEnum* metaclass instance. Moreover, each exhibition is specified by the *Exhibition* secondary context element and comprises the *name, description, date and time* properties. The name and description attributes are *String* primitive datatypes, while the date and time attributes are complex context datatypes; i.e. *ContextDatatype* metaclass instances.

Aside from static context sources, sensed and profiled sources are associated to the *Person* entity. The *location* context source associates each user to the *Location* context information, which describes essentially the position of the user within the museum. As aforementioned, the museum is separated into four virtual zones that depict the possible locations of the user within the museum. These locations are captured as raw data by proximity sensors and processed to obtain a high-level description. Consequently, the

section attribute can be assigned any of the literals defined by the *Section* context enumeration that represents actually the location's high-level context description.

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

Furthermore, the context model includes the *currentActivity* source that associates the user with the *Activity* context information, derived on the basis of the *Location* context. More specifically, this context source denotes that the activity of the user is directly related to his/her current location. The context derivation rule is expressed in the model via the *expression* property of the *ContextAssocation* metaclass in the form of an OCL constraint. This OCL expression is presented next and allows deriving the *status* context information from the related *Location* context information. More specifically, the *status* property value, which denotes the current activity of the user, is derived by evaluating the conditional part of the OCL expression. In accordance to the specified condition if the user is located at the exit of the museum the status *"Leaving"* is obtained, which denotes that the user is currently leaving the museum premises.

```
context Entity
    derive inv: if self.ECAsource->any(c: ContextAssociation | c.name =  'location').CACtarget.
    conproperties->any(p: Property | p.name = 'section').enumeration.literals->any(l: EnumLiteral |
    l.value = 'Exit').value = 'Exit' then
    self.ECAsource->any(c: ContextAssociation | c.name ='currentActivity') .CACtarget.conproperties
    ->any(p: Property | p.name = 'status').enumeration.literals->any(l: EnumLiteral | l.value
    = 'Leaving').value else '' endif
```

Additional rules are defined in the model for deriving the activity status of the user in accordance to the historic site the user is currently located. The presence of the user in

any of the virtual zones (e.g. Greece, Rome) denotes that the user is browsing the museum historic sites. In particular, the following OCL constraint defines that the activity status of the user is derived to be *"Browsing"* when the user is actually located at the historic site of *"Greece"*. The derivation rules for the other two sites are analogous to the OCL constraint illustrated below. Note that, the rules for deriving the current activity context information are defined merely as textual constraints in the model. Hence, these rules can only be evaluated using the OCL engine (i.e. Interactive OCL console) of the EMF component. The EMF-based implementation of the OCL specification provides the capability to define well-formed invariant constraints, derived attribute and reference constraints and operation constraints. This allows expressing accurately the required context-aware conditions in the context model.

```
context Entity
    derive inv: if self.ECAsource->any(c: ContextAssociation | c.name = 'location').CACtarget.
    conproperties->any(p: Property | p.name = 'section') .enumeration.literals->
    any(l: EnumLiteral | l.value = 'Greece').value = 'Greece' then
    self.ECAsource->any(c: ContextAssociation | c.name = 'currentActivity').CACtarget.conproperties
    ->any(p: Property | p.name = 'status') .enumeration.literals->any(l: EnumLiteral | l.value =
    'Browsing').value else '' endif
```

Each user of the service is also related to his corresponding device, which is defined in the model as an instance of the entity metaclass; i.e. *Device* entity. The association depicts the ownership relationship of the user with his device, on which the pervasive service is deployed and executed. Respectively, the *Device* entity is associated to corresponding information that characterise the user's device. First, the device is associated via the definition of the *currentTime* source to the *Time* context information, which is profiled by the user on his device. This information can be used to adapt the service behaviour in the case that time-specific events are detected, such as notifying the user that the museum would be closing down soon.

Furthermore, each device is associated using the *batteryPower* sensed context source to the *Power* context information, which defines the device power level and it is defined in the model using the *level* property. The property is associated to the *PowerLevel* context enumeration that defines a high-level description (i.e. *"Low"* or *"High"*) for the device power level and is assumed to be acquired, processed and derived using sensors. Finally, each user device is associated using the *categoryType* static source to the *Category* secondary context that denotes the actual type of the device. The *type* property of the *Category* context information obtains its literal values from the *DeviceType* enumeration and can be set to either mobile, netbook or laptop. Both the power level and the device type context allow adapting accordingly the service and determining the appropriate way to present historical information to the user; i.e. textual or media content.

The heterogeneous sources introduced in the model realize the main prerequisite, which is to differentiate between the diverse classes of context information and manage this information as required. Apart from entities, sources and context information the model comprises of contextual situations that are defined explicitly for each entity. Contextual situations designate the necessary condition(s) that must be valid, so as to enable the execution of a corresponding action. The OCL constraint presented next defines one of the contextual situations expressed in the model for the *Person* entity.

```
context Entity
    init inv: self.situations->any(c:ContextSituation | c.attribute = 'LeavingCS').attribute >
    derive inv: if self.ECAsource->any(c: ContextAssociation | c.name = 'currentActivity').CACtarget.
    conproperties->any(p: Property | p.name = 'status').enumeration.literals->any(l: EnumLiteral |
    l.value = 'Leaving').value = 'Leaving' then true else false endif
```

In particular, the above expression describes the following contextual situation: *"When the user decides to leave the museum details of forthcoming exhibition tours within the current calendar month must be presented to the user"*. The evaluation of the logical

expression determines the occurrence (or not) of the contextual situation, in order to undertake correspondingly the necessary action. Note that, the specified constraint defines merely the condition that determines the occurrence of the situation and does not describe the action that should be triggered as a result. In the context of this thesis the dynamic behaviour of the pervasive service that involves the execution of actions is defined using the Petri net process model.

The OCL expression defines that in the case the activity of the user changes from browsing to leaving then the state of the *LeavingCS* variable is set to *true*. This denotes that the user is currently leaving the museum premises. Subsequently, monitoring the variable state using different instances of the generated situation implementation class allows detecting the context event and reacting accordingly. For instance, in the case of the above contextual situation the action triggered as a result of the context change will present to the user the scheduled exhibitions tours taking place within the upcoming calendar month.

Moreover, twelve additional contextual situations are defined for the person entity in the form of OCL expressions. These situations describe the conditions that must be valid so as to execute the appropriate action. For instance, the *"EgyptMobileLowCS"* situation illustrated next defines the following: *"When the user enters the historic site of Egypt and his mobile device power level is currently low, he should be presented merely with a textual description that describes this particular site"*. This constraint describes a contextual situation that provides the capability to detect the location context change and react in accordance to the type and power level of the device, so as to present to the user the appropriate historical information. Accordingly, the rest of the contextual situations

are specified in the model using OCL constraints that describe the conditions that drive the adaptation of the service behaviour.

```
context Entity
    init inv: self.situations->any(c: ContextSituation | c.attribute = 'EgyptMobileLowCS').attribute >
    derive inv: if self. ECAsource -> any(c: ContextAssociation | c.name = 'location').CACtarget.
    conproperties->any(p: Property | p.name = 'section').enumeration.literals->any(l: EnumLiteral |l.value
    = ' Egypt').value = 'Egypt'
    and self.ERsource->any(r:Relation |r.name='owns').REtarget.ECAsource->any(c:ContextAssociation
    | c.name = 'categoryType').CACtarget.conproperties->any(p: Property | p.name = 'type').enumeration.
    literals->any(l: EnumLiteral | l.value = 'Mobile').value = 'Mobile'
    and self.ERsource->any(r:Relation |r.name='owns').REtarget.ECAsource->any(c:ContextAssociation
    | c.name = 'batteryPower').CACtarget.conproperties->any(p: Property | p.name = 'level').enumeration.
    literals->any(l: EnumLiteral | l.value = 'Low').value = 'Low'
    then true else false endif
```

This concludes effectively the model definition phase and kicks off the model validation phase. Consequently, via the use of the capabilities of the developed CMF the validation of the model is performed in accordance to the specified metamodel level constraints. The model validation assists primarily the designer in the definition of unambiguous models, since it allows detecting inconsistencies and refining the models accordingly. Moreover, it aids the developer of the pervasive service since non-erroneous implementations can be generated from the context models. Hence, the resulting code conforms precisely to the operational semantics of the programming language, since it is produced from the validated models and in accordance to the template definition. The enforcement of OCL constraints and the definition of the templates are fundamental in order to ensure that the context model captures the correct semantics.

6.2.3 Petri Net Process Model

This subsection presents the process model of the pervasive service that enables the integration of the modelling domains. The process model represents the states, actions and operations that depict the overall dynamic behaviour of the pervasive service. Apart

from the integration, the model provides a formal method that facilitates the validation of the service behaviour by means of model execution. More specifically, any process model defined using the PN-PMF can be effectively transformed to an equivalent PNML representation. Hence, the generated PNML file can be imported into the Petri Net software tool that provides the capability to simulate the functionality of the service. The validated process model can be then transformed to the corresponding implementation technology, which is considered the lowest-level model that describes the pervasive service behaviour.

Figure 6.6 illustrates the pervasive museum service behaviour in the form of an object-oriented Petri Net model. The objects circulating through the process model are the *Person* and *Device* entities defined in the context model and the *PsMuseumGui* top-level display of the presentation model. These objects are created as object net instances and are represented within the main service net shown in Figure 6.6 via the object transitions t_1, t_2 and t_3. This hierarchical structure of the model provides the capability to design a complex pervasive service net that is composed of various object nets. In the context of object-oriented programming these object references allow the generated process implementation class (i.e. service net) to access the variables and methods defined in these objects. The object references become tokens of the Petri Net model and are depicted via the corresponding *p, d and pmsg* inscriptions.

Subsequently, transition t_4 depicts a downlink transition that is considered the initiator of a communication channel. The definition of this transition is marked by an expression that denotes the service net, the channel and no arguments for the synchronisation; i.e. *"this:getLoginDialog()"*. Correspondingly, the uplink transition t_{19} is the terminating

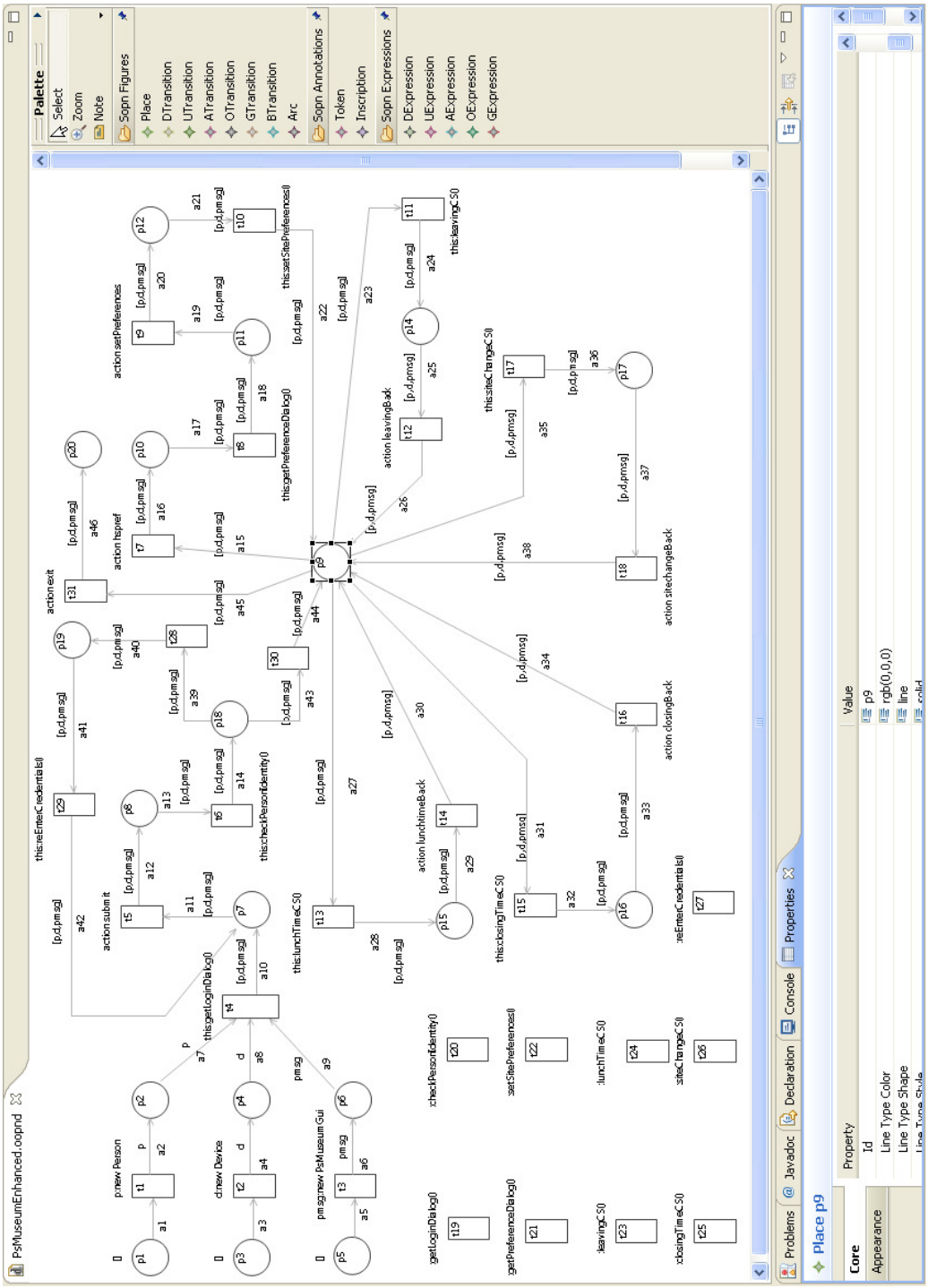


Figure 6.6: Pervasive Museum Service Petri Net Process model.

element of the channel that serves requests originated by the downlink transition. In terms of object-oriented programming the downlink transition is considered as the method call that delegates a request for creating the initial login frame. Accordingly, the uplink transition is considered as the actual method that satisfies the aforementioned request. Hence, since the process implementation class includes reference to the GUI class, it can utilise the corresponding methods that allow creating the initial login frame.

The user is provided consequently with a login frame that allows to enter the relevant details and click the submit button to perform the authentication. This particular action generated by the user is defined in the process model via the action transition t_5, which fires directly and invokes respectively the method that authenticates the user. Both transitions t_6 and t_{20} define the synchronous communication channel that refers to the method call and the actual authentication method being invoked.

At this point the non-deterministic nature of the Petri Net model execution is exhibited since two different transitions are enabled and consequently anyone may ultimately fire. The non-deterministic execution of the service net demonstrates the applicability of the Petri Net formalism for modelling concurrency. Consequently, in the case that the user login details are not provided correctly the basic transition t_{28} fires and the execution path that allows the user to input again the relevant login information is followed. In terms of the service net this enables actually the downlink transition t_{29} that delegates a request that is handled effectively by the corresponding uplink transition t_{27}.

On the contrary, in the case the user authentication is successful the basic transition t_{30} fires and subsequently the service state p_9 is reached. The service state p_9 is considered

as the most important state of the pervasive service net since different execution paths can be undertaken. The non-deterministic execution of the service net is again apparent at this particular state since any of the two explicit user actions or the four different implicit context events may occur independently. Therefore, the complex and dynamic behaviour of the pervasive museum service can be successfully defined without any restrictions being imposed due to non-determinism and/or concurrency requirements.

Foremost, the user can generate the appropriate action that creates the corresponding preference frame. This allows the user to select preferred sites for which he wants to be presented with relevant historical information. In terms of the process model the action transition t_7 represents the user generated action (i.e. click the preference button), which enables the downlink transition t_8 and the uplink transition t_{21}. The downlink and the uplink transitions represent the method call and the corresponding method being invoked, which create the required preference frame.

Hence, once the user has completed the selection of the preferred sites the subsequent user generated action can be executed and information on the selected sites is presented to the user. This behaviour is defined in the net via the action transition t_9 and the synchronized downlink and uplink transitions; i.e. t_{10} and t_{22}. As a result the pervasive service returns to the original state p_9 from which the execution of the aforementioned functionality was initiated.

Apart from the selection of the preferred sites for which the user requires to be presented with historical information, the user is allowed to terminate the pervasive service at any given time. This is essentially depicted in the model via the action transition t_{31} that denotes the user generated action, which provides the capability to exit the software

service; i.e. click the exit button. In particular, state p_{20} is considered as the terminating state of the pervasive service execution, which means that no other state can be reached from that particular state.

The behaviour depicted in the model defines also different contextual situations that might occur when the pervasive service is found at state p_9. These contextual situations occur when a change in context information relevant to the pervasive service is detected. In most cases the change in context information is due to the occurrence of a particular context event. For instance, the context event generated when the user reaches the exit of the museum alters context information related to the location of the user.

Therefore, the change in the location information triggers an additional context event, which affects correspondingly the derived activity context of the user. Hence, the activity status of the user is effectively changed to *"Leaving"* and subsequently the appropriate museum exhibition tours are displayed to the user. Note that, these context events are emulated since it is assumed that the appropriate software infrastructure exists, which is responsible for detecting context events using sensors and acquiring and processing raw data to obtain a high level context description.

The Petri Net model defines the occurrence of the leaving situation via the downlink transition t_{11} and the uplink transition t_{23} that allows displaying to the user upcoming museum exhibitions tours. Subsequently, the user is permitted to dispose the frame displaying the exhibition tours and return back to the original state p_9. This user generated action is illustrated in the service net via the corresponding action transition t_{12}. Similarly to the leaving situation, the rest of the contextual situations are represented in the service net via the corresponding downlink, uplink and action transitions. The non-

deterministic behaviour defined for this part of the pervasive service permits the occurrence of any situation at any given time. This denotes that either of the transitions t_{11}, t_{13}, t_{15} and t_{17} may fire. Furthermore, as aforementioned the user might decide to terminate the pervasive museum service since this is permitted at the current state p_9 of the service execution.

The designed object-oriented Petri Net model facilitates primarily the definition of the dynamic behaviour of the pervasive service. Furthermore, the capability to validate the syntax and semantics of the model is provided using two distinct validation techniques. First, the imposed OCL constraints are enforced via the modelling editor of the PN-PMF, validating the static structure and syntax of the process model. Moreover, the capability to transform the validated model to the corresponding PNML format is provided so as to support the validation of the dynamic structure of the model.

Figure 6.7 presents the process model that is successfully imported into the Petri Net software tool. The figure illustrates actually a snapshot of the model execution, during which the runtime instance of the process model is found at state p_9. As aforementioned the transition to any of the subsequent states is possible, due to the non-deterministic behaviour exhibited at this particular state of the Petri Net model. Figure 6.8 illustrates two different snapshots captured during the model execution that demonstrate essentially the concurrent behaviour of the process model. The primary snapshot illustrates that the action transition t_7 fires and the execution path that allows the user to set the historic sites preferences is followed. Furthermore, the second snapshot illustrates a different simulation step during which the downlink transition t_{11} fires and the leaving contextual situation execution path is undertaken.

