بسم الله الرحمن الرحيم

Sudan University of Science and Technology
College of Graduate Studies

Declaration

I, the signing here-under, declare that I'm the sole author of the Ph.D. thesis entitled...
Methods for Mapping State Machines in
Model Driven Architecture
which is an original intellectual work. Willingly, I assign the copy-right of this work to the College of Graduate Studies (CGS), Sudan Universit of Science & Technology (SUST). Accordingly, SUST has all the rights to pub طرق k for scientific purposes.

Candidate's nam Rihabab Eltayeb Ahmed

Candidate's signature: Rihab Date: 1/9/2015

إقرار

أنا الموقع أدناه أقر بأنني المؤلف الوحيد لرسالة الدكتوراه المعنونة ....
طرق لتحويل آلات الحالات في المعماريه المبنيه
على النماذج

وهى منتج فكري أصيل . وباختياري أعطى حقوق طبع ونشر هذا العمل لكلية الدراسات العليا جامعه السودان للعلوم والتكنولوجيا ،عليه يحق للجامعه نشر هذا العمل للأغراض العلمية .

اسم الدارس : رحاب الطيب أحمد

توقيع الدارس : Rihab التاريخ : 1/9/2015

Sudan University of Science and Technology

COLLEGE OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

METHODS FOR MAPPING STATE MACHINES IN MODEL DRIVEN ARCITECTURE

طرق لتحويل آلات الحالات في المعمارية المبنية على النماذج

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy of Science

by
Rihab Eltayeb Ahmed

Supervised by
Prof. Robert M Colomb

Co-Supervised by
Dr. Abdelgafar Hamid Ahmed

August 2015

سورة 17 رقم

الإسراء

111 واياتها

بسم الله الرحمن الرحيم

وَيَسْأَلُونَكَ عَنِ الرُّوحِ قُلِ الرُّوحُ مِنْ أَمْرِ رَبِّي وَمَا أُوتِيتُمْ مِنَ

الْعِلْمِ إِلَّا قَلِيلًا (85)

وَقُلِ الْحَمْدُ لِلَّهِ الَّذِي لَمْ يَتَّخِذْ وَلَدًا وَلَمْ يَكُنْ لَهُ شَرِيكٌ فِي الْمُلْكِ وَلَمْ

DEDICATION

To

My

Family

I'm usually not at a loss for words, but this time I am.

ACKNOWLEDGEMENTS

I would never have been able to finish my dissertation without the will and mercy of ALLAH (swt). Am indebted to the support, encouragement and profound understanding of my lab members, colleagues , friends and family.

I would like to express my deepest gratitude to my supervisor, Prof. Robert M. Colomb, for his excellent guidance, caring and patience. Continents , oceans and my humble thinking capabilities doesn't prevent him from kindly giving hints, clues and detailed email messages. With sincerity and high sense he apologized one time of not being expressive enough, but really all his messages were self explanatory and all the time satisfied my hungry mind.

I would like to thank my co-supervisor Dr. Abdelgafar Hamid Ahmed, who let me experience the research and practical issues beyond the textbooks, patiently listened to us in the Saturday meetings and give his best suggestions.

Special thanks goes towards people who remain in my memory forever, trusted me, always there cheering me up and stood by me through the good times and bad.

Rihab Eltayeb Ahmed

Abstract

Model Driven Architecture (MDA) is an initiative of the Object Management Group that uses models as the first class artifacts in the software development process. MDA aims at deriving values from models that capture the system structural and behavioral aspects. One of the values of models is to derive an implementation from models in an automated fashion. Automation enables rapid response to changes, increases the efficency of software development and decreses its cost. The derivation involves a Platform Independent Model PIM, a targeted Platform Specific Model PSM and mapping transformation rules between the PIM and the PSM. The PIM washes away the technical details and focuses on the business logic of the system where as the PSM contains the technical detailed information. The main challenge is the transformation from PIM to PSM (different models). In practice the transformation process from PIM to PSM might be a lot more complex and challenging. Between the models, gaps can exist because of the difference in the abstraction layers exhibited in the models. The gaps may not be small enough to perform a direct transformation. Moreover there is still difficulties when the application behavior is addressed in MDA. In most cases, behavioral models are used for other purposes like documentation rather than complementing the structural models to facilitate automatic software generation. The problem is the lack of mechanisms for mapping behavior models from an abstraction level to another. This research study proposes a method for mapping UML behavior models from PIM to PSM. Both the PIM and the PSM are augmented with UML class model and state machine behavioral model. A transformation framework, taxonomy and guidelines were identified beside the suitable languages and tools based on MDA best practices and standards. The PIM models for two application domains were built using MDA compliant modeling tool. The PSM model for a standard messaging oriented platform was developed and used along with the proposed transformation framework to map the PIM models to the PSM. The work is completed by including the PSM to code translation. The resulted artifacts were transferred to an execution environment to run the program. The proposed method achieved an acceptable degree of automation of the software application development using the MDA approach.

المستخلص

المعمارية المبنية على النماذج (Model Driven Architecture (MDA هي مبادرة من مجموعة Object Management Group OMG وفيها يتم استخدام النماذج كأساس لا غنى عنه في عملية تطوير البرمجيات. تهدف MDA الى استخلاص اكبر فائدة من النماذج التي يتم عبرها تمثيل كل جوانب النظم الهيكلية والسلوكية. واحدة من طرق استخدام النماذج واستخلاص الفائدة منها هو اتمتة عملية تنفيذ البرامج اعتماداً على النماذج. الاتمتة تساعد على الاستجابة السريعة للتغيرات وتزيد من كفاءة البرمجيات بالاضافة الى تقليل تكلفة تطويرها. تحتوي عملية اتمتة التنفيذ بناءاً على النماذج على نموذج يعكس متطلبات النظام من وجهة نظر المستخدم ويتم فيه نمذجة تلك المتطلبات بعيداً عن

اية تفاصيل تقنيـة او تنفيذيـة ويسمى هـذا النمـوذج بـالنموذج المسـتقل Platform Independent Model (PIM) . يتم بناء نموذج آخر (او في الوضع المثالي اختياره من بين بقيـة النمـاذج المشـابهة المتفق على جودتها مسبقا) يحتـوي التفاصـيل التقنيـة الـتي تسمح بتنفيـذ التطبيقـات المعينـة فـي بيئـات محددة ويسمى النموذج المرتبط Platform Specific Model PSM. يتم ربـط النمـوذج المسـتقل مع النموذج المرتبط عن طريق عملية تحويل يتم فيها تحديد العلاقات بين النموذجين والقواعد الـتي علـى اساسها سيتم التحويل بينهما. تعتبر عملية التحويل احدى التحديات في مجال MDA وخاصة التحويـل من نموذج مستقل الى نموذج مرتبط لوجود اختلافات في تجريد النماذج من مستوى الى اخر ممـا يخلـق فجوة قد لا تكون صغيرة بما يكفي لاجراء تحويل مباشر بين النمـاذج. وعلاوة علـى ذلـك لا تـزال هنـاك صعوبات في MDA عند تناول سلوك البرمجيـات والـذي يصـف وظيفـة البرامـج واسـتجابتها للمؤثرات وتفاعلها مع البيئة المحيطة بها. في معظم الحالات، يتم استخدام النماذج السلوكية لأغـراض أخـرى مثـل التوثيق للبرنامج بدلاً عن تكميل صورة النظام مع النماذج التي توصـف هيكليته مما يمنـع عملية الاتمتة الكاملة التي تنتج النظام بصورته التنفيذية المطلوبة. المشكلة تكمـن فـي عـدم وجـود آليـات توضـح كيـف يمكن أن تتم عملية تحويل النماذج السلوكية من مستوى الى آخـر حيـث لا يوجـد توصـيف للعمليـة حـتى في الوثائق المعتمدة من MDA. نقترح في هذا البحث طريقة لتحويل نماذج السلوك المعدة بلغة النمذجة UML وذلك من مستوى النموذج المستقل الى مستوى النموذج المرتبط بحيث يتم تـدعيم كلا النمـوذجين بالنماذج الهيكلية والسلوكية معاً. تم تحديد اطار للحـل وتصـنيفات لانـواع التحـويلات الممكنـة بالاضـافة الى المبادئ التي يمكن الاستناد عليها في عملية التحويل. كما تـم ايضـاً تحديـد اللغـات المناسـبة والادوات

التي تتماشى مع أفضل الممارسات والمعايير في MDA. لاتمام عملية التحويل تم بناء نماذج مستقلة ونماذج مرتبطة من مجالات برمجية مختارة بعناية وتنفيذ خطوات عملية التحويل المقترحة عليها. كما تمت ترجمة النماذج الى صورة تنفيذية ليسهل تنفيذ البرنامج الموصف في البداية بالنموذج المستقل في صورة تمثل ما حدده نهج MDA باعتماد النماذج في كل عمليات انتاج البرمجيات. حققت الطريقة المقترحة درجة مقبولة من أتمتة عملية تطوير تطبيقات البرمجيات باستخدام نهج MDA.

Table of Contents

LIST OF TABLES

LIST OF FIGURES

List of Symbols /Abbreviations

MDA	Model-Driven Architecture
ERP	Enterprise Resource Planning system
OMG	Object Management Group
UML	Unified Modeling Language
CORBA	Common Object Request Broker
EJB	Enterprise Java Bean
PIM	Platform Independent Model
PSM	Platform Specific Model
API	Application Programming Interface
CIM	Computational Independent Model
MOF	Meta Object facility
BPMN	Business Process Modeling and Notation
OWL	Web Ontology Language
SQL	Structured Query Language
CWM	Common Warehouse Metamodel
OCL	Object Constraint Language
ATM	Automated Teller Machine
QVT	Query View Transform
MDD	Model Driven Development

DTD	Document Type Definition
XSLT	Extensible Style Sheet Language Transformation
MOF	Meta Object Facility
OO	Object Oriented
XMI	XML Metadata Interchange
IDE	Integrated Development Environment
EMF	Eclipse Modeling Framework
EMOF	Essential Meta-Object Facility
JMS	Java Messaging Service
SCXML	State Chart extensible Markup Language
W3C	World Wide Web Consortiums
CCXML	Call Control XML
MOM	Message Oriented Middleware

Introduction

Overview

This chapter gives a brief introduction to the concepts that will be referenced later in the thesis. Section 1.1 first gives a brief introduction to the Model-Driven Architecture (MDA) software development approach. Section 1.2 of this chapter presents the reasons motivating the research work to be done. It is followed by Section 1.3 presenting the problems faced in this domain.

The following section, Section 1.4, explains some research questions that will be answered through this thesis. Section 1.6 presents the main contribution of the research work, taking account of the objectives presented in Section 1.5. Section 1.7 presents the scope and context in which this research work has been developed. Finally Section 1.8 presents the outline of the whole thesis, describing each chapter in brief.

MDA in Brief

Software development is a complex process. The information technology market is faced by many challenges among them is the effect of technology "platforms" change (languages, operating-systems, interoperability solutions, architecture frameworks etc.). This is an issue for software companies, developers and even customers. Software companies are forced to follow technology change or be abandoned by their customers. Customers may follow technology change to find new and interesting things. Software developers may fear being left behind by technological changes. The result is that software market migrates to a new technology frequently, regardless of whether the technological change is beneficial or not (Flater 2002).

Business process modeling plays an important role in software intensive information systems. It become more vital specially when the information systems grow in scale and complexity. Nowadays many of the large scale and complex information systems are driven by models. Models are representations of reality. A model of a system in MDA is a description or specification of that system and its environment. Business process modeling is the basis of process centric systems such as Enterprise Resource Planning system (ERP). A software system like an ERP is not just an artifact. Moreover The enterprise focus of such a system made it hard to rely on conventional methods only.

Model Driven Architecture (MDA) (OMG 2014a) is a new development method that represents a positive effort from the Object Management Group (OMG) to overcome software development problems including but not limited to technology change. The philosophy of MDA regarding technology change is the separation of concerns. That is to capture the most valuable and reusable part of the system– conceptual design- and washing away technical details. The conceptual design of the system then can be realized on new technologies cheaply because the process is going only to add the “new technology” details. The same conceptual design of the system can be realized into a technology or another hence no more fear from technology change.

The conceptual design of the system in MDA is captured as a model that describes the structures and/or behaviors of the intended software application. A modeling language is used to create these models such as the Unified Modeling Language (UML). As the emphasis is increasingly shifted towards models; the role of standard models increases. Two applications may be implemented in different technologies but conforming to the same standard model. This will enable them to share a common understanding of the system. Conformance to well- designed standard models in software development using MDA will in turn increase the chances for interoperability. This was another part of the motivation for the MDA.

Based on the MDA document (OMG 2003), the following are the key terms in MDA. A **Model** is a formal specification of the function, structure and/or behavior of a system. A **Platform** represents the technological and engineering details that are irrelevant to the fundamental functionality of a software component. Example platforms are Common Object Request Broker (CORBA), Enterprise Java Bean (EJB), and Microsoft Dot Net Framework. In MDA, structures and behaviors based on the business functions are abstracted and modeled in a Platform Independent Model (**PIM**). The implementation specific structures and behaviors are modeled in a Platform Specific Model (**PSM**). The PIM is then realized into the PSM through a **transformation** process to generate the software. A **mapping** provides specification for transforming a PIM to a PSM in a specific platform.

Model transformation is the process of converting a source model into a target model according to some transformation specifications. Transformation specifications are the rules that specify how to transform the source to the target. A distinction can be made between horizontal and vertical transformation. Horizontal transformation is a transformation where the source and target models reside in the same abstraction level. The aim of a horizontal transformation may be an optimization to improve certain quality attributes of the system (performance) or a simplification and normalization to decrease the syntactic complexity. Vertical transformation is a transformation where the source and target models reside at a different abstraction level. Refinement is an example of a vertical transformation in which the higher level, more abstract source model (e.g. design) is gradually refined into a lower level, more concrete model (e.g. a model of a Java program) (Mens et al. 2005).

Figure 1MDA Model Transformation

Motivation

Software development using MDA is promising. One of the promises of MDA is the automatic generation of executables. The software development is model centric and no longer code centric. We create a model of the application problem and select a technology that performs the class of tasks. Then we establish a mapping between the model and the technology platform (middleware or API). The mapping will enable turning the model of the application problem into a working system automatically without (or with minimal) programming.

Raising the abstraction level of the system design into models leads to reusability which is another gain using MDA. The model represents the business in a technology-independent fashion. New technology means just a new transformation for the same (reusable) model. So the same system spans several platforms. Through modeling and transformation in MDA the productivity of system developments increases, the development time for new systems is reduced and time to market decreases. The reason is that the whole development process is simplified and the core asset (the model) is reusable. The system model serves the purpose of the documentation and is the enabler of the code generated for the system. The system documentation is consistent with the system itself. When changes occurred they are also applied to the models, resulting in consistent and more efficient change management.

Behavior execution -automatically verifying models on a computer- at PIM level is a remarkable feature to verify high level models against the requirements and to provide conformance for implementation at PSM and code level. One of the promises of MDA is the automatic generation of software based on models. But the static structural features of the modeled system are not always enough to generate a completely full automated application unless complemented by behavioral features of the system. One can conclude that behavior modeling is important to support MDA goals.

Significance of the study

The study will help in bridging the gap between the design and development phase and will support the developers in the software development process using MDA.

Since MDA is considered a young discipline, standard models and Meta models are not yet created for a wide range of application domains.

Problem background

The main idea of MDA is to make models the main driver of software development process. To build an application in MDA, the process starts with defining a Computational Independent Model (**CIM**) aka domain model. An enterprise architect will transform the CIM into a PIM by adding architectural information. A CIM represents the system within the environment. To complete the build process, the transformed PIM has to target a platform. A platform specialist will carry out the transformation from the PIM to the PSM. The resulted PSM is considered an implementation when it provides all needed information, structures and behaviors that construct a system and make it up and running.

UML is a formal modeling language that is standardized by the Object Management Group (OMG) and is the recommended language to build various types of models in MDA. UML provides diagrams to model structural and behavioral aspects of a system. Structure diagrams in UML show the static structure of the system. The class diagram is widely used to describe static structures while other diagrams such as object, component and deployment diagrams are also provided. A variety of mechanisms to specify behaviors are supported in UML such as automata (state machine), Petri-net like graphs (activity), informal description (use case) and sequence of events (interaction) (OMG 2011). These different behavior specification mechanisms differ in their expressive power and domain of applicability so the choice of one of them depends on convenience and purpose.

Almost every application contains functionality and behavior besides possessing a structure. Any **behavior** is the direct consequence of the action of at least one object called the host object (OMG 2011). A behavior has access to the structural features of its host object. Behavior modeling is used to visualize, specify and construct various dynamic aspects such as modeling a flow of control, an element behavior, a workflow or an operation. Fund transfer between two bank accounts is an operation example. The formal definition of the behavior of this operation remains the same despite whether it was implemented in different platforms by a CORBA object, an Enterprise Java Beans, or a SOAP operation. Modeling the operation behavior is important since modeling expresses the operation in a higher level of abstraction that in turn allows for reusing the operation specifications between different languages, frameworks and execution environments.

Although MDA acknowledge richer modeling, reusability, reliability and automation of software generation, they are still far from defining a real engineering approach to tackle the transformation process, not even the MDA first guide published by The OMG (Richter & Conti 2004).

Some transformations can be considered heavy and challenging such as code generation, compilation and parsing. Some can be considered as light such as changing the internal software architecture to provide modularity while maintaining the same software behavior. Heavy transformations need certain set of tools and techniques. Certain aspects of the source model has to be preserved in the transformation to the target model. In the horizontal transformation the observable behavior of the system is preserved while the internal structure changes. In the vertical transformation the correctness has to be preserved from an abstraction level to another.

The main challenge is the transformation from PIM to PSM (different models).In traditional approaches, the transformation is inefficient because of the lack of formal models. Using informal models will prevent the formalization of transformation hence the transformation cannot be automated. In practice the process from PIM to PSM might be a lot more complex and challenging. Between the models, gaps can exist because of the difference in the abstraction layers exhibited in the models. The gaps may not be small enough to perform a direct transformation.

The idea of MDA works for structural models(OMG 2014c), (Ahmed 2010) but there is still difficulties when behavior is addressed as (Abdalla & Abdullah 2011) had investigated. Current practice shows that, in most cases, behavioral models are used for other purposes like documentation rather than complementing the structural models to facilitate automatic software generation. The problem is the lack of mechanisms for mapping behavior models from an abstraction level to another, for example from PIM to PSM.

The mapping is an important part of the process of generating an implementation. There are different types of mappings. Model type mappings specify mappings of the instances of model types from the source model language to the instances of target model types. Model types can be specified using Meta -Object facility MOF (OMG 2006) or any other language including natural languages. Another kind of mapping is the model instance mappings. A mapping will identify model elements in the PIM that can be transformed. Most mappings, however, will consist of some combination of the above approaches. What we want to do is to relate M0 instances of the PIM to the M0 instances of the PSM as an implementation of the application. The OMG document (Richter & Conti 2004) doesn't address how to relate and transform models when developing software applications using the MDA approach.

Several attempts such as (Aksit et al. 2009) described various approaches to better support automatic software generation by using behavior models but a concrete method to map the behavior models (state machine) from PIM to PSM is still an open question and to be answered.

Research Questions

The following research question is formulated:

- **Research Question 1:** How to automate software application generation using UML behavior models in MDA approach?
  It can be seen that methods for mapping the state machine behavior model between PIM and PSM are strongly needed to facilitate the complete generation of fully automatic software applications using MDA

These research questions are analyzed and answered in Section 1.6.

Objectives

This research aims to provide a proposed MDA based engineering methods to map the UML behavior models. The method is aimed at achieving an acceptable degree of automation of software development. The general objectives are:

1. To find an engineering approach or a method for mapping state machine behavior models from PIM to PSM.

2. To investigate suitable transformation languages and metamodels that better facilitate the mapping process.

3. To evaluate the proposed approach by developing a system automatically using MDA best practice and transferring the generated artifacts (programs, configuration files and all the generated classes to a suitable environment to be executed.

Main Contributions

The main contribution of this research is the method for mapping UML state machine behavior models from PIM to PSM. This contributes to the MDA in bridging the gap between various design levels (PIM to PSM) and implementation (PSM to Code aka text).

Based on the method, transformation framework , guidelines, models and meta models an MDA approach can be successfully applied to other software applications. Contributions are summarized as follows:

1. Domain assets creation

   The careful design of various models including PIM in two domains is valuable since the MDA approach is model centric. The PIMs can serve as a working samples for PIMs in different domains thus contributing to the development knowledge in MDA.

   The formal design of The PSM created for the ATM machine is formed based on a standard and can be used with various implementation platforms. An equivalent effort was done in designing the messaging system PSM. Chapter 3 and Chapter 4 illustrate the designs in detail based on UML static and dynamic features. Moreover the QVT transformations and the generated codes, files and documents included in the appendix, would increase the scale of capital equipment available to software solution developers.

Considering the implementation of the system, the algorithm that define the mapping between UML state machine meta model and the SCXML provides a basis for translating between them. The algorithm in Chapter 6 can be applied in different programming languages to provide the translation in different platforms and execution environments other than the one tried in the case study thus providing a generic translation technique.

2. The strategic messaging system PSM could be re-used to afford many products from the domain.
   The formal and precise representation of the messaging system in a model enables the usage of the model in other types of software applications such as email and chat applications. This facilitates the mapping automation and adds a value of selecting among alternatives. The PSM is illustrated in detail in chapter 4.

3. Taxonomy and guidelines for state machine mappings
   The suggested framework , guidelines , naming conventions or the taxonomy for state machine elements and the transformation are the heart of the proposed engineering method. In software engineering and design science the designs and the foundations are recognized as contributions to the knowledge in the field. Extending the existing knowledge and applying what we already know in a new and creative way is existed in the case studies spaces and in the conclusions drawn from tackling them. The case studies were deeply illustrated in Chapter 3 and 4, the result discussed and summarized in Chapter 7.

4. State of the art tools in the MDA context was identified and used.
   The MDA approach promotes the use of standardized languages, models and toolsets. The conformance to the standard is always a

benefit to software development team members. In our study we tried to follow the MDA best practices and set of languages that support the design and coding when treating the models. UML, QVT and EMF that conforms to and uses the UML meta models are used in our case studies. The complete set of standards and tools used is discussed in Chapter 3.

Thesis Outlines

The thesis is structured as follows. Chapter 1 introduces the research motivations, background information on MDA and concepts that will be referenced throughout the thesis. Chapter 2 describes the MDA and present the Unified Modeling Language (UML) and the behavior modeling capabilities. The research methodology, tools, languages and case studies description are covered in Chapter 3. In Chapter 4 and Chapter 5 the case studies are described, observation and issues were identified. The model to code and application execution are detailed in Chapter 6. The proposed approach results are presented and discussed in Chapter 7. Finally, we conclude by summarizing the contributions of the thesis, answers to the research questions and how objectives were achieved in Chapter 8.

Literature Review

Overview

This chapter serves as an introductory, review and criticize of three main topics in this thesis: Model Driven Architecture, Unified Modeling Language and automation and the model to code transformation.

This chapter begins by providing an overview on Model-Driven Architecture (MDA) as a modern software development paradigm and how it can leverage the many concepts such abstraction, automation and reusability. Section 2.2 of this chapter is to present the Unified Modeling Language (UML) and the behavior modeling capabilities. It is followed by Section 2.3 presenting the automation and model to code transformation in the traditional manner and how MDA changes that.

Section 2.4 explains the relationship between UML structure and behavior models. Section 2.5 presents the basic concepts of UML 2 class and state machine diagram elements. Section 2.6 presents the tools and techniques that support state machine models. This chapter is ended with Section 2.7 to summarize and provide a brief description of the whole chapter.

Model Driven Architecture (MDA)

An organization need to make sure that its existing legacy software system will evolve and it can easily integrate what it is building with what is going to be built in the future. As the pace of technology continues to speed up, the organization needs an architecture as a base for its infrastructure. The bad news is that neither a single platform nor a single operating system nor a single programming language nor a single network architecture will be available to depend upon. In the other hand with new approaches to software development, the organization can still manage to build software systems in such changing environment. Model Driven Architecture (MDA) is an initiative of OMG Object Management Group that is intended to better deal with the complexity of software system development. Figure 2 OMG Model Driven Architecture (OMG 2015) lays out the MDA which is transparent to operating systems, programming languages and network protocols.

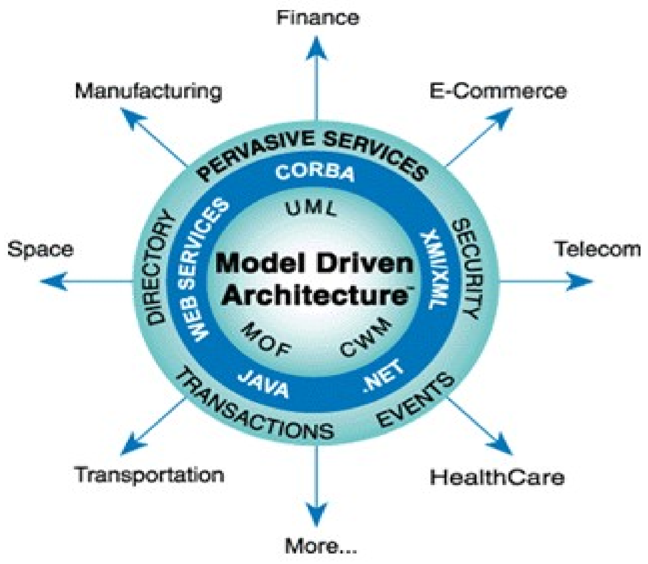


Figure 2 OMG Model Driven Architecture (OMG 2015)

Major MDA concepts

System

A system is a collection of parts and their relationships organized to achieve some purpose (OMG 2014a). In MDA, the term ‘system’ can refer to a software system or it can be generalized to include anything: software, hardware, people etc.

Model

MDA uses models as the first class artifacts in the software development process. It aims at deriving values from models that capture the system structural and behavioral aspects. A **model** in the context of MDA is information that represent a system based on a specific concern (OMG 2014a). The model also should include the integrity rules applied to the system beside the meaning of terms used. A model can represent business, domain, software, hardware and environments or any other aspect of the system. A physical system model could include representations of a hardware environment and a performance simulation. A software system model could include UML class diagram and screen shots of the user interface. A model of an enterprise may include business processes, services and resources. Models can provide a common understanding of the modeled system between different stakeholders. They can also be analyzed and evaluated to help in decision making. Models can simulate how the system being modeled is going to function. Moreover, models can be executed thus providing a design realization into a working system.

Metamodel

A metamodel is a model that set of models conforms to. It is the common foundation for the models expressed using such metamodel. The Meta Object facility (MOF) (OMG 2014b) is a key foundation to the Object Management Group MDA. MOF includes a family of specification and unifies the steps of development, integration and evolution of models. The key modeling concepts are Classifier , Instance (class and object) and the navigation between them. These concepts allow the traversal of any number of layers recursively. OMG defines a four layer architecture of model levels each of which conforms to (aka is an instance of) the one above it. The four levels are illustrated by an example as in Figure 2 An example of four-layer metamodel hierarchy (OMG 2014b).

Figure 2 An example of four-layer metamodel hierarchy (OMG 2014b)

Metamodels also specify the schema for a repository that stores model instances. A case tool can use the repository to create, store, browse, render , edit etc. model elements. A transformation process can take place where both the target and source models metamodels are present. The transformation specification will look at the source instances model that conforms to a source metamodel and try to produce the target instances that conforms to the target metamodel. More on transformation and their examples are illustrated in the next sections.

Modeling Language

To be useful for the system stakeholders, any model need to be expressed in a way that facilitate the communication of information about the system and also need to be correctly interpreted by the stakeholders and their technologies. A **modeling language** is used to express the structure, terms, notations, integrity rules, syntax and semantic of a model. OMG 's standard modeling language is the Unified Modeling Language (UML) (O M G 2011). SQL Schema, Business Process Modeling and Notation (BPMN), Web Ontology Language (OWL), and XML Schema are examples of well known modeling languages.

Model driven

Describes an approach to software development whereby models are used during the various development phases as the primary source artifact for documenting, analyzing, designing, constructing, deploying and maintaining a system.

Architecture

The architecture of a system is a specification of the parts of the system , its connectors and the rules that define the interactions of the parts using the connectors (Garlan & Shaw 1993). Within the context of MDA these parts, connectors and rules are expressed via a set of models.

In MDA an **architectural** process include understanding the stakeholders requirements , understanding the system scope and satisfying the requirements by a design of that system. MDA promotes modeling to the architectural process and formalizes the resulting artifacts (e.g. formal designs or models) so that developing systems or improving them could be less expensive and less error prone.

Platform

A platform is a set of subsystems and technologies that provide a rational set of functionality through interfaces and usage patterns. The users of a platform use it without the concern of how functions are done. Examples of platforms include operating systems, programming languages, databases, user interfaces, middleware solutions etc.

Platform Independence and Abstraction

Platform independence is a quality that a model may have. When a model is platform independent then it is expressed independently of the features of that platform. Independence is a relative indicator in terms of measuring the degree of abstraction (i.e. where one platform is either more or less abstract compared to the other).

An important basic concept in MDA is the **abstraction**. Abstraction is the concept of understanding the system in a general way and eliminating certain elements from the defined scope. Modeling and abstraction go well together. We can design a model of a system while abstracting away particular details such as those that tie the system to how it can be implemented in a specific platform or technology. The more abstract the system the more systems it can represent. The more specific the system the more bounded to specific details of the technology or platform that it represents.

CIM, PIM and PSM

When a model of a system is defined in terms of a specific platform it is called a “Platform Specific Model” (**PSM**). A model that is independent of such a platform is called a “Platform Independent Model” (**PIM**). A Domain is defined as a bounded area of knowledge. Domains relate to knowledge in two ways: vertically and horizontally. Vertical domains are the business domains such as banking, accounting, etc. Horizontal domains are specific software implementation technologies that are frequently used by vertical domains. In MDA, a Computation Independent Model (CIM) specifies the requirements of the system and includes the domain model which is in a level higher than a PIM.

Implementation

An implementation is a specification that provides all the information required to construct a system and to put it into operation.

Model Transformation

Model transformation is the process of converting one model to another (PIM to PIM , PIM to PSM, PSM to PSM and PSM to Text). One of the MDA capabilities is the automation of transforming the models from abstraction level to another e.g. from a PIM (closer to business concepts) to a PSM (closer to technology). Given an abstract concept in one model such as a class in UML , we could transform and produce a SQL table representation of the class in an Oracle data base system. The transformation specifies the definition of the pattern and parameters that are applied to the source element in order to produce the target element. Note that for the same PIM we can specify different patterns and parameters thus support different technologies.

Figure 2 MDA Pattern(OMG 2003) illustrates the MDA pattern by which a PIM is transformed to a PSM. It is a generic pattern and there are many ways to carry out the transformation. The empty box represents additional information that can be supplied to the transformation according to the MDA chosen style and along with the PIM.

Figure 2 MDA Pattern(OMG 2003)

The conversion of the source model to the target model will be carried out by standard mappings. In Figure 2 OMG Model Driven Architecture (OMG 2015) the target platforms are represented by the thin ring surrounding the core. Automation of the mapping is a goal, however some hand coding may be necessary because of the immaturity and lack of MDA tools.

One of the steps in creating an application using MDA is to produce the application artifacts.

Figure 2 MDA- based Software Development Process Example (OMG 2003) depicts an example for full MDA process that includes the execution environments. For example in a component based environments, the necessary files will be generated such as interfaces, component definitions and configuration files. The platform independent model reflects the general model of the application. The more complete this reflection is, the more complete the application structural and behavioral features can be included in the specific

model hence the more complete application can be generated. In a mature MDA environment, code and related files production can be significant and even complete. Deriving code and implementation from models is one of the uses of MDA. Automation enables rapid response to changes, increases the efficency of software development and decreses its cost.