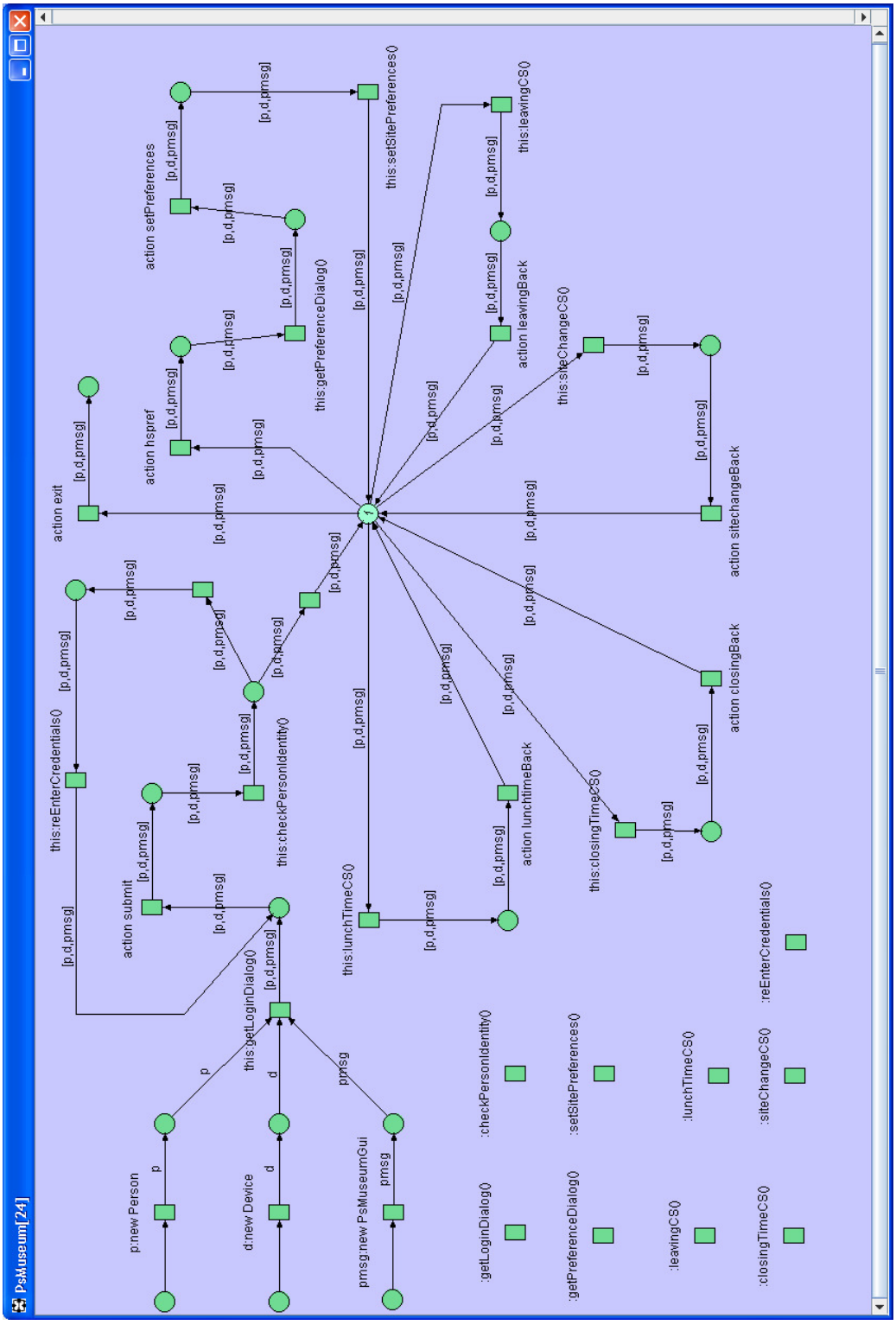


Figure 6.7: Pervasive Museum Service Petri Net Process model simulation.

The simulation of the pervasive service behaviour provides the capability to validate the operational semantics of the model. Consequently, the validated semantics of the Petri Net model can be transformed using the template-based mapping (i.e. code generators) to the operational semantics of the corresponding implementation. The development of the pervasive museum service is therefore simplified and expedited due to the automatic generation of tedious repetitive source code from the pervasive service models. In the next subsection an evaluation is performed using selected software metrics and a parametric software cost estimation model, in order to reveal the benefits of creating pervasive services using the proposed MDPNF.

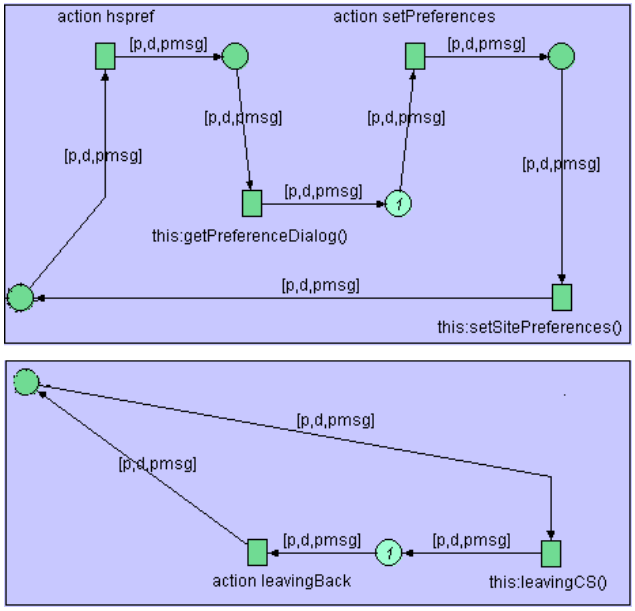


Figure 6.8: Snapshots of the process model simulation.

6.3 Quantitative Evaluation using a Pervasive Service Prototype

The evaluation of the MDPNF using the pervasive museum service aims to assess the applicability and efficiency of the process and the PSCE when creating pervasive services. First, a subjective evaluation is performed in terms of the applicability to define and enforce respectively the functional requirements of pervasive services via the

designed models and the generated implementation. The functional properties refer to the adaptability of the service, the seamless interaction of the user with the service and the dynamic behaviour of the service. Furthermore, a quantitative evaluation is applied to assess non-functional properties such as the service portability and the time, effort and cost required to develop this type of services using the proposed approach.

Primarily, the functional properties are evaluated as per the ability of the domain-specific modelling languages to define appropriately the essential characteristics of the pervasive service. The pervasive service models introduced in the previous subsections showcase the applicability of the proposed modelling languages and their supporing frameworks in modelling consistently the GUIs, the context-awareness characteristic and the dynamic behaviour of the pervasive museum service. Moreover, the static and dynamic validation capabilities provide the capability to guarantee that the necessary functional properties are unambiguously defined in the pervasive service models.

Besides the modelling and validation phases, the implementation phase is critical for the assesment of the approach with the help of the developed pervasive service prototype. The importance of the pervasive service implementation is twofold. First, it allows confirming that the functional properties defined in the models are enforced at runtime via the generated pervasive service implementation. Furthermore, it allows performing a quantitative evaluation to determine if the non-functional requirements are satisfied via the proposed approach. Note that, the formality of the qualitative evaluation is rather limited due to the absence of well-established methods for evaluating software quality when developing pervasive services [35]. Consequently, the evaluation technique applied in this work is predominantly quantitative rather than qualitative.

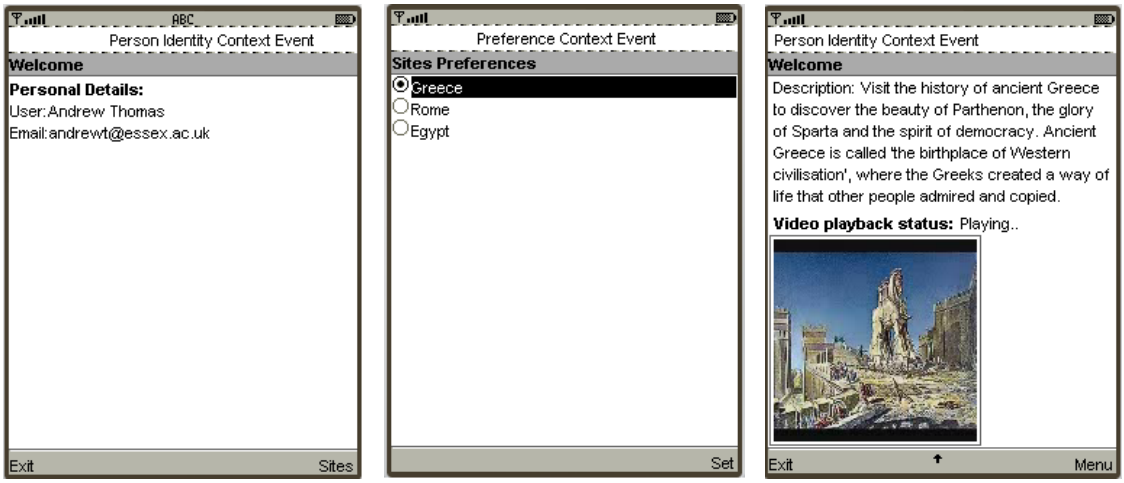


Figure 6.9: Pervasive museum service user-profiled context event.

Figure 6.9 illustrates the pervasive service running on a J2ME mobile device emulator, demonstrating in particular different states of the pervasive service execution. This allows determining if the adaptability and the behaviour of the service defined respectively in the context and process models are enforced via the generated service implementation. Furthermore, it allows establishing the flexibility the approach provides in developing advanced GUIs that aid the seamless interaction of the user with the service.

The three snapshots illustrated in Figure 6.9 showcase the occurrence of the profiled preference context event. In particular, from the initial state of the service execution the user clicks the *Sites* command button and is presented accordingly with the historic sites preference list. Consequently, the user explicitly requests for the information required by selecting the desired historic site from the preference list. In particular, the implementation of this context management task is generated from the *historicSites* profiled context source and its associated *Preference* information defined in the context model. This task describes specifically that the user inputs manually the necessary information (i.e. preference), so as to guide accordingly the service execution.

Moreover, the *"GreeceMobileHighCS"* situation is valid during the occurrence of the above explicit context event since the user has selected to be displayed with information on the historic site of Greece using his mobile device that is currently high on power. These conditions describe this particular contextual situation that denotes that the user should be displayed with a textual description and a low quality video that describes ancient Greece; as illustrated in the third snapshot. This is performed by executing the necessary context management task (i.e. Appendix L) generated from the definition of the *sites* static context source, its associated *Site* context information and the *"GreeceMobileHighCS"* contextual situation defined in the context model. Furthermore, the implementation of the pervasive service behaviour is generated via the transformation of the process model; i.e. starting from transition t_7 following the execution path back to the original state p_9.

```
public Form getIdentityForm() {
        identity = new Form("Welcome");
        identity.setTicker(new Ticker("Person Identity Context Event"));
        details = new StringItem("Personal Details:", null);
        hspref = new Command("Sites", Command.OK, 0);
        exit = new Command("Exit", Command.EXIT, 0);
        /** TODO starts
        */
        String forename = personIdentity.getForename();
        String surname = personIdentity.getSurname();
        String email = personIdentity.getEmail();
        details.setText("User:" + forename + " " + surname + "\nEmail:" + email);
        /** TODO ends
        */
        identity.addCommand(hspref);
        identity.addCommand(exit);
        return identity;
}
```

Figure 6.10: The J2ME implementation of an example graphical user interface.

Apart from the context and process models, the presentation model introduced in subsection 6.2.1 defines the GUIs illustrated in the captured snapshots. More specifically,

the generated source code comprises a considerable part of the implementation of the GUIs, which facilitate the interaction of the user with the service. For instance, the code presented in Figure 6.10 represents the primary snapshot illustrated in Figure 6.9, where a substantial part of the implementation is generated from the presentation model. Consequently, merely the code that invokes the required context management task for acquiring the relevant identity information should be manually implemented by the developer; i.e. TODO source code branch.

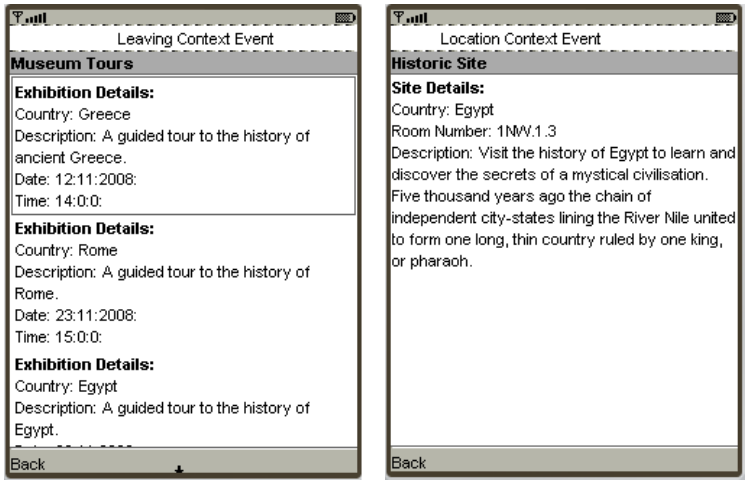


Figure 6.11: Pervasive museum service leaving and location contextual situations.

Figure 6.11 illustrates two additional snapshots of the service execution, which depict respectively the occurrence of the leaving contextual situation and of a location specific situation. In particular, the primary snapshot shows the occurrence of the *"LeavingCS"* contextual situation related to the activity of the user. This situation determines that when the user is located at the exit of the museum the sensed context event has an effect on the activity of the user. In fact the change of the user's location denotes that the derived activity context of the user is set to leaving. Correspondingly, the change in the activity of the user triggers the appropriate context management task that displays to the user a

list of museum exhibitions tours undertaken within the upcoming calendar month. The implementation classes that enforce this implicit functionality are generated from the *currentActivity, location and exhibitions* context sources, their associated information (*i.e. Location, Activity, Exhibitions*) and the *"LeavingCS"* contextual situation. Hence, these generated implementation classes aid as a result the realisation of the required pervasive service adaptation.

Moreover, the second snapshot of Figure 6.11 presents the occurrence of the *"EgyptMobileLowCS"* contextual situation, which denotes a change in the user's location (i.e. Egypt zone) while the mobile device running the service is currently low on power. For the example scenario, the presence of the user in the Egypt virtual zone is detected by proximity sensors. This generates as a result a context event that is detected at the application level via the *Person* context receiver implementation class (i.e. Appendix M). Consequently, since the user's mobile device is low on power the histrorical information is presented to the user merely in the form of a textual description. Correspondingly, the implementation that enforces this specific dynamic behaviour and the associated GUIs is also generated from the process and presentation models.

Figure 6.12 illustrates the same pervasive service running on a Java-enabled laptop device and showcases the occurrence of the *"RomeLaptopHighCS"* contextual situation. This situation determines that the presence of the user is detected at the historic site of ancient Rome using proximity sensors, while the battery power level of the laptop device is currently high. Consequently, the occurrence of this situation adapts the pervasive service by adding an additional selection (i.e. Rome) on the list shown on the left of the frame. This allows double-clicking and loading the appropriate information about the

historic site of Rome, which at this particular situation returns a textual description and a high quality video due to the conditions that are currently valid. On the contrary, if we assume that the power level of the laptop device is low then the same textual description is presented to the user but a low quality video is loaded instead.

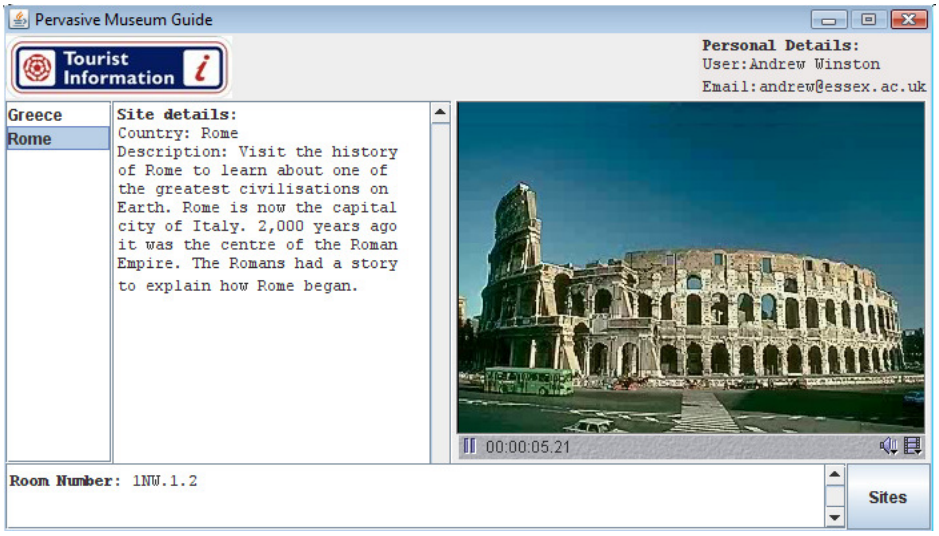


Figure 6.12: Pervasive museum service leaving and location contextual situations.

The transformation of the models generates the necessary application-level code, which delivers the implementation of context adaptation tasks, graphical user interfaces and the dynamic service behaviour. For the models transformation the developed template-based code generators are used to enable the translation of the models semantics to the operational semantics of different programming languages. The transformation results in the generation of repetitious code for both the J2ME and Java platform, on the basis of the defined generators. Hence, the service portability requirement is also satisfied since the percentage of the generated source code reduces significantly the prerequisite to re-write the same code when porting to another platform.

Apart from the transformation of models to code, the manual implementation of the complex functionality is required in order to deliver the complete implementation of the

service. This denotes that the pervasive service functionality showcased in Figure 6.9 and Figure 6.12 is the output of both code generation and manual implementation. Using such a hubrid approach simplifies and reduces the manual work required by the developer since it eradicates the requirement to physically implement repetitious code [35]. In specific, the automatic generation of a significant part of the implementation reduces the time, effort and cost introduced when creating pervasive services. In this thesis two complementary methods are used to perform the quantitative evaluation of the MDPNF, on the basis of the developed pervasive museum service prototype.

Initially the evaluation is restricted to the comparison of the generated code with the overall service code in a purely quantitative manner. The comparison is based on well-established software metrics that are namely: (i) Lines of Code (LoC) and (ii) McCabe's cyclomatic complexity [124]. For the acquisition of the necessary evaluation results the CCCC analysis tool [125] was used since it supports the analysis of source code files and allows generating extensive reports on different software metrics. The selection of the CCCC software tool was based on the competence and the open-source availability of the tool as attested also in [126].

The LoC metric was selected since it provides the capability to evaluate the efficiency of the approach in terms of reducing the time, effort and cost required to implement the pervasive service. In particular, the LoC metric is typically used to predict the effort required to develop a software application and estimate the overall software productivity. Moreover, the code complexity metric was chosen so as to illustrate that the architecture of the code generated by the defined templates (i.e. code generators) is fundamental for the reduction of code complexity. Cyclomatic complexity is a software metric that

determines the stability and the confidence level of a program by measuring the number of independent paths through a software application module. This metric inherently depicts that software applications with lower complexity are easier to comprehend, less risky to modify and can be subjected easily to conventional testing [124].

Table 6.1: Evaluation results of the pervasive service implementation phase.

Implementation	**Metric**	**Generated service code**		**Overall service code**	
		Overall	*Per Module*	*Overall*	*Per Module*
J2ME	Number of modules	58	-	61	-
	Lines of Code	2441	42.086	3121	51.164
	Cyclomatic Number	176	3.034	245	4.016
	Lines of Comment	804	13.862	817	13.393
Java	Number of modules	60	-	72	-
	Lines of Code	2019	33.650	2801	38.903
	Cyclomatic Number	125	2.083	201	2.792
	Lines of Comment	789	13.150	827	11.486

Table 6.1 presents the results acquired from the evaluation of the developed pervasive service prototype using the CCCC analysis tool. The table presents the results obtained from the evaluation of the different platform specific implementations of the pervasive service. Foremost, the LoC software metric presents the number of non-comment and non-blank lines of code. From the table it is apparent that 2441 LoC have been generated directly from the pervasive service models, where 680 LoC were manually implemented for the case of the J2ME implementation. Therefore, the percentage of the generated service code compared to the overall implemented service code yields to a value of 78.21%. Accordingly, for the case of the Java implementation 2019 LoC have been generated from the pervasive service models and merely 782 LoC were manually implemented. This denotes that 72.08% of the pervasive service implementation was automatically generated from the models.

Although the percentages calculated for the case study do not serve as an explicit baseline for future cases studies, they allow deriving the necessary conclusions. First, a clear separation requires to be made between the two different roles of the designer and the developer, so as to identify the individual benefits. In the case of the designer the abstract nature of the modelling languages aids and simplifies the complexity of the design process. This reduces the modelling time and effort since the designer is kept distinct from the implementation specific complexities introduced by different platforms. Moreover, in the case of the developer the calculated percentages that refer to the LoC metric indicate that the time and effort required from the developer to fully implement the pervasive service have been considerably reduced. Subsequently, the reduction in time and effort to design and implement the pervasive service denotes as a result a decrease of the associated service creation costs.

Code complexity is an additional valuable metric that denotes the degree of code understandability, code amenability to modification and it's also a dominant indicator of the code testability [124]. For instance, for the case of the J2ME platform the analysis results indicate that the complexity of the generated code is lower (i.e. 176) than the complexity of the overall code (i.e. 245). This denotes that it is easier to modify and test the generated code rather than the overall service code. An additional valuable conclusion is that by optimising the generators structure (i.e. code architecture), the complexity of the produced code can be effectively decreased. Consequently, the developer can be aided further since the generated code will be more comprehensible and subsequently its extension will be easier to accomplish. In contrast, in the case the pervasive service is

manually implemented from scratch (i.e. by different developers) it is very difficult to achieve optimisation and reduce as a result the complexity of the service code.

The LoC is indeed an intuitive metric that can express the amount of effort required to develop a service of a specific software size. Despite that fact, the LoC metric has received a lot of criticism mainly due to the lack of accountability and the lack of cohension with functionality. The lack of accountability denotes that the effort required to develop a pervasive service should not be based solely on the effort required to implement the service. Furthermore, the lack of cohesion with functionality denotes that experienced developers maybe able to implement the same service functionality writing less lines of code. This denotes that although the LoC metric is highly correlated to development time and effort, it is not well correlated with functionality and subsequently remains merely an estimator of software productivity.

Due to the aforementioned and other miscellaneous criticisms, this work introduces a complementary evaluation technique that considers additional parameters to determine software productivity. The evaluation is based on a non-proprietary, well documented and widely accepted software cost model, named the COnstructive COst MOdel II (COCOMO II) [127]. COCOMO II is a formal parametric model that allows estimating the effort, time and cost related to software development. The main advantage of the model over counterparts, such as the Software Lifecycle Management (SLIM) and the System Evaluation and Estimation of Resources – Software Estimation Model (SEER-SEM), is that it is an open model with several published data [128]. The model is based on various parameters that affect the estimation of the effort, time and cost required for completing a software service development activity. In particular the COCOMO II model

includes three tailorable submodels which are the: *Early Design Model, Application Composition model and Post-Architecture Model*.

The Post-Architecture model is the most detailed and mature model among the three and is defined to be used when the software lifecycle architecture is developed [127], [129]. In specific the parametric model allows estimating the development effort in Person-Months (PM) and the time to develop (TDEV) a software application. For the estimation of the effort and the time to develop, the model accepts a set of inputs such as the software size, five Scale Factors and seventeen Effort Multipliers. The following equations describe the effort and the time to develop taking into consideration the aforementioned inputs.

$$PM = A \times (Size)^{E} \times \prod_{i=1}^{17} EM_i \ , \ where \ E = B + (0.01 \times \sum_{j=1}^{5} SF_i) \ ,$$
$$A = 2.94 \ and \ B = 0.91 \ (COCOMOII.2000) \ \ (\boldsymbol{Eq}.1)$$

$$TDEV = C \times (PM)^{F} \ , \ where \ F = D + 0.2 \times (E - B) \ ,$$
$$C = 3.67 \ and \ D = 0.28 \ (COCOMOII.2000) \ \ (\boldsymbol{Eq}.2)$$

The primary equation (Eq.1) denotes the effort in Person Months derived by the software size defined in thousand of lines of code (KLoC), the exponent E that defines the sum of the Scale Factors (SF), the Cartesian product of the Effort Multipliers (EM) and the constant value A calibrated from several software projects [127]. In addition the second equation depicts the time required to develop a software service derived by the nominal effort (PM), the sum of the SF and the constant values calibrated from several software projects evaluated in COCOMO II. Appendix N presents the rating scales of the scale factors and the effort multipliers used in this work to derive the effort and the time required to develop the pervasive service prototype for the J2ME platform.

In particular, the evaluation applied in this work takes into consideration an extension [130] of the Use of Software Tools (TOOLS) effort multiplier defined in the Post-Architecture model. The TOOLS extension, calibrates and divides the multiplier into three complementary multipliers that are namely the completeness of tool coverage (TCOV), the degree of tool integration (TINT) and the tool maturity (TMAT) multiplier. Using this approach the critical role of Computer-Aided Software Engineering tools is heavily considered in the estimation of the effort and time required to develop a software application. In specific, the extended parametric model allows to evaluate the creation of the pervasive museum prototype service, by taking into serious consideration the CASE tools (i.e. PSCE) that are intensively utilised in the proposed process.

The TCOV effort multiplier provides the capability to define, on the basis of a rating scale, the coverage of activities undertaken in the software process by the supporting tools. Moreover, the TINT effort multiplier allows defining the degree of integration of the software tools used throughout the process and the effectiveness in achieving this integration. Finally, the TMAT effort multiplier provides the capability to specify the maturity of the adopted toolset in accordance to the time this particular tool set is used in the market and the technical support provided. Therefore, the extended model provides a more comprehensive estimate of the TOOL effort multiplier via the normalisation of the proposed parameters using the following equation [130]:

$$TOOL = 0.51 \times TCOV + 0.27 \times TINT + 0.22 \times TMAT \quad (Eq.3)$$

Consequently, the extended Post-Architecture model is utilised to derive the effort and the time required to develop the pervasive museum service prototype for the J2ME platform. This is performed sequentially taking into consideration the utilisation or not of

the PSCE when developing the pervasive service prototype. The software size considered in the calculations is the overall size of the J2ME service code ($Size = 3.121 KLOC$) illustrated in Table 6.1. Initially, the TOOL multiplier is derived using the normalisation equation (Eq. 3) of the extended model, for the two individual cases.

$$TOOL_{PSCE} = 0.51 \times 0.78 + 0.27 \times 0.78 + 0.22 \times 0.9 \Rightarrow TOOL_{PSCE} = 0.8064$$

$$TOOL_{NOPSCE} = 0.51 \times 1.17 + 0.27 \times 1.17 + 0.22 \times 1 \Rightarrow TOOL_{NOPSCE} = 1.1327$$

The ratings used for the TCOV, TINT and TMAT effort multipliers are derived from Tables 1 & 3 of Appendix N. These ratings denote respectively the coverage of the model-driven Petri Net based process provided by the adopted tool set (i.e. PSCE), the degree of the software tools integration and the maturity of the tool set. Using the same reasoning the corresponding ratings are derived for the case that the PSCE is not used and the service is developed from scratch using low-level code editing software tools. The TOOL rating derived for the two respective cases is used in the corresponding calculation of the nominal effort required for developing the J2ME pervasive service prototype.

Hence, using Table 1 (i.e. Appendix N) the ratings for the scale factors are derived for the two respective cases to calculate the exponent E [130]. The exponent E captures in fact the relative economies or diseconomies of scale encountered for software projects of different sizes [130]. Furthermore, the ratings of the rest of the effort multipliers are derived from Tables 1 & 2 (i.e. Appendix N) to define additional parameters that affect the development of the pervasive service prototype [131]. Detailed information about Scale Factors and Effort Multipliers is available in [127], [131]. Using the corresponsing scale factors and effort multipliers derived for the individual cases the nominal effort and the time to develop are calculated as shown onto the next page.

(1) – PSCE Support

$$E = 0.91 + [0.01 \times (1.24 + 3.04 + 5.65 + 1.1 + 4.68)] \Rightarrow E = 0.91 + [0.01 \times 15.71] \Rightarrow$$
$$E = 0.91 + 0.1571 \Rightarrow E = 1.0671$$

$$\mathrm{PM}_{\mathrm{PSCE}} = 2.94 \times (3.121)^{1.0671} \times [1.10 \times 1.14 \times (1 \times 1.17 \times 1 \times 1.17 \times 1) \times 1 \times 0.95 \times 1 \times 1 \times 0.87$$
$$\times 0.85 \times 0.88 \times 1 \times 0.88 \times 0.91 \times 0.91 \times 0.8064 \times 1 \times 1] \Rightarrow \mathrm{PM}_{\mathrm{PSCE}} = 2.94 \times 3.369 \times 0.624 \Rightarrow$$
$$\mathrm{PM}_{\mathrm{PSCE}} = 6.18 \text{ Person-Months}$$

$$F = 0.28 + 0.2 \times (1.0671 - 0.91) \Rightarrow F = 0.28 + 0.2 \times 0.1571 \Rightarrow F = 0.311$$

$$\mathrm{TDEV} = 3.67 \times (6.18)^{0.311} \Rightarrow \mathrm{TDEV} = 6.47 \text{ Months}$$

(2) – Without PSCE Support

$$E = 0.91 + [0.01 \times (1.24 + 3.04 + 5.65 + 1.1 + 4.68)] \Rightarrow E = 0.91 + [0.01 \times 15.71] \Rightarrow$$
$$E = 0.91 + 0.1571 \Rightarrow E = 1.0671$$

$$\mathrm{PM}_{\mathrm{NOPSCE}} = 2.94 \times (3.121)^{1.0671} \times [1.10 \times 1.14 \times (1 \times 1.17 \times 1 \times 1.17 \times 1) \times 1 \times 0.95 \times 1 \times 1 \times 0.87$$
$$\times 0.85 \times 0.88 \times 1 \times 0.88 \times 0.91 \times 0.91 \times 1.1327 \times 1 \times 1] \Rightarrow \mathrm{PM}_{\mathrm{NOPSCE}} = 2.94 \times 3.369 \times 0.876 \Rightarrow$$
$$\mathrm{PM}_{\mathrm{NOPSCE}} = 8.68 \text{ Person-Months}$$

$$F = 0.28 + 0.2 \times (1.0671 - 0.91) \Rightarrow F = 0.28 + 0.2 \times 0.1571 \Rightarrow F = 0.311$$

$$\mathrm{TDEV} = 3.67 \times (8.68)^{0.311} \Rightarrow \mathrm{TDEV} = 7.19 \text{ Months}$$

The above calculations indicate clearly that both the effort and time for developing the same prototype service for the J2ME platform are reduced significantly in the case the PSCE is used throughout the design and development process. Therefore, the decrease in the development effort and time denotes also a decrease in the development costs. This can be verified if the assumption is made that in both cases the average monthly work rate is \$10k. Hence, the cost in the case of advanced software tools support is \$61.80k (i.e. Effort*\$10k), while the use of low-level code editing tools increases the cost to \$86.80k. The results clearly state that the use of the PSCE benefits the process by reducing the effort, time and costs necessary for the creation of pervasive services.

Although the COCOMO II model is a widely used model calibrated through data obtained from various software projects, it still has a degree of uncertainty and risk due to the parametric inputs of the model. In order to cope with the risk and uncertainty, the evaluation employs also the Monte Carlo Simulation method via the use of a Software Cost Analysis Tool [131], [132] developed by the National Aeronautics and Space Administration (NASA). The Monte Carlo Simulation method provides the capability to cope with the uncertainty and lack of knowledge involved when modelling phenomena such as the calculation of the effort, time and cost for software service development. In particular, Monte Carlo Simulation methods are described as sampling methods that use sets of random numbers as inputs and generate data that can be represented as probability distributions graphs or histograms.

The application of the Monte Carlo Simulation involves primarily the definition of an input range for each scale factor and effort multiplier defined in the model. Note that, the ranges are defined from the actual inputs (i.e. rating scales) derived from the corresponding SFs and EMs tables; i.e. Appendix N. Subsequently, the simulation tool generates random inputs within the input range specified for each parameter and performs a deterministic computation (i.e. using a mathematical formula) with the aid of the random inputs. Finally, the generated output data are aggregated into Cummulative Distribution Functions (CDFs) that represent respectively the effort and the cost to develop the pervasive museum service prototype.

Figure 6.13 illustrates the CDF graphs generated from the simulation, which present the total effort with and without advanced tools support. From the distributions it is apparent that for the entire set of probabilities the effort required to develop the prototype service

without the support of the PSCE is always higher. The mean value (i.e. expected value) derived using the Monte Carlo sampling method is also higher for the case that advanced tools support is not provided. In specific, when using the PSCE the nominal effort is equal to 6.115 Person-Months and when the PSCE is not used the nominal effort is equal to 8.73 Person-Months. Note also that the mean nominal effort is computed with a likehood of occurrence of 58%, which is almost identical for the two recorded cases.

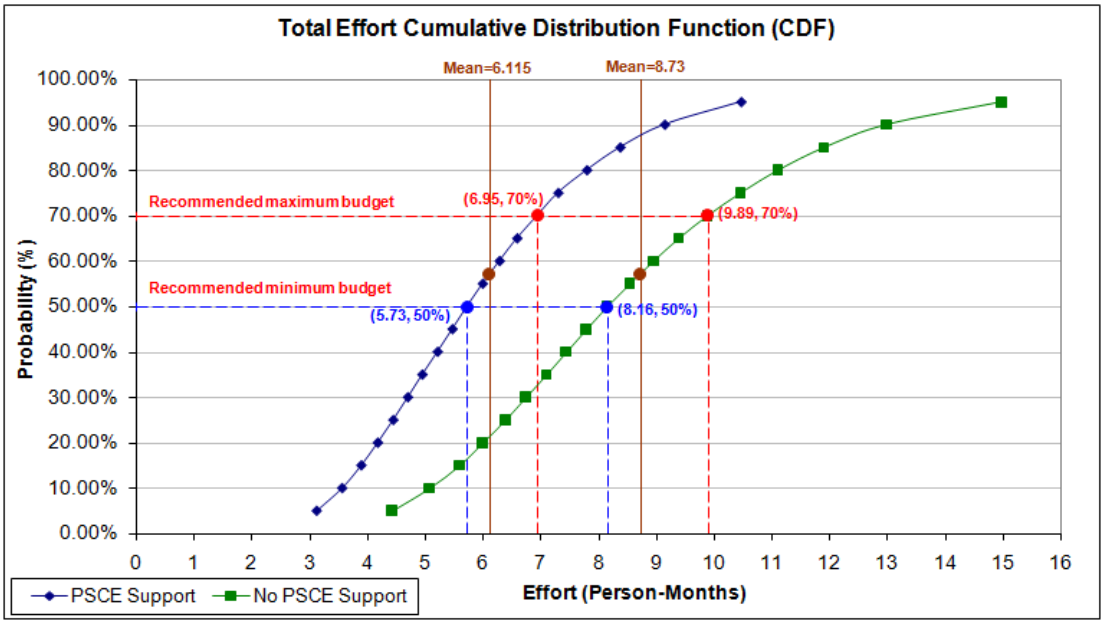


Figure 6.13: Total Effort Cumulative Distribution Function.

Figure 6.14 illustrates the corresponding CDF graphs that represent the matching costs for the aforementioned cases. The results indicate that the most cost-effective solution is provided when using the PSCE throughout the service development process. As in the case of the effort CDF graphs, the costs for the entire set of propabilities is always higher in the case that the PSCE is not used. Moreover, the expected cost (i.e. mean) derived is also much higher and the derivation is based actually on the previous assumption of an average monthly work rate of $10k. The CDF curves provide also the capability to validate the budget being imposed for the development of a software service. In the case

the budget estimate is between the recommended minimum and maximum ranges (i.e. 50%-70%), then it is considered feasible to complete the design and development of the pervasive service prototype with the specified budget [132].

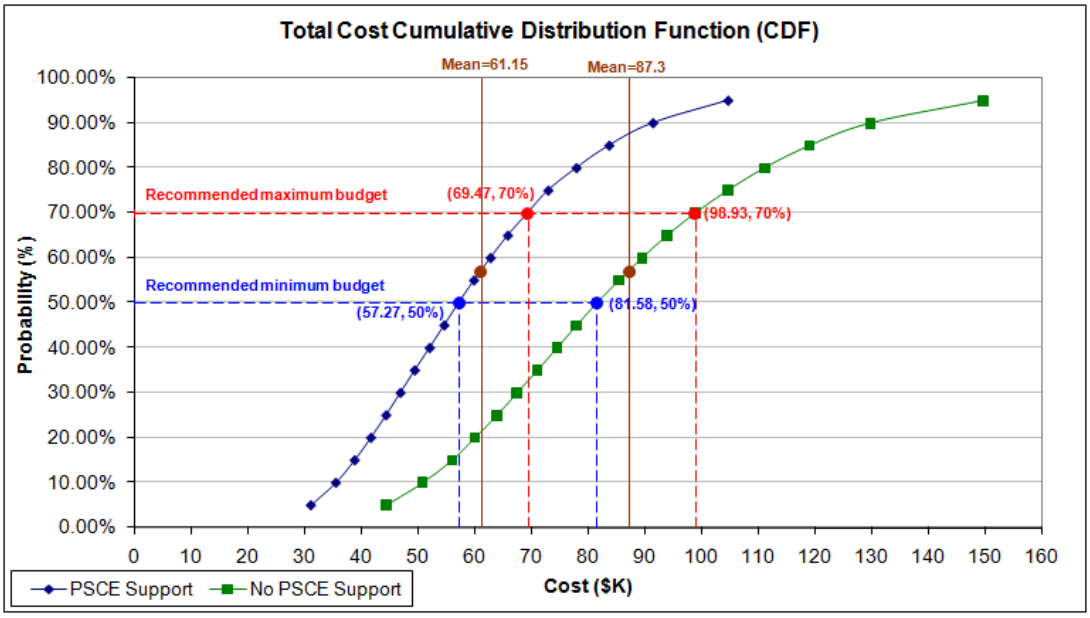


Figure 6.14: Total Cost Cumulative Distribution Function.

The generated CDFs verified once again the benefits of using the abstract PSCE and its associated development process for enabling the rapid and cost-effective design and development of pervasive services. Apart from the benefits, there are some shortcomings that have an effect on pervasive service creation. An initial overhead is introduced since the designers must be educated and become fully acquainted with the different modelling languages. In this approach though, the utilisation of a common meta-metalanguage with well established semantics decreases the learning-curve required by the designers to become fully competent with the modelling languages.

In addition, the service developers have to define the template-based code generators that enable the model-to-code transformation. This imposes a once-off implementation load and cost overhead when defining the code generators. The long-term benefits obtained

though from the code generation process diminish significantly the stated shortcoming. Furthermore, in the case of future extensions and/or modifications of the modelling languages only minor refinements of the code generators are required, which are simplified by the use of the oAW component of the integrated PSCE.

6.4 Summary

The pervasive museum service presented in this chapter serves as an example prototype, which facilitates the evaluation of the proposed Petri Net based model-driven process and it's supporting PSCE. Via the use of the MDPNF the pervasive service specification is captured in the form of presentation, context and Petri Net process models. In addition, the modelling frameworks of the PSCE facilitate aside from modelling, the validation of the static structure and syntax of the models by enforcing the defined metamodel level constraints. Furthermore, the Petri-Net process model supports both the integration of the discrete modelling domains and provide a formal method that facilitates the validation of the operational semantics of the pervasive service. Hence, using the two validation levels design errors are detected and resolved before the model-to-code generation phase.

Following, a quantitative evaluation of the MDPNF is performed using well-established software metrics and an extended version of the COCOMO II software cost estimation model. The acquired analysis results indicate essentially a fundamental decrease in effort, time and cost when creating pervasive services at the static compile time using the MDPNF. In particular, the evaluation performed confirms that the use of a model-driven Pervasive Service Creation Environment critically benefits the pervasive service creation process. In the following chapter this thesis concludes with an extensive discussion on the benefits, predicaments and future directions of this thesis research work.

Chapter 7 Conclusions

This thesis addresses the creation of advanced services using high-level service creation environments with particular emphasis on pervasive service creation. First, this research work derives a set of Model-Driven Development (MDD) requirements essential for defining a conceptual framework, which comprises a model-driven methodology and a generic MDD environment. Hence, the conceptual framework allows definition of the modelling languages and automatic generation of the modelling frameworks that support the definition of the pervasive service. In particular, the generated modelling frameworks are integrated with the generic environment to compose a suitable Pervasive Service Creation Environment (PSCE). The developed PSCE and the proposed model-driven Petri Net based process compose as a result the Model-Driven Petri Net based Framework (MDPNF). The MDPNF supports the creation of pervasive services at compile time and reduces the time, effort and costs incurred during the service creation process.

This chapter presents a summary of the technical contributions and the conclusions drawn from this work. Potential future directions of this work are then considered and discussed.

7.1 Summary

The conceptual framework proposed and defined in this thesis is the key to unlocking the potential of MDD for domain-specific (i.e. pervasive) service creation. The conceptual framework is formulated on the basis of the derived MDD requirements that allow a comprehensive methodology to be defined and a generic environment to be designed. The selected model-driven software tools composing the generic environment are vital in simplifying the MDD methodology and thereby decreasing the service creation overheads. In essence, the framework provides the capability to support and automate to a great extent the software tasks involved in the analysis, modelling, validation and implementation phases of the service creation process. The most significant tasks are the generation of the modelling frameworks that compose the domain-specific SCE (i.e. PSCE) and the automation of the service implementation phase.

The use of the widely-accepted Meta-Object Facility (MOF) modelling specification, which is the core of the conceptual framework, provides a common understanding of the defined modelling languages and eases the comprehension of the models composing the service. In addition, the definition of domain rules using a common constraint language (i.e. Object Constraint Language) ensures the design of unambiguous service models. Finally, the use of an established transformation language (i.e. Query/View/Transform and MOF Model to Text compliant) facilitates mapping modelling language semantics to operational semantics of a programming language and increases understanding of the models and the resulting service implementation. This research has revealed that the use of proprietary languages and software tools for defining modelling languages, imposing

constraints and defining transformations further complicates the analysis, design, validation and implementation phases of the service creation process.

Therefore, the definition of the methodology and the design of the generic environment solely conform to the Model Driven Architecture (MDA) specifications, so as to support the service creation process. The widely-used Eclipse modelling implementations of the MDA standards are the apposite candidates for the design and implementation of the generic model-driven environment that supports the methodology. Moreover, the integration of the environment's software components onto a well-known platform such as the Eclipse provides the added benefit of accelerated adoption by the designers and developers. In particular, the architecture of the Eclipse platform forms the basis for the design of the generic environment and the integration of domain-specific SCEs (e.g. PSCE).