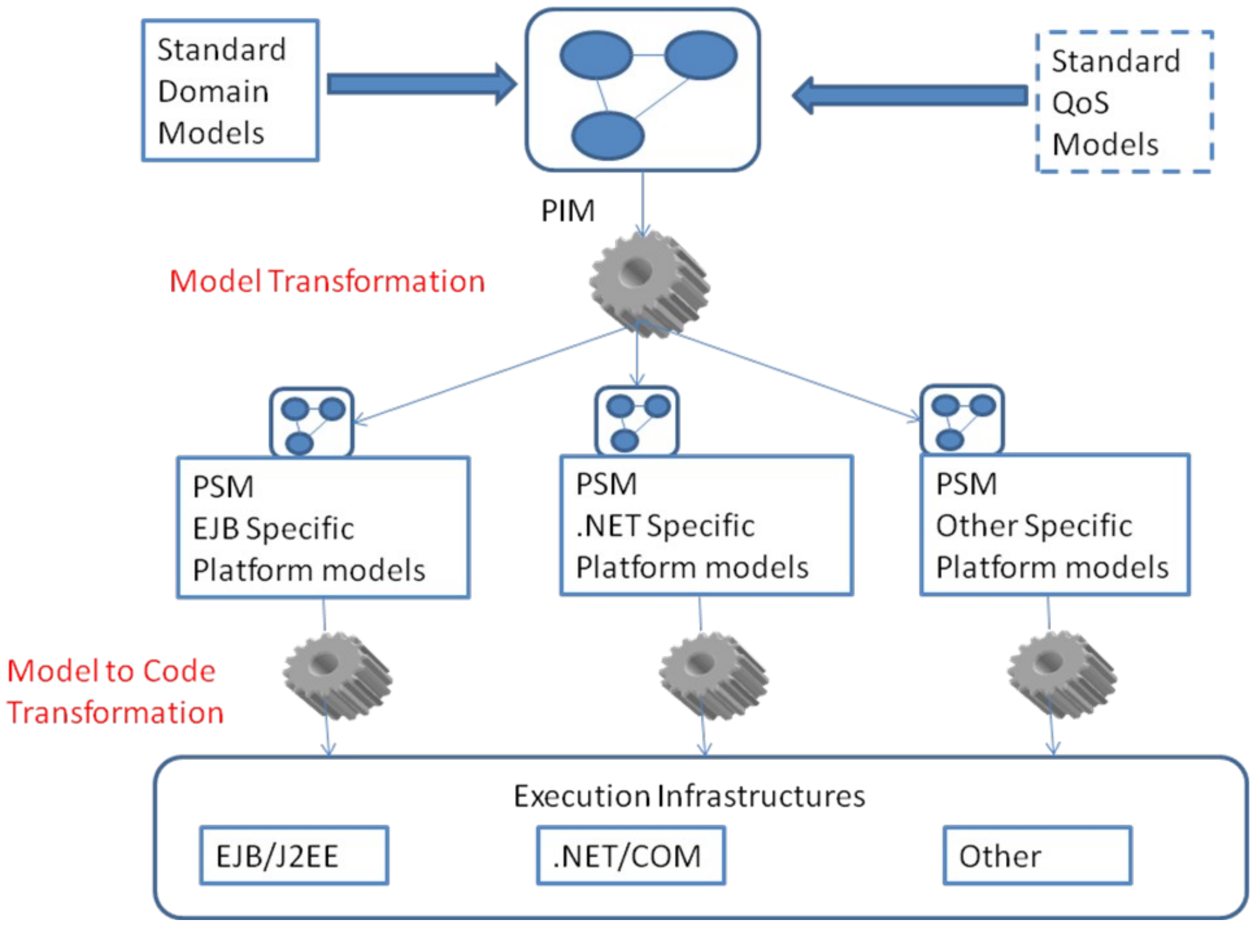

Standard Domain Models
Standard QoS Models
PIM
Model Transformation
PSM EJB Specific Platform models
PSM .NET Specific Platform models
PSM Other Specific Platform models
Model to Code Transformation
Execution Infrastructures
EJB/J2EE
.NET/COM
Other

Figure 2 MDA- based Software Development Process Example
(OMG 2003)

QVT is the standard OMG transformation language and is expressed in section .

MDA Adoption and Promises

MDA has been advantageously implemented in small and large organizations for different types of systems. some companies prefer to keep their success a secret to their competition, while many have agreed to publish their accomplishments, as can be seen on the OMG website (OMG 2014c) as well as in various articles.

MDA is a software development method that promises to facilitate the creation of formal models to achieve the long term flexibility in terms of:

- **Technology Proven**: new implementation technologies can easily integrated and supported by existing designs. **Separation of concerns** is an old engineering principle. Dijkstra is generally credited for bringing this idea to the attention of software community (Dijkstra 1976)

  Dijkstra: "*I have a small mind and can only comprehend one thing at a time*."

  Separating the business logic from technical details allows both PIM and PSM models to change without affecting each other and provide a solution to the software churn that burdens developers, system venders and users.
- **Portability**: existing designs and functionality can easily migrated to different environments and platforms.
- **Productivity and time to market**: automating tedious development tasks would free the developers and architects to

focus on the business logic. The resulted system development would be faster and less error prone.

- **Quality**: Formality, separation of concerns, consistency and reliability of artifacts produced all contribute to the quality of the produced system.
- **Testing and simulation:** models can be validated against the requirement and also tested against different infrastructures and platforms. They can be used to simulate the system behavior too.

OMG Adopted Standards for MDA

In order to enable the MDA approach, a set of technologies were adopted by OMG. Including UML as a standard modeling language , Meta Object Facility (MOF) (OMG 2014b) as a repository for model manipulations and Common Warehouse Metamodel (CWM) that enable the interchange of warehouse and business metadata. These are the core models of the architecture represented in Figure 2 OMG Model Driven Architecture (OMG 2015). Each core model represents the common features of all the platforms in its category, technically it is a metamodel of the category.

UML and Behavior Modeling

This section is the second one in the literature review specifically related to modeling in UML because models are the building blocks of MDA. The Unified Modeling Language (UML) is an OMG standard for modeling systems. UML provides diagrams to model structural and behavioral aspects of a system. Structure diagrams in UML represent the static structure of the system. The class diagram is widely used to describe static structures while other diagrams such as object, component and deployment diagrams are also provided. A variety of mechanisms to specify behaviors are supported in UML such as automata (state machine), Petri-net like graphs (activity), informal description (use case) and sequence of events (interaction) (OMG 2011). These different behavior specification mechanisms differ in their expressive power and domain of applicability so the choice of one of them depends on convenience and purpose.

Almost every application contains functionality that describes its features. Beside that the application has behavior and possess a structure. Behavior modeling is used to visualize, specify and construct various dynamic aspects such as modeling a flow of control, an element behavior, a workflow or an operation. Depositing an amount of money into a bank account is an operation example. The formal definition of the behavior of this operation remains the same despite whether it was implemented in different platforms or in d9ifferent programming languages. Modeling the operation behavior helps to express the operation in a higher level of abstraction. The abstract level description as a result allows for reusing the operation specifications between different languages, frameworks and execution environments.

Behavior execution -automatically verifying models on a computer- at PIM level is a remarkable feature to verify high level models against the requirements and to provide conformance for implementation at

PSM and code level. Considering that the promise of MDA is the automatic generation of software based on models, the static structural features of the modeled system are not always enough to generate a completely full automated application unless complemented by behavioral features of the system. One can conclude that behavior modeling is important to support MDA goals.

Current practice shows that MDA approach works quite well (OMG 2012) but, in most cases, behavioral models are used for other purposes like documentation rather than complementing the structural models to facilitate automatic software generation. The problem is the lack of mechanisms for mapping behavior models from an abstraction level to another, for example from PIM to PSM.

There are different approaches for modeling and executing behavior in the UML at PIM level. According to the study in (Riccobene & Scandurra 2009) they may mainly fall into the following mentioned categories. In the first category , behavior is not included in the PIM at all, but instead it is added as code to structural code skeletons later in the MDA process. This, however, prevent validating the system at earlier stages.

A notion of behavior is represented in the second category by the use of the Object Constraint Language (OCL) (OMG 2010) to add behavioral information (such as pre- and post-conditions) to other more structural UML modeling elements. This representation came at its own cost because OCL does not allow the change of a model state, though it allows describing it. In other words OCL is side-effect free.

In the last category, UML behavioral diagrams such as state machines, activity diagrams, sequence diagrams are used for behavior modeling and representation. However the purpose of that is for documenting the user requirements. The various mentioned diagrams are not used as a facilitator for automatic code generation. As a consequence behavioral models are separated from the code, which finally even leads to dead models.

Some effort was dedicated to enhance and extend UML diagrams as in (Kalnins et al. 2009). Their work was based on extending two UML behavior modeling notations, the sequence diagram and the activity diagram. The extension aims to provide more expressivity to the activity diagram and to allow the sequence diagram to represent behavior of multiple classes.

In our study , we are going to use UML statechart diagram to represent the behavior of the classes represented in the case study. We do believe that realization of the MDA vision requires that the business logic behavior of an application be represented explicitly in the PIM. State-machines provide the suitable basis for such representation (Mcneile & Simons 2004).

UML Structure and Behavior Models Relation

In order to simplify the semantic considerations, we are going to give an overview of the relation between UML classes and its behavior, considering both the activities and states.

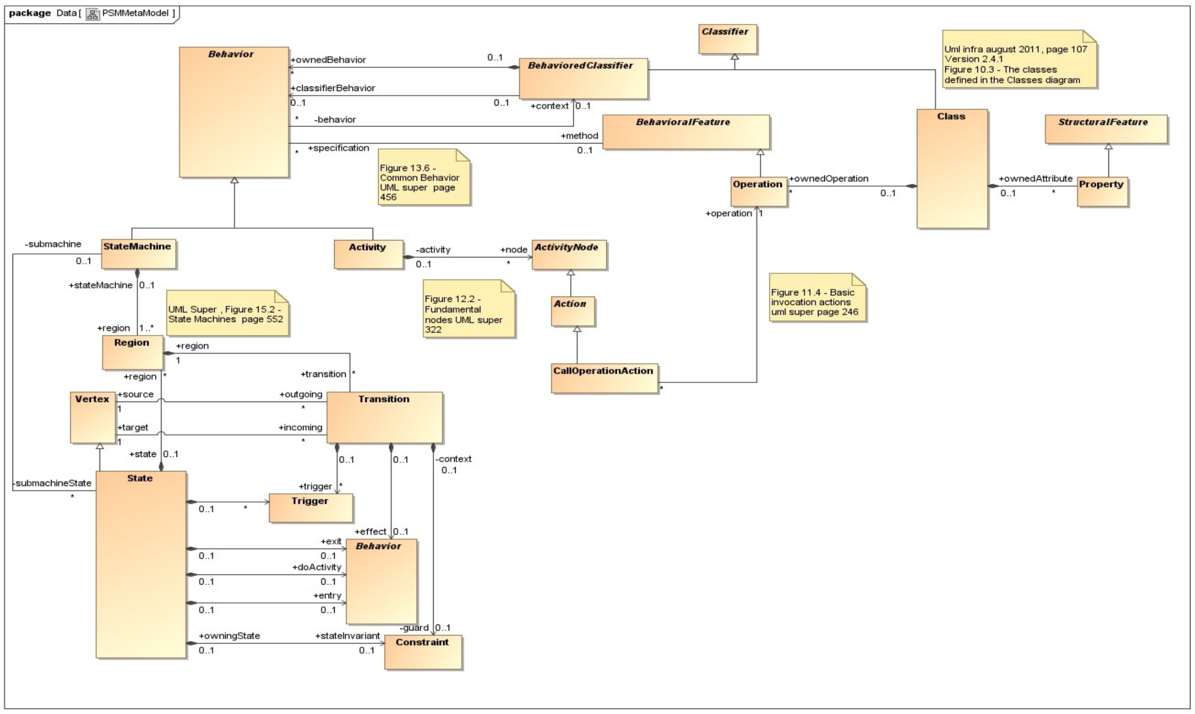


Figure 2 Simplified UML Class and Behavior Models Relationship

Figure 2 Simplified UML Class and Behavior Models Relationship is a diagram constructed from the BasicBehaviors, Kernel and BehaviorStateMachines Packages of UML (OMG 2011). **Class** can have zero or more owned attribute of type **Property** which is a ***StructuralFeature***. It can also be associated with zero or more owned operations of type **Operation** which is a ***BehavioralFeature***. *ownedAttribute* and *ownedOperation* can belong to at most one Class. The specification mechanisms used to specify the behavior of an **Operation** can be a **StateMachine**, an **Activity** or any concrete sub class of the abstract class ***Behavior***. The behavior (eg. activity) in this case is considered the implementation of the feature (the computation that generates the effects) that the class is modeling by its operation.

A **StateMachine** is a ***Behavior*** that has at most one ***BehavioredClassifier*** as its context. The ***BehavioredClassifier*** is a ***Classifier***. Since the **Class** is also a ***Classifier***, then navigating from the state machine using its context attribute will link the state machine back to its class. Navigation from the Class to the associated behavior(s) is possible using the directed association relations **ownedBehavior** and **classifierBehavior** respectfully**.** In a similar way we can navigate between the **Activity** and the **Class** to its context class. We conclude that within a particular model instance, UML pretty well integrates various diagrams.

The **StateMachine** is composed of one or more regions which in turn composed of zero or more states. Each state may compose of three owned elements, **entry, exit** and **doActivity** of type **Behavior** as an effect of a transition related to the state region. In other words a state has the ability to do a behavior before it transitions.

A fundamental unit of behavior is an **Action** that can modify the system state in which it is executed. Behaviors provide the action context and determine when actions to be executed and with what parameters (property values of objects). Actions can perform calls to operations specified in the model; the called operations may be bound to activities, state machines or other behavior.

Basic UML 2 Concepts

Class and State Machine

In this part we briefly introduce UML 2 state machines that represent the current state of the art in the long evolution of these techniques. The intention is not to give a complete, formal discussion of UML state machines, which the official OMG specification (OMG 2004) covers comprehensively and with formality. Rather, the goal in is to lay a foundation by establishing basic terminology, introducing basic notation, clarifying semantics, and giving some examples. This section is restricted to only a subset of those state machine features that are arguably most fundamental. The emphasis is on the role of UML state machines in the practical everyday programming rather than mathematical abstractions.

For illustration purposes Figure 2   Simplified ATM Class represents a simplified class diagram with one class - the ATM class which is represented in an Automated Teller Machine (ATM) software system. The ATM allows users (i.e., bank customers) to perform basic financial transactions. The first case study in provides a concise, carefully paced, complete analysis and design experience.

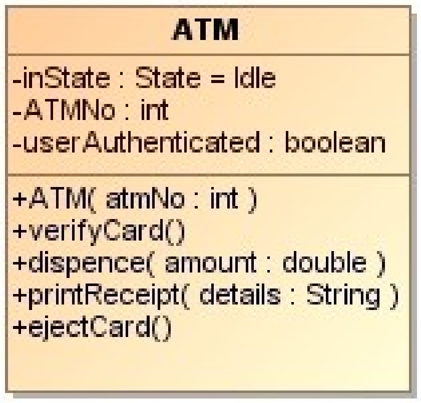

ATM
-inState : State = Idle
-ATMNo : int
-userAuthenticated : boolean
+ATM(atmNo : int)
+verifyCard()
+dispence(amount : double)
+printReceipt(details : String)
+ejectCard()

Figure 2 Simplified ATM Class

The ATM objects (instances) have both behavior and static structures or, in other words, they do things and they know things. The ATMNo is an attributes of the class representing static structures. The verifyCard method is considered a behavior. Beside that the class has a state machine that describe its life time and shown in Figure 2 ATM State Machine Diagram.

UML state machine diagram is a behavior diagram that is used to visualize, specify and construct various dynamic aspects of a designed system through nodes (states) and edges (transitions). State machine diagrams can also be used to specify the usage protocol of a system. UML provide behavior state machine and protocol state machine. The behavior state machine which we will refer to as state machine, is an object based variant of Harel state charts (Harel 1987).

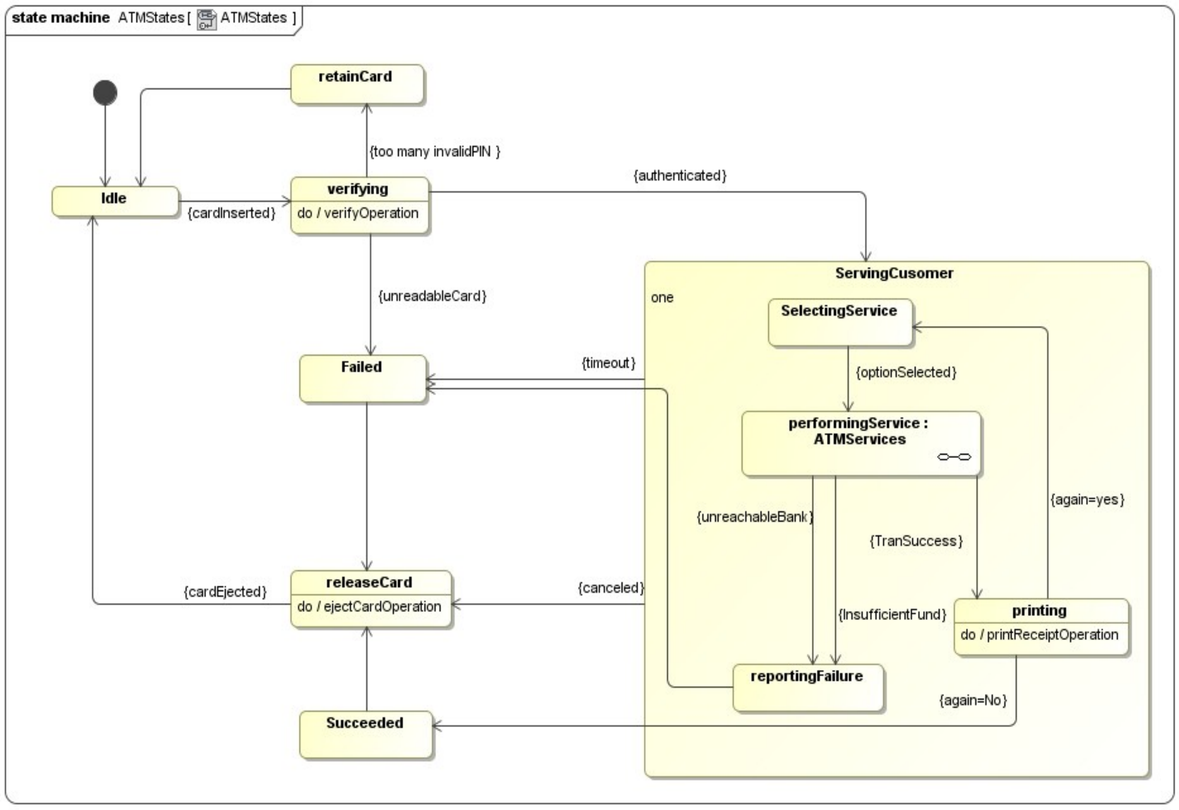


Figure 2 ATM State Machine Diagram

A state represents a stage in the behavior pattern of an object. During the life time of the object, a state satisfies some condition, performs some activity or waits for events to occur. It is possible to have initial state and final state. When the object is created it is placed in the initial state. The final state has no transitions going out of it.

A transition is a relation between two states, the source and target states. A transition indicates that the object will move (transit) from the source state and enter the target state when an event occurs, a condition is satisfied or an action is performed. A self-transition is a transition whose source and target states are the same.

Figure 2 ATM State Machine Diagram presents an example state machine diagram for the ATM class. The rounded rectangles represent states where the arrows with stick arrow head represent transitions. The instances of ATM can be in one of the modeled states such as Idle, verifying, or ServingCustomer states. The instance can start in the initial state, represented by the closed circle, and can end up in the Idle state again.

A state machine can change from one state configuration to another in a response to an occurrence of an event. An event is the specification of a significant occurrence. The name or description of the event that cause the transition is written the line that corresponds to the transition. For example, the ATM object changes from verifying state to servingCustomer state after the bank database authenticates the user.

A transition may be associated with at most one guard which is a constraint (condition) that controls the firing of the transition. The guard condition is a boolean expression that is evaluated when the event occurred, if the evaluation result is true ,the transition is enabled or otherwise it is disabled. A user class which is not shown here is authenticated by comparing the account number and PIN entered by the user with those of the corresponding account in the database. If the bank database indicates that the user has entered a valid account number and correct PIN, The ATM object transitions to servingCustomer state and changes its authenticated attributes to a value of true.

In order to model complex behaviors, sub states cab be grouped into a composite state. The state ServingCustomer is a composite state that is having SelectingService as a sub state. Another type of states is the compound state which indicates that the details of the PerformService sub-machine are shown in a separate diagram.

Constraints

A constraint is a restriction on UML models and model elements. As Figure 2 UML Constraints Metamodel (OMG 2011) depicts, the constraint is associated with an element and it has at most one specification.

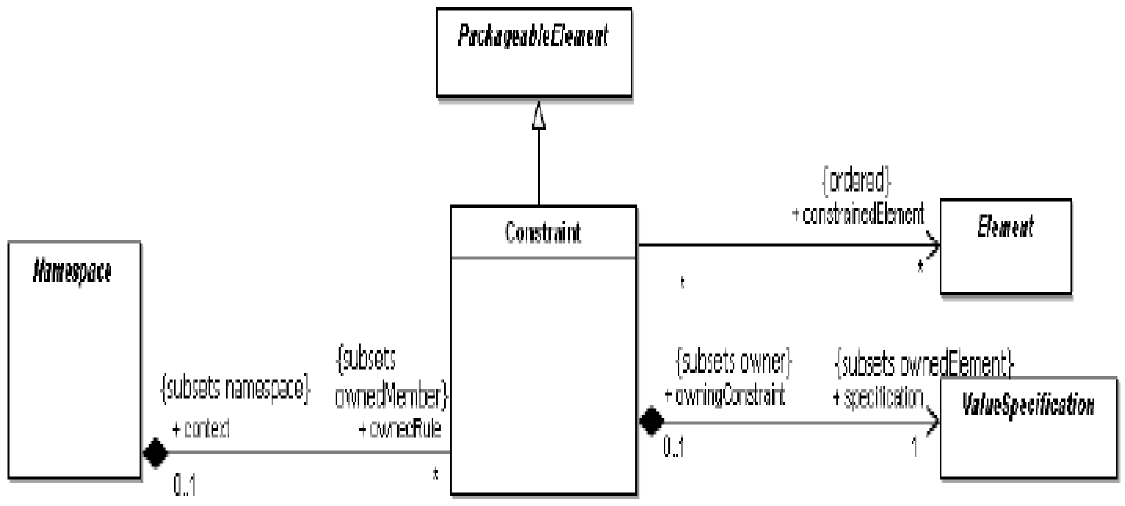
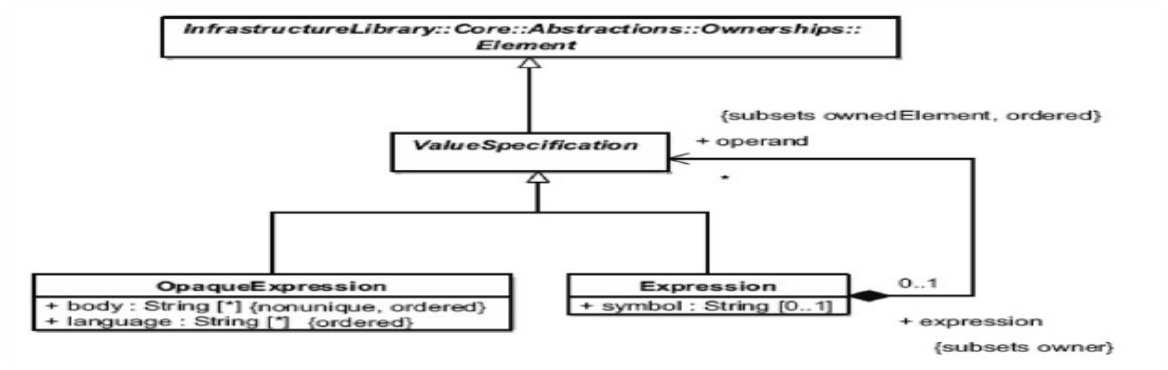


Figure 2 UML Constraints Metamodel (OMG 2011)

Specifying constrains is enabled by the flexibility of the ValueSpecification class and the OpaqueExpression extension as denoted in Figure 2 Elements defined in UML Expression Package (OMG 2011). The metamodel specifies the usage of a ValueSpecification wherever a value can be provided by a variety of technologies.

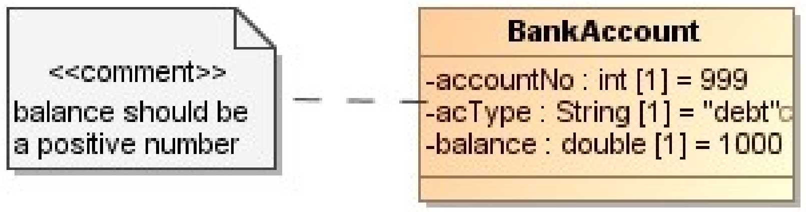

InfrastructureLibrary::Core::Abstractions::Ownerships::Element
ValueSpecification
{subsets ownedElement, ordered}
+ operand
*
OpaqueExpression
+ body : String [*] {nonunique, ordered}
+ language : String [*] {ordered}
Expression
+ symbol : String [0..1]
0..1
+ expression
{subsets owner}
Figure 9.20 - The elements defined in the Expressions package

Simple specification values can be provided by a string literal in any language including natural languages. More values can be provided by an OpaqueExpression that has two attributes, one of language names [language attribute] and the other of string bodies in the corresponding language [body attribute]. The attributes provide an ability to present implementations in a variety of languages. If the language name is omitted, an implementation default of The Object Constraint Language OCL is assumed. OCL is a precise text language that provides constraint and object query expressions on MOF model or meta-model. It is a key component in the OMG QVT specifications.

Specification of a behavior such as “name.toUpper()” can be achieved by an OpaqueExpression in which the language value is ‘OCL’ and the body is ‘name.toUpper()’. The OCL is therefore embedded in a textual form that has no knowledge of the classes in OCL metamodel. Users have the choice to use programming languages API such as the OCL Java API. The benefit is to avoid the need to incur OCL parsing costs by exploiting OCL’s ExpressionInOCL class that extends ValueSpecificaion and delegates functionality to an OCLExpression.

Specifying Constraints in UML

Some constraints can be effectively specified using the graphical UML features. Some types of constrains can't be represented by UML. Using UML comments to add constraints in the form of text was previously used but that was a source of ambiguity, informal specification and none interpreted constraints. Figure 2 Account Constraint: Positive Balance shows a comment used to express a constraint.



<<comment>>
balance should be a positive number
BankAccount
-accountNo : int [1] = 999
-acType : String [1] = "debt"c
-balance : double [1] = 1000

Figure 2 Account Constraint: Positive Balance

OCL is a language that is intended to provide a formal and comprehensive specification of model constraints. It has a precise syntax that enables the construction of unambiguous constraints and can avoid the inherent difficulty of using complex mathematics too. OCL can be applied to UML models and is used in MOF and QVT.

OCL statements are constructed in four parts:

1. a context in which the constraint is to be evaluated.
2. a property that defines some characteristics of the context (e.g., if the context is a class, a property might be an attribute)
3. an operation (e.g., arithmetic, set-oriented) that manipulates or qualifies a property, and
4. keywords (e.g., if, then, else, and, or, not, implies) that are used to specify conditional expressions.

There are four types of constraints on an object that can be specified using OCL: invariants, pre conditions , post conditions and guards.

Invariant

Invariants are constraints that applies to all instances of a class and evaluates to true if a condition is met. An invariant constraint consists of an OCL expression of type Boolean. The expression must be true for each instance of the classifier at any moment in time. For the ATMCard class in Figure 2 UML Constraints Metamodel (OMG 2011) the invariant expirationDate.isAfter(today) ensures the validity of the card when used by checking the class property expirationDate.

The syntax of an invariant is as follows:

context <class name> **inv**: <Boolean OCL expression>

Multiplicity constraints can be understood as simple cases of invariants. Specifying the multiplicity in associations can constraint the relation between the association ends instances. Each association end is a property whose type is a class. The association between the BankCustomer and an ATMCardas in Figure 2 ATM Card and Customer Association Multiplicity is named *customerCard* . An instance of the BankCustomer class can have one or more instances of ATMCard (*myCard*) denoted by 1..* , where an instance of ATMCard can only belong to one BankCustomer(*holder*).

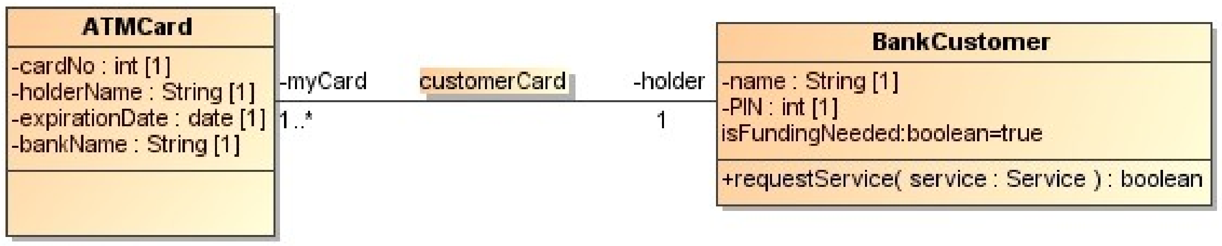


Figure 2 ATM Card and Customer Association Multiplicity

Precondition

A precondition is a constraint that may be associated with an operation of a classifier. The purpose of a precondition is to describe the conditions that has to hold before executing the operation by an instance. The precondition consists of an OCL expression of type Boolean evaluated to true whenever the operation is executed. Figure 2 UML Constraints Metamodel (OMG 2011) shows the placement of a precondition in the UML meta model.

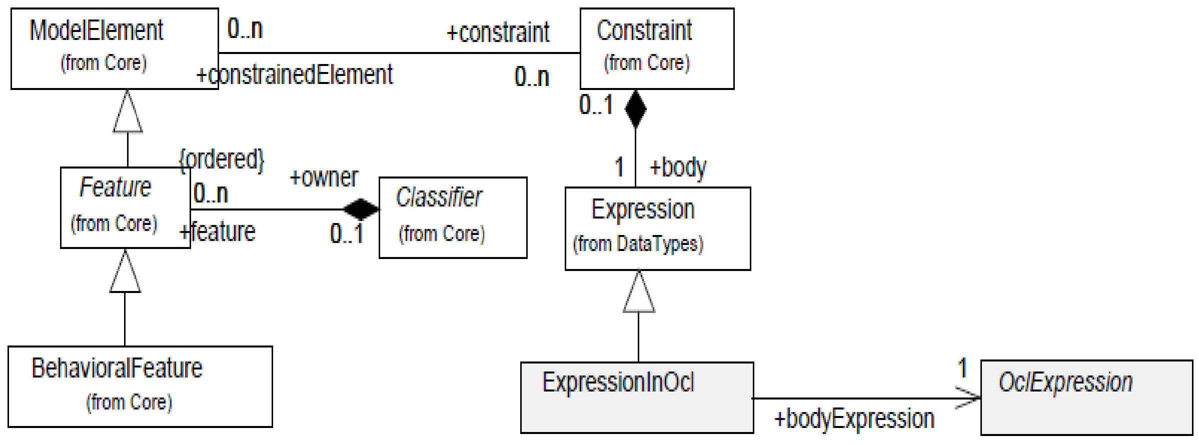


Figure 2 An OCL ExpressionInOcl used as a pre or postcondition

Postcondition

As the precondition constraint, the postcondition is a constraint that may be associated with an operation of a classifier. The purpose of a postcondition is to describe the conditions that has to hold after executing the operation by an instance. The precondition consists of an OCL expression of type Boolean evaluated to true whenever the operation stops executing. The mark "@*pre*" can be used to refer to values before execution time and the variable *result* refers to the returned value of the operation if any.

The OCL syntax to denote a precondition, a postcondition or a pair of them for an operation is:

context <class name> :: <operation> (<parameters>)

pre: <Boolean OCL expression>

post: <Boolean OCL expression>

Let us assume that the withdraw operation of ATM class is as follows:

Preconditions:

1. The ATM must not be in an error state

2. it must hold some card

3.The amount to be withdrawn is positive

4. The balance covers the withdrawal amount

Post-conditions:

After withdraw has been executed, the right amount of money must have been spent or some error has occurred.

The OCL statements are as follows:

> *context ATM::withdraw(amount : Integer)*
>
> *pre: (state = #ok) and (cardId <> 0) and (amount > 0) and (balance > amount+100)*
>
> *post: (balance = balance @pre - amount) or (state = #error)*

The post-condition expression makes use of the OCL operator @pre that yields an expression's value at pre-condition time.

Guard

A guard is an expression that can be linked to an association in a state machine. It places a restriction on the transition to the target state. Whenever the transition is attempted, its value must evaluate to true. The value of the guard is of Boolean type. The context of the guard is a classifier which is the owner of the state machine.

OCL syntax is simple. It defines an OCL expression, which always has a type. The classes defined in the class diagram can be used in OCL expressions. These types are called model types in the OCL literature. Typical operations for class types deal with the properties of a class type, i.e. its attributes, operations and associations.

Traditional Support of State Machine in Software Development

There are a few different techniques to implement state machines in different programming languages such as C, C++ and Java. These can be categorized into the following:

- native language support
- hand-coded implementation
- tabular implementation
- unintentional state machines
- State pattern

- use of a library
- model-based code generation

Native Language Support:

Some languages has built in support for state machines such as Erlang(Anon n.d.). Erlang is a programming language designed at the Ericsson Computer Science Laboratory. It contains libraries of code for building robust fault-tolerant distributed applications.

Hand-coded Implementation

A program that contains a switch statement where the code for each state is written and the next state is determined.

Tabular Implementation

A state transition table of entries represented as (source state, destination state, input condition) and the table is processed in the application. Then for each update of the state machine, the table is used to determine the next state.

Hidden State Machines

The logic is added in a program that contains a flag that switches between two states. The source code is considered the specification of the state machine behavior.

State Pattern

The state pattern is a behavioral software design pattern, also known as the objects for states pattern. This pattern is used in computer programming to encapsulate varying behavior for the same routine based on an object's state .It is a way for an object to change its behavior at runtime in class instances that encapsulate the behavior in each state, including determining which state is next.

Use of a Library

Some programming languages provide libraries to create and implement state machine such as the free C++ Boost libraries.

Model-based Code Generation using state charts

The state charts are drawn and the code is generated directly from them using some tools. Example of tools are Stateflow and StateMate that vary in their support and price. Most of the tools available in the market can generate the static parts of a model aka classes.

In MDA approach, sets of transformations are applied to a platform independent model (PIM) in order to derive a platform specific models. Query View Transform (QVT) standard addresses the model to model transformation (e.g., PIM to PIM, PIM to PSM and PSM to PSM). In order to complete the process of software development using models, those models has to be transformed to text artifacts such as code, deployment specifications, reports, documents, etc. The MOF Model to Text (mof2text) standard addresses how to translate a model to a text representation using a template base approach (Object Management Group (OMG) 2008). A Template is a text template that contains placeholders(expressions) for data extracted from metamodels entities through queries. For example, the following Template specification generates a Java definition for a UML class.

```
 [template public classToJava (c : Class)]
class [c.name/] {
// Constructor
[c.name/]() { }
} [/template]
```

For a class 'ATM' (shown in Figure 2 ATM State Machine Diagram), the following text will be generated:

```
class ATM {
        // Constructor
        ATM() { }
}
```

Automation and Model to Code Transformation

This section illustrates the third part of the literature review. One of the challenges in software engineering industry is to determine the software mistakes or at least to find mistakes (bugs) early during the requirement or design phases and not after delivery. Automatic code generation can provide a solution to the problem especially when it is based on a human-built model or design as investigated in (Burke & Sweany 2008). The authors concluded that the use of Model Driven Development (MDD) with automatic code generation can contribute significantly in decreasing the development costs and at the same time increase the reliability of products. As a result software development becomes faster, better, and cheaper.

Automation provides an increase in productivity. Generators can produce many application artifacts in short time. Tedious and boring parts of code can be also generated instead of hand written. Automation can also provide architecture consistency when programmers work within the architecture. Beside that automation lifts the problem to a higher level thus providing an easier porting to different languages and platforms. In contrast to the mentioned advantages, generators themselves - programs that produce programs- have to be written first. So there will always be hand coding required.

A research study (Domínguez et al. 2012) provides a systematic literature review that focuses on the code generation from state machine specifications in the context of MDD. The state machine specifications include UML state machines, finite state machines and Harel statecharts (Harel 1987). These constitute the most widely used specifications to specify the dynamic behavior of a system. The study analyzed the elements of the state machine specification supported by research and how they are implemented. The former analysis is denoted by element based comparison, the latter is referred to as pattern based comparison. Additionally the software feature that is desirable in the software development is considered and denoted as feature based comparison.

The review put it clear that the state machine specifications (UML state machines, finite state machines and Harel statecharts) constitute the most widely used to specify the dynamic behavior of a system. Moreover the UML state machines are the most common form used in automatic code generation in MDD. The results of the review show that the techniques of automatic code generation from state machine specifications can be classified into two groups , those based on design patterns and those not. Design pattern specifies a general solution for recurring design problems. Regarding the element based comparison, the review concluded that most of the implementations focus on elements such as simple states, events, guards, and actions in transitions. Specific elements of state machine such as simple and orthogonal composite states were less investigated by research. A key finding in the review regarding the feature based comparison is that many implementation strategies do not care about features like maintenance, reusability, or modularity.

Another conclusion drawn from the review is that code generation from state machine specification is one of the most challenging tasks. Because there is a gap between the modeling languages and the programming languages. Another reason is the dynamic nature of the state machine. Additionally, concepts such as states and events are not directly supported by most of the object oriented programming languages. MDA resolves what programming languages failed to handle by building a model of the system using modeling languages. UML - a standard modeling language- is having the concepts of states, transitions, actions, events and more. Traditionally modeling and programming are viewed to be different. Moreover there was a gap between the design phase and the implementation phase. The value of using the models created in the design phase is not gained and stops along the line in the software development process. As a result there was a difference between a model and a program. Our interest in MDA let us say that both the model and program are descriptions of the software system. MDA concepts rely on platform independent and platform specific models that can be seen as models and programs at a first glance. At a point in the software development process , the existing modeling artifacts are transformed to programming languages artifacts, after which the development method can proceed. This is only one benefit of gaining a value from models and there are more discussed in section

(Sunitha, E. V. 2012) presents a method to convert behavioral models to implementation code. The method concentrates on behavioral models which includes state machines, sequence diagrams and activity diagrams. The approach used was an MDA approach where the system is designed as a PIM using UML and mapped to a PSM using transformation. The implementation language targeted was Java. The method implementation, UML Code, includes a UML modeler, model processor, XMI generator and code generator.

In their method , the activity diagram reflects the business process flow. Each activity is explained using sequence diagram. The states of objects in the activity diagram is explained by state machine. The various diagrams contents are stored in a single XML file that conforms to a specific Document Type Definition (DTD). The DTD document shows how to express UML 2.0 activity diagram in XML. The XML file is parsed to identify behavioral objects and to transform them into Java classes using Extensible Style Sheet Language Transformation (XSLT). The generated code constructs were compared to similar tools output. The approach used in the study shows that 80% of source code can be produced automatically.

The mentioned method is parsing the files, in MDA the models are transformed from abstraction level to another to produce target artifacts. They also used XSLT which is a common and powerful language for XML transformations, but not suitable for transformations of semantically complex models due to its low level syntax. The method does not apply the principles of MDA, hence the benefits of MDA such as reusability and interoperability are not achieved. Although the mentioned method is not under the ideal MDA approach, but it assures that incorporating and using behavioral constructs is going to pave the way strongly between system designs and the generated code. The promise is to provide a complete code and not only code skeletons. In a comparable way we are suggesting a method that implements the MDA best practices and concentrates in the PIM to PSM mapping. The method could benefit from the several available UML modeling tools in addition to the other formal and de facto OMG standards such as QVT (OMG 2011a) and OCL (OMG 2012). Query View Transform (QVT) is the standard language that the OMG specified to carry on the transformation from model to model and from model to code. Object Constraint language (OCL) is a standard language that is used to specify constraints on models and model elements.

Summary

MDA places modeling at the heart of the software development process. Various models are used to capture various aspects of the system in a platform independent manner. Sets of transformations are then applied to these platform independent models (PIM) to derive platform specific models (PSM).

Deriving code and implementation from models is one of the uses of MDA. Automation enables rapid response to changes, increases the efficiency of software development and decreases its cost.

UML is an OMG standard for modeling systems and it provides a rich representation for different aspects of any under development software system. Behavioral models in UML complement the static models and provide the full picture of the system. Along with other OMG standards such as MOF, QVT, OCL the automatic code generation from models can be feasible.

UML state machine diagram is a behavior diagram that is used to visualize, specify and construct various dynamic aspects of a designed system through nodes (states) and edges (transitions). Models can be constrained by adding constraints to them using OCL which has its own limitations.

Code generation from state machine specification is one of the most challenging tasks. Because there is a gap between the modeling languages and the programming languages. Another reason is the dynamic nature of the state machine. Additionally, concepts such as states and events are not directly supported by most of the object oriented programming languages.

Research Methodology

Overview

This chapter presents the research methodology applied in order to complete this research. There are two case studies were conducted. Those are briefly described. The tools, languages, environments which were used are identified and described in this chapter too. The chapter also depicts the main steps for both case studies.

Case Study Methodology

The research methodology is basically based on (Hevner et al. 2004) to build and evaluate system techniques and methods iteratively and incrementally based on cases. A *language*, *model* or *guidelines* are to be identified in order to define the method under study. The strategy of the research is based on case study. (Creswell 2012) define case study as "*researcher explores in depth a program, an event, an activity, a process, or one or more individuals*" (p. 15). Leedy and Ormrod (2009) stated that , using case studies the researcher is attempting to learn "*more about a little known or poorly understood situation*" (p.149).

Brief Description of the Proposed Case Studies

First we have started by a case study that examines and tests the issues and problems when mapping Behavior models from PIM to PSM. We had reported on that in (Ahmed et al. 2013) by giving examples from real world and trying to highlight the importance of behavior in models. We concluded that a concrete method to map the behavior models (state machine) from PIM to PSM was still an open question. Eventually using a case study to raise issues helped us to understand how to tackle the problem and to improve model driven development. The second case study provides a concrete detailed and practical way of finding solutions to the issues raised. It describes an end to end model driven software development that incorporates both static and behavioral models with more focus on behavioral part. More details are provided in next chapters.

Languages and Tools Used in Case studies

In order to carry on the various steps in the two case studies some tools and languages were needed. The tools and languages were chosen because they are OMG standard or they comply with OMG standards. A modeling language , an IDE to create models that allow also the exchange of models in a standard way, a transformation language and an IDE that supports instances generation are specified in the sections below.

UML 2 Metamodel

The Unified Modeling Language™ - UML - is a specification and standard from the Object Management Group OMG's. It is used to model application structure, behavior, and architecture, business process and data structure. UML and Meta Object Facility (MOF) are corner stone in MDA. After its first release versions as UML 1.x, UML has gone through various improvements. UML arrived at version UML 2.x. specification that had four parts: UML Superstructure (OMG 2011b) for diagrams and elements description , UML Infrastructure (O M G

2011) that defines the core metamodel on which the Superstructure is based, the Object Constraint Language (OCL) (OMG 2010) for defining constraints for model elements and UML Diagram Interchange that defines how to exchange the diagrams.

Software system exhibits two characteristics:

- *Static (structural)*: Logical Structure, e.g., relationship between classes, attributes of a class, etc. UML provide the use case and class diagram for describing system static structures.
- *Dynamic*: Behavior of the system, e.g., how to respond to a certain event, how to initiate an action, etc. This view includes sequence diagram, activity diagram, state machine diagram, Object diagram and collaboration diagram.

Magic Draw

Magic Draw UML Personal Edition 16.5 SP 1 from No Magic, Inc was used. It is a development tool that facilitates analysis and design of Object Oriented (OO) systems and databases (Anon n.d.). Designed for business and software analysts, programmers, and Quality Assurance engineers. Major MDD vendors recommend using it in the world of Model Driven Architecture. It is used to create, visualize edit and export various UML diagrams including class, state, activity, package, and UML metamodel for PIM and PSM.

XMI

The XML Metadata Interchange (XMI) (OMG 2014d) is a standard and a trade mark for OMG. It is a framework for defining, interchanging, manipulating and integrating XML data and objects. It is mainly used as interchange format for UML tools and to integrate tools, applications, repositories and data warehouses. XMI also defines rules for schema definition and the rules for metadata generation of document production — how is a model mapped onto text.

Magic Draw tool has the support for XMI 2.x in many options. One option is to store native files in XMI format. Another option is to import from XMI and to export UML 2.x models to XMI. In the case studies the PIM and PSM models were exported as XMI documents to integrate them into the Eclipse Modeling Framework to further transform them.

QVT

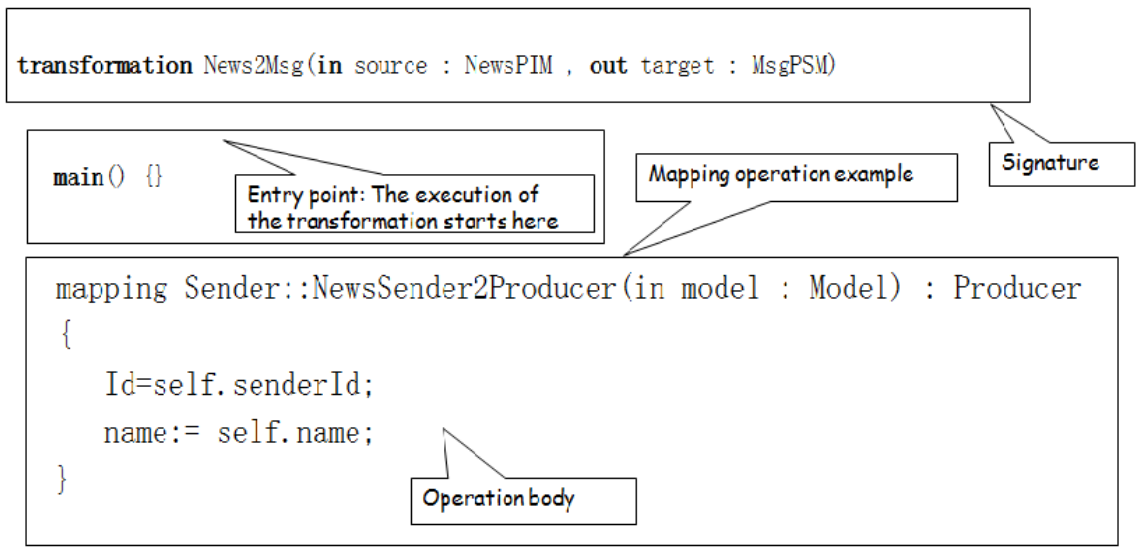
QVT (OMG 2011a) is another standard set of languages from OMG to Query, View and Transform models. QVT standard defines three languages: QVT-Operational, QVT-Relation and QVT-Core. Model transformation is a program which operates on models and contains transformation rules with model elements to be matched and transformed.

Figure 3 QVT Operational Context

In Figure 3 QVT Operational Context the abstract syntax of the language is defined as MOF 2.0 metamodel. The program of the transformations (T_{ab}) are defined on the base of (MM_a, MM_b) metamodels. Transformations are executed on instances of MM_a metamodels (M_a) in order to produce instances of MM_b metamodels (M_b).

Transformation as depicted by Figure 3 Structure of A Simple QVT Program can consists of mapping operations that form the

transformation logic. A mapping operation maps one or more source elements into one or more target elements. It matched the source elements on the base of a type and executes operations in its body to create target elements



```
transformation News2Msg(in source : NewsPIM , out target : MsgPSM)
```

```
main() {}
```

```
mapping Sender::NewsSender2Producer(in model : Model) : Producer
{
   Id=self.senderId;
   name:= self.name;
}
```

Signature
Entry point: The execution of the transformation starts here
Mapping operation example
Operation body

Figure 3 Structure of A Simple QVT Program

In the case studies a transformation program was written to transform the PIM instances model into the PSM instances model.

Eclipse Modeling Framework EMF

Eclipse is an Integrated Development Environment (IDE) written in Java programming language and can be used to develop applications (The Eclipse Foundation n.d.). The Modeling project in Eclipse (EMF) contains projects that focus on model-based development technologies and provide modeling and code generation facilities. Models as an input to EMF can be specified as UML or XMI documents then imported into the framework. From the specified model document, EMF will generate Java classes for the model along with adapter classes to instantiate them. Beside that an editor is generated to manipulate model elements. The meta model for EMF is Ecore which is a reference implementation of the OMG's simplified version EMOF (Essential Meta-Object Facility). An extract of a small part of the metamodel is shown in

Figure 3 Ecore EMF's Metamodel Sample (The Eclipse Foundation n.d.) Applications might consider using Ecore or defining their own metamodels based on it.

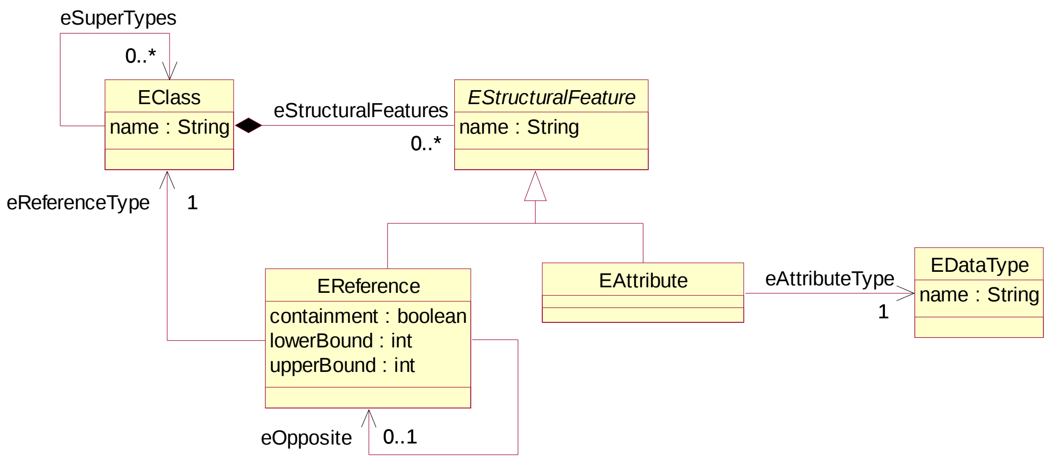


Figure 3 Ecore EMF's Metamodel Sample (The Eclipse Foundation n.d.)

Case Studies Main Steps

The case studies are an attempt to develop an entire small but rich enough application to illustrate the MDA approach. The PIM and the PSM are developed as UML2 class’s model, with the dynamics developed using UML2 State machine and Activity models. The developed models are not claimed to be the best but they were selected and modeled because they have facilities to exercise the proposed method, familiar to the developers and readers and big enough and not trivial.

1. Input Models

   a. PIM meta model

   First we analyzed and build a model with a high level of abstraction for a software system. The PIM is augmented with structural model (Class Diagram) and behavioral model (State machine Diagram).

   b. PSM meta model

In this step the meta models of a chosen platform specific model (PSM) of a software system are analyzed and modeled. A PSM is tailored to specify the system in terms of the implementation constructs that are available in one specific implementation technology. The PSM is augmented with the structural model (Class Diagram) and behavioral model (State machine Diagram).

2. Transformation in the first case study

The relationship between the PIM and PSM constructs were investigated and studied. A manual mapping(not by tools) was conducted to identify the relationships among structural features. For the behavioral features the process is carried out by creating the equivalence classes between PIM states and PSM states for particular objects. In mathematics when an equivalence relationship exists in a set , this denotes the natural grouping of elements related to each other. The issues raised by conducting the first case study were discussed and reported to the research community in (Ahmed et al. 2013).

Two stages of transformations were suggested, the first was the transformation from the PIM to the PSM and the second stage was the translation from PSM to code. The second stage is a rendering of the output into code and code alike constructs as prove of concepts. The benefit of the first case study is to check the feasibility of conducting more investigations and research on the topic. Moreover, to gain confidence on applying a more concrete case study and use the proper languages and tools.

3. Transformation in the second case study

The second case study provides a concrete detailed and practical end to end model driven software development that incorporates both static and behavioral models. The main idea is to model the structural and behavioral features of the news system in a PIM and to model the common features of a messaging system in a generic PSM. The models were prepared as step 1 above -defining the input- suggested. The generic messaging system enables the news application to be implemented in more than one platform such as Sun’s Java Messaging Service (JMS((Oracle 2013) Microsoft’s MSMQ, or IBM’s MQSeries. The PIM to PSM transformation was conducted using OMG's QVT transformation language. The final step is to transform a PSM to code. The complex step is the one in which a PIM is transformed to a PSM. The Apache Active MQ implementation of JMS is chosen as the execution environment for the resulting software system.

Summary

The research methodology used was the case study methodology. The case study approach facilitate the exploring and examining of the case under the study. Moreover provides a way to conduct a detailed solution to the issues raised while learning about new or poorly understood situations.

The models and meta-models are developed using UML 2 specifications in an environment called Magic Draw. The Eclipse Modeling Framework (EMF) provides a modeling and code generation framework.

The tools for developing and executing transformations are based on the Eclipse M2M project. This implementation is not completely finished and contains some bugs. In addition, the available documentation and tutorials about QVT are a bit limited ,not always clear or as practical as could be. Because of these limitations it can be difficult to make an optimal transformation, however the environment is certainly suitable to model easy to average transformations. It is very likely the environment will be able to manage more complex transformations in the near future, because it has a high potential.

Two case studies were conducted and research publications on initial results and issues were reported to the research community in (Rihab Eltayeb Ahmed 2012) and (Ahmed et al. 2013)

Case Study: Financial System Services

Overview

This chapter introduces the first case study of a financial system that is used to develop an application using MDA paradigm. The system is taken as an attempt to develop an entire application to illustrate the MDA approach. The PIM and the PSM are developed as UML2 classes model, with the dynamics developed using UML2 state machine. The observations and issues raised by the case study are identified and discussed.

Models of the System

In this case study we begin by the design of an object oriented automated teller machine (ATM) software system for a major bank. The requirement document of the system determines what functionality the system must include. For simplification and scope we are not going through the details of the document rather we limit our design to the basic financial transactions each ATM is capable of. Each user can have one account at the bank. ATM users can view account balance, withdraw , deposit and transfer money between accounts and more.

Financial System PIM

The UML class diagram in Figure 4 Financial Services PIM Class Model shows the implementation classes for the financial service system at the PIM level with default values for class attributes. It shows the internal structure of the system with the essential details at this stage. From a structural point of view, there is: a customer (**BankCustomer** class) with a specific bank account (**BankAccount**) and who is a holder of an ATM card (**ATMCard** class). With the ATM card the customer can benefit from various services such as getting the remaining balance, withdraw some amount of money, transfer fund between accounts, buy mobile credit and prepaid electricity. These services are concrete subclasses of the abstract (**Service** Class).

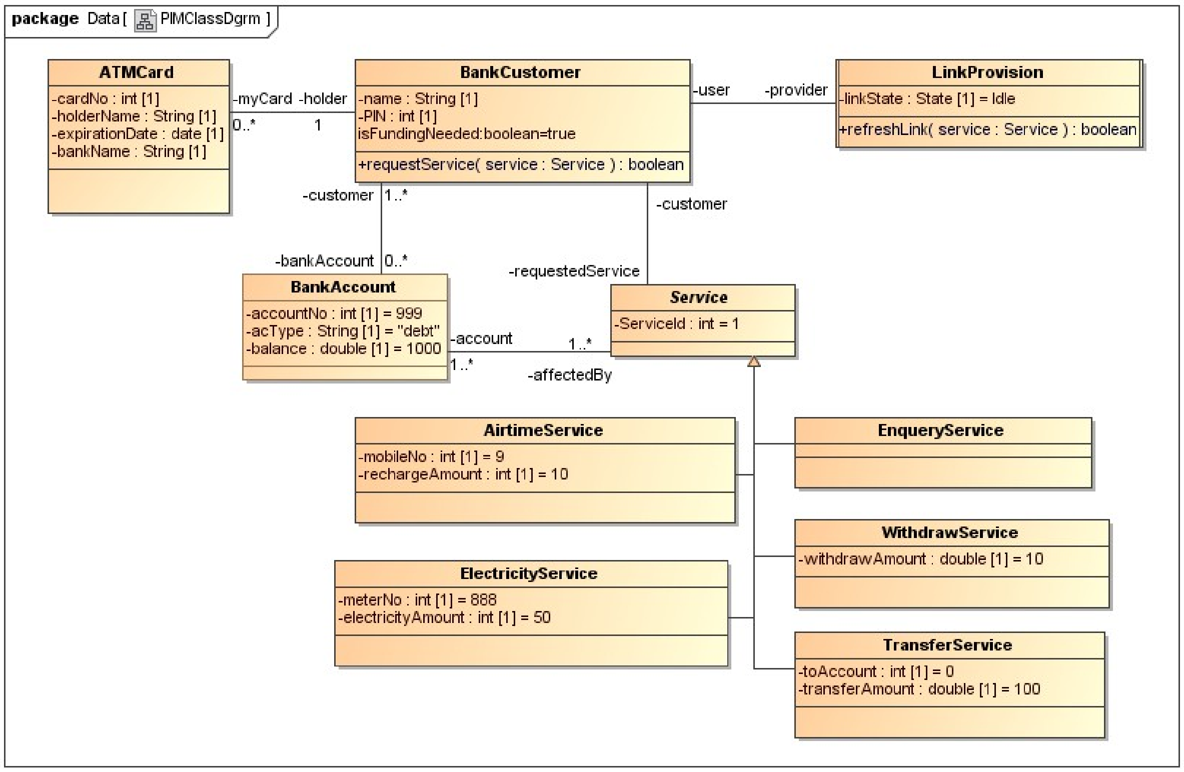


Figure 4 Financial Services PIM Class Model

LinkProvision is the provider of the service which a customer can request. Its behavior is specified by a state machine diagram. LinkProvision is Idle by default as indicated by the linkState attribute in the class diagram. To put LinkProvision in the proper state to serve the customer, a customer will demand a service by setting its own attribute isFundingNeeded to true. That is going to set the isFundingNeeded guard to true, hence triggering the transitions of LinkProvision from “Idle” to “validateUser” state as in Figure 4 State Machine Behavior Diagram of PIM LinkProvision Class.

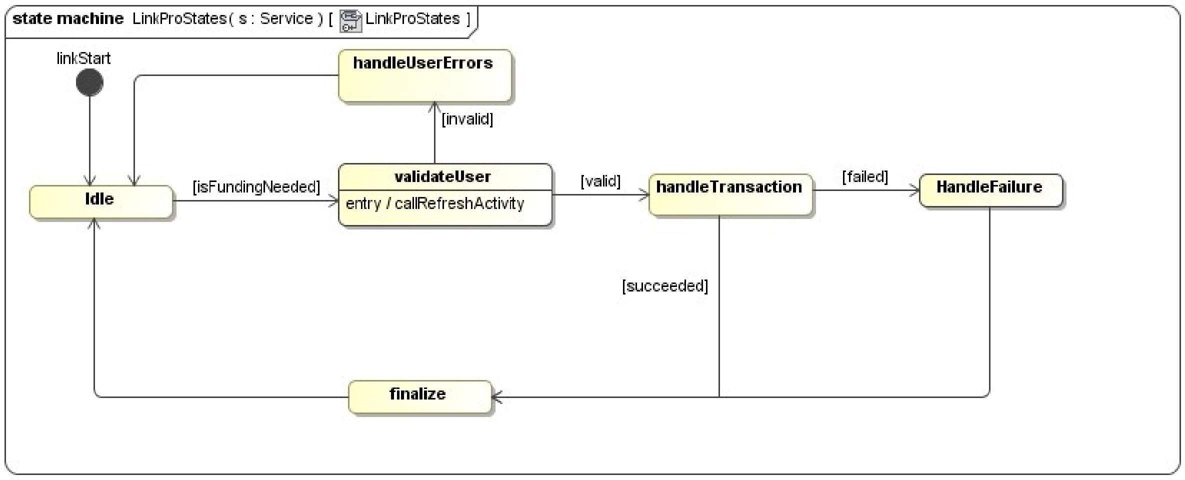


Figure 4 State Machine Behavior Diagram of PIM LinkProvision Class

A Customer is the initiator of the service he demands. The LinkProvision is the provider for the service by which the Customer is got served. The Customer has the state "Requesting" as in Figure 4 BankCustomer States that upon entry will call the requestService operation of the Customer. The behavior of the requestService operation is specified by an activity as in Figure 4 RequestService Operation Behavior that in turn creates a BehaviorAction call to put the LinkProvision in the proper state to serve the Customer. This Behavior Action call will set the isFundingNeeded guard to true, hence transitions the LinkProvision to "validateUser" state as in Figure 4 State Machine Behavior Diagram of PIM LinkProvision Class. Upon entry of the "validateUser" there is a call to refreshLink operation. The implementation of the refreshLink is provided by the ATM in the PSM models.

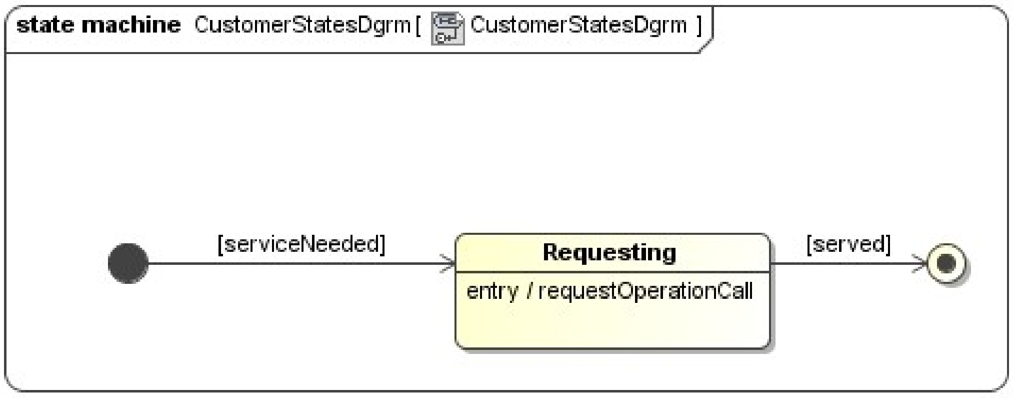

state machine CustomerStatesDgrm [CustomerStatesDgrm]
[serviceNeeded]
Requesting
entry / requestOperationCall
[served]

Figure 4 BankCustomer States

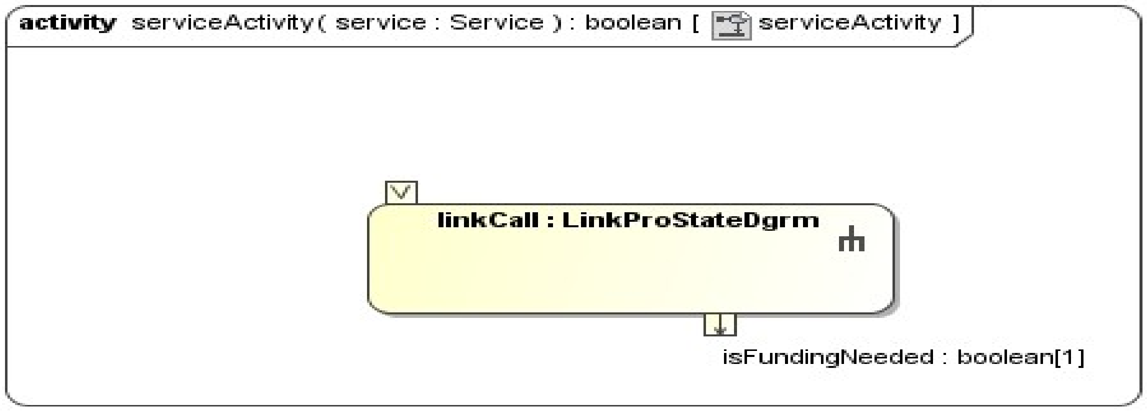

activity serviceActivity(service : Service) : boolean [serviceActivity]
linkCall : LinkProStateDgrm
isFundingNeeded : boolean[1]

Figure 4 RequestService Operation Behavior

PIM Class Model Instances

The Object diagram in UML shows possible configurations of instances of the class diagram. Similar to the class diagram it shows the static view of the system but this static view is a snapshot of the system at a particular moment. The class diagram is abstract while the object diagram is more concrete because it is more close to the actual behavior of the system.

UML specify that a *Classifier* can have zero or more InstanceSpecification that describe its instances. These instances are considered level zero instances. The diagram in Figure 4 Extract from

The UML Kernel Package below is the instances diagram from UML superstructure specification v2.4.1.(OMG 2011b).

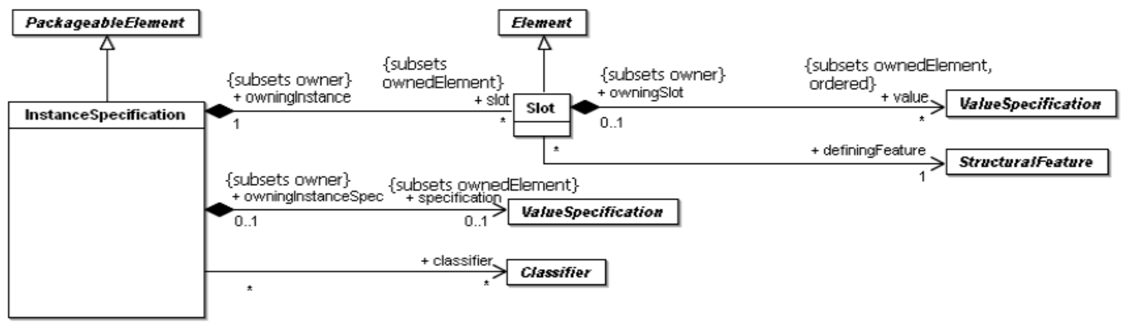


Figure 7.8 - Instances diagram of the Kernel package

Figure 4 Extract from The UML Kernel Package

According to the specifications in Figure 4 Extract from The UML Kernel Package, the object diagram in Figure 4 PIM Model Instances can show an object's classifier (e.g. ATMCard class) and instance name (e.g. card1), as well as attributes and other structural features using slots. Each slot corresponds to a single attribute or feature, and may include a value for that entity. For example the accountNo attribute with the value 9999. Figure 4 PIM Model Instances instantiates the model presented in the class diagram in Figure 4 Financial Services PIM Class Model.