The only limitation is that designers need to become fully acquainted with the modelling implementations of the MDA specifications (i.e. EMF, GMF) included in the generic environment, to effectively carry out the analysis, design and validation phases of the service creation process. Similarly, developers must familiarise themselves with the implementations (i.e. ATL, oAW) of the generic environment, which provide the software capabilities that allow undertaking the implementation phase; i.e. via model transformations and code generation. Despite that fact, the conformance of the Eclipse implementations to the MDA specifications makes the transition a relatively easy task to accomplish and reduces the initial learning curve. Furthermore, both designers and developers only need to be educated once on how to use these implementations: they can

then apply their knowledge in diverse domains for effortlessly and cost-effectively creating domain-specific services; e.g. pervasive services, web services.

In this thesis a Model-Driven Petri Net based Framework is also defined that introduces pervasiveness into the proposed and defined model-driven conceptual framework so as to support and simplify the development of pervasive services. The combination of the Petri net formalism with the MDA paradigm in the MDPNF allows defining the required modelling languages, integrating these discrete pervasive service modelling domains and provides additional dynamic validation capabilities. The MDA paradigm facilitates the definition of the pervasive modelling languages, which refer to context, presentation and process modelling. Furthermore, the Petri Net formalism provides dynamic validation capabilities that complement the static OCL validation capabilities of the conceptual framework and allow integrating the discrete pervasive service modelling domains.

Unlike existing pervasive service creation approaches that consider only the context-awareness characteristic, additional pervasive service requirements such as the interaction of the user with the service and the dynamic behaviour of the pervasive service are also considered in this thesis. Therefore, using the conceptual framework the domain-specific modelling languages are defined and from the definition their supporting modelling frameworks are automatically generated. As a result, this reduces the time, effort and costs required to develop these modelling frameworks, which facilitate the definition and validation of the pervasive service characteristics in the form of context, presentation and process models. Additionally, the generated modelling frameworks are integrated into the generic environment to compose the overall PSCE that supports and simplifies the pervasive service creation process.

Firstly, from the abstract and concrete syntax of the Context Modelling Language the corresponding Context Modelling Framework is automatically generated, which supports the design and validation of context models. The CML forms the backbone of the CMF and allows associating an entity (e.g. Person) with context information via the definition of diverse input context sources. These sources include the required properties that indicate how context information is obtained, managed and distributed to achieve the adaptation of the pervasive service. The context information associated with these input sources is defined in the model either as atomic or composite context. Atomic context is comprised mainly of primitive context properties, while composite context is defined as atomic context and primitive properties.

As well as entities, context sources and context information, the context model includes also contextual situations and temporal constraints. Contextual situations are required to model conditions that adapt the execution of the dynamic behaviour of the pervasive service and affect the interaction of the user with the service. Furthermore, the context sources defined in the model are associated to temporal constraints, so as to determine the validity of context information obtained from the sources; i.e. whether information is outdated. Hence, from the model definition an implementation can be generated that enforces at runtime the context-aware mechanisms described in the model. The generated implementation performs context management tasks such as querying, managing and dissiminating information to pervasive services to facilitate their adaptation.

As in the case of the CMF, the Presentation Modelling Framework is automatically generated from the abstract and concrete syntax definition of the defined Presentation Modelling Language. The PMF supports the definition and validation of presentation

models that represent abstract notions of Graphical User Interfaces (GUIs). In particular, the PMF supports the specification of GUI concepts such as displays, containers, components and their corresponding associations and graphical properties. Hence, from the model definition an implementation is automatically generated providing GUIs through which the user can interact seamlessly with the service. Furthermore, the efficiency with which the PMF is developed allows easy extension of the modelling language and regeneration of the framework, incorporating additional GUI concepts.

Finally, the Petri Net Process Modelling Framework is automatically generated from the abstract and concrete syntax definition of the Petri Net Process Modelling Language. The PN-PMF facilitates the definition and validation of the pervasive service behaviour in the form of an object-oriented Petri Net process model. In particular, the process model describes the states, object instances, user-generated actions and synchronised methods that compose the dynamic behaviour of the pervasive service. Consequently, from the process model definition a corresponding implementation can be generated that enforces the dynamic behaviour of the pervasive service during runtime.

The aforementioned analysis phase provides the capability to identify the concepts and constraints of the pervasive domain, so as to define the necessary modelling languages and generate their supporting modelling frameworks that compose the overall PSCE. *Subsequently, a model-driven Petri Net based process is defined that formulates along with the PSCE the aforesaid Model-Driven Petri Net based Framework.* The MDPNF supports as a result the execution of the remaining phases of pervasive service creation. Following the definition and static validation of the pervasive service models using the modelling frameworks, the PSCE allows validating the dynamic structure of the

pervasive service. This is performed by transforming the Petri Net model that integrates the modelling domains into an equivalent PNML representation, which allows the dynamic behaviour of the pervasive service to be imported and validated. Hence, the completeness and coherency of the pervasive service specification are guaranteed since both the static syntax and the operational semantics of the models have been validated. Finally, the software capabilities of the PSCE (i.e. the oAW component) facilitate the transformation of the pervasive service models to the required implementation, enforcing during service execution the pervasive service requirements that were captured at the modelling level (i.e. static compile time).

The model-driven nature of the pervasive service creation process provides also the value-added benefit (i.e. capability) of automatically generating different pervasive service implementations from the same platform-independent models. Therefore, apart from the enforcement of the pervasive service functional requirements, the non-functional requirement of service portability is satisfied since most of the service code is automatically generated for different platforms. The only prerequisite is the definition of the different template-based code generators, which support the transformation of the pervasive service models to the corresponding implementation technologies.

The example case study presented in Chapter 6 demonstrates the application of the MDPNF for the creation of a pervasive service prototype. Firstly, the respective context, presentation and process models that compose the pervasive service specification are defined. Then the static structure of the pervasive service models is validated by enforcing the OCL constraints imposed on each language during the analysis phase. The Petri Net Markup Language (PNML) representation is then generated from the Petri Net

model, which allows validating the dynamic behaviour of the service; this ensures the coherency of models and enables generation of the analogous pervasive museum service implementation. Finally, any subsequent changes performed in the designs can be reflected automatically in the generated code thereby minimising discrepancy between service design and implementation.

On the basis of the case study the evaluation of the MDPNF is performed using selected software metrics and a software cost estimation model. The Lines of Code (LoC) metric indicates that the workload of the developer is reduced significantly since a high-percentage of the service implementation is generated from the defined pervasive service models. Furthermore, the cyclomatic complexity metric shows that the complexity of the generated implementation can be further reduced by optimising the code generators. Consequently, the developer is better able to comprehend and extend or modify the generated pervasive service code, so as to deliver the complete service implementation. By contrast, code optimisation and the reduction of code complexity are difficult to achieve when manual implementation by different developers is involved.

The use of the parametric model provides also a formal evaluation method that determines the time, effort and the cost required to implement the pervasive museum service with and without the aid of the MDPNF. In particular, the calculated analysis results indicate that the use of the PSCE improves significantly the efficiency of the pervasive service creation process. Consequently, the effort, time and cost required to manually develop the pervasive service from scratch without the use of the PSCE are considerably higher.

The complementary Monte Carlo simulation applied in this thesis deals with the uncertainty and the risk imposed by the use of the parametric model when calculating the analysis results. The simulation generates the effort and cost Cummulative Distribution Functions (CDFs) that display the likehood of completing the development of the pervasive service prototype with a specific effort and budget; with and without the use of the PSCE. The effort distribution graph indicates that for all computed probabilities the required effort is always less when developing the pervasive service using the proposed process and the supporting PSCE. This also applies for the cost estimation distribution graphs since for the entire set of computed propabilities the development cost is descreased when the pervasive service creation process is supported by the PSCE.

The only shortcomings of the approach arise from the requirements to familiarise with the modelling languages, develop the code generators and understand the generated implementation so as to be able to extend or modify it. Therefore, in order to realise the concepts of the pervasive domain, developers should closely co-operate with designers that are highly competent with the specified modelling languages. This interaction will enable the developers to learn how the modelling languages are used so as to define the most appropriate code architecture (i.e. structure) within the code generators. Hence, the optimisation of the code generators will help to produce well-structured code that can be realised and extened (or modified) easily by the developers to deliver the complete pervasive service functionality. These shortcomings introduce an initial overhead to the process but are more than offset by the long-term efficiency of the pervasive service creation process.

7.2 Future work

This thesis proposes a conceptual framework that facilitates the development of a PSCE and the definition of a model-driven Petri Net based process, which compose the Model-Driven Petri Net based Framework for pervasive service creation. As part of future work the following research directions can be undertaken to extend the current work.

Future work should devote a reasonable effort in expanding the modelling languages defined that describe currently the core features of pervasive services. Therefore, by examining supplementary pervasive service scenarios the capability to identify additional concepts that could be expressed in the languages is provided. Moreover, with the help of these scenarios, the existing concepts captured in the modelling languages can be modified if required, thus optimising the languages. This optimisation would necessitate extending the code generators in order to transform the new or updated concepts into a pervasive service implementation that delivers wider functionality. For instance, incorporating modelling concepts in the CML that enable recording and ranking user preferences is an immediate extension, which would allow generating an implementation that enforces improved adaptation mechanisms at runtime. Finally, shifting the focus to particular categories of pervasive services such as e-health, Intelligent home and e-learning allows improving, tailoring and extending the pervasive modelling languages to accommodate these domains and satisfy their explicit requirements.

In order to deliver a fully integrated PSCE that supports pervasive service creation, future research work should also consider the development of an OCL generator and a Petri Net software validation capability. The OCL generator could be implemented in the form of supplementary templates or as an Eclipse plug-in (e.g. OCL4Java). This would facilitate

the transformation of the textual OCL constraints (i.e. contextual situations) expressed in the context model to implementation code. Consequently, the generated code would improve the pervasive service adaptability. Additionally, the Petri Net-based validation capability should be better embedded into the PSCE, avoiding transformation to the intermediary PNML format and use of an external Petri Net software tool to perform the process model simulation. The evaluation revealed that having a unified PSCE is beneficial and preferential in terms of development effort, time and cost.

Model-to-model transformations could be also defined to explore the capabilities of the MDPNF in performing pervasive service configuration management tasks. This requires the definition of transformations that allow translating outdated service models to new models that conform to an extended version of the modelling language(s). With this approach the capability to configure (i.e. update) the models is provided, which allows regeneration of the service implementation without having to redesign the models from scratch. Finally, model transformations could be investigated to enable the translation of domain models (i.e. context models) to diverse domain models (i.e. relational models), which would ease the generation of miscellaneous pervasive service implementations; e.g. relational database implementation.

List of Publications

Journal papers (Published/Accepted):

1. Achilleos, K. Yang and N. Georgalas, "Context modelling and a context-aware framework for pervasive service creation: a Model-driven approach", *Special Issue of Pervasive Mobile Computing (PMC) Journal*, Elsevier 2009.
2. N. Georgalas, A.Achilleos, V. Freskos and D.Economou, "Agile Product Lifecycle Management for Service Delivery Frameworks: History, Architecture and Tools", *BT Technology Journal*, Vol. 27, No.1, Springer, 2009.

Book Chapters (Published/Accepted):

1. A. Achilleos, K. Yang and N. Georgalas, *Model-driven engineering of non-functional properties for Pervasive service creation*, IGI Global Book Series on Methodologies for Non-Functional Requirements in Service Oriented Architecture, to be published 2010.

Conference Papers (Published/Accepted):

1. A. Achilleos, K. Yang and N. Georgalas, "A Model-driven Approach to Generate Service Creation Environments", in: Proceedings *of the IEEE Global Communications Conference (Globecom)*, New Orleans, USA, Nov. 2008, pp. 1-6.
2. A. Achilleos, K. Yang, N. Georgalas and M. Azmoodeh, "Pervasive Service Creation using a Model Driven Petri Net based Approach", in: *Proceedings of the IEEE International Wireless Communications and Mobile Computing Conference*, Crete, Greece, Aug. 2008, pp. 309-314.

3. A. Achilleos, N. Georgalas, K. Yang, "An Open Source Domain-Specific Tools Framework to Support Model Driven Development of OSS", in: *Proceedings of European Conference on Model Driven Architecture – Foundations and Applications, Lecture Notes in Computer Science (LNCS),* Vol. 4530, pp. 1-16, Springer, 2007.

Journal papers (Under review):

1. A. Achilleos, K. Yang and N. Georgalas, "Model-Driven Petri Net based Framework for Pervasive Service Creation", Special Issue on Software Services and Service-Based Systems, IEEE Transactions on Software Engineering, 2010.

References

[1] Y. V. Reddy, "Pervasive Computing: Implications, Opportunities and Challenges for the Society", in: *Proceedings of the IEEE International Symposium on Pervasive Computing and Applications*, Phuket, Thailand, Aug. 2006, pp. 5-5.

[2] D. Saha and A. Mukherjee, "Pervasive Computing: A Paradigm for the 21st Century", *IEEE Computer Society Press*, Vol. 36, No. 3, pp. 25-31, Mar. 2003.

[3] V. Dhingra and A. Arora, "Pervasive Computing: Paradigm for New Era Computing", in: *Proceedings of the IEEE International Conference on Emerging Trends in Engineering and Technology*, Nagpur, India, July 2008, pp. 349-354.

[4] M. Weiser, "The Computer for the 21st Century", *ACM SIGMOBILE Mobile Computing and Communications Review*, Special Issue Dedicated to Mark Weiser, Vol. 3, Issue 3, pp. 3-11, July 1999.

[5] M. Satyanarayanan, "Pervasive Computing: Vision and Challenges", *IEEE Personal Communications*, Vol. 8, Issue 4, pp. 10-17, Aug. 2001.

[6] K. Yang, S. Ou, M. Azmoodeh and N. Georgalas, "Policy-based Model-driven Engineering of Pervasive Services and the Associated OSS", *British Telecom Technical Journal (BTTJ)*, Vol. 23, No. 3, pp. 162-174, July 2005.

[7] L. Kolos-Mazuryk, G-J Poulisse and P. van Eck, "Requirements Engineering for Pervasive Services", in: *Proceedings of the Workshop on Building Software for Pervasive Computing*, Position Papers, Oct. 2005, pp. 18-22, No publisher.

[8] W. Sitou and B. Spanfelner, "Towards Requirements Engineering for Context Adaptive Systems", in: *Proceedings of the IEEE Annual International Computer Software and Applications Conference,* Vol. 02, Beijing, China, July 2007, pp. 593-600.

[9] R. H. Glitho, F. Khendek and A. De Marco, "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*, Vol. 33, No. 4, pp. 446-457, Nov. 2003.

[10] D. X. Adamopoulos, G. Pavlou and C. A. Papandreou, "Advanced Service Creation Using Distributed Object Technology", *IEEE Communications Magazine*, Vol. 40, No. 3, pp. 146-154, Mar. 2002.

[11] Object Management Group (OMG), "Model Driven Architecture (MDA) Guide Version 1.0.1", [Online] Available: http://www.omg.org/docs/omg/03-06-01.pdf [Accessed September 21, 2009].

[12] A. Kleppe, J. Warmer and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*, Boston, USA, Addison-Wesley Professional, 2005.

[13] D. S. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*, New York, USA, Wiley Publishing Incorporation, 2003.

[14] Y. Singh and M. Sood, "Model-Driven Architecture: A Perspective", in: *Proceedings of the IEEE International Advanced Computing Conference*, Patiala, India, Mar. 2009, pp. 1644-1652.

[15] Object Management Group (OMG), "Object Constraint Language (OCL) Specification Version 2.0", [Online] Available: http://www.omg.org/docs/formal/06-05-01.pdf, 2005, [Accessed March 06, 2009].

[16] A. Kleepe, J. Warmer, and S. Cook, "Informal Formality? The Object Constraint Language and Its Application in the UML Metamodel", in: *The Unified Modeling Language «UML»'98: Beyond the Notation – Selected Papers, Lecture Notes In Computer Science (LNCS)*, Vol. 1618, 1998, pp. 148-161.

[17] C. A. Petri, "Kommunikation mit Automaten", Ph.D. Thesis, Bonn: Institut für Instrumentelle Mathematik, University of Bonn, In German, 1962.

[18] M. P. Gervals and A. Diagne, "Enhancing Telecommunications Service Engineering with Mobile Agent Technology and Formal Methods", *IEEE Communications Magazine*, Vol. 36, Issue 7, pp. 38-43, July 1998.

[19] S. Mulik, S. Ajgaonkar and K. Sharma, "Where Do You Want to Go in Your SOA Adoption Journey?", *IT Professional*, Vol. 10, Issue 3, pp. 36-39, May-June 2008.

[20] J. Lennox, J. Rosenberg and H. Schulzrinne, "Common Gateway Interface for SIP, Internet Engineering Task Force", RFC3050, Jan. 2001.

[21] K. Singh and H. Schulzrinne, "Peer-to-Peer Internet Telephony using SIP", in: *International Workshop on Network and Operating System Support for Digital Audio and Video*, Washington, USA, June 2005, pp. 63-68.

[22] J. Lennox and H. Schulzrinne, "Call Processing Language (CPL): A Language for User Control of Internet Telephony Services", Internet Engineering Task Force, RFC3880, Oct. 2004.

[23] J. P. Tolvanen and Steven Kelly, "Defining Domain-Specific Modeling Languages to Automate Product Derivation: Collected Experiences", in: *Software Product Lines, Lecture Notes on Computer Science (LNCS),* Springer Berlin/Heidelberg, Oct. 2005, pp.198-209.

[24] J. L. Bakker and R. Jain, "Next Generation Service Creation using XML Scripting Language", in: *IEEE International Conference on Communications*, Vol. 4, Alaska, USA, Aug. 2002, pp. 2001-2007.

[25] B. Steffen, T. Margaria, A. Claßen, V. Braun and M. Reitenspieß, "A Constraint-Oriented Service Creation Environment", in: *International Conference on Practical Application of Constraint Technology*, New York, USA, Apr. 1996, pp. 283-298.

[26] T. Strang and C. Linnhoff-Popien, "A Context Modelling Survey", in: *International Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp - Sixth International Conference on Ubiquitous Computing*, Tokyo, Japan Sept. 2004, pp. 34-41.

[27] C. R. G. de Farias, "Using UML for Service Creation", in: *Proceedings of the Workshop on Future Trends of Distributed Computing Systems*, San Juan, Puerto Rico, May 2003, pp. 86-92.

[28] S. A. Thibault, R. Marlet and C. Consel, "Domain-Specific Languages: From Design to Implementation Application to Video Device Drivers Generation", IEEE Transactions on Software Engineering, Vol. 25, No. 3, pp. 363-377, May/June 1999.

[29] R. Zbib, A. Jain, D. Bassu and H. Agrawal, "Generating Domain-Specific Graphical Modelling Editors from Metamodels", in: *Proceedings of the Annual IEEE Computer Software and Applications Conference*, Chicago, USA, Sept. 2006, pp. 129-138.

[30] A. K. Dey and G. D. Abowd, "Towards a Better Understanding of Context and Context-Awareness", in: *Proceedings of the Conference on Human Factors in Computing Systems*, The Hague, The Netherlands, April 1-6, 2000, pp. 79-80.

[31] A. K. Dey, "Understanding and Using Context", *Personal and Ubiquitous Computing, Special issue on Situated Interaction and Ubiquitous Computing*, Vol. 5, No. 1, pp. 4-7, Feb. 2001.

[32] A. K. Dey and G. D. Abowd, "The Context Toolkit: Aiding the Development of Context-Aware Applications", in: *ICSE Workshop on Software Engineering for Wearable and Pervasive Computing, International Conference on Software Engineering*, Limerick, Ireland, June 2000.

[33] A. Schmidt, K. A. Adoo, A. Takaluoma, U. Tuomela, K. Van Laerhoven and W. Van De Velde, "Advanced Interaction in Context", in: *Proceedings of the International Symposium on Handheld and Ubiquitous Computing, Lecture Notes in Computer Science (LNCS)*, Vol. 1707, Springer, Sept. 1999, pp. 89-101.

[34] G. Chen and D. Kotz, "Context Aggregation and Dissemination in Ubiquitous Computing Systems", in: *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, New York, USA, June 2002, pp. 105-114.

[35] K. Henricksen and J. Indulska, "Developing Context-Aware Pervasive Computing Applications: Models and Approach", *Pervasive and Mobile Computing Journal*, Vol. 2, No. 1, pp. 37-64, Feb. 2006.

[36] T. McFadden, K. Henricksen and J. Indulska, "Automating Context-Aware Application Development", in: *Proceedings of the International Workshop on Advanced Context Modelling, Reasoning and Management - UbiComp*, Nottingham, England, Sept. 2004, pp. 90-95.

[37] S. Pokraev, J. Koolwaaij, M. Van Setten, T. Broens, P. D. Costa, M. Wibbels, P. Ebben and P. Strating, "Service Platform for Rapid Development and Deployment of Context-Aware Mobile Applications", in: *Proceedings of the IEEE International Conference on Web Services*, Florida, USA, July 2005, pp. 639-646.