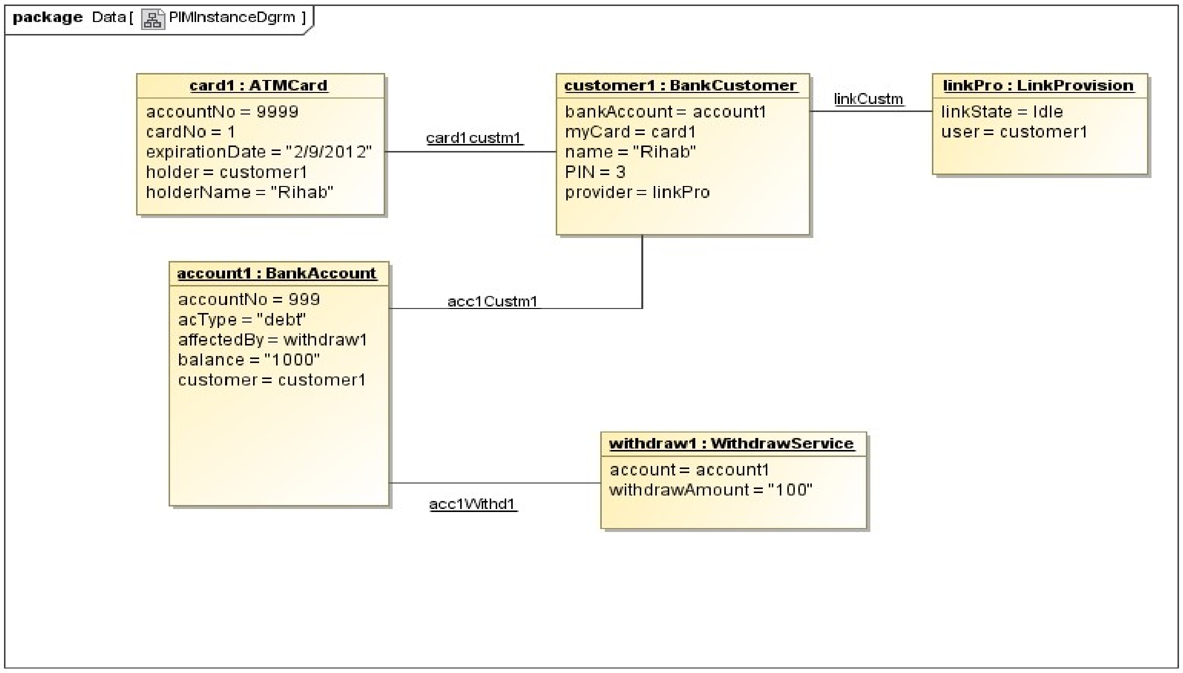


Figure 4 PIM Model Instances

PIM Behavior Model Instances

A state machine is a *Behavior. Behavior* is an abstract class that inherits from the concrete class *Class* that also inherits from *Classifier* from kernel package in UML superstructure (OMG 2011b). Hence a state machine is a Classifier and can have instance specification to represent it an object. A state is a concrete sub class of the abstract super class *Vertex.* Each implementation of a vertex can have a name because it inherits from the class *NamedElement.* Since a state is not a *Classifier* neither one of its super classes then a state cannot have instances (at level zero). A state named "Idle" is a level one (meta model M1) instance of the State meta class and cannot be represented in an object diagram. This can be considered a limitation because the states are not represented in most of the tools editors that represent the exported models. Using the Eclipse Modeling Framework, and when trying to show the XMI file contents diagrammatically, all the states were lost. The only thing that represented was the state machine itself.

Another important factor was that EMF is based on Ecore which is a simplified representation of UML.

Amendments to the state diagram model in UML or Ecore can solve the issue. A model element that has an instance specification with an association to the State class can be a valid solution. A solution is presented in (Eric Cariou n.d.), they extended the UML meta model for the state machine and provided the OCL constraints on that as shown in Figure 4 UML Meta-model Extension for State Machine Instance Specification (Eric Cariou n.d.).

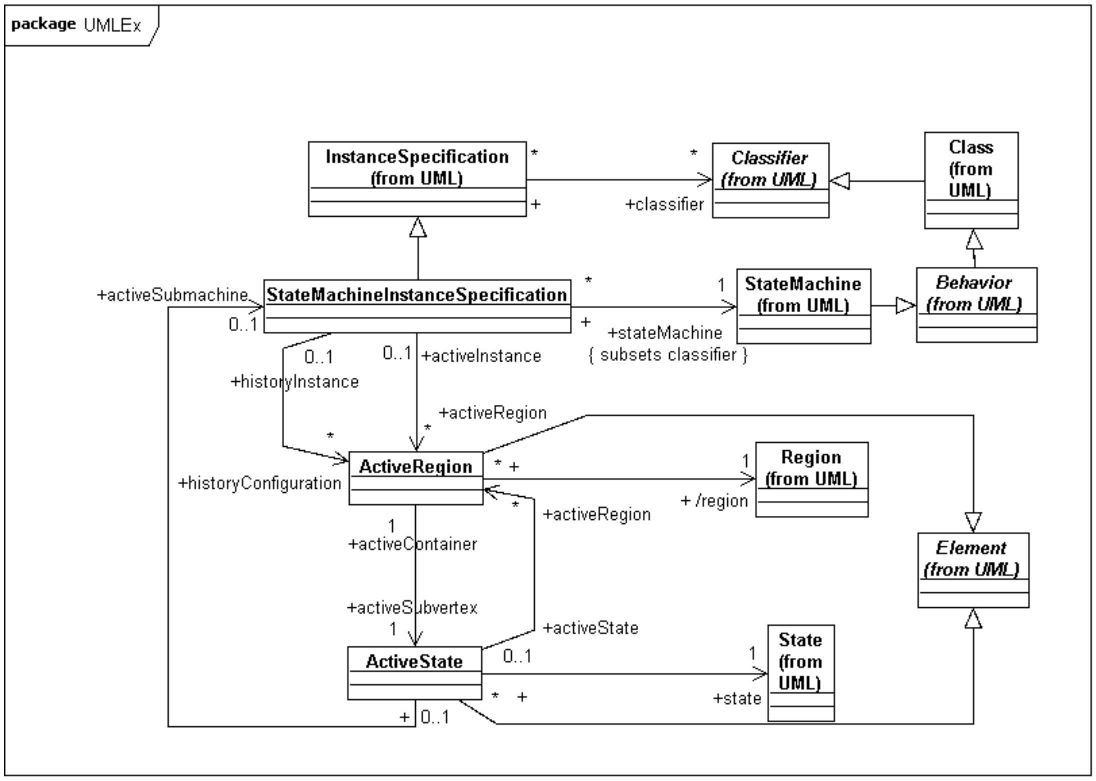


Figure 4 UML Meta-model Extension for State Machine Instance Specification (Eric Cariou n.d.)

Financial System PSM

An ATM machine model is used as a PSM for the financial system. The detailed class model can include two parts. The first is the ATM hardware and how that is managed, and the second is the banking part related to achieving the financial services. The **ATM** class in the PSM class model is associated with a **DeviceManager** class through which it manages the composed financial devices. Financial devices have in common attributes and operations inherited from the base class **FinancialDevice.** Classes such as **CardReader**, **Display**, **CashDispenser** and **ReceiptPrinter** are sub classes of the **FinancialDevice** super class, each of which is specialized in facilitating part of the **ATM** job. This part is modeled in UML and can further be implemented for example through the J/XFS Java eXtensions for Financial Services for the JavaTM platform. J/XFS provides a set of standard Java interfaces in support of the input/output peripheral devices used in the finance industry such as Cash Dispenser, Recycler and ATM Interface (Members 2004).

In the second part of the PSM class model shown in Figure 4 PSM Class Model of an ATM with regard to the financial services, a session (**ATMUserSession**) will be started for an inserted card (**Card** class) inside the ATM machine. A session is associated with a bank (**Bank** class) through a connection (**BankConnection** class) to provide a channel for reflecting the user selections back to the bank account (**Account** class). An account is associated with one or more transactions (**Transaction** class) and a transaction can affect one or more accounts.

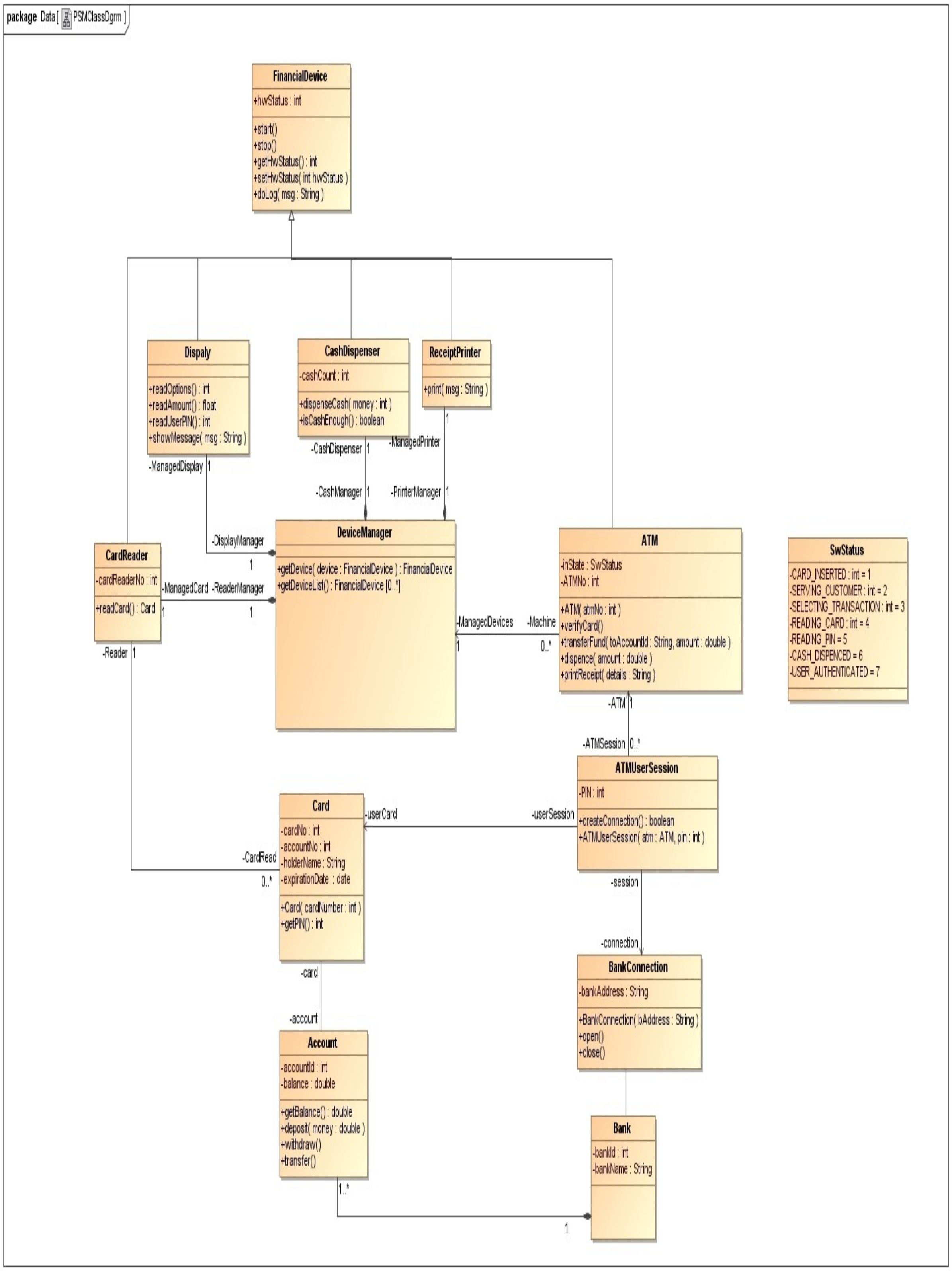


Figure 4 PSM Class Model of an ATM

The specification mechanism used to specify the behavior of the ATM class is the state machine in Figure 4 ATM State Machine Diagram. When the guard cardInserted is true, the ATM state will change from "Idle" to "verifying". According to the verification result, the ATM state would change from "verifying" to either "servingCustomer", "retainCard" or "Failed" state. The servingCustomer state is a composite state that has sub states to describe its behavior shown in Figure 4 Substate Machine Behavior of performingService Composite State

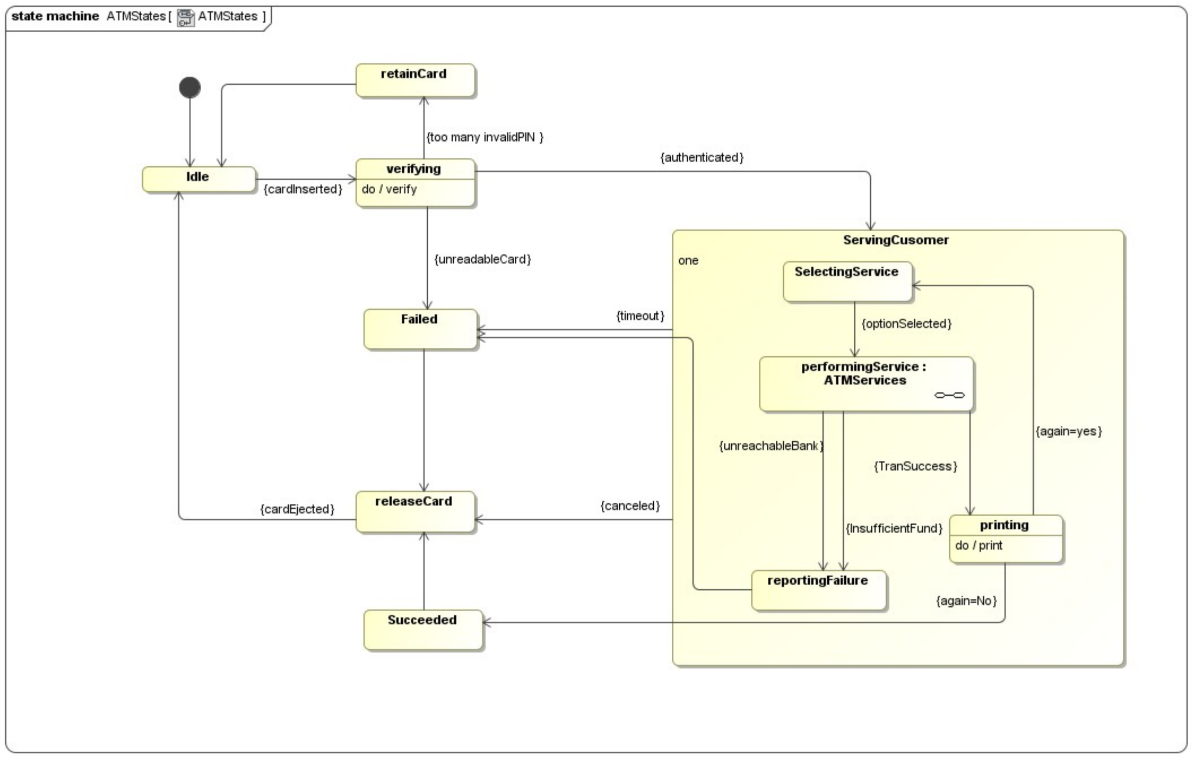


Figure 4 ATM State Machine Diagram

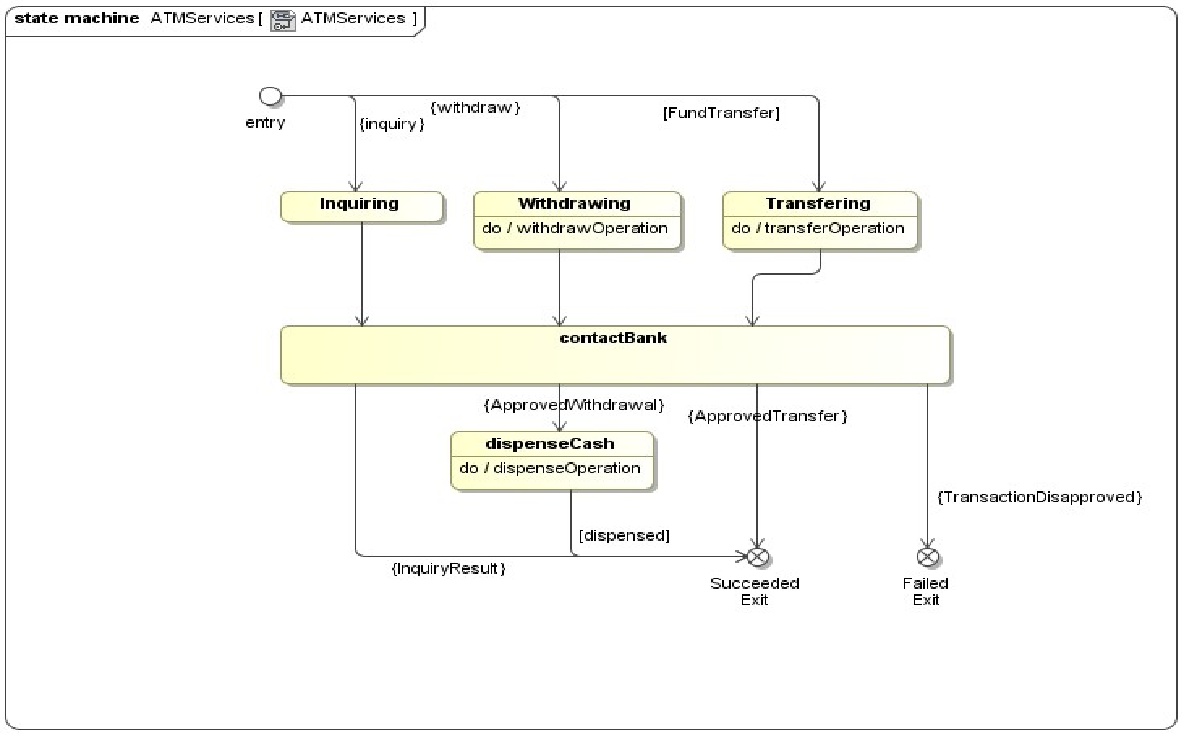


Figure 4 Substate Machine Behavior of performingService Composite State

Suggested Mapping Process

The mapping between PIM model and the PSM model have to be identified in order to generate the application. Given the PIM class model and the PIM instances model, the mapping expressed as transformation rules is going to generate the PSM instances.

Table 4. PIM to PSM Class Model Mapping represents the mapping between class models from PIM to PSM.

Table 4. PIM to PSM Class Model Mapping

PIM	PSM	Using
BankCustomer	ATMUserSession	PIM BankCustomer - PIN
ATMCard	Card	
BankAccount	Account	
LinkProvision	ATM	
-	DeviceManager	PSM Specific
-	CardReader	PSM Specific
-	Display	PSM Specific
-	CashDispenser	PSM Specific
-	ReceiptPrinter	PSM Specific
-	Bank	PIM ATMCard-bankName
-	BankConnection	PSM Specific
WithdrawService	-	PIM Specific

g

To map the behavior models we assume that the following rules are known:

1) *PIM Idle is equivalent to PSM Idle*
2) *PIM isFundingNeeded is equivalent to PSM cardInserted*
3) *PIM valid is equivalent to PSM authenticated*
4) *PIM succeeded is equivalent to PSM tranSSuccess*
5) *PIM invalid is equivalent to PSM tooManyInvalidPins*
6) *PIM failed is equivalent to PSM unreadableCard*

We have two state models, A as in Figure 4 State Machine Behavior Diagram of PIM LinkProvision Class and B as in Figure 4 ATM State Machine Diagram. Model B belongs to the PSM, so is an implementation of Model A, which belongs to the PIM.

Assume, as in the case study, that there are fewer states in model A than in model B, and that every state in model B corresponds to exactly one state in model A. This means that we can divide the states in model B into groups, indexed by the state of model A they correspond to. If *a* is a state of A, then the states of B corresponding to *a* form an equivalence class. Let's call that B(a).

If given an a state of A, there is one state in B(a), then call that state b, and the PIM/PSM mapping maps a into b. If given an *a* state of A, there are several states in B(a), then map *a* to those states if possible. The mapping is **one state to many**.

Assuming that every state of A corresponds to at least one state of B. Otherwise, the **orphan state** of A cannot be implemented. This could form part of an evaluation of the suitability of a PSM for implementing a given PIM. What happens if there are states in B that don't correspond to some state in A? Let's call them **forbidden states**. Whether this is a problem would depend on whether the transitions mapped from A ever take a state in B to the forbidden state.

The mapping process would begin by first mapping the transitions and constrains (guards). Then construct the concrete mapping from the PIM

state model to the PSM state model according to the above framework. We can entail the following using rule 1:
[idle]={ idle}

The PIM idle state will transition to validateUser when isFudingNeeded is true and idle will transition to verifying in the PSM model. Using rule 2 validateUser can be mapped to verifying forming an equivalence class as follows:
[validateUser]={ verifying }

using rule 3:

[handleTransaction]= {servingCustomer (composite state) }

using rule 5:

[handleErrors]={ retainCard}

using rule 6:

[handleFailure]={ failed}

using PIM state model the state handleFailure transitions without a guard to finalize state, the same is true for the path from state failed to releaseCard in the PSM. We can entail that finalize can be mapped to releaseCard

[finalize]={ releaseCard}

Using 4: PIM.succeeded is equivalent to PSM.tranSuccess

We can propose to map the PSM Succeeded state as follows:

[handleTransaction]= { servingCustomer, succeeded}

The problem here is that the guard condition again=No is not explicitly mapped to any guards from the PIM. Hence the Succeeded State can also be considered as forbidden.

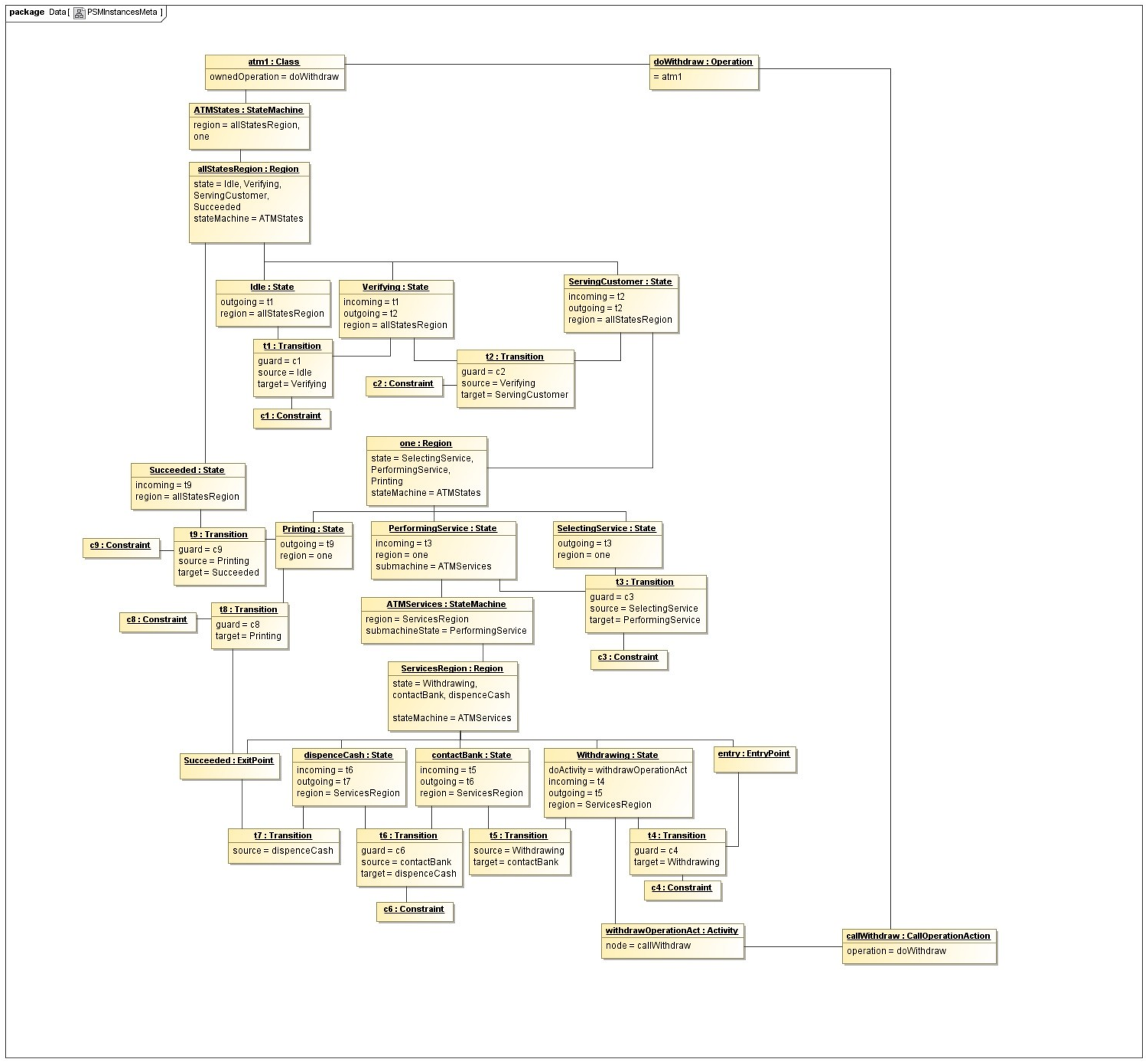


Figure 4 PSM Model Instances mapped Manually - Part of the behavior for withdraw Service

the PIM into a working program

Apparently the state machine depicts the flow of control an object has. In the context of the case study what we need is the application flow of the PIM instances hence their behavior and how that is achieved through PSM instances.

If we concentrate on the PSM instances which are mapped manually , especially the guards and operation calls, we get a sequence of calls guarded by conditions.

We are going to render the state machine of PIM and PSM as following:

- A state is rendered as a comment with state name. example //** Idle **//
- A guard condition is rendered as “Evaluate “+ guard specification
- A do action of a state is rendered as “Call ”+ the operation if the action is a CallOperationAction type.

Table 4. Rendering of Behavior Instances

PIM instance Behavior	PSM instance Behavior
//** Idle **// Evaluate isFundingNeeded If true //**Active**// Call refreshLink(s:Service):boolean Evaluate isSucceeded If true //** succeeded **// Evaluate isFailed If true //**failed **//	//** Idle **// Evaluate cardInserted If true //** verifying **// call verifycard() Evaluate too Many invalidPINs If true //** retainCard**// Evaluate unReadableCard If true //** failed**// //** releaseCard**// call

	ejectCard() Evaluate authenticated If true //**ServingCustomer **//
<u>PSM instance behavior Rendering Options</u>	<u>PSM instance Behavior To Java Code</u>

Since the transformed state machine is a program, it is better to represent it <u>visually</u> as
• Activity Diagram
	If (cardInserted)
	{ verifycard(); return ;}
	If(tooManyInvalidPINs)
	{ retainCard(); return ;}
	If (unReadableCard)
	{ ejectCard(); return }

The result of the rendering is what we can call a high level algorithm, in other words a high level program specified as model elements. The goal is to verify that the state machine mapping can generate an application (from PIM to PSM) that can further be transformed to code (PSM to code), yet applying the MDA concepts.

and Interpretation

A state machine can change from one state configuration to another in a response to an occurrence of an event through a transition. A transition may be associated with at most one guard which is a constraint that controls the firing of the transition. The guard is evaluated when the event occurred, if The evaluation result is true ,the transition is enabled or else disabled (O M G 2011)

UML as a modeling language defined some constraints to impose restrictions on various models and model elements. A user defined constraint (in our case a guard) is often expressed as a text string in some language including natural language as the two figures Figure 4 The Elements defined in the Constraints UML Package (O M G 2011) and Figure 4 The Element defined in the Expressions UML Package (O M G 2011) depict. As a result the syntax and interpretation of the constraints are out of the UML scope and they are language and tool dependent. If a formal (machine readable) language such as OCL is used, then tools may be able to verify some aspects of the constraints. OCL is

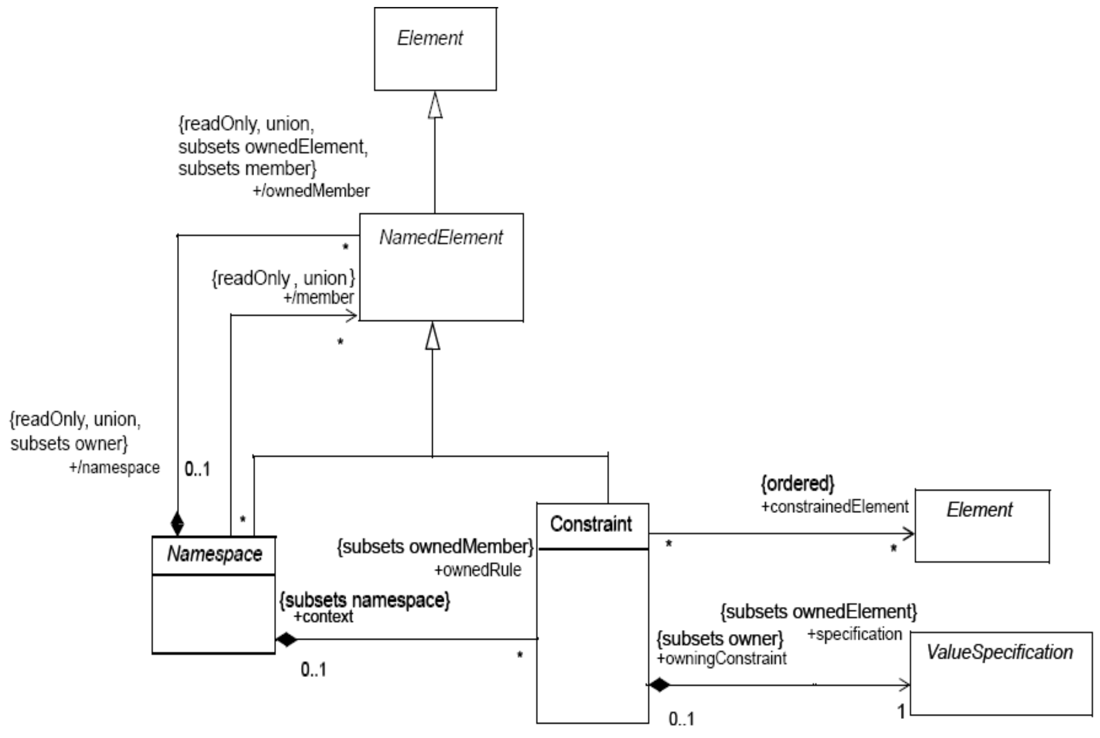


Figure 9.13 - The elements defined in the Constraints package

usually but not necessarily used to constrain a model.

Figure 4 The Elements defined in the Constraints UML

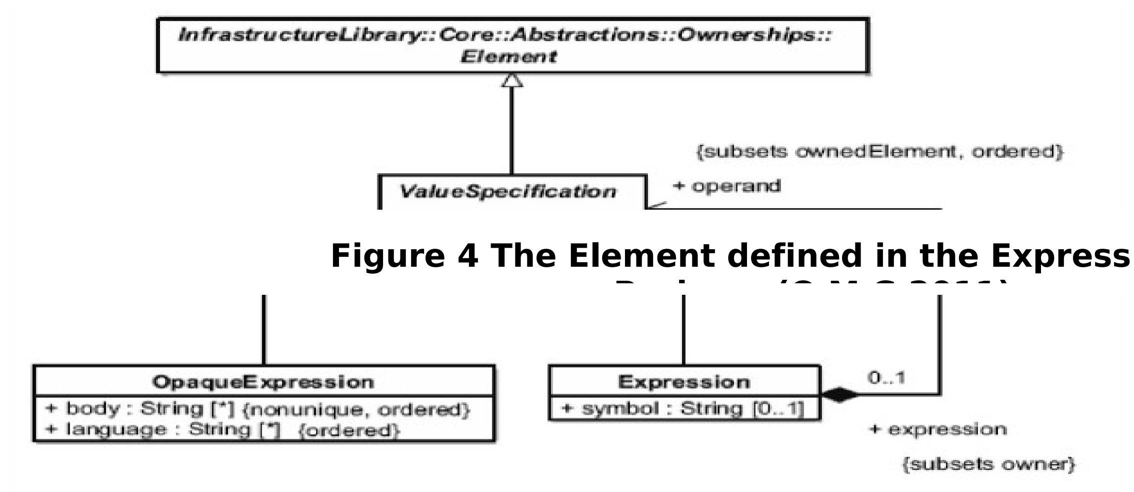

InfrastructureLibrary::Core::Abstractions::Ownerships::
Element
{subsets ownedElement, ordered}
+ operand
ValueSpecification
OpaqueExpression
+ body : String [*] {nonunique, ordered}
+ language : String [*] {ordered}
Expression
+ symbol : String [0..1]
0..1
+ expression
{subsets owner}

Figure 4 The Element defined in the Expressions UML

Figure 9.20 - The elements defined in the Expressions package

Another aspect of guards is that they should not include expressions causing side effects (OMG 2011b) . Being side effect free means that the state of the system will never change because of an expression even though expressions are used to specify such a state change (when true). Specifying constraints will not change elements as well as relationship among elements in the model. The same restrictions applies to an OCL expression. Whenever an OCL expression is evaluated, it simply delivers a value.

As a consequence of the semantic of constraints in the standard UML and even OCL , modelers must firstly transform the constraints into formal language if they want the constraints to have effects at run time. Secondly a language is needed to evaluate the formal constraint and provide an effect which is of a considerable value to the process we are proposing (mapping method).

Observations and Issues

1. The PSM has more states than the PIM.
2. Some PSM states such as ServingCustomer state is a composite state aka a state of states. The same process can be applied to map this kind of states also.
3. The given PSM platform is capable of implementing the PIM specification because every state or transition is mapped to at least one state or transition which indicates that the PSM state machine is indeed a superset of the PIM state machine.
4. A decisions has to be specified for the guard conditions that are not mapped to the PIM ones.
5. A decision is needed for the forbidden states.
6. Mapping the constraints to each other involve relating the attributes that are constrained. For example cardInserted is a constraint with a guard condition that checks the boolean property "cardInserted" in the PSM Card class. "fundingNeeded" is a property of the class BankCustomer in the PIM which is also constrained. BankCustomer is mapped to ATMUserSession in the class model mapping . Each ATMUserSession is associated with a user card of Card class. So the relation between the two

constrains, the PIM one and the PSM's involves the mapping of PIM class model to the PSM class model first.