[38] G. M. Kapitsaki, D. A. Kateros and I. S. Venieris, "Architecture for Provision of Context-aware Web Applications based on Web Services", in: *Proceedings of the IEEE International Symposium on Personal, Indoor and Mobile Radio Communications*, Cannes, France, Sept. 2008, pp. 14-18.

[39] M. Saternus, T. Weis, M. Knoll and F.Durr, "A Middleware for Context-Aware Applications and Services based on Messenger Protocols", in: *Proceedings of the IEEE International Conference on Pervasive Computing and Communication Workshops*, New York, USA, Mar. 2007, pp. 467-471.

[40] T. Gu, H. K. Pung, and D. Q. Zhang, "A Middleware for Building Context-Aware Mobile Services", in: *Proceedings of the IEEE Vehicular Technology Conference*, Vol. 5, Los Angeles, USA, May 2004, pp. 2656-2660.

[41] F. Ay, "Context Modelling and Reasoning Using Ontologies", Technical Report, University of Technology, Berlin, July 2007, pp. 1-9.

[42] X. H. Wang, D. Q. Zhang , T. Gu and H. K. Pung, "Ontology based Context Modelling and Reasoning using OWL", in: *Proceedings of the IEEE International Conference on Pervasive Computing and Communication Workshops*, Florida, USA, Mar. 2004, pp. 18-22.

[43] J. Bauer, "Identification and Modelling of Contexts for Different Information Scenarios in Air Traffic", Diplomarbeit, Faculty of Electrical Engineering and Computer Sciences, Technische Universitat Berlin, 2003.

[44] C. Simons and G. Wirtz, "Modelling Context in Mobile Distributed Systems with the UML", *Journal of Visual Languages and Computing*, Vol. 18, No. 4, pp. 420-439, Aug. 2007.

[45] Eclipse Foundation Incorporation, Eclipse Modelling Framework (EMF) [Online] Available: http://www.eclipse.org/modeling/emf/ [Accessed March 06, 2009].

[46] C. R. G de Farias, M. M. Leite, C. Z. Calvi, R. M. Pessoa and J. G. Pereira Filho, "A MOF metamodel for the Development of Context-Aware Mobile Applications", in: *Proceedings of the ACM symposium on Applied computing*, Seoul, Korea, Mar. 2007, pp. 947-952.

[47] Object Management Group (OMG), "Meta Object Facility (MOF) Core Specification version 2.0" [Online] Available: http://www.omg.org/docs/formal/06-01-01.pdf [Accessed February 16, 2009].

[48] K. Henricksen, J. Indulska and A. Rokotonirainy, "Modelling Context Information in Pervasive Computing Systems", in: *Pervasive Computing, Lecture Notes on Computer Science (LNCS)*, Vol. 2414, Springer Berlin/Heidelberg, 2002, pp. 167-180.

[49] K. Henricksen and J. Indulska, "A Software Engineering Framework for Context-Aware Computing", in: *Proceedings of the IEEE Annual Conference of Pervasive Computing and Communications*, Florida, USA, Mar. 2004, pp. 77-86.

[50] K. Henricksen, J. Indulska and T. McFadden, "Modelling Context Information with ORM", in: *On the Move to Meaningful Internet Systems 2005: OTM Workshops, Lecture Notes in Computer Science (LNCS)*, Vol. 3762, Springer Berlin/Heidelberg, 2005, pp. 626-635.

[51] A.K. Dey, G.D. Abowd, "CybreMinder: A Context-Aware System for Supporting Reminders", in: *Proceedings of the International Symposium on Handheld and Ubiquitous Computing, Lecture Notes in Computer Science (LNCS)*, Vol. 1927, Springer London, 2000, pp. 172–186.

[52] G. Broll, H. Hubmann, G. N. Prezerakos, G. Kapitsaki and S. Salsano, "Modelling Context Information for Realizing Simple Mobile Services", in: *Proceedings of the IST Mobile and Wireless Communications Summit*, Budapest, Hungary, July 2007, pp. 1-5.

[53] Q. Z. Sheng and B. Benatallah, "ContextUML: A UML-Based Modeling Language for Model-Driven Development of Context-Aware Web Services", in: *Proceedings of the International Conference on Mobile Business*, Sydney, Australia, July 2005, pp. 206-212.

[54] F. Seyler, C. Taconet and G. Bernard, "Context Aware Orchestration Meta-Model", in: *Proceedings of the International Conference on Autonomic and Autonomous Systems*, Athens, Greece, June 2007, Pages: 17.

[55] L. Baresi and M. Pezze, "On Formalising UML with High-Level Petri Nets", in: *Concurrent Object-Oriented Programming and Petri Nets, Lecture Notes in Computer Science (LNCS)*, Springer-Verlag Berlin/Heidelberg, 2001, pp.276-304.

[56] N. Milanovic and M. Malek, "Current Solutions for Web Service Composition", *IEEE Internet Computing*, Vol. 8, No. 6, pp. 51-59, November/December 2004.

[57] A. Nait-Sidi-Moh, C. Dumez, J. Gaber, and M. Wack, "Petri Net based Verification and Validation of UML-S models", *Submitted to the IEEE International Conference on Advanced Information Networking and Applications (AINA2009)*, 2008.

[58] C. Dumez, A. Nait-sidi-moh, J. Gaber and M. Wack, "Modelling and Specification of Web Services Composition using UML-S", in: *Proceedings of the International Conference on Next Generation Web Services Practices*, Oct. 2008, pp. 395-398.

[59] Y. Yang, Q. Tan, J. Yu and F. Liu, "Transformation BPEL to CP-Nets for Verifying Web Services Composition", in: *Proceedings of the International Conference on Next Generation Web Services Practices*, Seoul, Korea, Aug. 2005, pp. 6.

[60] H. Dun, W. Zhao and L. Wang, "Transform BPEL to Synchronisation Net", Unpublished.

[61] H. Dun, H. Xu, L. Wang, "Transformation of BPEL Processes to Petri Nets", in: *Proceedings of IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering*, Nanjing, China, June 2008, pp.166-173.

[62] E. Andonoff, L. Bouzguenda and C. Hanachi, "Specifying Workflow Web Services using Petri nets with Objects and Generating their OWL-S Specifications", in: *E-Commerce and Web Technologies, Lecture Notes in Computer Science (LNCS),* Vol. 3590, Springer Berlin/Heidelberg, 2005, pp. 41-52.

[63] K. Lano, *Advanced Systems Design with Java, UML and MDA*, Oxford, United Kingdom, Elsevier Butterworth-Heinemann, 2005.

[64] Object Management Group (OMG), Unified Modelling Language (UML) 2.0 Specification. [Online] Available: http://www.omg.org/docs/formal/05-07-05.pdf [Accessed February 16, 2009].

[65] A. Evans, R. B. France, K. Lano and B. Rumpe, "The UML as a Formal Modelling Notation", *in: The Unified Modeling Language «UML»'98: Beyond the Notation – Selected Papers, Lecture Notes In Computer Science (LNCS)*, Vol. 1618, 1998, pp. 336-348.

[66] B. Graaf and A. V.Deursen, "Visualisation of Domain-Specific Modelling Languages Using UML", in: *Proceedings of the Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, Tucson, Arizona, USA, Mar. 2007, pp. 586-595.

[67] A. V. Deursen, P. Klint and J. Visser, "Domain-Specific Languages: An Annotated Bibliography", *ACM SIGPLAN Notices*, Vol. 35, No. 6, pp.26-36, June 2000.

[68] J. Evermann and Y. Wand, "Toward Formalizing Domain Modelling Semantics in Language Syntax", *IEEE Transactions on Software Engineering*, Vol. 31, No.1, pp. 21-37, Jan. 2005.

[69] T. Clark, A. Evans, P. Sammut and J. Willans, "An eXecutable Metamodelling Facility for Domain-Specific Language Design", in: *Proceedings of the Object-Oriented Programming, Systems, Languages, and Applications Workshop on Domain-Specific Modelling*, 2004.

[70] N. Iscoe, G. B. Williams and G. Arango, "Domain Modelling for Software Engineering", in: *Proceedings of the IEEE International Conference on Software Engineering*, Austin, Texas, USA, May 1991, pp. 340-343.

[71] M. Staron, “Adopting Model Driven Software Development in Industry - A Case Study at Two Companies”, in: *Proceedings of the International Conference on Model Driven Engineering Languages and Systems, Lecture Notes In Computer Science (LNCS)*, Vol. 4199, Springer Berlin/Heidelberg, Oct. 2006, pp. 57-72.

[72] M. Afonso, R. Vogel and J. Texeira, “From Code Centric to Model Centric Software Engineering: Practical Case Study of MDD Infusion in a Systems Integration Company”, in: *Workshop on Model-Based Development of Computer-Based Systems and International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, Potsdam, Germany, Mar. 2006, pp.125-134.

[73] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Springle and G. Karsai, “Composing Domain-Specific Design Environments”, IEEE Computer Society Press, vol. 34, no. 11, pp. 44-51, November 2001.

[74] A. Sreenivas, R. Venkatesh and M. Joseph, “Meta-Modelling for Formal Software Development”, *Electronic Notes in Theoretical Computer Science*, Vol. 42, Jan. 2001, pp. 1-11.

[75] J. Abd-Ali and K. El Guemhioui, “Do MDA Transformations Preserve Meaning? An Investigation into Preserving Semantics”, *in: Proceedings of Software Engineering Applications*, 2006, pp.13-22.

[76] J. P. Nytun, A. Prinz and M. S. Tveit, “Automatic Generation of Modelling Tools”, in: *Proceedings of the European Conference Model Driven Architecture – Foundations and Applications, Lecture Notes on Computer Science (LNCS)*, Vol. 4066, Springer Berlin/Heidelberg, July 2006, pp. 268-283.

[77] C, Cetina, E. Serral, J. Munoz and V. Pelechano, “Tool Support for Model Driven Development of Pervasive Systems”, in: *Proceedings of the IEEE Workshop on Model-Based Methodologies for Pervasive Embedded Software*, Braga, Portugal, Mar. 2007, pp. 33-44.

[78] M. Antkiewicz and K. Czarnecki, “Framework-Specific Modelling Languages with Round-Trip Engineering”, in: *Proceedings of International Conference on Model Driven Engineering Languages and Systems, Lecture Notes on Computer Science (LNCS)*, Vol. 4199, Springer-Verlag, Oct. 2006, pp. 692-706.

[79] M. Azmoodeh, N. Georgalas and S. Fisher, “Model-Driven Systems Development and Integration Environment”, *BT Technology Journal*, Vol. 23, No. 03, pp. 96-110, July 2005.

[80] S. Ou, N. Georgalas, M. Azmoodeh, K. Yang and X. Sun, “A Model Driven Integration Architecture for Ontology-Based Context Modelling and Context-Aware Application Development”, in: *Proceedings of European Conference on Model Driven Architecture - Foundations and Applications, Lecture Notes on Computer Science (LNCS)*, Vol. 4066, Springer-Verlag, July 2006, pp. 188-197.

[81] N. Georgalas, S. Ou, K. Yang and M. Azmoodeh, “Towards a Model-Driven Approach for Ontology-Based Context-Aware Application Development: A Case Study”, in: *Proceedings of IEEE International Workshop on Model-based Methodologies for Pervasive and Embedded Software*, Braga, Portugal, March 2007, pp. 21-32.

[82] S. Kelly and R. Pohjonen, “Worst Practices for Domain-Specific Modeling”, *IEEE Software*, Vol. 26, No. 4, pp. 22-29, July/Aug. 2009.

[83] K. Balasubramanian, A. Gokhale, G. Karsai, J. Sztipanovits and S. Neema, "Developing Applications Using Model-Driven Design Environments", *IEEE Computer*, Vanderbilt University, 2006.
[84] Borland Together Integrated and Agile Design Solutions, "Getting Started Guide for Borland Together 2006 for Eclipse", [Online] Available: http://techpubs.borland.com/together/tec2006/en/GettingStarted.pdf [Accessed September 28, 2009].
[85] International Business Machines Corp. "Eclipse Platform Technical Overview", [Online] Available: http://www.eclipse.org/whitepapers/eclipse-overview.pdf [Accessed September 28, 2009].
[86] C. O' Halloran, "Model Based Code Verification", in: Formal Methods of Software Engineering, Lecture Notes on Computer Science (LNCS), Vol. 2885, November 2003, pp. 16-25.
[87] Eclipse Foundation Inc., Graphical Modelling Framework (GMF) [Online] Available: http://www.eclipse.org/modeling/gmf/ [Accessed September 28, 2009].
[88] INRIA Research Institution, Atlas Transformation Language (ATL) [Online] Available: http://www.eclipse.org/m2m/atl [Accessed September 28, 2009].
[89] openArchitectureWare.org and Eclipse Foundation Inc., openArchitectureWare (oAW) [Online] Available: http://www.eclipse.org/workinggroups/oaw/ [Accessed September 28, 2009].
[90] A. Gerber and K. Raymond, "MOF to EMF: There and Back Again", in: *Proceedings of the Object-Oriented Programming, Systems, Languages and Applications Workshop on Eclipse Technology eXchange*, California, USA, Oct. 2003, pp. 60-64.
[91] M. Mohamed, M. Romdhani and K. Ghedira, "MOF-EMF Alignment", in: *Proceedings of the International Conference on Autonomic and Autonomous Systems*, Athens, Greece, June 2007, pp. 1-7.
[92] M. MacHenry and J. Matthews, "Topsl: A Domain-Specific Language for On-Line Surveys", in: *Proceedings of the SIGPLAN ACM Workshop on Scheme and Functional Programming*, Snowbird, Utah, September 2004.
[93] D. Harel and B. Rumpe, "Meaningful Modelling: What's the Semantics of "Semantics"?", *IEEE Computer*, Vol. 37, No. 10, pp. 64-72, October 2004.
[94] P. Kearney, "Managing Assurance, Security and Trust for Services (MASTER) Project Preliminary Specification and Design of Graphical Workbench", British Telecom (BT) Internal Technical Report, 2009.
[95] G. Ortiz, B. Bordbar and J. Hernandez, "Evaluating the Use of AOP and MDA in Web Service Development", in: *IEEE International Conference on Internet and Web Applications and Services*, Athens, Greece, June 2008, pp. 78-83.
[96] J. Jelinek and P. Slavik, "GUI Generation from Annotated Source Code", in: *Proceedings of the Annual ACM Conference on Task Models and Diagrams – Session: Generating the User Interface*, Prague, Czech Republic, 2004, pp. 129-136.
[97] S. Sauer, M. Dürksen, A. Gebel, and D. Hannwacker, "GuiBuilder - A Tool for Model-Driven Development of Multimedia User Interfaces", in: *Proceedings of the Workshop on Model Driven Design of Advanced User Interfaces – MoDELS 2006*, Genova, Italy, Oct. 2006.

[98] J. M. Heines and M. J. Schedlbauer, "Teaching Object-Oriented Concepts Through GUI Programming", Department of Computer Science, University of Massachusetts Lowell, May 2007.

[99] S. Link, T. Schuster, P. Hoyer and S. Abeck, "Focusing Graphical User Interfaces in Model-Driven Software Development", in: *Proceedings of the International Conference on Advances in Computer-Human Interaction*, Sainte Luce, Martinique, Feb. 2008, pp. 3-8.

[100] D. Harel, "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming*, Vol. 8, No. 3, pp. 231-274, June 1987.

[101] R. Eshuis and R. Wieringa, "A Comparison of Petri Nets and Activity Diagram Variants", in: *Proceedings of International Colloquium on Petri Net Technologies for Modelling Communication Based Systems*, Berlin, Germany, Sep. 2001, pp. 93-104.

[102] J. A. Ferreira and J. P. E. de Oliveira, "Modelling Multidisciplinary Systems with Hybrid Statecharts", in: *Proceedings of the IEEE Conference on Computer Aided Control Systems Design*, Munich, Germany, Oct. 2006, pp. 422-427.

[103] R. Fehling, "A Concept of Hierarchical Petri Nets With Building Blocks", in: *Advances in Petri Nets, Lecture Notes in Computer Science (LNCS)*, Vol. 674, Springer Berlin/Heidelberg, 1993, pp. 148-168.

[104] J. P. Lopez-Grao, J. Merseguer and J. Campos, "From UML Activity Diagrams to Stochastic Petri Nets: Application to Software Performance Engineering", in: Proceedings of the International Symposium on Computer and Information Sciences, Jan. 2004, pp. 25-36.

[105] H. Goldsby, B. H. C. Cheng, S. Konrad and S. Kamdoum, "A Visualization Framework for the Modelling and Formal Analysis of High Assurance Systems", in: *Proceedings of the Conference on Model Driven Engineering Languages and Systems, Lecture Notes on Computer Science (LNCS)*, Vol. 4199, Springer Berlin/Heidelberg, Oct. 2006, pp. 707-721.

[106] C. Girault and R. Valk, "Petri Nets for System Engineering: A Guide to Modelling, Verification and Applications", Springer, 2003.

[107] J. L. Peterson, "Petri Net Theory and the Modelling of Systems", Prentice Hall, Englewood Cliffs, 1981.

[108] R. Tan and S-U. Guan, "A Dynamic Petri Net Model for Iterative and Interactive Distributed Multimedia Presentation", *IEEE Transactions on Multimedia*, Vol. 7, No.5, pp. 869-879, Oct. 2005.

[109] D. Xu and K. E. Nygard, "Threat-Driven Modelling and Verification of Secure Software Using Aspect-Oriented Petri Nets", *IEEE Transactions on Software Engineering*, Vol. 32, No.4, pp. 265-278, Apr. 2006.

[110] K. Jensen, "A Brief Introduction to Coloured Petri Nets", *in: Proceedings of the International Workshop on Tools and Algorithms for Construction and Analysis of Systems, Enschede, The Netherlands, Lecture Notes in Computer Science (LNCS)*, Vol. 1217, Springer-Verlag, 1997, pp. 203-208.

[111] T. Miyamoto and S. Kumagai, "A Survey of Object-Oriented Petri Nets and Analysis Methods", *IEICE Transactions on Fundamentals of Electronics,*

Communications and Computer Sciences, Vol. E88, No.11, pp. 2964-2971, Nov. 2005.

[112] C. Lakos, "Object Oriented Modelling with Object Petri Nets", in: *Concurrent Object-Oriented Programming and Petri Nets - Advances in Petri Nets, Lecture Notes in Computer Science (LNCS)*, Springer Berlin/Heidelberg, Vol. 2001, Jan. 2001, pp. 1-37.

[113] O. Biberstein, D. Buchs and N. Guelfi, "Object-Oriented Petri Nets with Algebraic Specifications: The CO-OPN/2 Formalism", Concurrent Object-Oriented Programming and Petri Nets, Lecture Notes in Computer Science (LNCS), Springer Berlin/Heidelberg, Vol. 2001, Jan. 2001, pp. 73-130.

[114] H. Motameni, A. Movaghar, B. Shirazi, M. Aminzadeh and H. Samadi, "Analysis Software with an Object-Oriented Petri Net Model", *World Applied Sciences Journal*, Vol. 3, No.4, pp. 565-576, 2008.

[115] R. Valk, "Petri Nets as Token Objects - An Introduction to Elementary Object Nets", in: Proceedings of the International Conference on Application and Theory of Petri Nets, Lecture Notes In Computer Science (LNCS), Vol. 1420, Springer Berlin/Heidelberg, June 1998, pp. 1-25.

[116] O. Kummer, F. Wienberg and M. Duvigneau, "Reference Net User Guide", Theoretical Foundations Group, Department of Informatics, University of Hamburg, [Online] Available: http://www.informatik.uni-hamburg.de/TGI/renew/renew.pdf, [Accessed September 28, 2009].

[117] O. Kummer, "Tight Integration of Java and Petri Nets", Theoretical Foundations Group, Department of Informatics, University of Hamburg, 1999.

[118] O. Kummer, "Introduction to Petri Nets and Reference Nets", Sozionik Aktuell, No.1, 2001, pp.7-16.

[119] V. Kulkarni and S. Reddy, "Separation of Concerns in Model-Driven Development", *IEEE Software*, Vol. 20, No. 5, pp. 64-69, September/October 2003.

[120] E. Kindler, "Software and Systems Engineering – High-level Petri Nets: Part2 Transfer Format - Proposed Draft Addendum to International Standard", ISO/IEC 15909 Part 2 – Version 0.6.3, June 2005.

[121] L. Hillah and F. Kordon, "Model engineering on Petri nets for ISO/IEC 15909-2: API Framework for Petri Net types Metamodels", Petri Nets Newsletter, 2005, pp. 22-40.