7. The PIM doesn't express explicitly that the successful completion of a transaction would result in a print of a receipt describing the transaction. The PIM and PSM state machines are structurally different, the PSM had additional states and logic. By mapping the constraints , the states and transitions ends up in the idle state of the PIM state machine, while it continues to print and release the card in the PSM. In this situation we are going to map the additional PSM states to the PIM one in order to complete the application logic.

Summary

Some issues were raised by attempting the development of an entire small application using the MDA approach. A service in a financial system is taken as a case study. The PIM and the PSM are developed as UML2 classes model, with the dynamics developed using UML2 State machine and Activity models.

One of the issues is the guards representation and interpretation. A state machine can change from one state configuration to another in a response to an occurrence of an event through a transition. A transition may be associated with at most one guard which is a constraint that controls the firing of the transition.

The guards in the PIM state machine are specified using the terminology of the PIM Classes model, while the guards in the PSM state machines are specified using the terminology of the

corresponding PSM Classes models. It is therefore necessary to map the guard expressions from PIM terminology to PSM terminology, using the mapping of the PIM Classes model to the PSM Classes models. The possible mapping of state guards can be carried out by examining the participating instances attributes. Involving instances in the mapping as well as classes is another research issue.
A second problem is that the PIM and PSM state machines may be structurally different. In order for a PSM to implement an application specified in the PIM, its state machine must be a superset of the PIM state machine, otherwise the application's specification cannot be met.

A related question is to be able to test whether a given platform is capable of implementing a PIM specification, which involves testing whether the PSM state machine is indeed a superset of the PIM state machine. One way to use this information is to help select from a number of potential platforms. Another way might be in a circumstance where only a deficient platform is available. Mapping back from the PSM state model to the PIM might help the designers alter the specifications to make them implementable.

Case Study: News Application

Overview

A news software application is taken as an attempt to develop an entire small application to illustrate the MDA approach. The PIM and the PSM are developed as UML2 classes model, with the dynamics developed using UML2 State machine.

The main idea is to model the structural and behavioral features of the news system in a PIM and to model the common features of a messaging system in a generic PSM. The generic messaging system enables the news application to be implemented in more than one platform such as Sun’s Java Messaging Service (JMS) (Oracle 2013)**,** Microsoft’s MSMQ, or IBM’s MQSeries,

Deriving code and implementation from models is one of the uses of MDA and may be fully or partially automated. Automation enables rapid response to changes, increases the efficency of software development and decreses its cost. As Figure 5 Detailed mapping From PIM to PSM and to Execution Environment depicts, the Apache Active MQ implementation of JMS is chosen as the execution environment for the resulting software system.

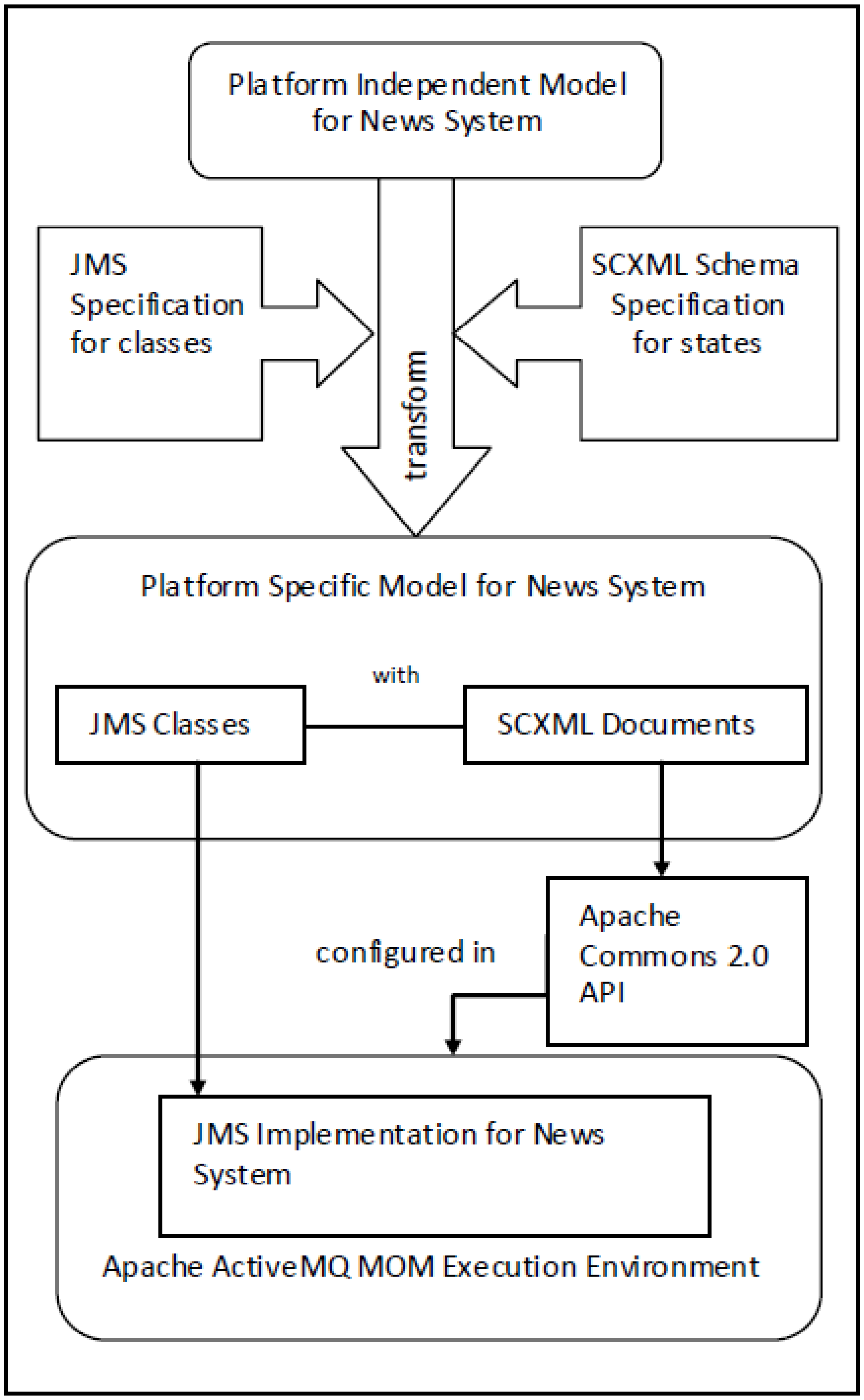

Platform Independent Model for News System
JMS Specification for classes
SCXML Schema Specification for states
transfom
Platform Specific Model for News System
with
JMS Classes
SCXML Documents
Apache Commons 2.0 API
configured in
JMS Implementation for News System
Apache ActiveMQ MOM Execution Environment

Figure 5 Detailed mapping From PIM to PSM and to Execution Environment

Approaching the Problem

In order to generate the software application we tried two different approaches. The first approach is by writing Java code programs that reads in the meta models files and transform them. The second is by writing QVT rules. The first approach is considered as a guidance because of the tools limitations we had faced when trying the second approach.

<u>Major Steps: Part 1 using Java Programs for transformation</u>

PIM

1. Model the News system PIM structures and behaviors
2. Generate Java classes for PIM
    a. Generate PIM instances and serialize them into XMI using EMF
    b. Writing a Java program for structures read from the model exported to EMF frame work.

PSM(s)

1. Model the generic messaging platform PSM structures and behaviors
2. Generate Java classes for PSM
3. Generate PSM instances and serialize them into XMI using EMF
    a. writing a Java program for structures read from the model exported to EMF frame work.
    b. Generate the simple program (send & receive) from the behaviors read from the original Magic Draw file. The exported version of the model in step 3 above does not include the behaviors part.

Transformation

1. Transform the PIM instances to PSM instance to generate Java classes containing both attributes (structure) and methods (behavior) for the system
2. Transform the resulted classes to JMS native classes .
3. Transfer The main program and all the generated classes to the Apache ActiveMQ environment to be executed

<u>Major Steps: Part 2 using QVT for transformation</u>

PIM to PSM, PSM to JMS API

1. Model the News system PIM and the JMS PSM structures and behaviors using Magic Draw 16.5
2. Export the models in Eclipse Modeling Framework EMF to generate Java classes
3. Programmatically generate PIM instances and serialize them into XMI using EMF
4. Write QVT mapping rules to transform the model elements from PIM to PSM
    a. QVT mapping style for structures is UML level zero to UML level zero instances. The transformation uses the PIM , PSM and PIM instances to generate PSM class instances.
    b. QVT mapping style for behaviors is UML level one to UML level one. The mapping is done in more than one step and finally map the state machine diagrams to SCXML document.
5. Export the result and write a Java program to start the execution.
6. Install and configure The Apache Commons inside the Apache ActiveMQ server.
7. Transfer The main program and all the generated classes to the Apache ActiveMQ environment to be executed.

Models Of The System

News System Platform Independent Model

Figure 5 PIM Class Model for News System shows a simple class's model for the news system platform independent model. The classes identified are ***NewsSender*** that sends the message using the method ***writeData.*** The ***NewsReceiver*** class represents a receiver side that receive the news through the ***readData*** method. The message that is

represented by the ***NewsMessage*** class. Each ***NewsMessage*** has a content and a status. The ***DataLink*** class represents the link established in order to send and receive the messages. It is used for delivering the data.

The ***PIMClient*** is an active class. Active classes and hence active objects initiate and control their own flow of behavior, while passive classes store data and serve other classes. Rather than being invoked or activated by other objects, active objects can operate standalone and define their own thread of behavior. In UML, active classes are rendered with a thicker border. The system is represented by the ***PIMClient*** class that controls the other classes. It generates the message, establish the link, call the ***NewsSender's writeData*** method for sending news, call the ***NewsReceiver readData*** method to receive news when available and display them.

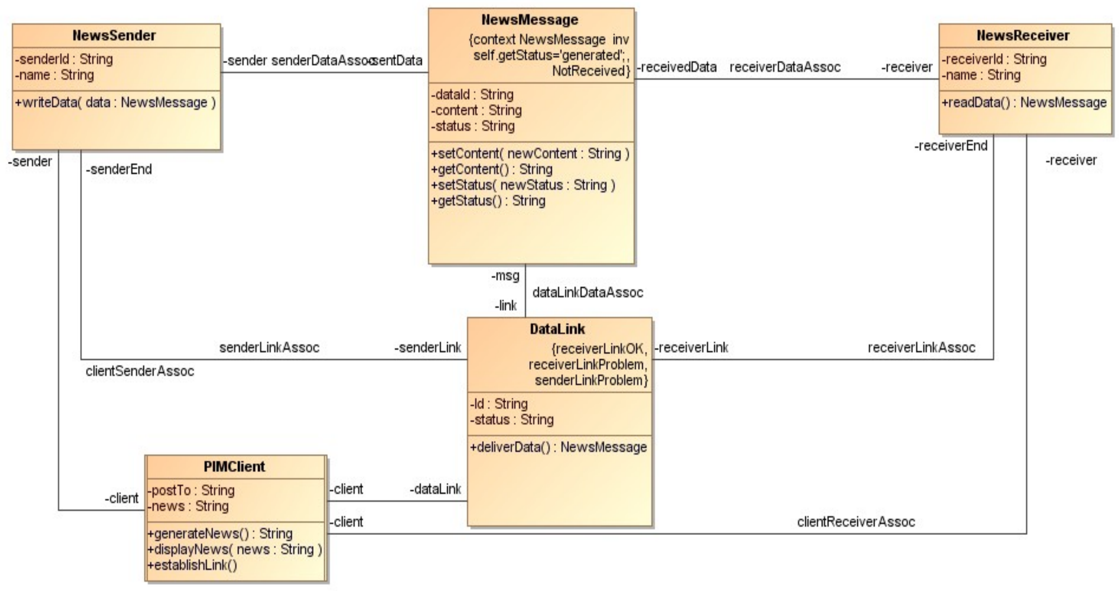


Figure 5 PIM Class Model for News System

Messaging System Platform Specific Model

The Java Message Service JMS (Oracle 2013) provides a common way for Java programs to create, send, receive and read an enterprise messaging system's messages. JMS is a set of interfaces and associated semantics that define how a JMS client accesses the facilities of an enterprise messaging product. Among JMS objectives is to provide portable application across products within the same messaging domain.

The basic building blocks of a JMS application are shown in Figure 5 JMS API Programming Model (Oracle 2013). It consists of Administered objects: Connection factories and destinations, Connections, Sessions, Message producers, Message consumers and Messages. Figure 5 JMS API Programming Model (Oracle 2013) shows how all these objects fit together in a JMS client application.

Figure 5 JMS API Programming Model (Oracle 2013)

The PSM for the messaging system aims to provide a standardized model to send and receive messages in a vendor-neutral manner. It formally defined many concepts and artifacts from the world of messaging:

> ***Client*** - An application modeled to create, send and receive messages.

Producer - A client application that sends messages

Consumer - A client application that receives messages.

Message - The most fundamental concept of PSM; sent and received by clients.

ConnectionFactory - Clients use a connection factory to create connections.

Destination - A generic object to which messages are addressed and sent and from which messages are received.

The two styles of messaging that include point-to-point and publish/subscribe are supported in the PSM. Accordingly there are two types of producers, ***QueueSender*** and ***TopicPublisher***. Two types of destination, ***Queue*** and ***Topic***. Two types of consumers, ***QueueReceiver*** and ***TopicPublisher***.

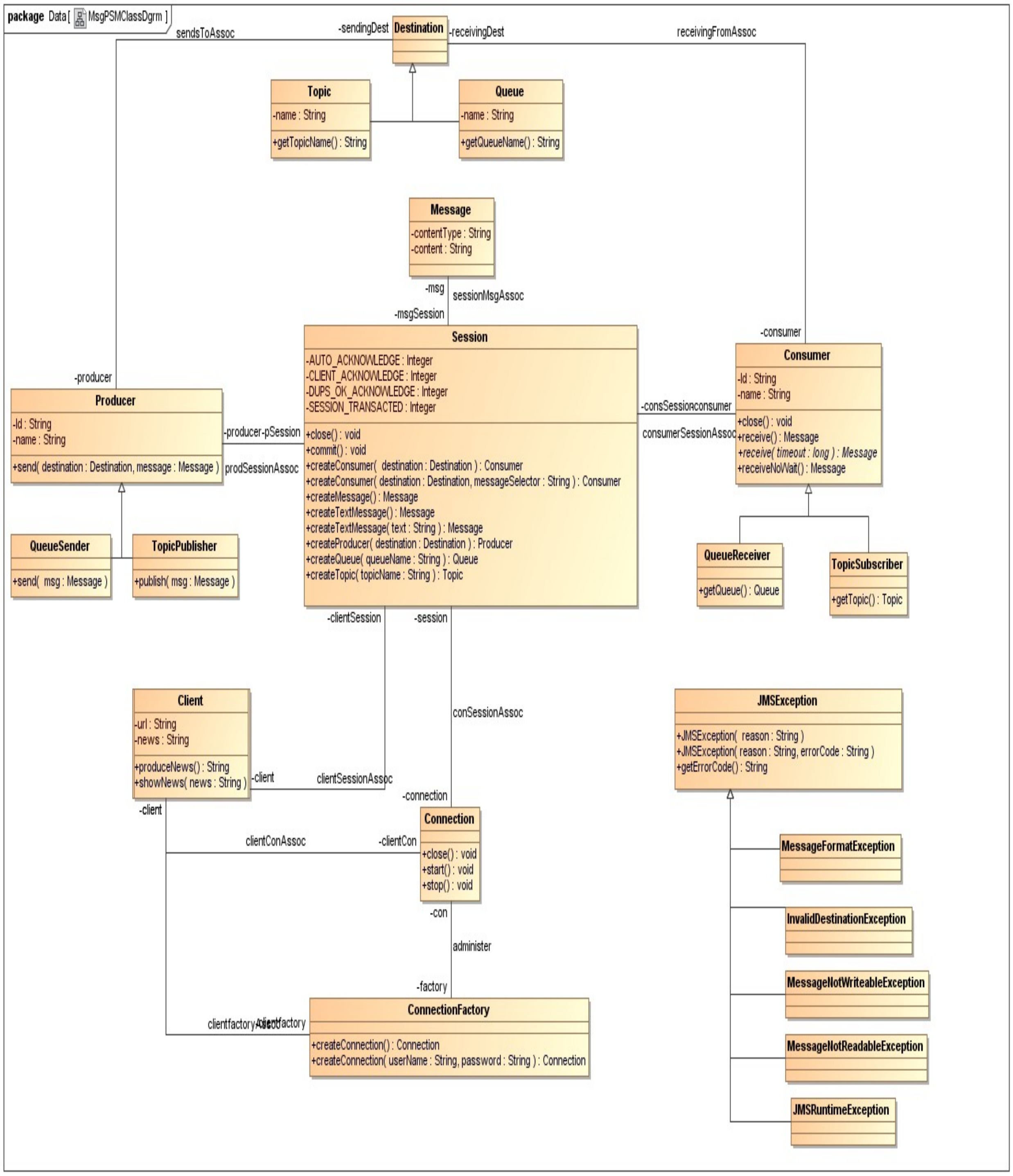

package Data[MsgPSMClassDgrm]
Destination
-sendingDest
-receivingDest
sendsToAssoc
receivingFromAssoc
Topic
-name : String
+getTopicName() : String
Queue
-name : String
+getQueueName() : String
Message
-contentType : String
-content : String
-msg
sessionMsgAssoc
-msgSession
Session
-AUTO_ACKNOWLEDGE : Integer
-CLIENT_ACKNOWLEDGE : Integer
-DUPS_OK_ACKNOWLEDGE : Integer
-SESSION_TRANSACTED : Integer
+close() : void
+commit() : void
+createConsumer(destination : Destination) : Consumer
+createConsumer(destination : Destination, messageSelector : String) : Consumer
+createMessage() : Message
+createTextMessage() : Message
+createTextMessage(text : String) : Message
+createProducer(destination : Destination) : Producer
+createQueue(queueName : String) : Queue
+createTopic(topicName : String) : Topic
-producer
Producer
-Id : String
-name : String
+send(destination : Destination, message : Message)
-producer
-pSession
prodSessionAssoc
QueueSender
+send(msg : Message)
TopicPublisher
+publish(msg : Message)
-consumer
Consumer
-Id : String
-name : String
+close() : void
+receive() : Message
+receive(timeout : long) : Message
+receiveNoWait() : Message
-consSession
consumer
consumerSessionAssoc
QueueReceiver
+getQueue() : Queue
TopicSubscriber
+getTopic() : Topic
-clientSession
-session
conSessionAssoc
Client
-url : String
-news : String
+produceNews() : String
+showNews(news : String)
-client
clientSessionAssoc
-connection
-client
clientConAssoc
-clientCon
Connection
+close() : void
+start() : void
+stop() : void
-con
administer
-factory
clientfactoryAssoc
-clientfactory
ConnectionFactory
+createConnection() : Connection
+createConnection(userName : String, password : String) : Connection
JMSException
+JMSException(reason : String)
+JMSException(reason : String, errorCode : String)
+getErrorCode() : String
MessageFormatException
InvalidDestinationException
MessageNotWriteableException
MessageNotReadableException
JMSRuntimeException

Figure 5 PSM Class Model for a Messaging System

Class Model Mapping

Table 5. PIM to PSM mappings represents the class model mapping between the PIM and the PSM. The PIM instances are provided as an ecore file.The QVT transformation and mapping rules are going to manipulate these instaces to create the PSM instances.

Table 5. PIM to PSM mappings

PIM Classes	**PSM Classes**
NewsSender	Producer(QueueSender or TopicPublisher)
NewsReceiver	Consumer(QueueReceiver, TopicSubscriber
NewsMessage	Message
DataLink	Connection
PIMClient	Client
-	Destination(Queue or Topic)
-	ConnectionFactory
-	Session
-	Exceptions

Table 5. Simplified XMI File of PIM Instances

```
<?xml version="1.0" encoding="UTF-8"?>

<xmi:XMI      xmi:version="2.0"      xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:Data="http:///NewsPIMData.ecore"
xsi:schemaLocation="http:///NewsPIMData.ecore NewsPIMData.ecore">

  <Data:NewsSender senderId="S001" name="sender1" sentData="/6"/>

          <Data:NewsReceiver     receiverId="R001"     name="receiver1"
receivedData="/6"/>

  <Data:PIMClient postTo="TESTQUEUE" news="Hello"/>

  <Data:DataLink Id="DL001" status="OK"/>
  <Data:NewsMessage dataId="Msg1" content="Hello" status="generated"/>

       <Data:NewsMessage   dataId="Msg2"   content="The   second   msg"
status="generated"/>

       <Data:NewsMessage   dataId="Msg3"   content="The   third   msg"
status="generated" receiver="/1" sender="/0"/></xmi:XMI>
```

Table 5. Example of a Transforming Program in QVT

```
transformation trans( source : NewsPIMData, target : NewsPSMData) {


      top relation senderToProducer {

            varName, identity : String;

            checkonly domain source s : NewsPIMData::NewsSender   {

                  name = varName,

                  senderId = identity

            };

            enforce domain target p : NewsPSMData::Producer {

                  name = varName,

                  Id = identity

            };

      }


      top relation receiverToConsumer {

            rName, rId : String;
```

```
        checkonly domain source rcvr : NewsPIMData::NewsReceiver   {
              name = rName,
              receiverId = rId
        };
        enforce domain target con : NewsPSMData::Consumer {
              name = rName,
              Id = rId
        };
  }
  top relation PIMClientToPSMClient {
        to,n: String;
        enforce domain source c:NewsPSMData::PIMClient  {
         postTo=to,
         news=n
        };
        enforce domain target PSMC:NewsPSMData::Client {
        url=to,
        news=n
        };
  }
  top relation DataLinkToConnection {
        id,st: String;
        enforce domain source d:NewsPSMData::DataLink  {
         Id=id,
         status=st
        };
        enforce domain target c:NewsPSMData::Connection {


        };
  }
  top relation NewsMessageToMessage {
        dId,con,st: String;
        enforce domain source d:NewsPSMData::NewsMessage  {
         dataId=dId,
         content=con,
         status=st
        };
        enforce domain target c:NewsPSMData::Message {
        content=con
        };
  }
 top relation createSession{
```

```
            enforce domain target s:NewsPSMData::Session {

            };

      }


      top relation createQueue{

            enforce domain target q:NewsPSMData::Queue {

            };

      }


      top relation createConnectionFactory{

            enforce domain target cf:NewsPSMData::ConnectionFactory {

            };

      }

}
```

Table 5. Generated XMI File of PSM Instances-Simplified

```
<?xml version="1.0" encoding="UTF-8"?>

<xmi:XMI         xmi:version="2.0"         xmlns:xmi="http://www.omg.org/XMI"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:NewsPSMData="http:///NewsPSMData.ecore"

xsi:schemaLocation="http:///NewsPSMData.ecore NewsPSMData.ecore">
```

```
  <NewsPSMData:Consumer Id="R001" name="receiver1"/>

  <NewsPSMData:Producer Id="S001" name="sender1"/>

  <NewsPSMData:Client url="TESTQUEUE" news="Hello"/>

  <NewsPSMData:Connection/>

  <NewsPSMData:Message content="The third msg"/>

  <NewsPSMData:Message content="The second msg"/>

  <NewsPSMData:Message content="Hello"/>

  <NewsPSMData:ConnectionFactory/>

  <NewsPSMData:Queue/>

  <NewsPSMData:Session/>

</xmi:XMI>
```

Behavioral Models Mapping

PIMClient and Client Classes

The ***PIMClient*** behavior is started and placed initially in the ***setup*** state as shown in figure 3.The ***setupNotOK*** and *setupOK* are constraints on the transitions going out of the setup state. The constraints specification expressed in OCL 2.0 syntax [transition names are not shown]. The constraints are checking the DataLink class to find if the status attribute is 'ok' or 'notOK'. When the ***setupNotOK*** constraint is true, the state behavior transitions to the ***handleProblems*** state. After that it can either transition to do the setup again or to exit.

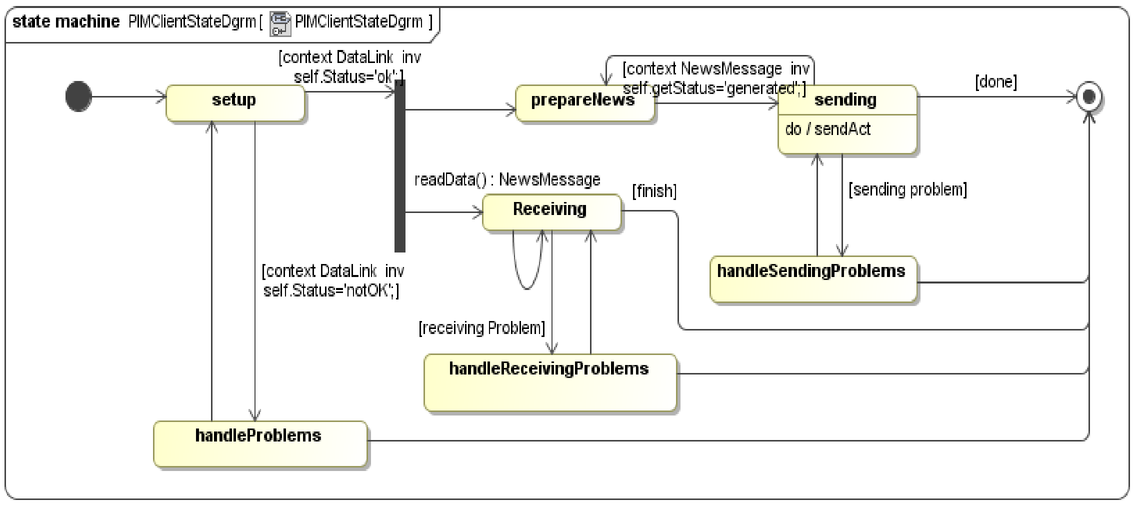

state machine PIMClientStateDgrm [PIMClientStateDgrm]
setup
[context DataLink inv self.Status='ok';]
prepareNews
[context NewsMessage inv self.getStatus='generated';]
sending
do / sendAct
[done]
readData() : NewsMessage
Receiving
[finish]
[sending problem]
handleSendingProblems
[context DataLink inv self.Status='notOK';]
[receiving Problem]
handleReceivingProblems
handleProblems

Figure 5 PIM Client State Diagram

When the ***setupOK*** constraint is satisfied, the ***PIMClient*** can transition to the next state. The transition took place either to the ***prepareNews*** state or to the ***receiving*** state when a call to ***readData*** method is specified. In the ***prepareNews*** state and when the news are generated, their status is set to "Generated" which leads to the firing of the next transition to ***sending*** state. The sending state is going to do an action that calls the ***writeData*** method in the ***Sender*** class in order to send the newly created news message. If there are problems in sending which is indicated by the constraint ***sendingProblems*** evaluated to true, the state ***handleProblems*** is going to be visited. If the sending is to be repeated, the next transition is going to be to the ***prepareNews*** state again. When sending is finished, the behavior will end in the final state.

After the setup, the ***Client*** PSM class behavior can transition to the ***receiving*** state by specifying a call to ***readData*** method of the ***Receiver*** class. The receiving can be repeated, in which case transitioning back to the ***receiving*** state itself. When receiving is completed the transition ended in the final state.

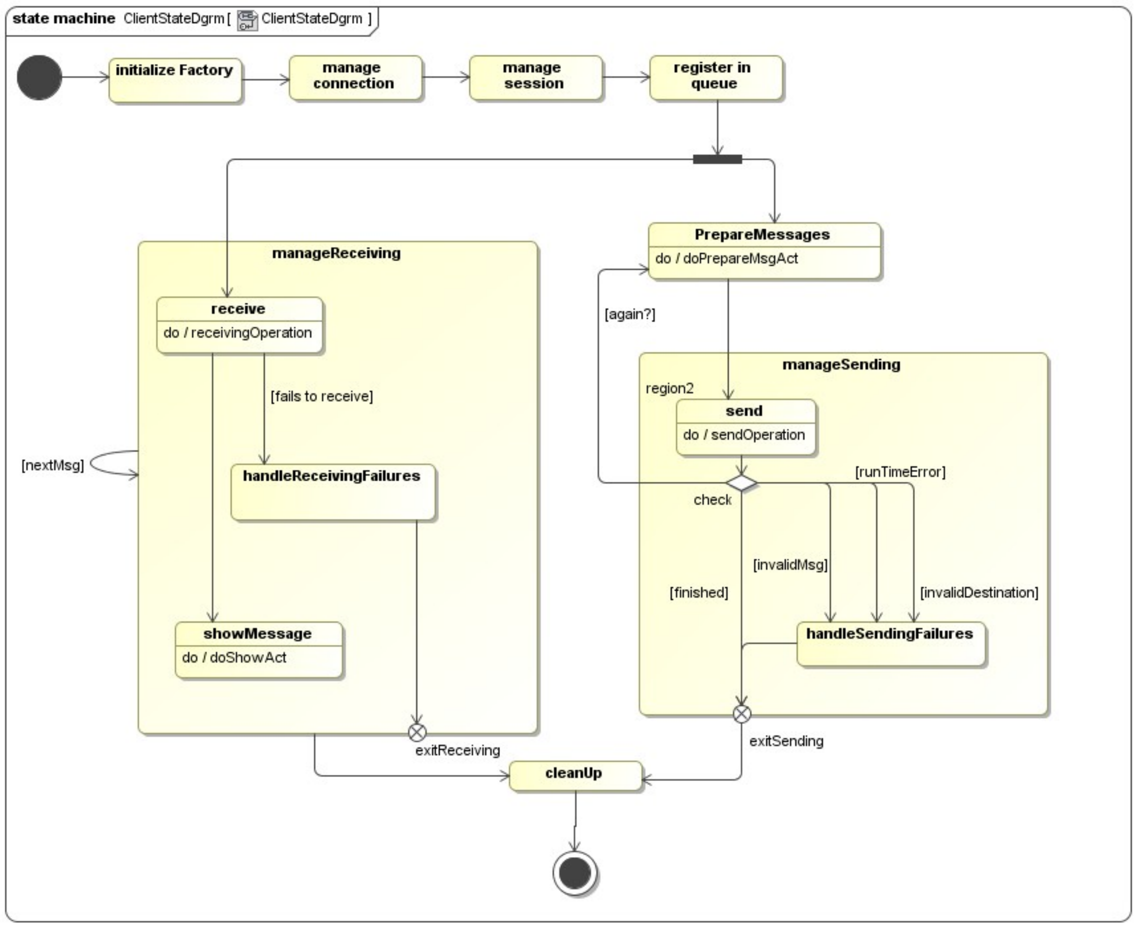


Figure 5 PSM Client State Diagram

Behavioral Mapping of PIMClient to Client

Setup State

PIMClient behavior starts in the initial node and transitions to the ***setup*** state. The PSM starts in the initial node and transitions to ***initialize factory***. Moreover the PSM continues to transition to other states to manage the ***Connection***, ***session*** and ***Queue***. Referring to the class model mapping, we find that the ***DataLink*** class is equivalent to the ***Connection*** class. The ***Session*** , ***connectionFactory*** and ***Queue*** are PSM specific and they do not correspond to any class in the PIM. The setup process according to the PIM means creating the DataLink class and ensuring that its status is 'ok' to proceed. On the other hand the preparation in the PSM include preparing the ConnectionFactory to establish the connection, assign a session for the user and registering in a queue. The PSM ***Connection*** class is created and started but its status is not checked explicitly - using an associated attribute or a method- but is checked internally by raising exceptions when an error in starting or stopping the connection occur. The same applies to the ***Session*** , ***connectionFactory*** and

Queue classes. So the transitions between the states to manage these classes are not guarded.

we can entail that:

Setup={initialize factory, manage Connection, manage session, register in queue}

The decision here is to create a new composite state with a new region and add the states to it as the transformation result.

Table 5. Setup State Transformation

Name	Type	Equivalence Class (PSM)	Transformation Decision
Setup	Single State to more than one state	={initialize factory, manage Connection, manage session, register in queue}	Create a new composite state with name=Setup, Create a new region with name=SetupRegion and add to it the
Reason/Justific	PIMClient is	-	

ation	initially in the setup state, Client PSM class is initially transitioned between several states.		equivalence class states as sub states
Transitions	-	-	-
Guards	setupOk	No equivalent	Environment internal check is carried out using exception mechanism.
Actions	-	-	-

Table 5. HandleProblems State Transformation

Name	Type	Equivalence Class (PSM)	Transformation Decision
Handle Problems	Orphan state	={ }	Ignore the orphan state
Reason/Justification	-	Handling problems is implicitly done through raising exceptions when an error occurred.	
Transitions	To setup and to End		
Guards	setupNotOK	={ }	Ignore the guard

Actions	-	-

The PIM specify that if problems occur in the setup process then the application will transition to the handleProblems state. The PSM mechanism in dealing with problems/failures or abnormal conditions is by raising/throwing exceptions. An exception is an event that occurs during the execution of a program that disrupts the normal flow of control. Exceptions are represented by classes such as MessageFormatException, InvalidDestinationException and MessageNotWritableException. Exceptions can be caught by handlers. Uncaught exceptions may be handled by the environment and can cause the termination of the thread of control. Handling the exceptions according to the PSM is carried out by the rules and not by an explicit state. The decision for the handleProblems PIM state is simply not transforming it to a state in the PSM because its functionality is implemented in another way in the PSM.