[122] M. Weber, E. Kindler and C. Stehno, "Petri Net Markup Language", Department of Computer Science, Humboldt Universität zu Berlin, 2004. [Online] Available: http://www2.informatik.hu-berlin.de/top/pnml/pnml.html, [Accessed January 23, 2009].

[123] S. Christensen and N. D. Hansen, "Coloured Petri Nets Extended with Channels for Synchronous Communication", Technical Report DAIMI PB–390, Aarhus University, 1992.

[124] P. V. Bhansali, "Complexity Measurement of Data and Control Flow", *ACM SIGSOFT Software Engineering Notes*, Vol. 30, No.1, pp. 1-2, Jan. 2005.

[125] T. Littlefair, "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, 2001.

[126] R. Lincke, J. Lundberg and W. Löwe, "Comparing Software Metrics Tools", in: *Proceedings of International Symposium on Software Testing and Analysis*, Seattle, Washington, USA, July 2008, pp. 131-142.

[127] B. Boehm, C. Abts, B. Clark, S. Devnani-Chulani, E. Horowitz, R. Madachy, D. Reifer, R. Selby and B.Steece, "COCOMO II Model Definition Manual", Center for Systems and Software Engineering, University of Southern California.

[128] Z. Chen, B. Boehm, T. Menzies and D. Port, "Finding the Right Data for Software Cost Modelling", *IEEE Software*, Vol. 22, No. 6, pp. 38-46, November/December 2005.

[129] B. Clark, S. Devnani-Chulani and B. Boehm, "Calibrating the COCOMO II Post-Architecture Model", in: *Proceedings of the International Conference on Software Engineering*, Kyoto, Japan, April 1998, pp. 477-480.

[130] J. Baik, B. Boehm and B. M. Stecee, "Dissagregating and Calibrating the CASE Tool Variable in COCOMO II", *IEEE Transactions on Software Engineering*, Vol. 28, No.11, pp. 1009-1022, November 2002.

[131] K. Lum and E. Monson, "Software Cost Analysis Tool User Document", NASA-Jet Propulsion Laboratory Pasadena, California.

[132] K. Lum, M. Bramble, J. Hihn, J. Hackney, M. Khorrami and E. Monson, "Handbook for Software Cost Estimation", NASA-Jet Propulsion Laboratory Pasadena, California.

Appendices

Appendix A: Survey to MIDlet transformation mapping

```
module Survey2Midlets; -- Module Template
create OUT : midlets from IN : surveys;
rule SurveysMetamodel2MIDletMetamodel{
	from
		sm: surveys!SurveysMetamodel
	to
		mm: midlets!MIDletMetamodel (
			name <- sm.name,
			forms <- sm.pages,
			choicegroups <- sm.questions,
			elements <- sm.answers
		)
}
rule Page2Form{
	from
		page: surveys!Page
	to
		form: midlets!Form (
			name <- page.name,
			title <- page.title,
			ticker <- page.description,
			containsChoiceGroups <- page.hasQuestions
		)
}
rule Question2ChoiceGroup{
	from
		question: surveys!Question
	to
		choicegroup: midlets!ChoiceGroup (
			name <- question.name,
			label <- question.qcontext,
			index <- question.index,
			includesElements <- question.hasAnswers
		)
}
rule Answer2Element{
	from
		answer: surveys!Answer
	to
		element: midlets!Element (
			name <- answer.name,
			value <- answer.acontext,
			index <- answer.index
		)
}
```

Appendix B: SML code generation template – J2ME

```
«REM»Import the file that includes code generation extension functions.«ENDREM»
«EXTENSION templates::j2me»
«DEFINE Root FOR surveys::SurveysMetamodel»
«EXPAND Page FOREACH pages»
«ENDDEFINE»
«DEFINE Page FOR surveys::Page»
«FILE this.name+"MIDlet.java"»
package survey_j2me;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class «this.name»MIDlet extends MIDlet implements CommandListener {
        private Command exitCommand,okCommand;
        private Form «this.name»,submittedForm;
        private boolean midletPaused = false;
        private StringItem stringItem;
«FOREACH hasQuestions AS question»
        private ChoiceGroup «question.name»;
«ENDFOREACH»
        public «this.name»getForm() {
                if («this.name» == null) {
                        «this.name» = new Form («this.title», null );
                        «this.name».setTicker(«this.description»);
«FOREACH hasQuestions AS question»
                        «this.name».insert(«question.index», getChoiceGroup«question.name»());
«ENDFOREACH»
                        «this.name».addCommand(getExitCommand());
                        «this.name».addCommand(getOkCommand());
                        «this.name».setCommandListener(this);
                }
                return «this.name»;
        }
«FOREACH hasQuestions AS question»
        public ChoiceGroup getChoiceGroup«question.name»(){
                if («question.name» == null) {
                        «question.name» = new ChoiceGroup("«question.qcontext»",
                                                        Choice.EXCLUSIVE);
«FOREACH question.hasAnswers AS answer»
                        «question.name».append("«answer.acontext»", null);
«ENDFOREACH»
«FOREACH question.hasAnswers AS answer»
                        «question.name».setSelectedIndex(«answer.index», false);
«ENDFOREACH»
«FOREACH question.hasAnswers AS answer»
                        «question.name».setFont(«answer.index», null);
«ENDFOREACH»
                }
                return «question.name»;
        }
«ENDFOREACH»
        «this.commonMIDletFunctions()»
}
«ENDFILE»
«ENDDEFINE»
```

Appendix C: SML code generation extension functions – J2ME

```
import surveys;

String commonMIDletFunctions(Page p):

        'public void startApp() {
                if (midletPaused) {
                        resumeMIDlet ();
                }
                else {
                        initialize ();
                        startMIDlet ();
                }
                midletPaused = false;
        }
        public void pauseApp() {
                midletPaused = true;
        }
        public void destroyApp(boolean unconditional){
        }
        private void initialize() {
        }
        public void startMIDlet() {
                switchDisplayable(null, getForm());
        }
        public void resumeMIDlet() {
        }
        public void switchDisplayable(Alert alert, Displayable nextDisplayable) {
                Display display = getDisplay();
                if (alert == null) {
                        display.setCurrent(nextDisplayable);
                }
                else {
                        display.setCurrent(alert, nextDisplayable);
                }
        }
        public Display getDisplay () {
                return Display.getDisplay(this);
        }
        public StringItem getStringItem() {
                if (stringItem == null) {
                        stringItem = new StringItem("Survey:", "Completed succesfully");
                }
                return stringItem;
        }
        public Form getSubmittedForm() {
                if (submittedForm == null) {
                        submittedForm = new Form("", new Item[] { getStringItem() });
                        submittedForm.addCommand(getExitCommand());
                        submittedForm.setCommandListener(this);
                }
                return submittedForm;
        }
```

```
public Command getOkCommand() {
        if (okCommand == null) {
                okCommand = new Command("Ok", Command.OK, 0);
        }
        return okCommand;
}
public Command getExitCommand() {
        if (exitCommand == null) {
                exitCommand = new Command("Exit", Command.EXIT, 0);
        }
        return exitCommand;
}
public void exitMIDlet() {
        switchDisplayable (null, null);
        destroyApp(true);
        notifyDestroyed();
}
public void commandAction(Command command, Displayable displayable) {
        if (displayable == form || displayable == submittedForm) {
                if (command == exitCommand) {
                        exitMIDlet();
                }
                else if (command == okCommand) {
                        switchDisplayable(null, getSubmittedForm());
                }
        }
}'
```

Appendix D: SML code generation template – Java

```
«DEFINE Root FOR surveys::SurveysMetamodel»
«EXPAND Page FOREACH pages»
«ENDDEFINE»
«DEFINE Page FOR surveys::Page»
«FILE this.name+"JApplet.java"»
package survey_java;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class «this.name»JApplet extends JApplet implements ActionListener {
«FOREACH hasQuestions AS question»
        private JLabel «question.name»;
«ENDFOREACH»
«FOREACH hasQuestions.hasAnswers AS answer»
        private JRadioButton «answer.name»;
«ENDFOREACH»
«FOREACH hasQuestions AS question»
        private ButtonGroup group«question.name»;
«ENDFOREACH»
        private JPanel pane;
        private JLabel description;
        private JButton submit;
        public void init(){
                this.setName("«this.title»");
                description = new JLabel("«this.description»");
                description.setForeground(Color.RED);
                pane = new JPanel();
                pane.add(description);
«FOREACH hasQuestions AS question»
                createQuestion«question.name»();
«ENDFOREACH»
                createGUI();
        }
«FOREACH hasQuestions AS question»
        public void createQuestion«question.name»(){
                «question.name» = new JLabel("«question.qcontext»");
                «question.name».setForeground(Color.BLACK);
«FOREACH question.hasAnswers AS answer»
                «answer.name» = new JRadioButton("«answer.acontext»");
                «answer.name».setBackground(Color.WHITE);
                «answer.name».setActionCommand("«answer.name»");
                «answer.name».addActionListener(this);
«ENDFOREACH»
                group«question.name» = new ButtonGroup();
«FOREACH question.hasAnswers AS answer»
                group«question.name».add(«answer.name»);
«ENDFOREACH»
        }
«ENDFOREACH»
        public void createGUI(){
«FOREACH hasQuestions AS question»
                pane.add(«question.name»);
```

```
«FOREACH question.hasAnswers AS answer»
                    pane.add(«answer.name»);
«ENDFOREACH»
«ENDFOREACH»
                    submit = new JButton("Submit Survey");
                    submit.addActionListener(this);
                    pane.add(submit);
                    pane.setBackground(Color.WHITE);
                    pane.setBorder(BorderFactory.createMatteBorder(1,1,2,2,Color.black));
                    setContentPane(pane);
          }
          public void actionPerformed(ActionEvent e) {
                    // TODO implementation code
          }
}
«ENDFILE»
«ENDDEFINE»
```

Appendix E: Context Modelling Language constraints

1. **context** Entity

   inv: Entity.allInstances()->forAll(e1: Entity, e2:Entity | e1 <> e2 **implies** e1.type <> e2.type)
2. **context** Entity

   inv: if not self.type.oclIsUndefined() **then self**.type.size() > 0 **else self**.type.oclIsInvalid() **endif**
3. **context** Entity

   inv: self.situations->forAll(cs1: ContextSituation, cs2: ContextSituation | cs1 <> cs2 **implies**
   cs1.attribute <> cs2.attribute)
4. **context** Entity

   inv: Entity.allInstances()->forAll(e1: Entity, e2:Entity |e1<>e2 **implies** e1.ERsource <> e2.ERsource)
5. **context** Entity

   inv: not Entity.allInstances()->exists(e1: Entity | e1.ERsource.REtarget->exists(e2:Entity | e2 = e1))
6. **context** Entity

   inv: self.ECAsource->size() > 0
7. **context** ContextAssociation

   inv: ContextAssociation.allInstances()->forAll(ca1: ContextAssociation, ca2: ContextAssociation |
   ca1 <> ca2 **implies** ca1.name <> ca2.name)
8. **context** ContextAssociation

   inv: if not self.name.oclIsUndefined() **then self**.name.size() > 0 **else self**.name.oclIsInvalid() **endif**
9. **context** ContextAssociation

   inv: not self.CACtarget->isEmpty()
10. context ContextAssociation

    inv: self.CACtarget->size() = 1
11. **context** Context

    inv: Context.allInstances()->forAll(c1: Context, c2: Context | c1 <> c2 **implies** c1.name <> c2.name)
12. **context** Context

    inv: if not self.name.oclIsUndefined() **then self**.name.size() > 0 **else self**.name.oclIsInvalid() **endif**
13. **context** Context

    inv: self.conproperties->forAll(p1: Property, p2: Property | p1 <> p2 **implies** p1.name <> p2.name)
14. **context** Context

    inv: 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)
15. **context** Property

    inv: if not self.name.oclIsUndefined() **then self**.name.size() > 0 **else self**.name.oclIsInvalid() **endif**
16. **context** Property

    inv: if not self.type.oclIsUndefined() **then self**.type.size() > 0 **else self**.type.oclIsInvalid() **endif**

17. **context** Property
    inv: if self.complextype.oclIsUndefined() **then true else self**.type = **self**.complextype.name **endif**
18. **context** Property
    inv: if self.enumeration.oclIsUndefined() **then true else self**.type = **self**.enumeration.name **endif**
19. **context** ContextDatatype
    inv: ContextDatatype.allInstances()->forAll(cd1: ContextDatatype, cd2: ContextDatatype | cd1 <> cd2 **implies** cd1.name <> cd2.name)
20. **context** ContextDatatype
    inv: if not self.name.oclIsUndefined() **then self**.name.size() > 0 **else self**.name.oclIsInvalid() **endif**
21. **context** ContextDatatype
    inv: self.cdproperties->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') **and** (p.complextype = **null and** p.enumeration = **null**))
22. **context** ContextDatatype
    inv: self.cdproperties->forAll(p1: Property, p2: Property | p1 <> p2 **implies** p1.name <> p2.name)
23. **context** ContextEnum
    inv: ContextEnum.allInstances()->forAll(ce1: ContextEnum, ce2: ContextEnum | ce1 <> ce2 **implies** ce1.name <> ce2.name)
24. **context** ContextEnum
    inv: if not self.name.oclIsUndefined() **then self**.name.size() > 0 **else self**.name.oclIsInvalid() **endif**
25. **context** ContextEnum
    inv: self.literals->forAll(lit1: EnumLiteral, lit2: EnumLiteral | lit1 <> lit2 **implies** lit1.value <> lit2.value)
26. **context** ContextSituation
    inv: if not self.attribute.oclIsUndefined() **then self**.attribute.size() > 0 **else self**.attribute.oclIsInvalid() **endif**
27. **context** ContextSituation
    inv: if not self.expression.oclIsUndefined() **then self**.expression.size() > 0 **else self**.expression.oclIsInvalid() **endif**
28. **context** Relation
    inv: Relation.allInstances()->forAll(r1: Relation , r2: Relation | r1 <> r2 **implies** r1.name <> r2.name)

29. **context** Relation
    inv: if not self.name.oclIsUndefined() **then self**.name.size() > 0 **else self**.name.oclIsInvalid() **endif**
30. **context** Constraint
    inv: Constraint.allInstances()->forAll(c1: Constraint, c2: Constraint | c1 <> c2 **implies** c1.name <> c2.name)
31. **context** Constraint
    inv: if not self.name.oclIsUndefined() **then self**.name.size() > 0 **else self**.name.oclIsInvalid() **endif**
32. **context** Absolute
    inv: if not self.startTime.oclIsUndefined() **then self**.startTime.size() > 0 **else self**.startTime.oclIsInvalid() **endif**
33. **context** Absolute
    inv: if not self.endTime.oclIsUndefined() **then self**.endTime.size() > 0 **else self**.endTime.oclIsInvalid() **endif**
34. **context** Comparative
    inv: if not self.expireTime.oclIsUndefined() **then self**.expireTime.size() > 0 **else self**.expireTime.oclIsInvalid() **endif**

Appendix F: ContextAssociation code generation template – J2ME

```
«EXTENSION extensions::accessors»
«DEFINE Root FOR cml::DocumentRoot»
«EXPAND Entity FOREACH entities»
«ENDDEFINE»
«DEFINE Entity FOR cml::Entity»
«FOREACH this.ECAsource AS ecas-»
«FILE ecas.ext1()+".java"»
 package j2me.context.sources;
 import java.util.Vector;
 import javax.microedition.rms.RecordStoreException;
 import j2me.context.management.RecordStoreManagement;
 import j2me.context.helpers.StringRenderer;
 import j2me.context.information.« this.ext2()+""+ecas.CACtarget.first().ext3()»;
«FOREACH ecas.CACtarget.conproperties AS cp-»
«IF cp.complextype != null-»
 import j2me.context.datatypes.«cp.complextype.name»;
«ELSEIF cp.enumeration != null-»
 import j2me.context.enumerations.«cp.enumeration.name»;
«ENDIF»
«ENDFOREACH»
 public class «ecas.ext1()»{
        private static final String atomic_context = "« this.ext2()+""+ecas.CACtarget.first().ext3()»";
        private static final String multiplicity = "«ecas.multiplicity»";
        private static final String multiplicityType = "«ecas.multiplicityType»";
        private static final String persistence = "«ecas.persistence»";
        private static final String permission = "«ecas.permission»";
«FOREACH ecas.CACtarget.conproperties AS cp-»
        private static final String «cp.name»= "«cp.name»";
«ENDFOREACH»
        private RecordStoreManagement rsm = null;
        private « this.ext2()+""+ecas.CACtarget.first().ext3()»
                «this.ext4()+""+ecas.CACtarget.first().ext5()» = null;
«FOREACH ecas.CACtarget.conproperties AS cp-»
«IF cp.complextype != null-»
        private «cp.complextype.name» «cp.complextype.ext9()» = null;
«ELSEIF cp.enumeration != null-»
        private «cp.enumeration.name» «cp.enumeration.ext11()» = null;
«ENDIF»
«ENDFOREACH»
        public void set«ecas.ext1()»(Vector «ecas.name») {
                int i=0;
                int j = 0;
                try {
                        if (rsm == null)
                                rsm = new RecordStoreManagement(atomic_context);
                        while ( i<«ecas.name».size()){
«FOREACH ecas.CACtarget.conproperties AS cp-»
                                rsm.put(«cp.name»+j,«ecas.name».elementAt(i).toString());
                                i++;
«ENDFOREACH»
                                j++;
                        }
                        rsm.save();
```

```
                }catch(RecordStoreException rse){
                        rse.printStackTrace();
                }
        }
«IF ecas.multiplicity == "0...1" || ecas.multiplicity == "1...1"»
        public «this.ext2()+""+ecas.CACtarget.first().ext3()» get«ecas.ext1()» (){
                int i=0;
                int j = 0;
                «this.ext4()+""+ecas.CACtarget.first().ext5()» = new
                        «this.ext2()+""+ecas.CACtarget.first().ext3()»();
«FOREACH ecas.CACtarget.conproperties AS cp-»
«IF cp.complextype != null»
                «cp.complextype.ext9()» =  new «cp.complextype.name»();
«ELSEIF cp.enumeration != null»
                «cp.enumeration.ext11()» =  new «cp.enumeration.name»();
«ENDIF»
«ENDFOREACH»
                try{
                        if (rsm == null)
                                rsm = new RecordStoreManagement(atomic_context);
                        int records = rsm.getSize();
                        while ( i<records){
«FOREACH ecas.CACtarget.conproperties AS cp-»
«IF cp.complextype != null»
                                «this.ext4()+""+ecas.CACtarget.first().ext5()».«cp.set1()»
                                                        («cp.complextype.ext9()»);
                                i++;
«ELSEIF cp.enumeration != null»
                                «cp.enumeration.ext11()».setState(rsm.get(«cp.name»+j));
                                «this.ext4()+""+ecas.CACtarget.first().ext5()».«cp.set1()»
                                                        («cp.enumeration.ext11()»);
                                i++;
«ELSEIF cp.enumeration == null && cp.complextype == null»
                                «this.ext4()+""+ecas.CACtarget.first().ext5()». «cp.set1()»
                                                        (rsm.get(«cp.name»+j));
                                i++;
«ENDIF»
«ENDFOREACH»
                                j++;
                                }
                }catch(RecordStoreException rse){
                        rse.printStackTrace();
                }
                return «this.ext4()+""+ecas.CACtarget.first().ext5()»;
        }
«ELSEIF ecas.multiplicity == "0...*" || ecas.multiplicity == "1...*"»
        public Vector get«ecas.ext1()»(){
                int i=0;
                int j = 0;
                Vector «ecas.name» = new Vector();
                try {
                        if (rsm == null)
                                rsm = new RecordStoreManagement(atomic_context);
                                int records = rsm.getSize();
                        while ( i<records-1){
                                String[] temp = null;
```

```
                                        StringRenderer sr = new StringRenderer();
                                        «this.ext4()+""+ecas.CACtarget.first().ext5()» =
                                                new « this.ext2()+""+ecas.CACtarget.first().ext3()»();
«FOREACH ecas.CACtarget.conproperties AS cp-»
«IF cp.complextype != null»
                                        «cp.complextype.ext9()» =  new «cp.complextype.name»();
«ELSEIF cp.enumeration != null»
                                        «cp.enumeration.ext11()» =  new «cp.enumeration.name»();
«ENDIF»
«ENDFOREACH»
«FOREACH ecas.CACtarget.conproperties AS cp-»
«IF cp.complextype != null»
                                        temp = new String[«cp.complextype.ext9()».size];
                                        temp = sr.renderLine(rsm.get(«cp.complextype.ext10()»+j),
                                                                «cp.complextype.ext9()».size);
«FOREACH cp.complextype.cdproperties AS cdp ITERATOR it-»
«IF cdp.type == "Integer"»
                                        «cp.complextype.ext9()».«cdp.set1()»
                                                (new Integer(Integer.parseInt(temp[«it.counter0»])));
«ELSEIF cdp.type == "String"»
                                        «cp.complextype.ext9()».«cdp.set1()»(temp[«it.counter0»]);
«ENDIF»
«ENDFOREACH»
                                        «this.ext4()+""+ecas.CACtarget.first().ext5()».«cp.set1()»
                                                                («cp.complextype.ext9()»);
                                        i++;
«ELSEIF cp.enumeration != null»
                                        «cp.enumeration.ext11()».setState(rsm.get(«cp.name»+j));
                                        «this.ext4()+""+ecas.CACtarget.first().ext5()».«cp.set1()»
                                                                («cp.enumeration.ext11()»);
                                        i++;
«ELSEIF cp.complextype == null && cp.enumeration == null»
                                        «this.ext4()+""+ecas.CACtarget.first().ext5()». «cp.set1()»
                                                                        (rsm.get(«cp.name»+j));
                                        i++;
«ENDIF»«ENDFOREACH»
                                        «ecas.name».insertElementAt
                                                («this.ext4()+""+ecas.CACtarget.first().ext5()», j);
                                        j++;
                                }
                        }catch(RecordStoreException rse){
                                rse.printStackTrace();
                        }
                        return «ecas.name»;
                }
«ENDIF»
                public static String getMultiplicity() {
                        return multiplicity;
                }
                public static String getMultiplicityType() {
                        return multiplicityType;
                }
                public static String getPersistence() {
                        return persistence;
                }
                public static String getPermission() {
```

```
                return permission;
        }
«FOREACH ecas.CACtarget.conproperties AS cp-»
        public static String «cp.get1()»() {
                return «cp.name»;
        }
«ENDFOREACH»
 }
«ENDFILE»
«ENDFOREACH»
«ENDDEFINE»
```