Table 5. Receiving State Transformation

Name	Type	Equivalence Class (PSM)	Transformation Decision
Receiving	Single State to part of a composite state	={manage receive:: receiving, manage receive:: showMessage }	Map the single state to the composite state
Reason/Justification	receive PSM state transitions to showMessage without a	-	

	condition		
Transitions	To self	={nextMsg?}	Map the transition
Guards	Receiving problem	={Fails to receive}	Map the guard
	Finish	Transition without a guard to the end	Map the transition
Actions	Call to readData method	Receiving method in Consumer class.	Map the calls
	-	doShowAct	Create the action

One of the capabilities of the PSM is message receiving. The PIM ***Receiving*** single state is equivalent to ***receiving*** PSM state which is a sub state in a composite state aka ***manageReceiving.*** Consumption of news messages in the PIM is not specified. In the PIM the message reception is followed by showing the message. The decision is to map the single state to the composite state that contains more functionality than specified in the PIM.

Table 5. HandleReceivingProblems State Transformation

Name	Type	Equivalence Class (PSM)	Transformation Decision
HandleReceivingProblems	Single state to a part of a composite state (manageReceivin	={handleReceiving Failures}	Map the single state to the

	g)		composite state as done with the ***Receiving*** state
Reason/Justification			
Transitions	To self	To composite state self	Map the transition
	To end	={To composite state exit, To end}	Map the transition
	Transition from HandleReceivingProblems back to the Receiving state	No equivalent, but the PSM has mechanisms to retry the message reception.	
Guards	Receiving problem	={Fails to receive}	Map the guard
	Finish	Transition without a guard to the end	Map the transition
Actions	Call to readData method	Receiving method in Consumer class.	Map the calls

HandleReceivingProblems is mapped to ***handleReceivingFailures*** in the PSM forming an equivalence class together with the ***receiving*** and ***showMessage*** PSM state. The application logic specified in the PIM is to receive a message, if problems in receiving occur then handle the situation and go back to receive again. The PSM does not go back to receive again, because there is no transition going back to the

receiving state from ***HandleReceivingFailures***. The PSM Consumer class provide overloaded ***receive*** methods, one of them has a timeout period as a parameter and can wait for the message to arrive. The default receive operation blocks indefinitely until a message is produced or until the message consumer is closed. The decision regarding the call action to the readData method is to map it to the receive method. This way the message reception will be tried until the message arrived or an error occurred.

Table 5. Forbidden State Transformation

Name	Type	Equivalence Class (PSM)	Transformation Decision
-	forbidden state	={ cleanUp}	Create the PSM state
Reason/Justification	The PIM transitions to the end when finished. The PSM cleans up the resources such as the connection object before ending.		The logic in PSM completes by closing resources.
Transitions	To end	To end	Map the transition
Guards	-	={ }	
Actions	-	-	

The PSM ensures the proper initialization of classes instances such as the ***Connection*** and ***Session*** through ***manageConnection*** and ***manageSession*** states. Class instances use computing resources that are finite therefore it is reasonable to free and release the resources when they are no longer needed. The PSM specify a cleanup state in order to close the opened connection, close the session and so on. In the other hand , the PIM doesn't specify such details. The decision is to create the cleanUp state in the resulting instances, because its functionality is recommended in the PSM in order to create applications that conforms to the best practices.

Table 5. PrepareNews State Transformation

Name	Type	Equivalence Class (PSM)	Transformation Decision
prepareNews	Single state to single state	={ prepareMessages }	Map the states
Reason/Justification			
Transitions			
Guards	Invariant Context Message -Self.getStatus='generated'	={ }	Ignore , PSM messages lacks a status property that is to be checked.

Actions	-	doPrepareMsg Act	Create the action

The state ***PrepareNews*** is equivalent to the state ***PrepareMessages***. The mapping is one to one. In the class model the ***News*** Class is mapped to the PSM ***Message*** class. The ***News*** class contains an attribute ***status*** that reflects the status of the message whether generated, sent or notReceived. On the other hand the ***Message*** class has no such attribute. Moreover the sending/receiving of a message is setting/getting attributes related to the queue, session, connection, News producers and consumers. The decision here is to keep the default behavior of the PSM regarding the message status from sending till receiving. The default behavior include default values for various attributes in other objects used to send/receive the message.

Table 5. Sending State Transformation

Name	Type	Equivalence Class (PSM)	Transformation Decision
Sending	Single state to single state	={send }	Map the state
Reason/Justification			
Transitions			
Guards	Done	={ finished}	Map the guard
	SendingProblem	={runtimeError, invalidMsg, invalidDestination}	Map the guards

Actions	-	={again?}	create the guard
	sendAct	sendOperation	Map the actions
	-		

The ***sending*** state is mapped to the ***send*** PSM state. The sending of the news fits exactly what specified in the PSM. The related actions, method calls and transitions are equivalent to the PSM ones. The guard sendingProblems is equivalent to more than one guard in the PSM. The PSM specify detailed and specific situations of problems/failures that can occur. The PSM guards trigger the creation of instances of exceptions. Exceptions is the mechanism that the PSM raise/handle various types of problems. In the PSM class model various types of exceptions are included while there is no equivalent classes in the PIM. The mapping of the guards between the PIM and the PSM assure that the decision to map the exceptions classes was a right decision although no equivalent classes are in the PIM.

Table 5. HandleSndingProblems State Transformation

Name	Type	Equivalence Class (PSM)	Transformation Decision
HandleSendingProblems		={handleSendingFailures }	Map the states
Reason/Justification			
Transitions	To end	={To exitSending, To end}	Map the guard
	to sending	={}	Ignore

	state	
Guards	-	-
Actions	-	-

HandleSendingProblems is mapped to ***handleSendingFailures*** in the PSM forming an equivalence class together with the ***send*** PSM state. The application logic specified in the PIM is to send a message, if problems in sending occur then handle the situation and go back to send again. The PSM does not go back to send again, because there is no transition going back to the ***send*** state from ***handleSendingFailures***. The PSM Producer class provide overloaded ***send*** methods, some of them has a timeToLive as a parameter that specify the length of time in milliseconds from its dispatch time that a produced message should be retained by the message system. The default send operation sends a message using the Producer's default delivery mode, priority, and time to live. The decision regarding the call action to the sendData method is to map it to the send method. This way the message retained till consumed or an error occurred.

Mapping of Other Types

Initial node , Final node, Fork node , Merge node with decision node , are mapped to the equivalent.

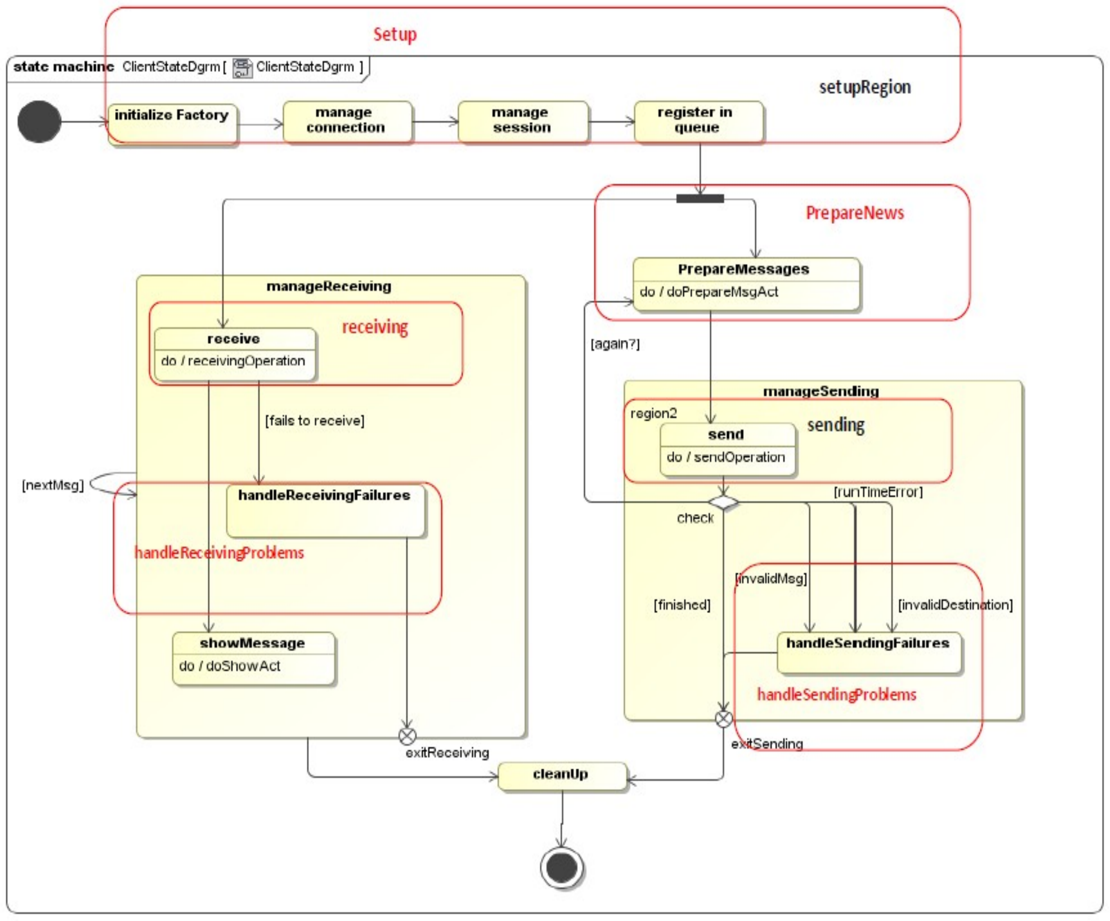


Figure 5 The mapping result of the PIM (in rectangles) to The PSM

The NewsSender and Producer Classes

The ***NewsSender*** class is responsible for sending messages. As depicted in Figure 5 PIM NewsSender State Diagram it is initially waiting for its clients as indicated by the ***waiting*** state. The clients request sending a message by calling the method ***writeData*** in the ***NewsSender*** class that triggers the change of the ***NewsSender*** state to the ***sending*** state. After writing the data, the ***NewsSender*** can go back to the waiting state when the ***wait*** constraint is true. When sending is not successful, the problem is raised so as to let the caller-the Client- handle the situation properly. The ***NewsSender*** ends its behavior when the ***Finished*** constraint is true.

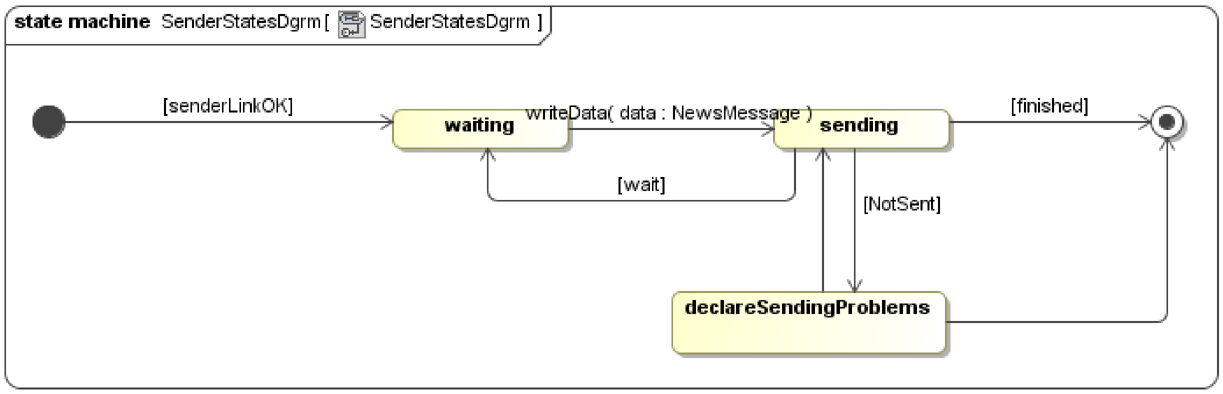

state machine SenderStatesDgrm [SenderStatesDgrm]
[senderLinkOK]
waiting
writeData(data : NewsMessage)
sending
[finished]
[wait]
[NotSent]
declareSendingProblems

Figure 5 PIM NewsSender State Diagram

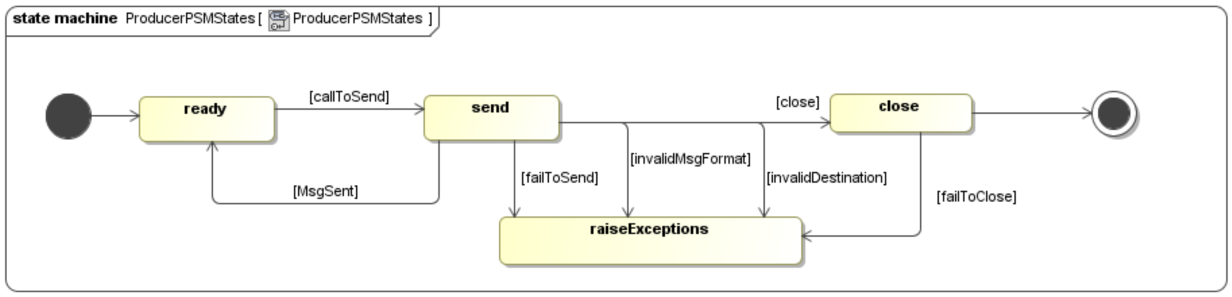


Figure 5 PSM Producer State Diagram

<u>Behavioral Mapping of NewSender and Producer</u>

Table 5. Waiting State Transformation

Name	Type	Equivalence Class (PSM)	Transformation Decision
Waiting	Single state to single state	={ ready}	Map the states
Reason/Justification	The producer is initially put on ready state without conditions		
Transitions			
Guards	senderLinkOk	={ }	Ignore the guard
	Call to writeData	={callToSend}	Map the calls
	Waiting	={msgSent}	Map the guard
Actions			

Table 5. Sending State Transformation

Name	Type	Equivalence Class (PSM)	Transformation Decision
Sending	Single state to single state	={send }	Map the states
Reason/Justification			
Transitions			
Guards	NotSent	={failToSend, invalidMsgFormat, invalidDestination }	Map the guards
	-	={close}	Map the guard
Actions			

Table 5. declareSendingProblems State Transformation

Name	Type	Equivalence Class (PSM)	Transformation Decision
declareSendingProblems	Single state to single state	={ raiseExceptions}	Map the states
Reason/Justification			

Transitions	-	-
Guards	-	-
Actions	-	-

Table 5. Forbidden State Transformation

Name	Type	Equivalence Class (PSM)	Transformation Decision
-	forbidden state	={ close}	Create the PSM state
Reason/Justification	The PIM transitions to the end when finished. The PSM closes the resources used before ending.		The logic in PSM completes by closing resources.
Transitions	To end	To end	Map the transition
Guards	-	-	
Actions	-	-	

The mapping process resulted in a state machine equivalent to the PSM state machine with no further changes to it.

The NewsReceiver and Consumer Classes

The ***NewsReceiver*** class shown in Figure 5 PIM NewsReceiver State Diagram below behaves similarly to the ***NewsSender*** class. It is responsible for receiving messages. It is initially placed in the **waiting** state ready to serve its clients. The clients request receiving a message by calling the method ***readData*** that triggers the change of the ***Receiver*** state into the ***receiving*** state. After writing the data, the ***NewsReceiver*** can go back to the waiting state when the ***wait*** constraint is true. When receiving is not successful, the problem is raised so as to let the caller- the Client- handle the situation as required. The ***NewsReceiver*** ends its behavior when the ***Finished*** constraint is true.

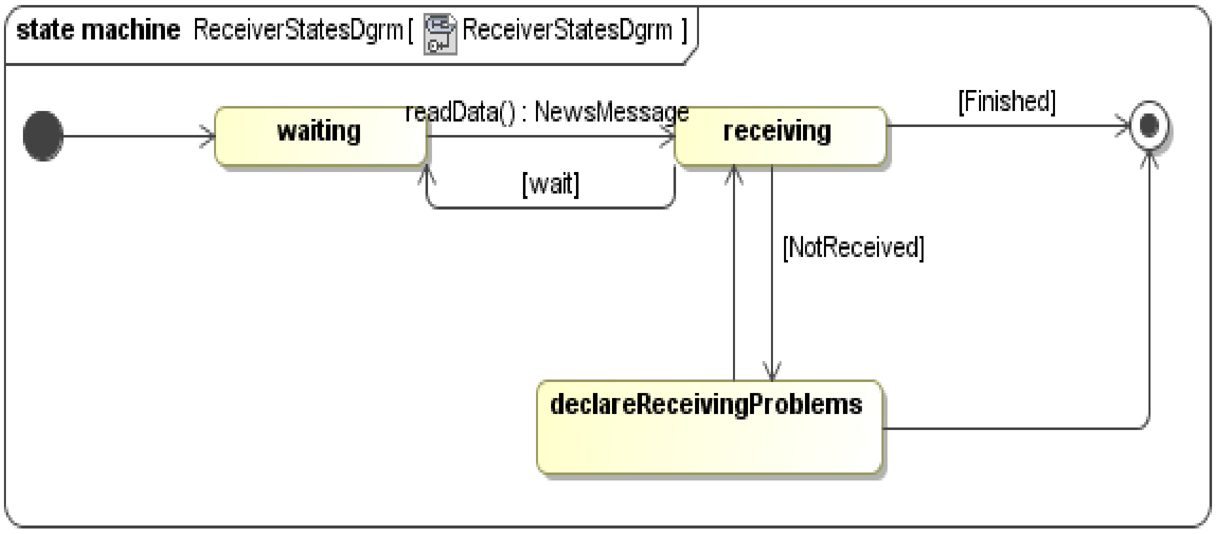


Figure 5 PIM NewsReceiver State Diagram

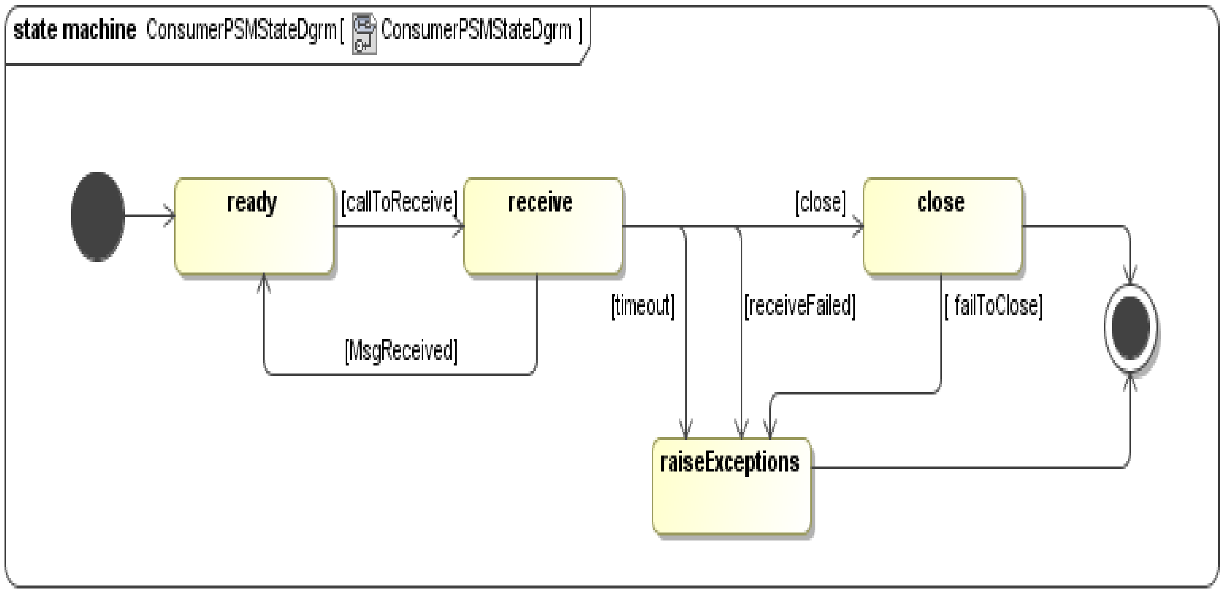


Figure 5 PSM Consumer State Diagram

Behavioral Mapping of NewsReceiver and Consumer

Table 5. Waiting State Transformation

Name	Type	Equivalence Class (PSM)	Transformation Decision
Waiting	Single state to single state	={ ready}	Map the states
Reason/Justification	The Receiver is initially put on ready state without conditions		
Transitions	-	-	
Guards	Call to	={callToReceive}	Map the call

	readData		
	Waiting	={msgReceived}	Map the guard
Actions	-	-	

Table 5. Receiving State Transformation

Name	Type	Equivalence Class (PSM)	Transformation Decision
Receiving	Single state to single state	={receive }	Map the states
Reason/Justification			
Transitions	To receiving	={}	ignore
Guards	NotReceived	={ timeout, receiveFailed }	Map the guards
	Finished	={close}	Map the guard
Actions	-	-	

Table 5. declareReceivingProblems State Transformation

Name	Type	Equivalence Class (PSM)	Transformation Decision
declareReceivingProblems	Single state to single state	={ raiseExceptions}	Map the states
Reason/Justification			

Transitions	-	-
Guards	-	-
Actions	-	-

Table 5. Forbidden State Transformation

Name	Type	Equivalence Class (PSM)	Transformation Decision
-	forbidden state	={ close}	Map the states
Reason/Justification	The PIM transitions to the end when finished. The PSM closes the resources used before ending.		
Transitions	To end	To end	Map the transition
Guards	-	={failToClose}	Map the

			guard
Actions	-	-	

The mapping process resulted in a state machine equivalent to the PSM state machine with further changes applied to it.

Summary

Model transformation is a young field and there are several competing, yet partly overlapping definitions of the terms. (Tratt 2005) defines model transformation very widely as ”*a program that mutates one model into another*”. The Object Management Group (OMG) defines model transformation in the context of the model-driven architecture (MDA) as ”*the process of converting a model into another model of the same system*” in the first MDA guide (OMG 2003). The second revision of MDA (OMG 2014a) extends this definition by also allowing several models as input or output and define model transformation as ” *Transformation deals with producing different models, viewpoints, or artifacts from a model based on a transformation pattern. In general, transformation can be used to produce one representation from another, or to cross levels of abstraction or architectural layers*”.

The proposed mapping process would begin by first mapping the class models that map the classes and their attributes. According to the class mapping , the behavior of the mapped classes would be mapped also. Behavior mappings begin with mapping of transitions and constrains (guards). Then construct the concrete mapping from the PIM state model to the PSM state model according to the following framework.

We have two state models, A and B. Model B belongs to the PSM, so is an implementation of Model A, which belongs to the PIM. Model A will describe the application states and model B describe technically how to use the available services provided in order to achieve some functionality. An example of model A is the behaviour model developed for the financial system application in Figure 4 State Machine Behavior

Diagram of PIM LinkProvision Class and the other one developed for the news application Figure 5 PIM Client State Diagram. Examples for Model B are Figure 4 ATM State Machine Diagram and Figure 5 PSM Client State Diagram.

Assume, as in the case studies, that there are fewer states in model A than in model B, and that every state in model B corresponds to exactly one state in model A. This means that we can divide the states in model B into groups, indexed by the state of model A they correspond to. If *a* is a state of A, then the states of B corresponding to *a* form an equivalence class. Let's call that B(a).

Model transformations of behavior models represented as UML state machine in this research can be classified into five categories:

1. Single State to Single State Transformation

   If given an *a* state of A, there is one state in B(*a*), then call that state *b*, and the PIM/PSM mapping maps *a* into *b*.

2. Single State to More Than One State Transformation

   If given an *a* state of A, there are several states in B(*a*), then map *a* to those states if possible. The mapping is **one state to many**.

3. Single State to part of a composite state Transformation

4. Orphan State Transformation

   Assuming that every state of A corresponds to at least one state of B. Otherwise, the **orphan state** of A cannot be implemented. This could form part of an evaluation of the suitability of a PSM for implementing a given PIM.

5. Forbidden State Transformation

What happens if there are states in B that don't correspond to some state in A? Let's call them **forbidden states**. Whether this is a problem would depend on whether the transitions mapped from A ever take a state in B to the forbidden state.

Table 5. Issues and Behavioral Mapping Decisions summarized

Type (state/transition)	**Mapping**
Single state to single state	Direct mapping
Single state to multiple states	Create a new composite state with name=the PIM single state name, Create a new region with name=PIM state name+"Region" and add to the newly created composite state the PSM equivalence class states as sub states
Orphan State	Ignored, in the case study, this type of states are handled implicitly by the PSM , so ignoring them is not going to create problems in the application logic.
Forbidden State	This type of states are important to complete the PSM logic, they are created in the resulting model.
Single transition to single transition	Direct mapping
Single transition to multiple transitions	Map the transitions (create the PSM transitions) and check the target states of each transition for equivalence with

	PIM.
Orphan transition	Ignored
Forbidden transition	Created in the resulting model

the same process was done to the guards, actions, Initial node , Final node, Fork node , Merge node with decision node , are mapped to the equivalent.

Models To Text and The Application Execution

Overview

The mapping of PIM behavior model- state Machine - to the PSM behavior model yield a behavior model expressed as PSM constructs (state machine). The generated state machine models are highly reusable since they are expressed in UML. The models then can be transformed into other forms to enable complementing the class model and provide the big picture as a complete application with class and behavior instances both available to be executed.

Figure 6 Models to Text Translation

In the following sections we are going to discuss different possibilities of mapping the PSM Behavior model instances into more usable constructs. The main objective is to find various ways to make a forward step towards application execution.

Possible Option 1 : Generic Mapping to prove concepts

Apparently the state machine depicts the flow of control an object has. In the context of the case study what we need is the application flow of the PIM instances hence their behavior and how that is achieved through PSM instances.

If we concentrate on the PSM instances which are mapped manually , especially the guards and operation calls, we get a sequence of calls guarded by conditions.

Proposed Rendering

We are going to render the state machine instances of the PIM NewsSender and the PSM Producer classes that depicted in Figure 5 PIM NewsSender State Diagram and Figure 5 PSM Producer State Diagram respectably as following:

- A state is rendered as a comment with state name. example //** Idle **//
- A transition with a guard condition is rendered as “Evaluate “+ guard specification
- A none guarded transition is rendered as " and"+ target state name
- A do action of a state is rendered as “Call ”+ the operation if the action is a CallOperationAction type.

PIM instance Behavior

```
Evaluate senderLinkOK

If true  //**waiting**//

Evaluate
writeData(data:NewsMessage)

If true //** sending **//

Evaluate finished

If true //** final **//

Evaluate NotSent

If true //**declareSendingProblems
**//

Evaluate wait

If true //** waiting **//
```

PSM instance Behavior

```
//** ready **//

Evaluate callToSend

If true //** send **//

 Evaluate close

If true //** close**// and
//**final**//

Evaluate failToSend

If true //** raiseException**//

Evaluate invalidMsgFormat

If true //** raiseException **//

Evaluate invalidDestination

If true //** raiseException **//

Evaluate failToClose

If true //** raiseException **//
```

The result of the rendering is what we can call a high level algorithm, in other words a high level program specified as model elements.

Possible Option 2 : Mapping of UML State Machine to SCXML

State Chart extensible Markup Language: State Machine Notation for Control Abstraction (SCXML) is a standard developed by the World Wide Web Consortiums (W3C) with the objective of generifying the state diagrams notations used in XML contexts. According to the working draft dated May 2014 (W3C n.d.), SCXML combines the concepts of the Call Control XML (CCXML) standard and Harel State Tables (Harel 1987). CCXML is an event based state machine language that supports the call control features in voice application. Harel State Tables are state machine notation that is included in UML and provide several extensions to the basic notions of the CCXML state machine.

UML state machine diagram is an object-based variant of Harel state chart tables.

SCXML= CCXML enhanced with Harel State Tables

SCXML=State machine + event handling syntax + standard call controls

SCXML provide core constructs to represent state machine concepts such as state, transition, parallel, history and other constructs. It also provide executable constructs such as if, elseif, foreach and log. Beside that it offers the capability of manipulating the state internal data as elements and initial values in an abstract representation that can be realized by various languages. SCXML also provide a way to communicate with external entities through events.

The Apache Foundation supports the SCXML specification by providing a working implementation, a set of APIs and an engine that can execute a SCXML state machine described as a document. The Apache Commons SCXML 2.0 (Apache n.d.) is the Java SCXML engine aligned and compliant with the latest SCXML specifications.

Since the concepts and terminology used in SCXML and UML state machine are both based on the Harel state charts table , a mapping process is feasible between them. This implies that the SCXML development tools, class library and runtime implementation of the

Apache Common SCXML can be used to create a platform model and also provide an execution environment for the behavior of an end to end application. The reason here is to provide constructs that can be executed since the UML state charts are not.

Figure 6 Detailed mapping From PSM to Execution

Apache ActiveMQ

In the JMS API architecture, a JMS Provider is a messaging system that implements the JMS interfaces and provides administrative and control features. Apache ActiveMQ is an open source, Java Message Service (JMS) 1.1–compliant, message oriented middleware (MOM) from the Apache Software Foundation (Apache 2015). ActiveMQ implements the JMS specification and offers additional features and value on top of this specification. The goal of ActiveMQ is to provide standards-based, message-oriented application integration across as many languages and platforms as possible.

Apache ActiveMQ in this case study is used as the execution environment where the JMS clients respectively the producer and consumer PSM instances are running and the administered objects are configured. The behavior instances can be executed in the Apache ActiveMQ environment by configuring the Apache Commons SCXML inside it to run the behavior models along with the class models.

UML, SCXML and the Apache Commons SCXML

A comparison of UML, W3C SCXML and the Apache Common SCXML was carried out. The differences existed because of the continuous improvements to the SCXML specification and its Apache implementation. The latest release of the Apache Commons SCXML is 0.9. Subsequent changes to the SCXML Draft may necessitate changes to portions of the Commons SCXML library API but the core APIs (SCXMLParser, SCXML Executor etc.) are stable (W3C n.d.).