Appendix G: ContextReceiver code generation template – J2ME

```
«EXTENSION extensions::accessors»
«DEFINE Root FOR cml::DocumentRoot»
«EXPAND Entity FOREACH entities»
«ENDDEFINE»
«DEFINE Entity FOR cml::Entity»
«FILE this.type+"Receiver.java"»
 package j2me.context.receivers;
 import java.util.Enumeration;
 import java.util.Vector;
 import j2me.context.base.Context;
 import j2me.context.events.ContextEvent;
 import j2me.context.listeners.ContextListener;
 public class «this.type+"Receiver"» {
        «this.extension15()»
«FOREACH this.ECAsource.CACtarget AS cact ITERATOR it-»
«IF it.firstIteration»
        if( _context == Context.«this.extension2()+"_"+cact.extension3()» ) {
                _context = Context.«this.extension2 ()+"_"+cact.extension3()»;
                _fireContextEvent();
        }
«ELSEIF !it.firstIteration»
        else if( _context == Context.«this.extension2()+"_"+cact.extension3()») {
                _context = Context.«this.extension2()+"_"+cact.extension3()»;
                _fireContextEvent();
        }
«ENDIF»
«ENDFOREACH»
        «this.extension16()»
 }
«ENDFILE»
«ENDDEFINE»
```

Appendix H: Presentation Modelling Language constraints

1. **context** DocumentRoot

   inv: self.displays->size() = 1

2. **context** Display

   inv: self.title.substring(1,1) = "'" **and self**.title.substring(**self**.title.size(),**self**.title.size()) = "'"

3. **context** Display

   inv: self.layout = 'default' **or self**.layout = 'grid'

4. **context** Display

   inv: self.dproperties->forAll(p1: Property, p2:Property | p1 <> p2 **implies** p1.name <> p2.name)

5. **context** Display

   inv: if self.layout = 'default'
   then self.discontainers->forAll(con: Container| con.position = 'CENTER')
   or self.discontainers->forAll(con: Container | con.position = 'EAST')
   or self.discontainers->forAll(con: Container | con.position = 'WEST')
   or self.discontainers->forAll(con: Container | con.position = 'NORTH')
   or self.discontainers->forAll(con: Container | con.position = 'SOUTH')
   else self.discontainers->forAll(con: Container | con.position.toInteger().oclIsTypeOf(Integer))
   endif

6. **context** Container

   inv: Container.allInstances()->forAll(con1, con2 | con1 <> con2 **implies** con1.name <> con2.name)

7. **context** Container

   inv: self.layout = 'default' **or self**.layout = 'grid'

8. **context** Container

   inv: self.conproperties->forAll(p1: Property, p2:Property | p1 <> p2 **implies** p1.name <> p2.name)

9. **context** Component

   inv: Component.allInstances()->forAll(comp1, comp2 | comp1 <> comp2 **implies** comp1.name <> comp2.name)

10. context Component

    inv: self.compproperties->forAll(p1: Property, p2:Property | p1 <> p2 **implies** p1.name <> p2.name)

11. **context** Property

    inv: Property.allInstances()->forAll(p: Property | p.name = 'text' **or** p.name = 'title' **or** p.name = 'message' **or** p.name = 'rows' **or** p.name = 'columns' **or** p.name = 'lineWrap' **or** p.name = 'stringArray' **or** p.name = command)

12. **context** Display

    inv: if self.oclIsTypeOf(Display) **and self**.layout = 'grid' **then self**.dproperties->exists(p:Property | p.name = 'rows') **and self**.dproperties->exists(p:Property | p.name = 'columns') **else true endif**

13. **context** Display
    inv: if self.discomponents->forAll(c:Component | !c.oclIsTypeOf(List) **or** !c.oclIsTypeOf(TextPane) **or** !c.oclIsTypeOf(Button)) **then true else false endif**
14. **context** Container
    inv: if self.oclIsTypeOf(Container) **then self**.conproperties->exists(p:Property | p.name = 'title') **and self**.conproperties->exists(p:Property | p.name = 'message') **else true endif**
15. **context** Container
    inv: if self.oclIsTypeOf(Container) **and self**.layout = 'grid' **then self**.conproperties->exists(p:Property | p.name = 'title') **and self**.conproperties->exists(p:Property | p.name = 'message') **and self**.conproperties->exists(p:Property | p.name = 'rows') **and self**.conproperties->exists(p:Property | p.name = 'columns') **else true endif**
16. **context** Component
    inv: if self.oclIsTypeOf(Label) **or self**.oclIsTypeOf(RadioButton) **or self**.oclIsTypeOf(CheckBox) **or self**.oclIsTypeOf(SelectionGroup) **then self**.compproperties->exists(p:Property | p.name = 'text') **else true endif**
17. **context** ComboBox->Component
    inv: if self.oclIsTypeOf(ComboBox) **then self**.compproperties->exists(p:Property | p.name = 'stringArray') **else true endif**
18. **context** Button->Component
    inv: if self.oclIsTypeOf(Button) **then self**.compproperties->exists(p:Property | p.name = 'text') **and self**.compproperties->exists(p:Property | p.name = 'command') **else true endif**
19. **context** TextPane->Component
    inv: if self.oclIsTypeOf(TextPane) **then self**.compproperties->exists(p:Property | p.name = 'text') **and self**.compproperties->exists(p:Property | p.name = 'message') **else true endif**
20. **context** List->Component
    inv: if self.oclIsTypeOf(List) **then self**.compproperties->exists(p:Property | p.name = 'title') **and self**.compproperties->exists(p:Property | p.name = 'message') **else true endif**
21. **context** Message->Component
    inv: if self.oclIsTypeOf(Message) **then self**.compproperties->exists(p:Property | p.name = 'title') **and self**.compproperties->exists(p:Property | p.name = 'message') **else true endif**

Appendix I: PML code generation template – J2ME

```
«EXTENSION templates::J2mePresentation»
«DEFINE Root FOR presentation::DocumentRoot»
«EXPAND Display FOREACH displays»
«ENDDEFINE»
«REM»Definition of the Display model element.«ENDREM»
«DEFINE Display FOR presentation::Display»
«REM»Create the new GUI java file for the Display model element.«ENDREM»
«FILE this.toFirstUpper()+"Gui.java"»
«REM»Declare the package name for the class.«ENDREM»
package j2me.context.presentation;
«REM»Import the J2ME package used by the class.«ENDREM»
import javax.microedition.lcdui.*;

«REM»Create a new GUI class for the Display model element.«ENDREM»
public class «this.toFirstUpper()+"Gui"»{
«REM»Create a new Form component foreach Container model element.«ENDREM»
«FOREACH discontainers AS discon-»
        protected Form «discon.name»;
«ENDFOREACH»

«REM»Create accordingly a component foreach Display child model element.
A TextBox, a List and a Button can be associated to a Display in J2ME. «ENDREM»
«FOREACH discomponents AS discomp-»
«IF discomp.metaType.name.matches("presentation::TextPane")-»
        protected TextBox «discomp.name»;
«ELSEIF discomp.metaType.name.matches("presentation::List")-»
        protected List «discomp.name»;
«REM»Create the command associated to each Display Button component.«ENDREM»
«ELSEIF discomp.metaType.name.matches("presentation::Button")-»
        protected Command «discomp.name»;
«ENDIF»
«ENDFOREACH»

«REM»Create accordingly a component foreach Container child model element.
TextBox and List components cannot be added to a container in J2ME. «ENDREM»
«FOREACH discontainers.concomponents AS concomp-»
«IF concomp.metaType.name.matches("presentation::Label")-»
        protected StringItem «concomp.name»;
«ELSEIF concomp.metaType.name.matches("presentation::TextField")-»
        protected TextField «concomp.name»;
«ELSEIF concomp.metaType.name.matches("presentation::ComboBox")-»
        protected ChoiceGroup «concomp.name»;
«ELSEIF concomp.metaType.name.matches("presentation::CheckBox")-»
        protected StringItem «concomp.name»;
«ELSEIF concomp.metaType.name.matches("presentation::RadioButton")-»
        protected StringItem «concomp.name»;
«ELSEIF concomp.metaType.name.matches("presentation::Message")-»
        protected Alert «concomp.name»;
«ELSEIF concomp.metaType.name.matches("presentation::SelectionGroup")-»
        protected ChoiceGroup «concomp.name»;
«REM»Create the command associated to each container Button component.«ENDREM»
«ELSEIF concomp.metaType.name.matches("presentation::Button")-»
        protected Command «concomp.name»;
```

```
«ENDIF»
«ENDFOREACH»

«REM»Create a method to return the Form component for each container.«ENDREM»
«FOREACH discontainers AS discon-»
        public Form get«discon.toFirstUpper()»Form() {
«REM»Create the Form component. Add a ticker message to the Form.«ENDREM»
«IF discon.conproperties.exists(ele.name.matches("title"))-»
                «discon.name» = new Form(«discon.conproperties.
                        select(ele.name.contains("title")).value.first()»);
«ENDIF»
«IF discon.conproperties.exists(ele.name.matches("message"))-»
                «discon.name».setTicker(new Ticker(«discon.conproperties.
                        select(ele.name.contains("message")).value.first()»));
«ENDIF»
«REM»Create the different components associated to the Form.«ENDREM»
«FOREACH discon.concomponents AS concomp-»
«IF concomp.metaType.name.matches("presentation::Label")-»
                «concomp.name» = new StringItem(«concomp.compproperties.
                        select(ele.name.contains("text")).value.first()» , null);
«ELSEIF concomp.metaType.name.matches("presentation::TextField")-»
                «concomp.name» = new TextField("", null, 20, TextField.ANY);
«ELSEIF concomp.metaType.name.matches("presentation::ComboBox")-»
                String «concomp.name»Elements[] = {«concomp.compproperties.
                        select(ele.name.contains("stringArray")).value.first()»};
                «concomp.name» = new ChoiceGroup("", Choice.EXCLUSIVE,
                        «concomp.name»Elements , null);
«ELSEIF concomp.metaType.name.matches("presentation::CheckBox")-»
                «concomp.name» = new StringItem(«concomp.compproperties.
                        select(ele.name.contains("text")).value.first()» , null);
«ELSEIF concomp.metaType.name.matches("presentation::RadioButton")-»
                «concomp.name» = new StringItem(«concomp.compproperties.
                        select(ele.name.contains("text")).value.first()» , null);
«ELSEIF concomp.metaType.name.matches("presentation::Message")-»
                «concomp.name» = new Alert((«concomp.compproperties.
                        select(ele.name.contains("title")).value.first()»,
                                        «concomp.compproperties.
                        select(ele.name.contains("message")).value.first()», null,
                                                AlertType.ERROR);
«ELSEIF concomp.metaType.name.matches("presentation::SelectionGroup")-»
                «concomp.name» = new ChoiceGroup(«concomp.compproperties.
                        select(ele.name.contains("text")).value.first()», Choice.EXCLUSIVE);
«ELSEIF concomp.metaType.name.matches("presentation::Button")-»
                        «concomp.name» = new Command(«concomp.compproperties.
                        select(ele.name.contains("text")).value.first()»,
                        Command.«concomp.compproperties.
                        select(ele.name.contains("command")).value.first()», 0);
«ENDIF»
«ENDFOREACH»
                /** TODO starts
                */

                /** TODO ends
                */
«FOREACH discon.concomponents AS concomp-»
```

```
«REM»Add the commands associated to the Form.«ENDREM»
«IF concomp.metaType.name.matches("presentation::Button")-»
                «discon.name».addCommand(«concomp.name»);
«ENDIF»
«ENDFOREACH»
                return «discon.name»;
        }
«ENDFOREACH»

«REM»Create a new method to return the created Display List component.«ENDREM»
«FOREACH discomponents AS discomp-»
«IF discomp.metaType.name.matches("presentation::List")-»
        public List get«discomp.toFirstUpper()»List() {
«REM»Create the List component. Add a ticker message to the List.«ENDREM»
                «discomp.name» = new List(«discomp.compproperties.
                                select(ele.name.contains("title")).value.first()» , List.MULTIPLE);
«IF discomp.compproperties.exists(ele.name.matches("message"))-»
                «discomp.name».setTicker(new Ticker(«discomp.compproperties.
                                select(ele.name.contains("message")).value.first()»));
«ENDIF»
                /** TODO starts
                */

                /** TODO ends
                */
«REM»Add the commands associated to the List.«ENDREM»
«FOREACH discomponents AS listdiscomp-»
«IF listdiscomp.metaType.name.matches("presentation::Button")-»
                «discomp.name».addCommand(«listdiscomp.name»);
«ENDIF»
«ENDFOREACH»
                return «discomp.name»;
        } «ENDIF»
«ENDFOREACH»

«REM»Create a new method to return the created Display TextBox component.«ENDREM»
«FOREACH discomponents AS discomp-»
«IF discomp.metaType.name.matches("presentation::TextPane")-»
        public List get«discomp.toFirstUpper()»List() {
«REM»Create the TextBox component. Add a ticker message to the TextBox.«ENDREM»
                «discomp.name» = new TextBox(«discomp.compproperties.
                                select(ele.name.contains("text")).value.first()»,
                                        "This is the editable text area!", 256 , TextField.ANY);
«IF discomp.compproperties.exists(ele.name.matches("message"))-»
                «discomp.name».setTicker(new Ticker(«discomp.compproperties.
                                select(ele.name.contains("message")).value.first()»));
«ENDIF»
                /** TODO starts
                */

                /** TODO ends
                */
«REM»Add the commands associated to the TextBox.«ENDREM»
«FOREACH discomponents AS txtboxdiscomp-»
«IF txtboxdiscomp.metaType.name.matches("presentation::Button")-»
                «discomp.name».addCommand(«txtboxdiscomp.name»);
```

```
«ENDIF»
«ENDFOREACH»
                return «discomp.name»;
        } «ENDIF»
«ENDFOREACH»

«REM»Create the accessor methods associated to each container Button component.«ENDREM»
«FOREACH discontainers.concomponents AS concomp-»
«IF concomp.metaType.name.matches("presentation::Button")-»
        public void «concomp.setComponent()»(Command «concomp.name»){
                this.«concomp.name» = «concomp.name»;
        }
        public Command «concomp.getComponent()»(){
                return this.«concomp.name»;
        }
«ENDIF»
«ENDFOREACH»

«REM»Create the accessor methods associated to each display Button component.«ENDREM»
«FOREACH discomponents AS discomp-»
«IF discomp.metaType.name.matches("presentation::Button")-»
        public void «discomp.setComponent()»(Command «discomp.name»){
                this.«discomp.name» = «discomp.name»;
        }
        public Command «discomp.getComponent()»(){
                return this.«discomp.name»;
        }
«ENDIF»
«ENDFOREACH»
}
«ENDFILE»
«ENDDEFINE»
```

Appendix J: Petri Net Process Modelling Language constraints

1. **context** Place
   inv: not **self**.token.oclIsUndefined()
2. **context** Place
   inv: Place.allInstances()->forAll(p1, p2 | p1 <> p2 **implies** p1.id <> p2.id)
3. context Transition
   inv: Transition.allInstances()->forAll(t1, t2 | t1 <> t2 **implies** t1.id <> t2.id)
4. **context** Arc
   inv: Arc.allInstances()->forAll(a1, a2 | a1 <> a2 **implies** a1.id <> a2.id)
5. **context** Arc
   inv: Arc.allInstances()->forAll(a1, a2 | a1 <> a2 **and** a1.arctarget = a2.arcsource
   implies a2.arctarget <> a1.arcsource)
6. **context** Arc
   inv: Arc.allInstances()->forAll(a1, a2 | a1 <> a2 **and** a1.arcsource = a2.arcsource
   implies a1.arctarget <> a2.arctarget)
7. **context** Arc
   inv: self.arcsource.oclIsTypeOf(Place) <> **self**.arctarget.oclIsTypeOf(Place)
8. **context** Arc
   inv: self.arcsource.oclIsTypeOf(Transition)<>**self**.arctarget.oclIsTypeOf(Transition)
9. **context** Downlink
   inv: not **self**.dexpression.oclIsUndefined()
10. **context** Uplink
    inv: not **self**.uexpression.oclIsUndefined()
11. **context** Action
    inv: not **self**.aexpression.oclIsUndefined()
12. **context** ObjectNet
    inv: not **self**.oexpression.oclIsUndefined()
13. **context** Guard
    inv: not **self**.gexpression.oclIsUndefined()
14. **context** Inscription
    inv: not **self**.value.oclIsUndefined()
15. **context** DExpression
    inv: self.value.substring(1,5)='this:'
16. **context** DExpression
    inv: self.value.substring(**self**.value.size()-1,**self**.value.size())='()'

17. context DExpression
    inv: self.value.substring(1,4) = 'this' implies self.funType = 'void' or
    self.value.substring(1,self.funType.size()) = self.funType
18. context UExpression
    inv: self.value.substring(1,1)=':'
19. context UExpression
    inv: self.value.substring(self.value.size()-1,self.value.size())='()'
20. context AExpression
    inv: self.value.substring(1,6)='action'
21. context GExpression
    inv: self.value.substring(1,5)='guard'
22. context Downlink
    inv: Uplink.allInstances()->forAll(u | u.uexpression->exists(uexpr | uexpr.value =
    self.dexpression.value.substring(5,self.dexpression.value.size())))
23. context Token
    inv: self.value = '[]' or not self.value.toInteger().oclIsInvalid() or self.value = true or self.value =
    false or self.value.oclIsTypeOf(String)