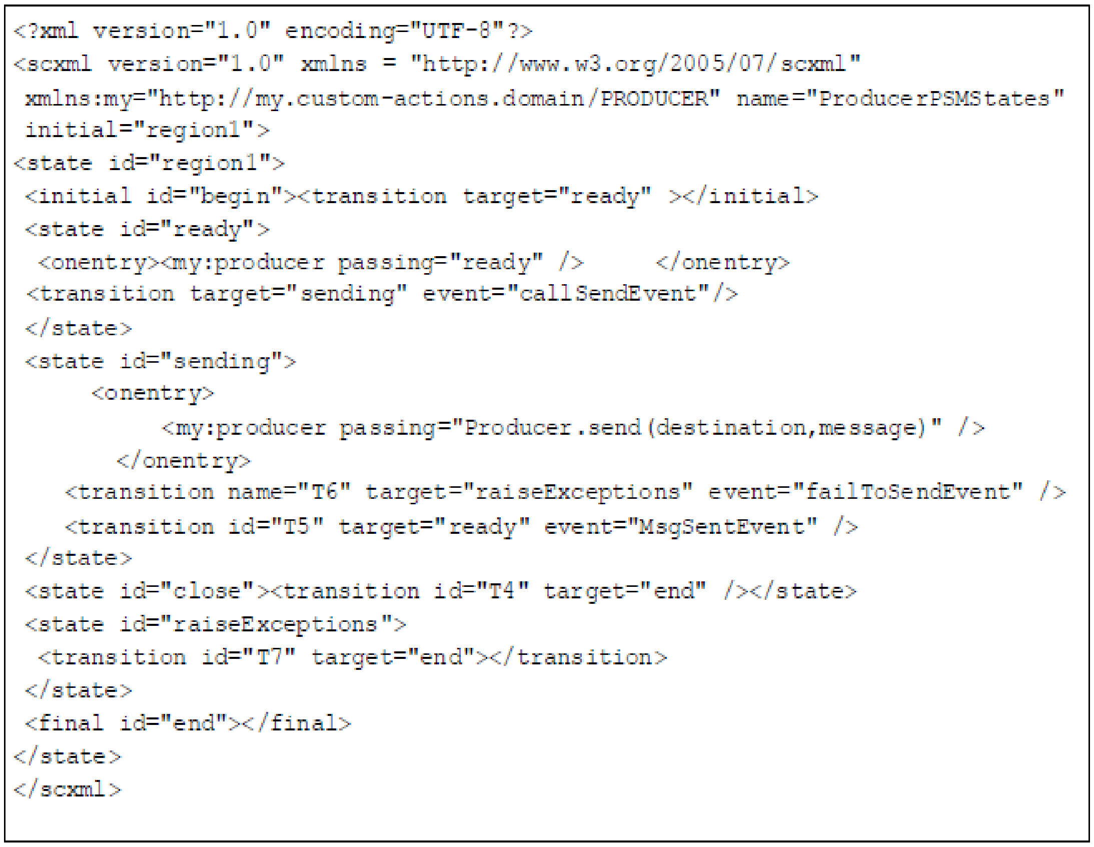
Table 6.Comparing some elements of UML, W3C SCXML and the Apache Common SCXML

UML State machine Constructs (OMG 2011c)	**SCXML CONSTRUCTS** (W3C n.d.)	**Apache Commons SCXML API** (Apache n.d.)
State Machine	Document <scxml > tag	org.apache.commons.scxml2.env.AbstractStateMachine
region	-	-
Simple state	<state>	org.apache.commons.scxml2.model.State
Initial Pseudostate	<initial>	org.apache.commons.scxml2.model.Initial
FinalState	<final>	org.apache.commons.scxml2.model.Final
History State	<history>	org.apache.commons.scxml2.model.History

composite state	A compound state is a <state> that has <state>, <parallel>, or <final> children (or a combination of these).]	-
Orthogonal State		-
submachine state		-
Transition	<transition>	org.apache.commons.scxml2.model.Transition
Guard /Constraint	Cond attribute of <transition>	String cond Property that specifies the trigger(s) for this transition class
Event /Trigger	event attribute of <transition>	String event Property that specifies the trigger(s) for this transition class
fork and join is a short heavy bar	<parallel>	org.apache.commons.scxml2.model.Parallel
entry: Behavior[0..1]	<onentry>	org.apache.commons.scxml2.model.OnEntry
doActivity: Behavior[0..1]	<invoke>	org.apache.commons.scxml2.model.Invoke
exit: Behavior[0..1]	<onexit>	org.apache.commons.scxml2.model.OnExit
	<raise>	org.apache.commons.scxml2.model.Raise
choice pseudostate	<if>	org.apache.commons.scxml2.model.If
diamond-shaped symbol	<elseif>	org.apache.commons.scxml2.model.ElseIf
	<else>	org.apache.commons.scxml2.model.Else

Transforming UML State Machine to SCXML

The UML class diagram and state machine diagrams were imported from the Magic Draw environment as an XML document. The XML document is very large so we present here the equivalent SCXML document for the diagram in Figure 5 PSM Producer State Diagram.



```
<?xml version="1.0" encoding="UTF-8"?>
<scxml version="1.0" xmlns = "http://www.w3.org/2005/07/scxml"
 xmlns:my="http://my.custom-actions.domain/PRODUCER" name="ProducerPSMStates"
 initial="region1">
<state id="region1">
 <initial id="begin"><transition target="ready" ></initial>
 <state id="ready">
  <onentry><my:producer passing="ready" />     </onentry>
 <transition target="sending" event="callSendEvent"/>
 </state>
 <state id="sending">
      <onentry>
           <my:producer passing="Producer.send(destination,message)" />
        </onentry>
    <transition name="T6" target="raiseExceptions" event="failToSendEvent" />
    <transition id="T5" target="ready" event="MsgSentEvent" />
 </state>
 <state id="close"><transition id="T4" target="end" /></state>
 <state id="raiseExceptions">
  <transition id="T7" target="end"></transition>
 </state>
 <final id="end"></final>
</state>
</scxml>
```

Figure 6 SCXML Document for the Producer State Machine Diagram in Figure 5 PSM Producer State Diagram

Suggested Algorithm to transform State machines to SCXML Document

```
Input:
      XML file representing the UML state diagram
Output:
      XML file representing the SCXML document
Steps:
   1. Read and parse the input file

   2. Get the owned behavior id from the behaviored class ( producer,
      sender ,...)

   3. Start with the element <ownedBehavior xmi:type="uml:StateMachine"

         a. map to <scxml> with name=name

         b. map the first state name to the scxml initial attribute

   4. Process the children of the <ownedBehavior>

         a. if the element is a <region>

               i. map it to an upper level state with id= region name

              ii. if the region is the first one map its name as in step
                  3, b part.

         b.  if the element is a <Pseudostate> map it to <initial>  with
            id= Pseudostate name

         c. if the element is a <subvertex xmi:type='uml:FinalState' >
            map it to <final>  with id= Final state name

         d. if the element is a <subvertex xmi:type='uml:State' >

               i. map it to <state> with id= state name

              ii. Find the nested state actions and map them.

             iii. Find the state transitions and map them. Match the
                  transition source=state id

         e. if the element is a <transition>
```

        i. map it to a <transition> with the target= UML transition target state name

        ii. find the target state name by matching using the specified id

        iii. find transition triggers <trigger xmi:type='uml:Trigger'

            1. map to event attribute with name = the UML trigger event name

            2. Search for an element matched by id to find event name

    f. if the element is <entry>

        i. map it to <onentry>

        ii. map the specification

            1. as a script inside the <onentry>, or code in a programming language of choice.

            2. if it is a method call try to locate the class operation name and parameters

Concrete Mapping Examples

UML tags are represented in the first line and the equivalent SCXML tag after it.

1. UML <OwnedBehavior> to SCXML <SCXML>

Each State machine diagram is mapped to an SCXML document with the root being the state machine.

```
<ownedBehavior xmi:type="uml:StateMachine" id="_16_5_1_b8d02e4_1391972952867_520512_1044" name="ProducerPSMStates" visibility="public">

<scxml version="1.0" xmlns="http://www.w3.org/2005/07/scxml" xmlns:my="http://my.custom-actions.domain/PRODUCER" name="ProducerPSMStates" initial="region1">
```

2. UML <region> to SCML <state>

Since there is no equivalent for a region in SCXML, it is mapped into a composite state that contains all the remaining states inside it.

```
<region xmi:type='uml:Region' xmi:id='_16_5_1_b8d02e4_1408362414529_426609_465'
name='region1' visibility='public'>
```

```
<state id="region1">
```

3. UML <PseudoState> to SCXML <initial>

In UML an initial pseudostate represents a default vertex. It is mapped to the initial state in the SCML document structure.

```
<subvertex xmi:type='uml:Pseudostate'
xmi:id='_16_5_1_b8d02e4_1408362443489_943063_484' name='begin' visibility='public'/>
```

```
<initial id="begin">
```

4. UML <state> to SCXML <state>

```
<subvertex xmi:type='uml:State' xmi:id='_16_5_1_b8d02e4_1408362443490_654880_488'
name='close' visibility='public'/>
```

```
<state id="close">
```

5. UML < transition> to SCXML <transition>

```
<transition xmi:type='uml:Transition' xmi:id='_16_5_1_b8d02e4_1408364127870_470179_638'
name='T1' visibility='public' source='_16_5_1_b8d02e4_1408362443489_943063_484'
target='_16_5_1_b8d02e4_1408362443490_261204_487'> </transition>
```

```
<transition target="ready" />
```

6. UML transition with trigger sub element to SCXML transition with an event attribute

```
<transition xmi:type='uml:Transition' xmi:id='_16_5_1_b8d02e4_1408364164846_157322_642'
name='T2' visibility='public' source='_16_5_1_b8d02e4_1408362443490_261204_487'
target='_16_5_1_b8d02e4_1408362443489_615661_485'>

	<trigger xmi:type='uml:Trigger' xmi:id='_16_5_1_b8d02e4_1408364175183_723161_643'
name='SendTrigger' visibility='public' event='_16_5_1_b8d02e4_1408364213530_593838_644'/>

</transition>

	<packagedElement xmi:type='uml:SignalEvent'
xmi:id='_16_5_1_b8d02e4_1408364213530_593838_644' name='callSendEvent' visibility='public'/>

<transition id="_16_5_1_b8d02e4_1392133664229_112965_1035" target="sending"

event="callSendEvent"/>
```

7. UML state with entry action to SCXML <onentry>

```
<subvertex xmi:type='uml:State' xmi:id='_16_5_1_b8d02e4_1408362443489_615661_485'
name='sending' visibility='public'>
	<entry xmi:type='uml:Activity' xmi:id='_16_5_1_b8d02e4_1408365151206_870767_804'
name='sendActivity' visibility='public'>
		<node xmi:type='uml:CallOperationAction'
xmi:id='_16_5_1_b8d02e4_1408365265134_660380_808' name='callSendOpAction' visibility='public'
operation='_16_5_1_b8d02e4_1408364861740_896861_711'>
		<argument xmi:type='uml:InputPin'
xmi:id='_16_5_1_b8d02e4_1408365301521_636072_809' name='destination' visibility='public'>
		<xmi:Extension extender='MagicDraw UML 16.5'>	<modelExtension
parameter='_16_5_1_b8d02e4_1408364861754_523511_734'/>
				</xmi:Extension></argument>
		<argument xmi:type='uml:InputPin'
xmi:id='_16_5_1_b8d02e4_1408365301522_117517_810' name='message' visibility='public'>
		<xmi:Extension extender='MagicDraw UML 16.5'>
		<modelExtension parameter='_16_5_1_b8d02e4_1408364861755_761884_735'/>
		</xmi:Extension></argument>
	</node>
	</entry>
</subvertex>



<state id="sending">
	<onentry>
		<my:producer passing="Producer.send(destination,message)" />
	</onentry>
```

Java Code to run the state machine

The UML class diagram contains the system classes. Referring to the mapping process , the classes were imported the Eclipse Modeling Framework EMF to generate Java classes. To add the behavioral features to a class, the Apache Commons SCXML Java API is used. A program is written in Java language in which the method startStateMachine() loads the SCXML file presented in Figure 6 SCXML Document for the Producer State Machine Diagram in . The method also inspects the current state which the object is in, logs the state name and responds accordingly.

```
public  void startStateMachine() throws Exception
     {
           // Read the SCXML document
           SCXML scxml = null;
           ErrorHandler errHandler = null;

           //url is the SCXML document path
                    scxml  =  SCXMLParser.parse(url,  errHandler);//,
customActions);
           SCXMLExecutor exec = null;

            exec = new SCXMLExecutor ();
            JexlEvaluator ev= new JexlEvaluator() ;
            exec.setEvaluator(ev);
            exec.setEventdispatcher(new SimpleDispatcher());
            SimpleErrorReporter er= new SimpleErrorReporter();
            exec.setErrorReporter(er);
            exec.setStateMachine(scxml);
            exec.addListener(scxml,new SimpleSCXMLListener() );
            Context rootCtx=ev.newContext(null);
            exec.setRootContext(rootCtx);
            exec.go();
                                                                  while
(exec.getCurrentStatus().getStates().iterator().hasNext())
            {
                 State              CurrentState      =      (State)
exec.getCurrentStatus().getStates().iterator().next();
                 if(CurrentState.isFinal())
                             break;
                 else
```

```
                        {
                              String stateId=CurrentState.getId();
                                         System.out.println("In  state:
"+stateId);
                                  switch (stateId)
                                  {
                                          case
"region1":display1("region1");
                      break;
                                          case              "begin":
display1("begin");
                  break;
                                          case              "ready":
getReady(exec);
            break;
                                          case            "sending":
sending(exec);
            break;
                                          case              "close":
display1("close");
     connection.jmsConnection.close();
                                    break;
                                          case   "raiseExceptions":

     display1("raiseExceptions");
                              break;
                                  }//switch
                        }//else
                    }//while
     }
```

The sending method is an example of the code that can represent the sending state. The method sends the message once and chooses to transit to close state. The same code can be written using the SCXML constructs in the SCXML document but this needs more investigation.

```
public      void   sending(SCXMLExecutor   ex)throws   ModelException,
JMSException
{
       String nextEvent;
       System.out.println("Producer is sending...");

       //call the object send method
       // Here we are sending the message!
       jmsProducer.send(data.message);
```

```
        System.out.println("Sent message '" + data.message.getText() +
"'");
        // if sent nextEvent="MsgSentEvent";
        nextEvent="closeEvent";
        //if failed  nextEvent="failToSendEvent";
        //if close nextEvent="closeEvent";
                TriggerEvent     event     =     new     TriggerEvent(nextEvent,
TriggerEvent.SIGNAL_EVENT);
        System.out.println("event..."+event.getName());
        ex.triggerEvent(event);
}
```

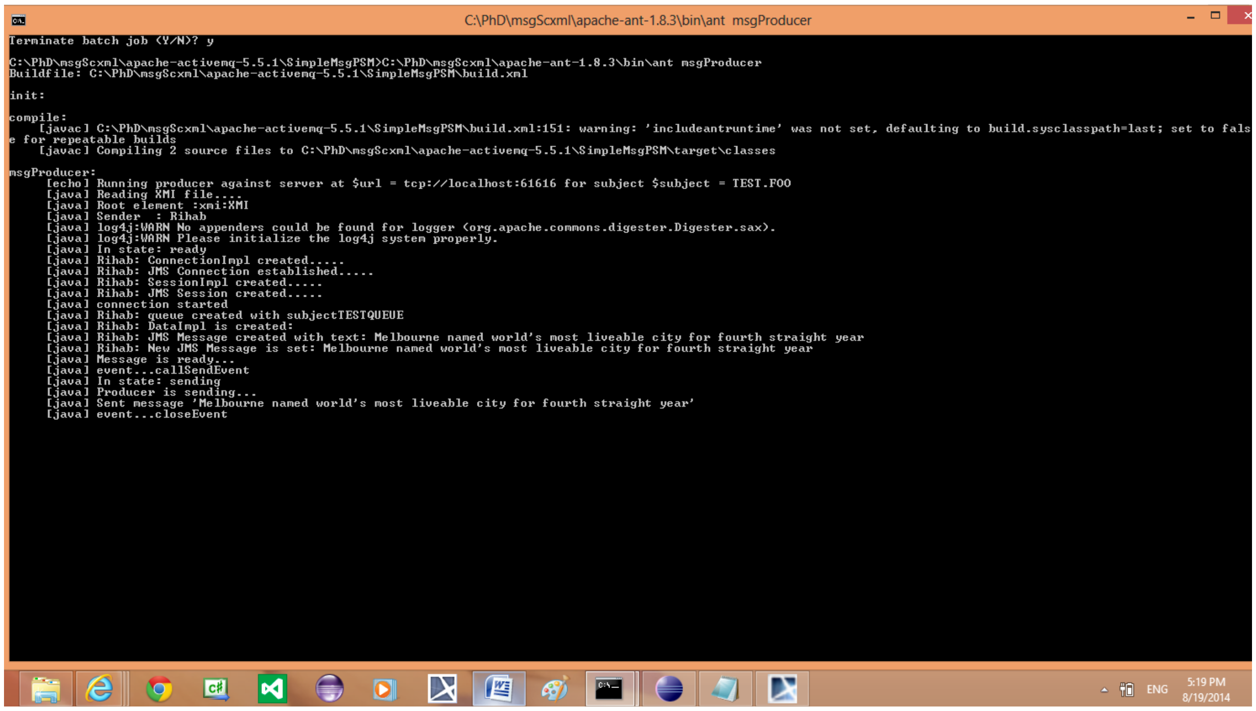
1. Sample Run of the application

The Apache Commons Engine log

```
Aug 19, 2014 4:19:36 PM org.apache.commons.scxml.io.SCXMLParser begin
WARNING: Ignoring element <producer> in namespace "http://my.custom-actions.domain/PRODUCER"
at file:/C:/Users/rahboni/workspace/RunSCXMLProj+/bin/producer.xml:36:44 and digester match
"scxml/state/state/onentry/producer"
Aug 19, 2014 4:19:36 PM org.apache.commons.scxml.io.SCXMLParser begin
WARNING: Ignoring element <producer> in namespace "http://my.custom-actions.domain/PRODUCER"
at file:/C:/Users/rahboni/workspace/RunSCXMLProj+/bin/producer.xml:82:64 and
digester match "scxml/state/state/onentry/producer"
Aug 19, 2014 4:19:36 PM org.apache.commons.scxml.env.SimpleSCXMLListener
onEntry
INFO: /region1
Aug 19, 2014 4:19:36 PM org.apache.commons.scxml.env.SimpleSCXMLListener
onEntry
INFO: /region1/ready
In state: ready
Producer is ready...
Aug 19, 2014 4:19:36 PM org.apache.commons.scxml.env.SimpleSCXMLListener
onExit
INFO: /region1/ready
Aug 19, 2014 4:19:36 PM org.apache.commons.scxml.env.SimpleSCXMLListener
onTransition
INFO: transition (event = callSendEvent, cond = null, from = /region1/ready, to =
/region1/sending)
event...callSendEvent
Aug 19, 2014 4:19:36 PM org.apache.commons.scxml.env.SimpleSCXMLListener
onEntry
INFO: /region1/sending
In state: sending
Producer is sending...
event...closeEvent
```

```
Aug 19, 2014 4:19:36 PM org.apache.commons.scxml.env.SimpleSCXMLListener
onExit
INFO: /region1/sending
Aug 19, 2014 4:19:36 PM org.apache.commons.scxml.env.SimpleSCXMLListener
onTransition
INFO: transition (event = closeEvent, cond = null, from = /region1/sending, to =
/region1/close)
Aug 19, 2014 4:19:36 PM org.apache.commons.scxml.env.SimpleSCXMLListener
onEntry
INFO: /region1/close
Aug 19, 2014 4:19:36 PM org.apache.commons.scxml.env.SimpleSCXMLListener
onExit
INFO: /region1/close
Aug 19, 2014 4:19:36 PM org.apache.commons.scxml.env.SimpleSCXMLListener
onTransition
INFO: transition (event = null, cond = null, from = /region1/close, to =
/region1/end)
Aug 19, 2014 4:19:36 PM org.apache.commons.scxml.env.SimpleSCXMLListener
onEntry
INFO: /region1/end
```

The point here is that the Apache Commons engine logs onEntry, onExit, events, transitions,...etc. The state machine is followed in each step of its lifetime. In the other hand, the programming part is able to log (in black color) the states having actions such as onEntry , onExit, ..etc. The Apache Commons log is expressive enough but its log info is not displayed in the Apache ActiveMQ log as shown below.

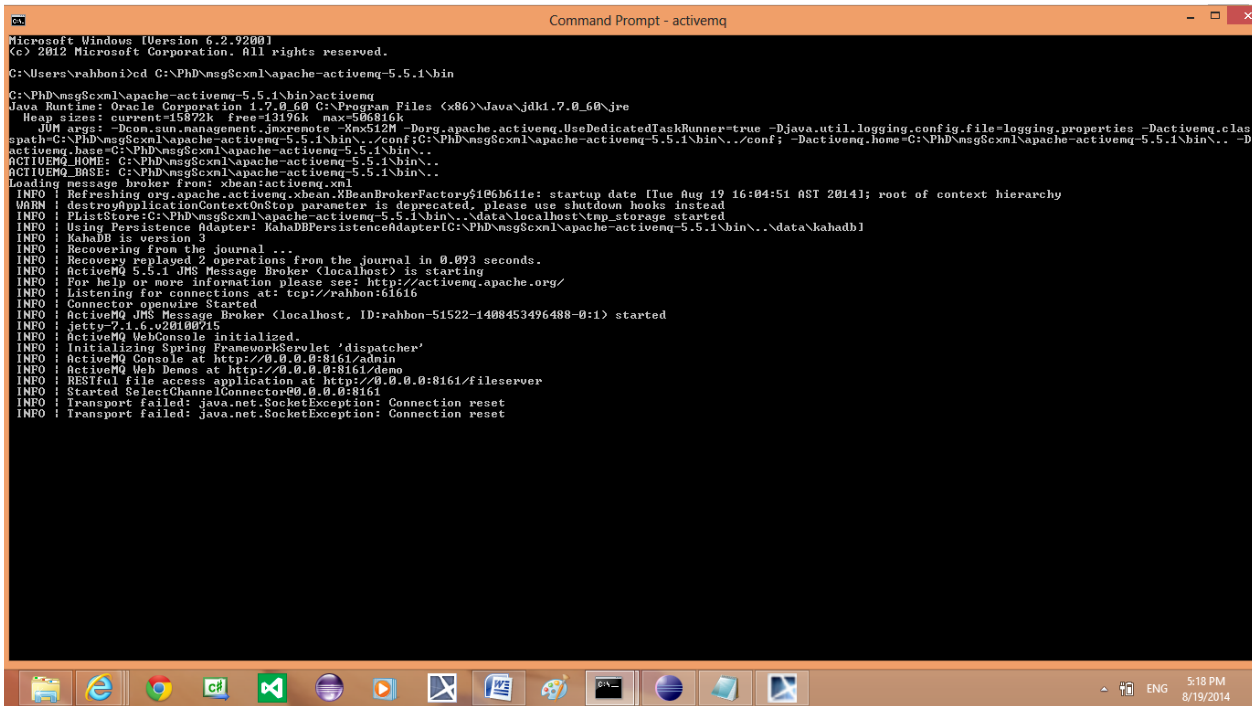


```
Terminate batch job (Y/N)? y

C:\PhD\msgScxml\apache-activemq-5.5.1\SimpleMsgPSM>C:\PhD\msgScxml\apache-ant-1.8.3\bin\ant msgProducer
Buildfile: C:\PhD\msgScxml\apache-activemq-5.5.1\SimpleMsgPSM\build.xml

init:

compile:
    [javac] C:\PhD\msgScxml\apache-activemq-5.5.1\SimpleMsgPSM\build.xml:151: warning: 'includeantruntime' was not set, defaulting to build.sysclasspath=last; set to fals
e for repeatable builds
    [javac] Compiling 2 source files to C:\PhD\msgScxml\apache-activemq-5.5.1\SimpleMsgPSM\target\classes

msgProducer:
     [echo] Running producer against server at $url = tcp://localhost:61616 for subject $subject = TEST.FOO
     [java] Reading XMI file....
     [java] Root element :xmi:XMI
     [java] Sender  : Rihab
     [java] log4j:WARN No appenders could be found for logger (org.apache.commons.digester.Digester.sax).
     [java] log4j:WARN Please initialize the log4j system properly.
     [java] In state: ready
     [java] Rihab: ConnectionImpl created.....
     [java] Rihab: JMS Connection established.....
     [java] Rihab: SessionImpl created.....
     [java] Rihab: JMS Session created.....
     [java] connection started
     [java] Rihab: queue created with subjectTESTQUEUE
     [java] Rihab: DataImpl is created:
     [java] Rihab: JMS Message created with text: Melbourne named world's most liveable city for fourth straight year
     [java] Rihab: New JMS Message is set: Melbourne named world's most liveable city for fourth straight year
     [java] Message is ready...
     [java] event...callSendEvent
     [java] In state: sending
     [java] Producer is sending...
     [java] Sent message 'Melbourne named world's most liveable city for fourth straight year'
     [java] event...closeEvent
```

Figure 6 Executing the ProducerImpl class

```
C:\PhD\msgScxml\apache-activemq-5.5.1\SimpleMsgPSM>C:\PhD\msgScxml\apache-ant-1.8.3\bin\ant
msgProducer
Buildfile: C:\PhD\msgScxml\apache-activemq-5.5.1\SimpleMsgPSM\build.xml

init:

compile:
            [javac]   C:\PhD\msgScxml\apache-activemq-5.5.1\SimpleMsgPSM\build.xml:151:   warning:
'includeantruntime' was not set, defaulting to build.sysclasspath=last; set to false for repeatable builds
               [javac]   Compiling   2   source   files   to   C:\PhD\msgScxml\apache-activemq-
5.5.1\SimpleMsgPSM\target\classes

msgProducer:
   [echo] Running producer against server at $url = tcp://localhost:61616 for subject $subject = TEST.FOO
   [java] Reading XMI file....
   [java] Root element :xmi:XMI
   [java] Sender  : Rihab
   [java] log4j:WARN No appenders could be found for logger (org.apache.commons.digester.Digester.sax).
   [java] log4j:WARN Please initialize the log4j system properly.
   [java] In state: ready
   [java] Rihab: ConnectionImpl created.....
   [java] Rihab: JMS Connection established.....
   [java] Rihab: SessionImpl created.....
   [java] Rihab: JMS Session created.....
   [java] connection started
   [java] Rihab: queue created with subjectTESTQUEUE
   [java] Rihab: DataImpl is created:
    [java] Rihab: JMS Message created with text: Melbourne named world's most liveable city for fourth
straight year
    [java] Rihab: New JMS Message is set: Melbourne named world's most liveable city for fourth straight
year
```

```
[java] Message is ready...
[java] event...callSendEvent
[java] In state: sending
[java] Producer is sending...
[java] Sent message 'Melbourne named world's most liveable city for fourth straight year'
[java] event...closeEvent
```



```
Microsoft Windows [Version 6.2.9200]
(c) 2012 Microsoft Corporation. All rights reserved.

C:\Users\rahboni>cd C:\PhD\msgScxml\apache-activemq-5.5.1\bin

C:\PhD\msgScxml\apache-activemq-5.5.1\bin>activemq
Java Runtime: Oracle Corporation 1.7.0_60 C:\Program Files (x86)\Java\jdk1.7.0_60\jre
  Heap sizes: current=15872k  free=13196k  max=506816k
    JVM args: -Dcom.sun.management.jmxremote -Xmx512M -Dorg.apache.activemq.UseDedicatedTaskRunner=true -Djava.util.logging.config.file=logging.properties -Dactivemq.clas
spath=C:\PhD\msgScxml\apache-activemq-5.5.1\bin\../conf;C:\PhD\msgScxml\apache-activemq-5.5.1\bin\../conf; -Dactivemq.home=C:\PhD\msgScxml\apache-activemq-5.5.1\bin\.. -D
activemq.base=C:\PhD\msgScxml\apache-activemq-5.5.1\bin\..
ACTIVEMQ_HOME: C:\PhD\msgScxml\apache-activemq-5.5.1\bin\..
ACTIVEMQ_BASE: C:\PhD\msgScxml\apache-activemq-5.5.1\bin\..
Loading message broker from: xbean:activemq.xml
 INFO | Refreshing org.apache.activemq.xbean.XBeanBrokerFactory$1@6b611e: startup date [Tue Aug 19 16:04:51 AST 2014]; root of context hierarchy
 WARN | destroyApplicationContextOnStop parameter is deprecated, please use shutdown hooks instead
 INFO | PListStore:C:\PhD\msgScxml\apache-activemq-5.5.1\bin\..\data\localhost\tmp_storage started
 INFO | Using Persistence Adapter: KahaDBPersistenceAdapter[C:\PhD\msgScxml\apache-activemq-5.5.1\bin\..\data\kahadb]
 INFO | KahaDB is version 3
 INFO | Recovering from the journal ...
 INFO | Recovery replayed 2 operations from the journal in 0.093 seconds.
 INFO | ActiveMQ 5.5.1 JMS Message Broker (localhost) is starting
 INFO | For help or more information please see: http://activemq.apache.org/
 INFO | Listening for connections at: tcp://rahbon:61616
 INFO | Connector openwire Started
 INFO | ActiveMQ JMS Message Broker (localhost, ID:rahbon-51522-1408453496488-0:1) started
 INFO | jetty-7.1.6.v20100715
 INFO | ActiveMQ WebConsole initialized.
 INFO | Initializing Spring FrameworkServlet 'dispatcher'
 INFO | ActiveMQ Console at http://0.0.0.0:8161/admin
 INFO | ActiveMQ Web Demos at http://0.0.0.0:8161/demo
 INFO | RESTful file access application at http://0.0.0.0:8161/fileserver
 INFO | Started SelectChannelConnector@0.0.0.0:8161
 INFO | Transport failed: java.net.SocketException: Connection reset
 INFO | Transport failed: java.net.SocketException: Connection reset
```

Figure 6 Apache Active MQ Server is running

Proposed Approach Results and Discussion

Overview

This chapter presents a set of model transformations on UML class and state machine models. Each transformation is provided with an explanation of its purpose, examples of its use and conditions necessary for its correct use. The results are presented and discussed with examples.

Results Summarized

Model transformations of behavior models represented as UML state machine in this research can be classified into five categories: Single State to Single State, Single State to more than one State, Single State to part of a composite state, orphan state and forbidden state transformations.

If the PSM has a forbidden state which can be entered for a given PIM, and the forbidden state has actions that involved changes to any PIM class instances, then the PIM must be enhanced to take account that PSM behaviour, otherwise a PIM state can map to a composite PSM state including the forbidden state. A test for this kind of situation would be valuable. Some observations are following:

a. A forbidden state with no guard predicate will generally do something necessary for the operation of the PSM which is not visible in the PIM, so the mapping is to a composite state.

b. A forbidden state with a guard predicate. A PIM may be constrained in such a way that the PSM guard predicate will always evaluate to true, in which case situation a above is obtained, or always false, in which case the forbidden state can never occur.

c. In fact, if a forbidden state has no action with an effect on the PIM database (PIM Classes model instances), then what it does would appear to be irrelevant to the PIM.

d. In the case where a forbidden state has a guard predicate which may evaluate to either true or false (this requires that the guard predicate include terms which involve mapped PIM class model instances), and the forbidden state has actions which change PIM class model instances, then the PSM behaviour is richer than the PIM, and the PIM needs to be enhanced to make the necessary specifications.

Another aspect in the state machine models is the constraints in various forms. A constraint is formulated on the level of classes, but its semantics is applied on the level of objects.

a. In the state models of both the PIM and PSM the predicates can take different forms.

b. PSM class may have more attributes than PIM class, if such attributes existed in the constraint they need a decision. If the PSM attributes are left in the expression as specified by the PSM, then we have to note that the values are PSM specific and are not specified by the PIM instance model.

c. PSM specific classes that are not part of the PIM classes may also have their own constraints. These classes may be part of an equivalence class too.

d. The relation between the attributes used in the guard expression involves the mapping of PIM class model to the PSM class model first in order to map the attributes values of instances accordingly.

e. The guard predicate in the PSM may be manually edited to find the corresponding semantically equal behavior as specified by the PIM.

Discussion

Forbidden States Mapping

The fifth type of transformation identified is the “forbidden state” transformations where the PIM state model has no equivalent for the PSM state.

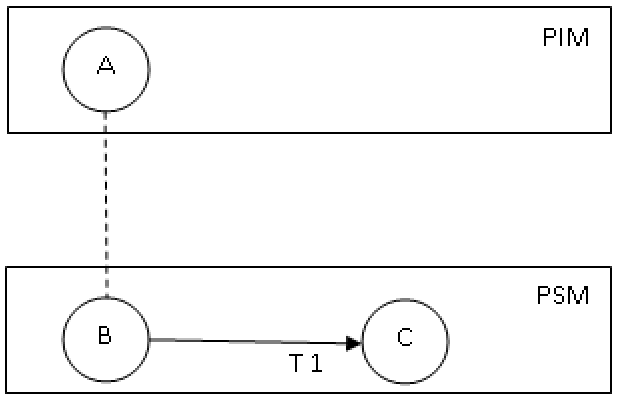


Figure 7: Forbidden state C targeted by transition T1

Figure 7 : Forbidden state C targeted by transition T1 shows state B which is equivalent to state A from the PIM state machine model. State B links to state C with the transition T1.

The mapping decisions can be as follows:

Non guarded Transitions from B to C:

When T1 has no guard condition:

- Safely ignore state C and do not include it in the PSM.

  In this case the PIM is followed strictly. The PSM contains more functionality specified by the more states it has. In order for a PSM to implement a PIM , its state should be superset of the PIM states. The forbidden states can be used to enhance the PIM and alter its specification by mapping back from the PSM states to the PIM states.
- Consider state C in the equivalence class of the PSM

  In this case the logic is to be completed by visiting the forbidden state C from state B, since the transition T1 has no condition.

  - The equivalence class would be A= { B, C }. It can be mapped as a composite state, or another new region containing both state B and C.
  - An example to this situation is the printing of a receipt in the first case study. The PIM doesn't express explicitly that the successful completion of a transaction would result in a print of a receipt describing the transaction. By mapping the constraints , the states and transitions ends up in the idle state of the PIM state machine, while it continues to print and release the card in the PSM. In this situation it is recommended to map the additional PSM states to the PIM one in order to complete the application logic.
  - Another example is the cleanUp forbidden state in the second case study. The PIM transitions to the end state when finished sending or receiving. The PSM cleans up the resources such as closing the connection object before ending. The cleanUp state is required in the PSM so it is mapped and added to the equivalence class.

<u>Guarded Transitions from B to C:</u>

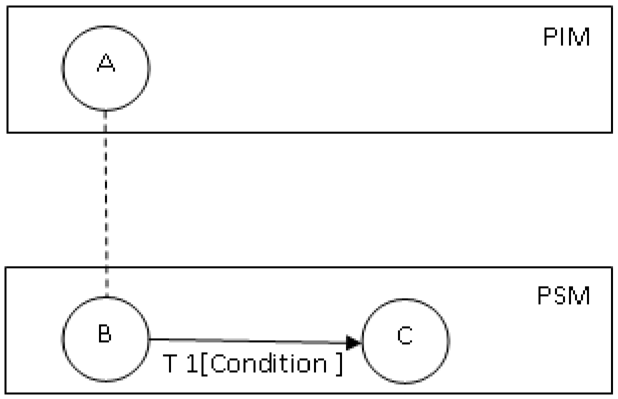


Figure 7 : Forbidden state C targeted by a guarded transition T1

In a state machine model, a guard condition is a boolean condition that is evaluated when the transition is initiated. The transition to the target state occurs when the guard condition is evaluated to true. In the UML notation, guard conditions are shown in square brackets.

It is possible that in an implementation of a particular PIM that the guard for a forbidden transition is always false. In this case, a forbidden state can safely be ignored as it can never be reached in that application.

UML Constraints Mapping

Invariant

Assuming the following invariant as follows:

context Card **inv**: : expirationDate.isAfter(today)

1. Determine the context of the constraint in the PSM class model, let us call it PSMContext
2. Determine the PIM class that is equivalent to the PSM class denoted by the <class name>, let us call it PIMContext.
3. Map the attributes of the PIMContext to the PSMContext
4. Check the OCL expression
5. For each attribute in the constraint expression, map the equivalent attribute from the PIMContext class.
6. Assess the OCL functions used (involves checking the semantic of the constraint)

According to the class model mapping of the PIM to PSM, the ATMCard is mapped to a Card class in the PSM. Since the Card is the context of the constraint, then we are going to map the equivalent class attribute value for each object of type ATMCard from the instances model into the PSM instance model. Note that the name of both attributes need not be the same.