Appendix K: PN-PML document generation template – PNML

```
«DEFINE Root FOR oopn::DocumentRoot»
«FILE value+".pnml"»
<pnml xmlns="RefNet">
<net id="«id»" type="«type»">
<name>
   <text>"«value»"</text>
</name>
«REM»Generate the PNML output for each Object Oriented Petri Net Place.«ENDREM»
«FOREACH places AS place-»
<place id= "«place.id»">
«IF place.token != null-»
        <initialMarking>
                <graphics>
                        <offset x="«place.token.offsetX»" y="«place.token.offsetY»"/>
                </graphics>
                        <text>«place.token.value»</text>
        </initialMarking>
«ENDIF»
        <graphics>
                <position x="«place.possitionX»" y="«place.possitionY»"/>
                <dimension x="«place.dimensionX»" y="«place.dimensionY»"/>
                <fill color="«place.fillTypeColor»"/>
                <line color="«place.lineTypeColor»"/>
        </graphics>
</place>
«ENDFOREACH»
«REM»Generate the PNML output for each Object Oriented Petri Net Basic Transition.«ENDREM»
«FOREACH basics AS basic-»
<transition id= "«basic.id»">
        <graphics>
                <position x="«basic.possitionX»" y="«basic.possitionY»"/>
                <dimension x="«basic.dimensionX»" y="«basic.dimensionY»"/>
                <fill color="«basic.fillTypeColor»"/>
                <line color="«basic.lineTypeColor»"/>
        </graphics>
</transition>
«ENDFOREACH»
«REM»Generate the PNML output for each Object Oriented Petri Net Downlink Transition.«ENDREM»
«FOREACH downlinks AS downlink-»
<transition id= "«downlink.id»">
«IF downlink.dexpression != null-»
<downlink>
        <graphics>
                <offset x= "«downlink.dexpression.offsetX»" y="«downlink.dexpression.offsetY»"/>
        </graphics>
                <text>«downlink.dexpression.value»</text>
        </downlink>
«ENDIF»
        <graphics>
                <position x="«downlink.possitionX»" y="«downlink.possitionY»"/>
                <dimension x="«downlink.dimensionX»" y="«downlink.dimensionY»"/>
                <fill color="«downlink.fillTypeColor»"/>
                <line color="«downlink.lineTypeColor»"/>
```

```
        </graphics>
</transition>
«ENDFOREACH»
«REM»Generate the PNML output for each Object Oriented Petri Net Uplink Transition.«ENDREM»
«FOREACH uplinks AS uplink-»
<transition id= "«uplink.id»">
«IF uplink.uexpression != null-»
<uplink>
        <graphics>
                <offset x= "«uplink.uexpression.offsetX»" y="«uplink.uexpression.offsetY»"/>
        </graphics>
                <text>«uplink.uexpression.value»</text>
        </uplink>
«ENDIF»
        <graphics>
                <position x="«uplink.possitionX»" y="«uplink.possitionY»"/>
                <dimension x="«uplink.dimensionX»" y="«uplink.dimensionY»"/>
                <fill color="«uplink.fillTypeColor»"/>
                <line color="«uplink.lineTypeColor»"/>
        </graphics>
</transition>
«ENDFOREACH»
«REM»Generate the PNML output for each Object Oriented Petri Net Action Transition.«ENDREM»
«FOREACH actions AS action-»
<transition id= "«action.id»">
«IF action.aexpression != null-»
<action>
        <graphics>
                <offset x= "«action.aexpression.offsetX»" y="«action.aexpression.offsetY»"/>
        </graphics>
                <text>«action.aexpression.value»</text>
        </action>
«ENDIF»
        <graphics>
                <position x="«action.possitionX»" y="«action.possitionY»"/>
                <dimension x="«action.dimensionX»" y="«action.dimensionY»"/>
                <fill color="«action.fillTypeColor»"/>
                <line color="«action.lineTypeColor»"/>
        </graphics>
</transition>
«ENDFOREACH»
«REM»Generate the PNML output for each Object Oriented Petri Net Object Transition.«ENDREM»
«FOREACH objects AS object-»
<transition id= "«object.id»">
«IF object.oexpression != null-»
<create>
        <graphics>
                <offset x= "«object.oexpression.offsetX»" y="«object.oexpression.offsetY»"/>
        </graphics>
                <text>«object.oexpression.value»</text>
</create>
«ENDIF»
        <graphics>
                <position x="«object.possitionX»" y="«object.possitionY»"/>
                <dimension x="«object.dimensionX»" y="«object.dimensionY»"/>
                <fill color="«object.fillTypeColor»"/>
```

```
                <line color="«object.lineTypeColor»"/>
        </graphics>
</transition>
«FILE object.oexpression.value.split("new").last().trim()+".pnml"»
<pnml xmlns="RefNet">
        <net id="«object.id»" type="«type»">
        <name>
                <text>"«object.oexpression.value.split("new").last().trim()»"</text>
        </name>
        </net>
</pnml>
«ENDFILE»
«ENDFOREACH»
«REM»Generate the PNML output for each Object Oriented Petri Net Arc.«ENDREM»
«FOREACH arcs AS arc-»
<arc id= "«arc.id»" source="«arc.source»" target="«arc.target-»">
«IF arc.attribute != null-»
        <inscription>
                <graphics>
                        <offset x= "«arc.attribute.offsetX»" y="«arc.attribute.offsetY»"/>
                </graphics>
                <text>«arc.attribute.value»</text>
        </inscription>«ENDIF»
        <type>
                <text>ordinary</text>
        </type>
        <graphics>
                <line color="«arc.lineTypeColor»" style="«arc.lineTypeStyle»"/>
        </graphics>
</arc>
«ENDFOREACH»
</net>
</pnml>
«ENDFILE»
«ENDDEFINE»
```

Appendix L: Context souce generated implementation – J2ME

```
package j2me.context.sources;

import java.util.Vector;
import javax.microedition.rms.RecordStoreException;

import j2me.context.management.RecordStoreManagement;
import j2me.context.helpers.StringRenderer;

import j2me.context.information.PersonSite;
import j2me.context.enumerations.Section;

public class Sites {

	private static final String atomic_context = "PersonSite";

	private static final String multiplicity = "1...*";
	private static final String multiplicityType = "Collection";
	private static final String persistence = "Fixed";
	private static final String permission = "Unrestricted";

	private static final String name = "name";
	private static final String room = "room";
	private static final String description = "description";

	private RecordStoreManagement rsm = null;
	private PersonSite ps = null;

	private Section s = null;

	public void setSites(Vector sites) {
		int i = 0;
		int j = 0;

		try {
			if (rsm == null)
				rsm = new RecordStoreManagement(atomic_context);

			while (i < sites.size()) {
				rsm.put(name + j, sites.elementAt(i).toString());
				i++;
				rsm.put(room + j, sites.elementAt(i).toString());
				i++;
				rsm.put(description + j, sites.elementAt(i).toString());
				i++;

				j++;
			}
			rsm.save();
		} catch (RecordStoreException rse) {
			rse.printStackTrace();
		}
	}
```

```
	public Vector getSites() {
		int i = 0;
		int j = 0;
		Vector sites = new Vector();
		try {
			if (rsm == null)
				rsm = new RecordStoreManagement(atomic_context);
			int records = rsm.getSize();

			while (i < records - 1) {
				String[] temp = null;
				StringRenderer sr = new StringRenderer();
				ps = new PersonSite();

				s = new Section();

				s.setState(rsm.get(name + j));
				ps.setName(s);
				i++;

				ps.setRoom(rsm.get(room + j));
				i++;

				ps.setDescription(rsm.get(description + j));
				i++;

				sites.insertElementAt(ps, j);
				j++;
			}
		} catch (RecordStoreException rse) {
			rse.printStackTrace();
		}
		return sites;
	}
	public static String getMultiplicity() {
		return multiplicity;
	}
	public static String getMultiplicityType() {
		return multiplicityType;
	}
	public static String getPersistence() {
		return persistence;
	}
	public static String getPermission() {
		return permission;
	}
	public static String getName() {
		return name;
	}
	public static String getRoom() {
		return room;
	}
	public static String getDescription() {
		return description;
	}
}
```

Appendix M: Context Receiver generated implementation – J2ME

```
package j2me.context.receivers;

import java.util.Enumeration;
import java.util.Vector;

import j2me.context.base.Context;
import j2me.context.events.ContextEvent;
import j2me.context.listeners.ContextListener;

public class PersonReceiver {
	private Object _context_object;
	private Context _context = null;
	private Vector _listeners = new Vector();

	public synchronized void receivePersonContext(Context _context, Object _context_object) {

		this._context = _context;
		this._context_object = _context_object;

		if (_context == Context. Person_Identity) {
			_context = Context.Person_Identity;
			_fireContextEvent();
		}
		else if (_context == Context.Person_Location) {
			_context = Context.Person_Location;
			_fireContextEvent();
		}
		else if (_context == Context.Person_Preference) {
			_context = Context.Person_Preference;
			_fireContextEvent();
		}
		else if (_context == Context.Person_Activity) {
			_context = Context.Person_Activity;
			_fireContextEvent();
		}
		else if (_context == Context.Person_Exhibition) {
			_context = Context.Person_Exhibition;
			_fireContextEvent();
		}
		else if (_context == Context.Person_Site) {
			_context = Context.Person_Site;
			_fireContextEvent();
		}
	}

	public synchronized void addContextListener(ContextListener cl) {
		_listeners.addElement(cl);
	}

	public synchronized void removeContextListener(ContextListener cl) {
		_listeners.removeElement(cl);
	}
```

```
        private synchronized void _fireContextEvent() {
                ContextEvent context_event = new ContextEvent(this, _context);
                Enumeration listeners = _listeners.elements();
                while (listeners.hasMoreElements()) {
                        ((ContextListener) listeners.nextElement()).contextAltered(
                                        context_event, _context_object);
                }
        }
}
```

Appendix N: COCOMO II Effort Multipliers and Scale Factors

Table 1: COCOMO II Parameters and Rating Scale

CATEGORY/ Parameters	Recommendations/Rating Scale					
LINES OF CODE						
Size	Enter your size estimates from Software Estimation Step #3 for each low-level element. Or if using analogy to historical data based on physical SLOC, convert physical SLOC to logical SLOC. In general, estimators tend to be overly optimistic on the amount of code that can be inherited from projects. Therefore, it is better to underestimate the size of inherited/reused software.					
% Design Modified	If there is heritage, enter % of inherited design to be modified.					
% Code Modified	If there is heritage, enter % of the inherited or reused code that will be modified.					
% Integration Modified	If there is heritage, enter % of the effort needed for integrating and testing the adapted software as compared to the normal amount of integration and test effort for software of comparable size.					
% Code breakage	Enter % of code thrown away due to requirements evolution and volatility.					
Post Architecture Effort Multipliers	**Very Low**	**Low**	**Nominal**	**High**	**Very High**	**Extra High**
PRODUCT ATTRIBUTES						
RELY Required Software Reliability	Effect of SW failure = slight inconvenience (0.82)	Effect of SW failure = low, easily recoverable losses (0.92)	Effect of SW failure = moderate, easily recoverable losses (1.00)	Effect of SW failure = high financial loss (1.10)	Effect of SW failure = risk to human life/public safety requirements (1.26)	
DATA Database Development Size		Testing DB Bytes/Program SLOC < 10 (0.90)	10 ≤ D/P < 100 (1.00)	100 ≤ D/P < 1000 (1.14)	D/P ≥ 1000 (1.28)	
CPLX Product Complexity	See Table 2					
DOCU Documentation Match to Life-Cycle Needs	Many life-cycle needs uncovered (0.81)	Some life-cycle needs uncovered (0.91)	Right-sized to life-cycle needs (1.00)	Excessive for life-cycle needs (1.11)	Very excessive for life-cycle needs (1.23)	
RUSE Developed for Reusability		None (0.95)	Across project (1.00)	Across program (1.07)	Across product line (1.15)	Across multiple product lines (1.24)
PLATFORM ATTRIBUTES						
TIME Execution Time Constraint			≤50% use of available execution time (1.00)	70% use of available execution time (1.11)	85% use of available execution time (1.29)	95% use of available execution time (1.63)
STOR Main Storage Constraint			≤50% use of available storage (1.00)	70% use of available storage (1.05)	85% use of available storage (1.17)	95% use of available storage (1.46)
PVOL Platform Volatility		Major change every 12 mo.; Minor change every 1 mo. (0.87)	Major change every 6 mo.; Minor change every 2 wk. (1.00)	Major change every 2 mo.; Minor change every 1 wk. (1.15)	Major change every 2 wk.; Minor change every 2 days (1.30)	
PERSONNEL ATTRIBUTES The personnel attributes are the most misused of the all the effort multipliers. If you do not know who you will be hiring, then assume Nominal, which would represent average capability and experience.						
ACAP Analyst Capability	15th percentile (1.42)	35th percentile (1.19)	55th percentile (1.00)	75th percentile (0.85)	90th percentile (0.71)	
PCAP Programmer Capability	15th percentile (1.34)	35th percentile (1.15)	55th percentile (1.00)	75th percentile (0.88)	90th percentile (0.76)	
PCON Personnel Continuity	Annual personnel turnover: 48%/year (1.29)	24%/year (1.12)	12%/year (1.00)	6%/year (0.90)	3%/year (0.81)	

APEX Applications Experience	≤2 months (1.22)	6 months (1.10)	1 year (1.00)	3 years (0.88)	6 years (0.81)	
PLEX Platform Experience	≤2 months (1.19)	6 months (1.09)	1 year (1.00)	3 years (0.91)	6 years (0.85)	
LTEX Language and Tool Experience	≤2 months (1.20)	6 months (1.09)	1 year (1.00)	3 years (0.91)	6 years (0.84)	
PROJECT ATTRIBUTES						
TOOL Use of Software Tools	Edit, code, debug (1.17)	Simple, frontend, backend, CASE, little integration (1.09)	Basic life-cycle tools, moderately integrated (1.00)	Strong, mature life-cycle tools, moderately integrated (0.90)	Strong, mature, proactive life-cycle tools, well integrated with processes, methods, reuse (0.78)	
SITE Multisite Development	Collocation: international; Communications: some phone, mail (1.22)	Collocation: multicity and multicompany; Communications: individual phone, fax (1.09)	Collocation: multicity or multicompany; Communications: narrow band email (1.00)	Collocation: same city or metro area; Communications: wideband electronic communication (0.96)	Collocation: same building or complex; Communications: wideband electronic communication, occasional video conf. (0.86)	Collocation: Fully collocated; Communications: Interactive multimedia (0.80)
SCED Required Development Schedule	75% of nominal (1.43)	85% of nominal (1.14)	100% of nominal (1.00)	130% of nominal (1.00)	160% of nominal (1.00)	
SCALE FACTORS	**Very Low**	**Low**	**Nominal**	**High**	**Very High**	**Extra High**
PREC Precedentedness	thoroughly unprecedented (6.20)	largely unprecedented (4.96)	Somewhat unprecedented (3.72)	generally familiar (2.48)	largely familiar (1.24)	thoroughly familiar (0.00)
FLEX Development Flexibility	Rigorous (5.07)	occasional relaxation (4.05)	Some relaxation (3.04)	General conformity (2.03)	Some conformity (1.01)	general goals (0.00)
RESL Architecture/Risk Resolution	little (20%) (7.07)	some (40%) (5.65)	often (60%) (4.24)	Generally (75%) (2.83)	mostly (90%) (1.41)	full (100%) (0.00)
TEAM	very difficult interactions (5.48)	some difficult interactions (4.38)	Basically cooperative interactions (3.29)	Largely cooperative (2.19)	Highly cooperative (1.10)	Seamless interactions (0.00)
PMAT Process Maturity	CMM Level 1 (Lower half) (7.80)	CMM Level 1 (Upper half) (6.24)	CMM Level 2 (4.68)	CMM Level 3 (3.12)	CMM Level 4 (1.56)	CMM Level 5 (0.00)

Table 2: COCOMO II Complexity Table

	Control Operations	Computational Operations	Device-dependent Operations	Data Management Operations	User Interface Management Operations
Very Low (0.73)	Straight-line code with a few non-nested structured programming operators: DOs, CASEs, IF-THEN-ELSEs. Simple module composition via procedure calls or simple scripts.	Evaluation of simple expressions: e.g., A = B + C * (D –E)	Simple read, write statements with simple formats.	Simple arrays in main memory. Simple COTS-DB queries, updates.	Simple input forms, report generators.
Low (0.87)	Straightforward nesting of structured programming operators. Mostly simple predicates	Evaluation of moderately level expressions: e.g., D = SQRT (B*2-4. * A * C)	No cognizance needed of particular processor or I/O device characteristics. I/O done at GET/PUT level.	Single file subsetting with no data structure changes, no edits, no intermediate files. Moderately complex COTS-DB queries, updates.	Use of simple graphic user interface (GUI) builders.
Nominal (1.00)	Mostly simple nesting. Some inter-module control. Decision tables. Simple callbacks or message passing, including middleware-supported distributed processing	Use of standard math and statistical routines. Basic matrix/vector operations.	I/O processing includes device selection, status checking, and error processing.	Multi-file input and single file output. Simple structural changes, simple edits. Complex COTS-DB queries, updates.	Simple use of widget set.
High (1.17)	Highly nested structured programming operators with many compound predicates. Queue and stack control. Homogeneous, distributed processing. Single processor soft real-time control.	Basic numerical analysis: multivariate interpolation, ordinary differential equations. Basic truncation, round off concerns.	Operations at physical I/O level (physical storage address translations; seeks, reads, etc.). Optimized I/O overlaps.	Simple triggers activated by data stream contents. Complex data restructuring.	Widget set development and extension. Simple voice I/O multimedia.
Very High (1.34)	Reentrant and recursive coding. Fixed-priority interrupt handling. Tasks synchronization, complex callbacks, heterogeneous distributed processing. Single-processor hard real-time control.	Difficult but structured numerical analysis: near-singular matrix equations, partial differential equations. Simple parallelization.	Routines for interrupt diagnosis, servicing, masking. Communication line handling. Performance-intensive embedded systems.	Distributed database coordination. Complex triggers. Search optimization.	Moderately complex 2D/3D, dynamic graphics, multimedia.
Extra High (1.74)	Multiple resource scheduling with dynamically changing priorities. Microcode-level control. Distributed hard real-time control.	Difficult and unstructured numerical analysis: highly accurate analysis of noisy, stochastic data. Complex parallelization.	Device timing-dependent coding, micro-programmed operations. Performance-critical embedded systems.	Highly coupled, dynamic relational and object structures. Natural language data management.	Complex multimedia, virtual reality.

Table 3: Tool Coverage (TCOV), Tool Integration (TINT) and Tool Maturity (TMAT) Rating Scales

Rating	TCOV
Very Low (1.17)	Text-Based Editor, Basic 3GL Compiler, Basic Library Aids, Basic Text-Based Debugger, Basic Linker
Low (1.09)	Graphical Interactive Editor, Simple Design Language, Simple Programming Support Library, Simple Metrics/Analysis Tool
Nominal (1.00)	Local Syntax Checking Editor, Standard Template Support Document Generator, Simple Design Tools, Simple Standalone Configuration Management Tool, Standard Data Transformation Tool, Standard Support Metrics Aids with Repository, Simple Repository, Basic Test Case Analyser
High (0.9)	Local Semantics Checking Editor, Automatic Document Generator, Requirement Specification Aids and Analyser, Extended Design Tools, Automatic Code Generator from Detailed Design, Centralised Configuration Management Tool, Process Management Aids, Partially Associative Repository (Simple Data Model Support), Test Case Analyser with Testing Process Manager, Test Oracle Support, Extended Reengineering and Reverse Engineering Tools
Very High (0.78)	Global Semantics Checking Editor, Tailorable Automatic Document Generator, Requirement Specification Aids and Analyser with Tracking Capability, Extended Design Tools with Model Verifier, Code Generator with Basic Round-Trip Capability, Extended Static Analysis Tool, Basic Associative, Active Repository (Complex Data Model Support), Heterogeneous N/W Support Distributed Configuration Management Tool, Test Case Analyser with Testing Process Manager, Test Oracle Support, Extended Reengineering and Reverse Engineering Tools
Extra High (N/A)	Groupware Systems, Distributed Asynchronous Requirement Negotiation and Trade-off Tools, Code Generator with Extended Round Trip Capability, Extended Associative, Active Repository, Spec-based Static and Dynamic Analyzers, Pro-active Project Decision Assistance

Rating	TINT
Very Low (1.17)	Individual File Formats for Tools (No Conversion Aid), No Activation Control for Other Tools, Different User Interface for Each Tool, Fundamental Incompatibilities Among Process Assumptions and Object Semantics
Low (1.09)	Various File Formats for Each Tool (File Conversion Aids), Message Broadcasting to Tools, Some Standartised User Interfaces Among Tools, Difficult Incompatibilities Among Process Assumptions and Object Semantics

Nominal (1.00)	Shared Repository, Point-to-Point Message Passing, Customisable User Interface Support, Largely Workable Incompatibilities Among Process Assumptions and Object Semantics
High (0.9)	Shared Repository, Point-to-Point Message Passing, Customisable User Interface Support, Largely Workable Incompatibilities Among Process Assumptions and Object Semantics
Very High (0.78)	Highly Associative Repository, Point-to-Point Message Passing Using Reference for Parameters, Some Level of Different User Interface, Largely Consistent Among Process Assumptions and Object Semantics
Extra High (N/A)	Distributed-Associative Repository, Extended Point-to-Point Message Passing for Tool Activation, Complete Set of User Interface for Different Level of Users, Fully Consistent Among Process Assumptions and Object Semantics

Rating	TMAT
Very Low (1.17)	Version in Pre-Release Beta-Test, Simple Documentation and Help
Low (1.09)	Version on Market/Available Less than 6 Months, Up-Dated Documentation, Help Available
Nominal (1.00)	Version on Market/Available Between 6 Months and 1 Year, Online Help, Tutorial Available
High (0.9)	Version on Market/Available Between 1 and 2 Years, Online User Support Group
Very High (0.78)	Version on Market/Available Between 2 and 3 Years, On-Site Technical User Support Group
Extra High (N/A)	Version on Market/Available More Than 3 Years, Expert On-Site Technical User Support Group