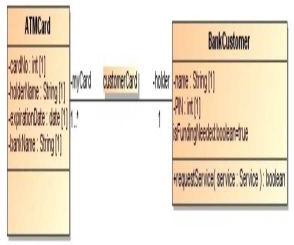


Figure 7 PIM ATMCard Class

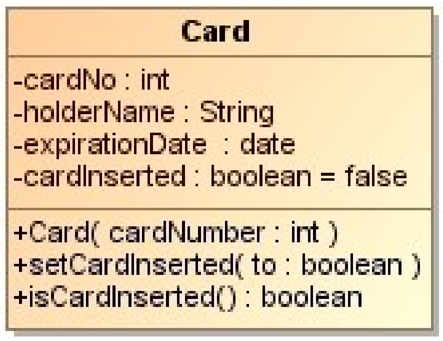

Card
-cardNo : int
-holderName : String
-expirationDate : date
-cardInserted : boolean = false
+Card(cardNumber : int)
+setCardInserted(to : boolean)
+isCardInserted() : boolean

Figure 7 PSM Card Class

Observations:

1. PSM class may have more attributes than PIM class, if such attributes existed in the constraint they need a decision. If the PSM attributes are left in the expression as specified by the PSM, then we have to note that the values are PSM specific and are not specified by the PIM instance model.
2. PSM specific classes that are not part of the PIM classes may also have their own constraints. These classes may be part of an equivalence class too.

Pre and Postcondition

context ATM::dispence(amount : Integer)

pre: self.inState=performingTransaction *or*

pre: oclInState(performingTransaction);

The pre condition specifies that the state machine that is owned by the context object- *ATM* object - is in a specific state in order to enable the execution of the operation *dispense*. In this case the mapping should check that the state specified is equivalent to some state in the PIM and if there is no constraint , a decision has to be made. Because the PSM constraints are stronger than PIM ones, the decision here can be to keep the constraint as it is in the PSM.

State Machine Constraint

A Constraint may be applied to a State machine in the same way as for a Class to specify an invariant of the State machine. The guard condition of a State machine transition may be specified by associating a constraint with a transition

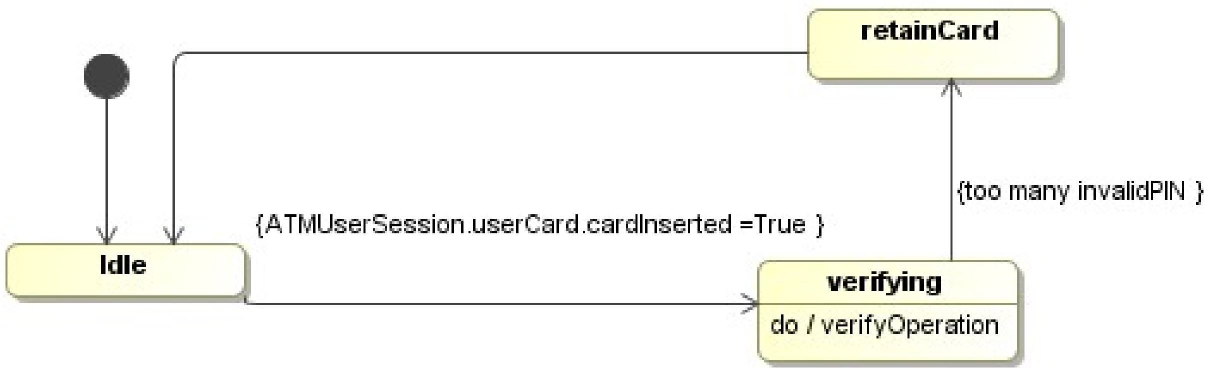


Figure 7 Part of the PSM ATM states model

In Figure 7 Part of the PSM ATM states modelFigure 7 Part of the PSM ATM states model above the transition from Idle state to Verifying state is constrained with a guard condition that checks the boolean property "cardInserted" in the PSM Card class. The navigation from the ATM context - who owns the states - to the class Card is done through the userCard association end that associated with the ATMUserSession class. The "fundingNeeded" is a property of the class BankCustomer in the PIM. BankCustomer is mapped to ATMUserSession in the class model mapping . Each ATMUserSession is associated with a user card of Card class. So the relation between the attributes used in the constraint expression involves the mapping of PIM class model to the PSM class model first in order to map the attributes values of instances accordingly.

Example 2

In the state models of both the PIM and PSM the predicates can take different forms. For example the PIMClient class in the messaging case study has an attribute newsCount that specify the number of messages generated and sent , with a default value set to 3, while the PSM Client class has a guard predicate again? which is true if we want to stop message generation and sending. The guard predicate in the PSM may be edited to find the corresponding semantically equal behavior as specified by the PIM.

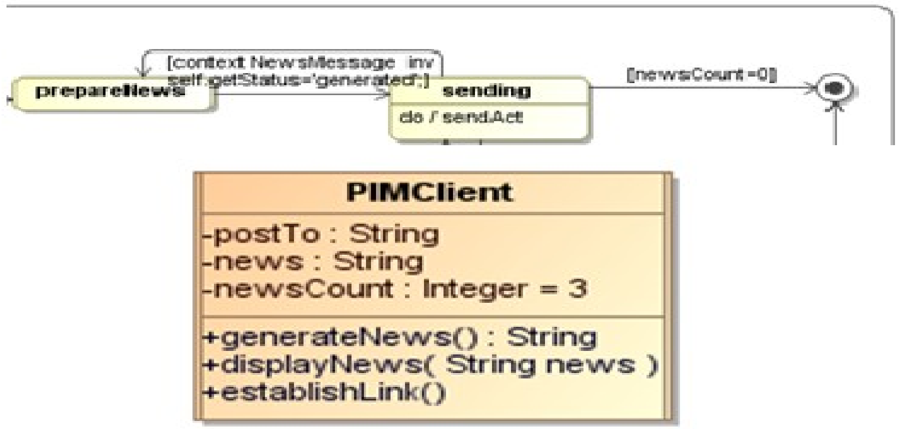


Figure 7 PIMClient and the sending state

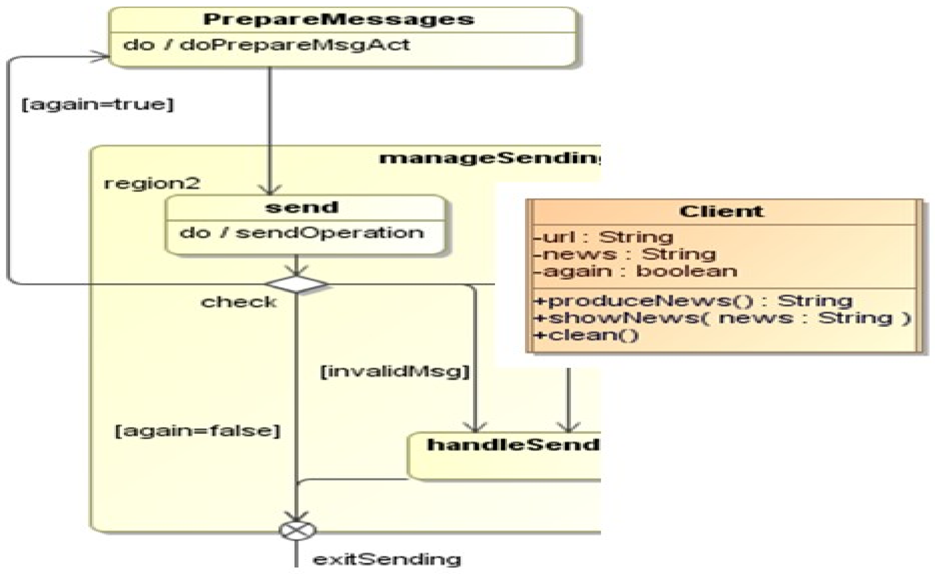


Figure 7 Client class and send state

A mapping solution can be as follows: The specification body of the PSM constrain has to include the check expressed by the PIM constraint.

<body> count=0 </body>

Beside that a new attribute or a complete data structure to hold the attributes and values from the PIM should be invented and attached to the PSM in order to preserve the semantic specified in the PIM.A semi manual approach is needed.

Conclusion

Overview

This chapter is a conclusion of the thesis. Answers to the research questions and explanation are drawn here. The chapter also presents how the objectives were achieved beside showing the limitations and future directions.

Summary of The Results

MDA is about using models as first class artifacts in the development process from designs to implementations thus providing an end to end complete process. Automating the path from models to executable systems is a featured proposition in MDA that reduce cost, time and improving their fitness for purpose. Our end to end engineering approach creates domain assets in the form of metamodels , models and QVT transformations for software solution developers.

The built PSM for messaging system could be re-used to afford many products from the domain although it is not MDA or OMG standard. Ideally, the standard PSMs will allow the software vendors to use them off the shelf and generate code automatically.

Taxonomy and guidelines for state machine mappings will also be valuable to the architects and developers. State of the art tools in the MDA context was identified and used that pave the way for developers who are examining the MDA process.

The research question was

How to automate software application generation using UML behavior models in MDA approach?

The answer is provided through the thesis and covered by the relevant literature. Automation is achieved by defining and discussing the mapping relations between the PIM and PSM and also from the PSM to code. The guidelines for doing the transformation is established and implemented successfully that resulted in executing the program modeled in the first place as a PIM thus providing the evidence of MDA concepts. In the next paragraphs our approach is compared to the ideal MDA and the traditional software development methods. The models were formally represented as UML models which is the standard modeling language from OMG.

Approaching the problem using case study methodology is considered an evaluation to the problem. Moreover and considering the second case study we had tried to build the same application in two ways: One that uses pure MDA approach with current tools and languages , we call it our approach. The other one is model driven but not MDA in the sense of no PIM , PSM nor transformation is used, we can call it traditional approach. In the recent future the MDA is going to be mature enough and the software development process can be as described by the MDA guides, we call this (dreamt) optimal MDA.

Table 8. Optimal MDA , our approach and traditional code generation approaches compared

	Optimal MDA	**Our Approach**	**Traditional Code generation from models**
PIM	Built	Built	Built
PSM	Standard and ready on the shelf	Built	-
Messaging PSM	Standard and ready on the shelf	Built	API level not model level
SCXML Document	Standard and ready on the shelf	Built	-
alternatives	Existed as other PSMs	Need investment (time and effort)	Hard coding of everything again
Code written	Transformation	QVT + minimal coding in Java for illustration	Code for relating classes in a specific language+ business logic

Regarding the PIM that captures the application logic, all the methods get the benefit of having a model of the system. The PIM model promotes the reuse and conformance with the requirement. The PSM in the optimal MDA process is ready and the architect may select one that suits the needs. One of the uses of a PSM is to suggest functionality that the application may need to use. For example with commercial accounting software, the software package reflects industry best practice, and the customer will often change their procedures to take advantage of the facilities provided by the package. In our approach and because of the lack of standard PSMs we had built the PSM. In the traditional approach there is no notion of a PSM. Specifically we had written code and glue code to link the objects created by the models and the object needed in the execution environment. So we were working in the API level and not the model level.

In the optimal MDA, the platform models or platform specific models for messaging systems and SCXML and alike technologies are going to be standardized and available to compare, select and use. In our approach we had built them so using an alternative is costly. On the other hand, there were examples of changes to the models being formulated, agreed and deployed in the working system. We had experienced the built models being enhanced as if we had chosen a deficient PSM at the beginning and also experienced working with the complete and stable PSM after it reaches its stability when it had the functions most applications look for. This experience is what we can find in the context of the optimal MDA when judging about which PSM to choose. The messaging platform chosen and modeled is a standard one (but not MDA standard) that has sufficient facilities to implement the PIM.

Augmenting the structural model with behavioral models in terms of state machine models allowed the application logic to be available in a higher level constructs that mapped to the API level objects (instances) by transformations not by code writing. The code written in our approach is the java code that initializes the state machine and loads the files that contains the instances. We used that code for illustration purposes only and to show the server log messages. Such code could easily be illuminated. The effort of programming -if we can say- is devoted to writing the transformation specification and rules.

The main objective of this research is to find an engineering method for mapping UML state machine behavior diagrams from PIM to PSM. The objectives were achieved by :

a) Designing and modeling a suitable software application PIM structures Using the UML class model.

b) The PIM is enriched and complemented with UML state machine models beside the UML class diagram. The state machine models assist in providing enough information in the PIM that enables the automatic generation of code artifacts.

c) Developing a generic model to present the implementation of the chosen software application functionality described in the PIM model. This generic model was used as PSM (Target Model).

d) Specifying the mapping rules to transform the PIM to the generic model PSM containing both attributes (structure) and methods (behavior) for the system.

e) Developing a module (using QVT) to carry out the mapping specification in the previous step. The module is used to execute the model transformation and provide the platform connection between the design model (PIM instances) and the instances of the generic PSM.

f) QVT was the suitable transformation languages and UML metamodels provide a base for better facilitating the mapping process.

g) The proposed approach was evaluated by developing a system using MDA best practices and transferring the generated artifacts (programs, configuration files and all the generated classes to a suitable environment to be executed.

The way we approached the problem, the development of our methodology, and the integration of our approach with the programming technologies, modeling tools, ,development frameworks and execution environments, remains the subject of future research. In the meantime, we hope that our success in applying Model Driven Architecture techniques in this study might inspire others to adopt a similar approach, and thus make a positive effort towards the quality, reliability, and maintainability of enterprise level information systems.

Limitations

Lack of Supporting Tool Set

MDA is a young discipline in which a mature set of tools are still required. In this research we had consulted many different set of tools each with its own strengths and weaknesses. Although the tools may support export/ import models and model elements but the resulted files were too big and error prone. Some features were not supported in another tool that would result in different representation of the same model. The Eclipse Modeling Framework EMF is based on Ecore which is a subset of UML that does not include behavioral features of classes. The modeling project of EMF would be an enhancement towards providing a complete model representation and generation.

Automating Code Generation

Automatic code generation provides an increase in productivity. Generators can produce thousands lines of codes in short time. Tedious and boring parts of code can be also generated instead of hand written. Automation can also provide architecture consistency when programmers work within the architecture. Beside that automatic code generation lifts the problem to a higher level thus providing an easier porting to different languages and platforms. In contrast to the mentioned advantages, generators themselves - programs that produce programs- have to be written first. So there will always be hand coding required. The code generation aspect fell partially in our research scope because Model-to-Model Transformation was considered as the main scope of this thesis. We had tested code generation to provide an end to end transformation and provide a complete MDA approach.

Lack of Standard Models

In both case studies we had built the platform independent models and the platform specific models. The process continually involves enhancing and enriching the models which can create a sort of bias. The MDA approach specify that the platforms should be standardized and can be utilized by many software builders. This is the optimal case.

Manual Work

In the MDA process, models are the heart. Because the whole MDA process is driven by the PIM, and the PIM is automatically transformed into a PSM, and from there to code, modeling in effect will become programming on a higher level. The PIM specifies the structure and behavior that need to be produced. In this research some guidelines were identified and a semi manual and sometimes manual contributions could not be avoided. This is because of the richness and complexity of behavior models beside the many under research issues. When the MDA become mature enough, there will be far less programming, or that remains to be done by hand. Programming, in the sense of building software systems, will eventually become modeling. All software development effort will be focused on producing a good, high level, independent model of the system.

Future Work

More work is to be carried out in the constraints parts of the state machine models. Moreover state machine elements such as actions, Initial node , Final node, Fork node , Merge node with decision node , has to be deeply investigated and decided upon how to be mapped.

In addition, we plan to test how to select a suitable platform specific model based on the structure and behavior specified in the platform independent model. Another interesting experiment would be to translate the transformed constructs into a working software using the MDA approach and tool sets in a fully automatic way.

Summary

This chapter concludes the mapping of UML state machine models from the PIM to the PSM and to code. It provides answers to the research questions and how the objective were achieved. Some limitations regarding the manual work, lack of tool support and the lack of standard models were discussed. Finally some future work were described.

Abdalla, O.M.M. & Abdullah, A. Bin, 2011. Mapping of Behavior Model using Model-Driven Architecture. *International Journal of Computer …*. Available at: http://www.ijcaonline.org/volume13/number8/pxc3872495.pdf [Accessed May 28, 2011].

Ahmed, abd elgaffar hamed, 2010. *Automating specification to implementation software development using model driven architecture ali universiti teknologi malaysia*.

Ahmed, R.E., Colomb, R.M. & Ahmed, A.H., 2013. A method for mapping state machine behavior models in MDA issues and challenges. In *2013 INTERNATIONAL CONFERENCE ON COMPUTING, ELECTRICAL AND ELECTRONIC ENGINEERING (ICCEEE)*. IEEE, pp. 404–409. Available at: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6633971 [Accessed March 22, 2015].

Aksit, M. et al., 2009. Behaviour Modelling in Model Driven Architecture. In *CTIT Workshop Proceedings Series WP09-04*.

Anon, Erlang Programming Language. Available at: http://www.erlang.org/ [Accessed June 26, 2015a].

Anon, MagicDraw. Available at: http://www.nomagic.com/products/magicdraw.html [Accessed March 20, 2015b].

Apache, 2015. Apache ActiveMQ ™ -- Index. Available at: http://activemq.apache.org/ [Accessed March 25, 2015].

Apache, SCXML - Commons SCXML. Available at: http://commons.apache.org/proper/commons-scxml/ [Accessed March 25, 2015].

Burke, P.W. & Sweany, P., 2008. Automatic Code Generation Through Model-Driven Design. *20th System and Software Technology Conference, Las Vegas NV*.

Creswell, J.W., 2012. *Educational research: Planning, conducting, and evaluating quantitative and qualitative research*,

Dijkstra, E.W., 1976. *A Discipline of Programming*, Prentice Hall.

Domínguez, E. et al., 2012. A systematic review of code generation proposals from state machine specifications. *Information and Software Technology*, 54, pp.1045–1066.

Eric Cariou, UML meta-model extension for state machine instance specification. Available at: http://ecariou.perso.univ-pau.fr/contracts/uml-state-machine-extension.html [Accessed March 24, 2015].

Flater, D., 2002. Impact of model-driven standards. *Proceedings of the 35th Annual Hawaii International Conference on System Sciences*.

Garlan, D. & Shaw, M., 1993. An Introduction to Software Architecture. In *Advances in Software Engineering and Knowledge Engineering*. pp. 1–40. Available at: http://www.cs.cmu.edu/afs/cs/project/able/ftp/intro_softarch/intro_softarch.pdf.

Harel, D., 1987. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3), pp.231–274. Available at: http://www.sciencedirect.com/science/article/pii/0167642387900359 [Accessed March 1, 2015].

Hevner, A.R. et al., 2004. Design Science in Information Systems Research. *MIS Quarterly*, 28, pp.75–105. Available at: http://dblp.uni-trier.de/rec/bibtex/journals/misq/HevnerMPR04.

Kalnins, A. et al., 2009. Behaviour modelling notation for information system design. *Proceedings of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture - BM-MDA '09*, pp.1–7. Available at: http://portal.acm.org/citation.cfm?doid=1555852.1555854.

Mcneile, A. & Simons, N., 2004. METHODS OF BEHAVIOUR MODELLING A Commentary on Behaviour Modelling Techniques for MDA. *A Commentary on Behavior Modelling Techniques for MDA. Metamaxim Ltd*, 201, pp.1–11. Available at: http://www.metamaxim.com/download/documents/Methods.pdf.

Members, C.J.W., 2004. J/eXtensions for Financial Services (J/XFS) for the Java Platform - Part 1: Base Architecture - Programmer's Reference. *CEN/ISSS J/XFS Workshop Event (London)*, Ref. No. C.

Mens, T., Czarnecki, K. & Gorp, P. Van, 2005. 04101 Discussion -- A Taxonomy of Model Transformations. *Language Engineering for ModelDriven Software Development*, pp.1–10. Available at: http://drops.dagstuhl.de/opus/volltexte/2005/11.

O M G, 2011. OMG Unified Modeling Language TM (OMG UML), Infrastructure. , (January).

Object Management Group (OMG), 2008. MOF Model to Text Transformation Language 1.0. *Formal/2008-01-16*, (January).

OMG, 2003. MDA Guide Version 1.0. 1. , (June), p.51. Available at: http://www.omg.org/docs/omg/03-06-01.pdf.

OMG, 2015. MDA OMG web page. Available at: http://www.omg.org/mda/ [Accessed July 5, 2015].

OMG, 2011a. Meta Object Facility (MOF) 2 . 0 Query / View / Transformation Specification. , version 1.(January), p.246. Available at: http://www.omg.org/spec/QVT/1.1.

OMG, 2006. Meta Object Facility (MOF) Core Specification. *Management*, 080907(January), pp.1–76. Available at: http://www.omg.org/spec/MOF/2.0/.

OMG, 2010. Object Constraint Language. *Language*, 03(December).

OMG, 2014a. Object Management Group, Model Driven Architecture (MDA). , (June), pp.1–15. Available at: http://www.omg.org/cgi-bin/doc?omg/03-06-01.

OMG, 2014b. OMG Meta Object Facility (MOF) Core Specification, Version 2.1.4. , 2(April).

OMG, 2011b. OMG Unified Modeling Language TM (OMG UML), Superstructure. , (January). Available at: http://www.omg.org/spec/UML/2.4/Superstructure.

OMG, 2011c. OMG Unified Modeling Language TM (OMG UML), Superstructure. , (August).

OMG, 2014c. Success Stories. Available at: http://www.omg.org/mda/products_success.htm [Accessed March 22, 2015].

OMG, 2004. UML 2.4.1 Superstructure Specification. *October*, 02(August), pp.1–786.

OMG, 2014d. XML Metadata Interchange (XMI) Specification. *Interchange*, 2(April), pp.1–112. Available at: http://www.omg.org/spec/XMI/2.4.2/PDF/.

Oracle, 2013. Java Message Service Concepts. , (March). Available at: http://docs.oracle.com/javaee/6/tutorial/doc/bncdq.html.

Riccobene, E. & Scandurra, P., 2009. Weaving executability into UML class models at PIM level. In *Proceedings of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture - BM-MDA '09*. pp. 1–9. Available at: http://portal.acm.org/citation.cfm?doid=1555852.1555853.

Richter, W. & Conti, M., 2004. The oligomerization state determines regulatory properties and inhibitor sensitivity of type 4 cAMP-specific phosphodiesterases. *Journal of Biological Chemistry*, 279, pp.30338–30348. Available at: http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:MDA+Guide+Version+1.0.1#0\nhttp://dret.net/biblio/reference/mda10.

Rihab Eltayeb Ahmed, N.S., 2012. BEHAVIOR MODELING, LANGUAGES AND DIAGRAMS IN COMPONENT BASED SOFTWARE DEVELOPMENT. *Journal of Asian Scientific Research*, 2(11), pp.773–781. Available at: http://www.aessweb.com/pdf-files/773-781.pdf [Accessed March 22, 2015].

Sunitha, E. V., and P.S., 2012. Translation of behavioral models to source code. In *12th International Conference on Intelligent Systems Design and Applications (ISDA)*. IEEE, pp. 598–603.

The Eclipse Foundation, Eclipse Modeling Project. Available at: http://www.eclipse.org/modeling/emf/ [Accessed March 22, 2015].

Tratt, L., 2005. Model transformations and tool integration. *Software and Systems Modeling*, 4(2), pp.112–122. Available at: http://link.springer.com/10.1007/s10270-004-0070-1 [Accessed February 15, 2015].

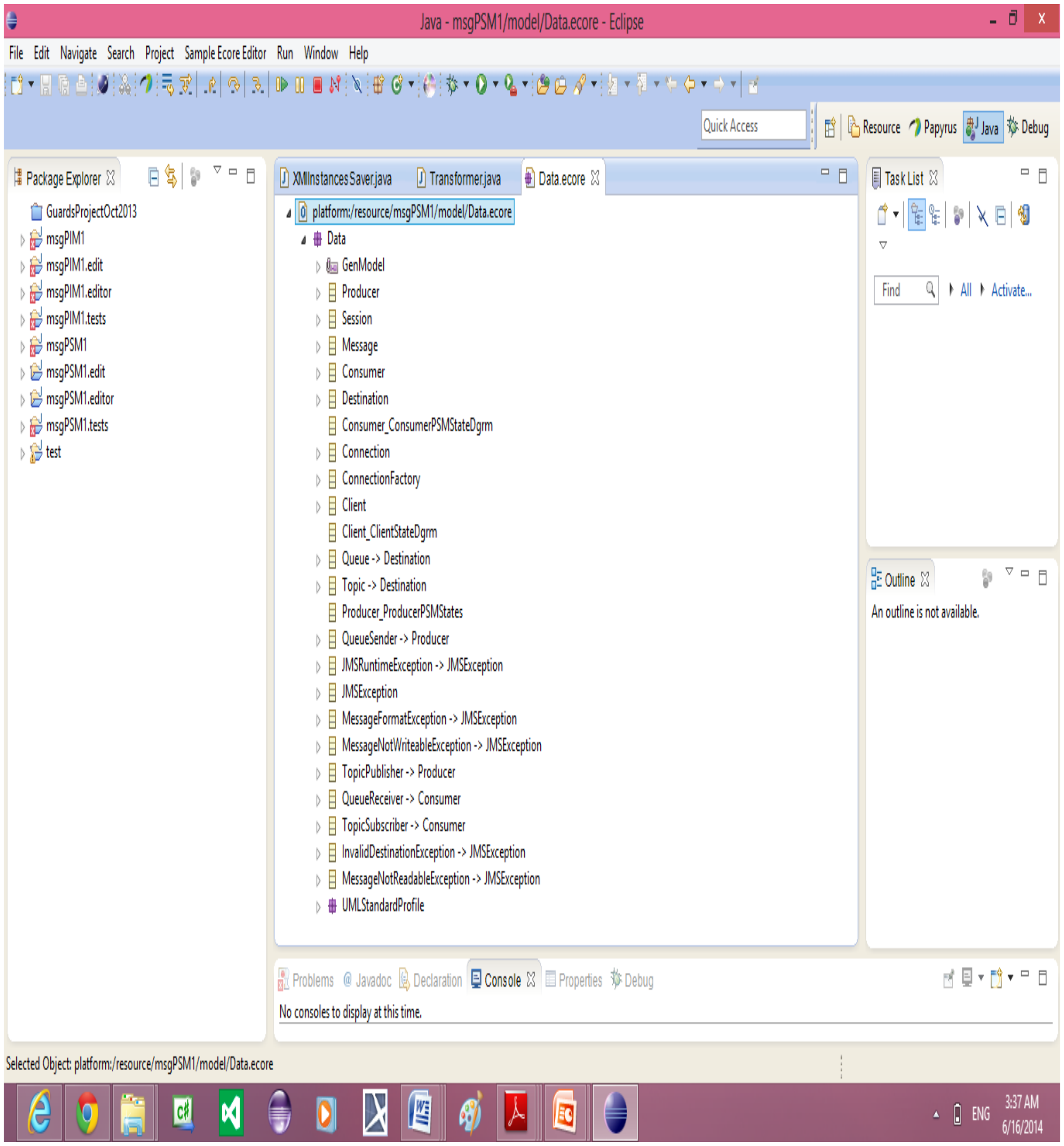
W3C, State Chart XML (SCXML): State Machine Notation for Control Abstraction. Available at: http://www.w3.org/TR/2014/WD-scxml-20140529/#Examples [Accessed March 25, 2015].

PUBLICATIONS

[1] Ahmed, R.E., Colomb, R.M. & Ahmed, A.H., 2013. A Method for Mapping State Machine Behavior Models in MDA, Issues and Challenges. In *2013 International Conference on Computing Electrical and Electronic Engineering (ICCEEE)*. IEEE, pp. 404–409. Available at: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6633971

[2] Rihab Eltayeb Ahmed, N.S., 2012. Behavior Modeling Languages and Diagrams in Component Based Software Development. *Journal of Asian Scientific Research*, 2(11), pp.773–781. Available at: http://www.aessweb.com/pdf-files/773-781.pdf

Screenshots of the Models


Java - msgPSM1/model/Data.ecore - Eclipse
File Edit Navigate Search Project Sample Ecore Editor Run Window Help
Quick Access
Resource Papyrus Java Debug
Package Explorer
XMIInstancesSaver.java
Transformer.java
Data.ecore
Task List
Find
All
Activate...
Outline
Problems Javadoc Declaration Console Properties Debug
ENG
3:37 AM
6/16/2014

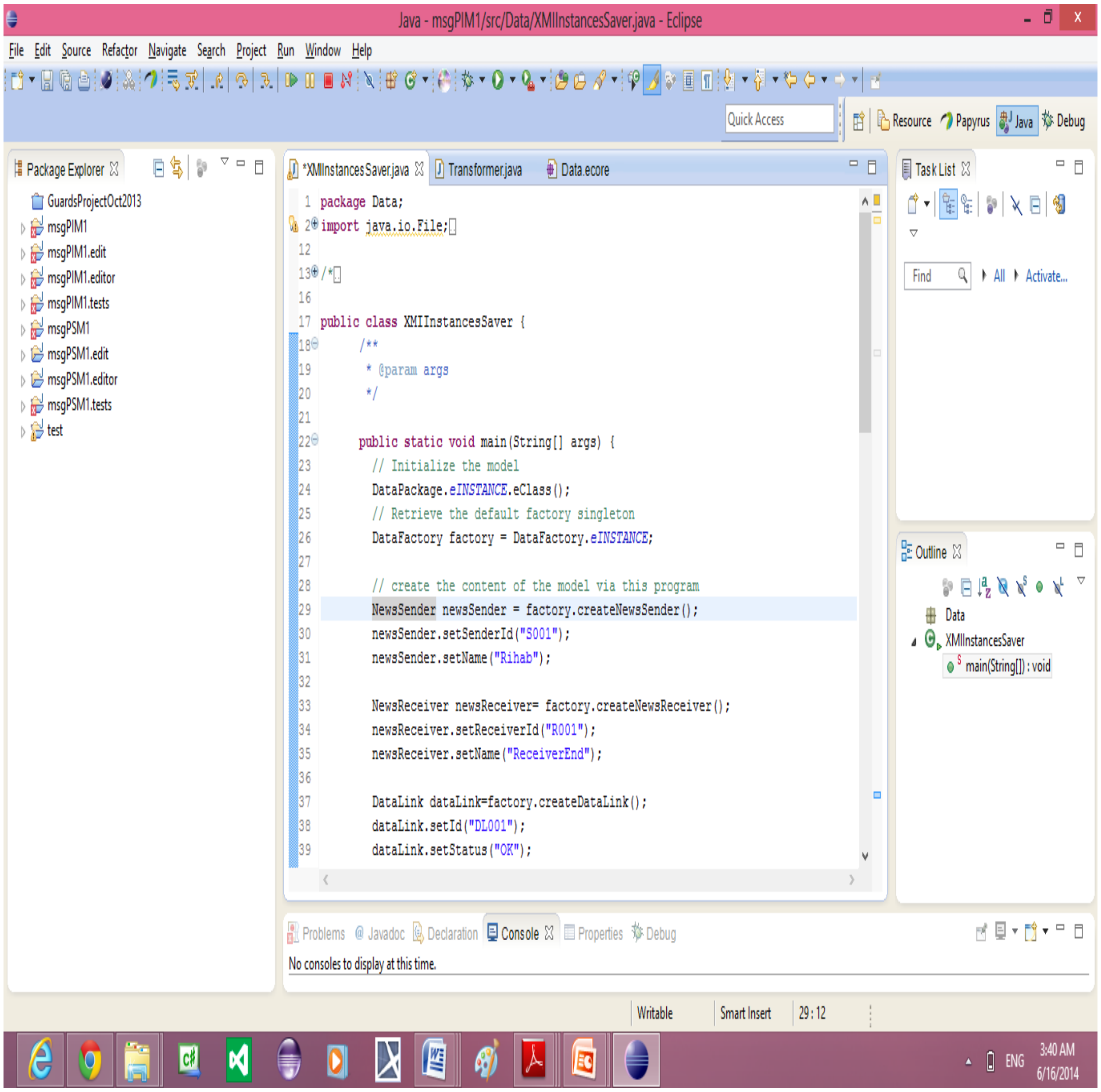
GuardsProjectOct2013
msgPIM1
msgPIM1.edit
msgPIM1.editor
msgPIM1.tests
msgPSM1
msgPSM1.edit
msgPSM1.editor
msgPSM1.tests
test

platform:/resource/msgPSM1/model/Data.ecore
- Data
  - GenModel
  - Producer
  - Session
  - Message
  - Consumer
  - Destination
  - Consumer_ConsumerPSMStateDgrm
  - Connection
  - ConnectionFactory
  - Client
  - Client_ClientStateDgrm
  - Queue -> Destination
  - Topic -> Destination
  - Producer_ProducerPSMStates
  - QueueSender -> Producer
  - JMSRuntimeException -> JMSException
  - JMSException
  - MessageFormatException -> JMSException
  - MessageNotWriteableException -> JMSException
  - TopicPublisher -> Producer
  - QueueReceiver -> Consumer
  - TopicSubscriber -> Consumer
  - InvalidDestinationException -> JMSException
  - MessageNotReadableException -> JMSException
  - UMLStandardProfile

An outline is not available.

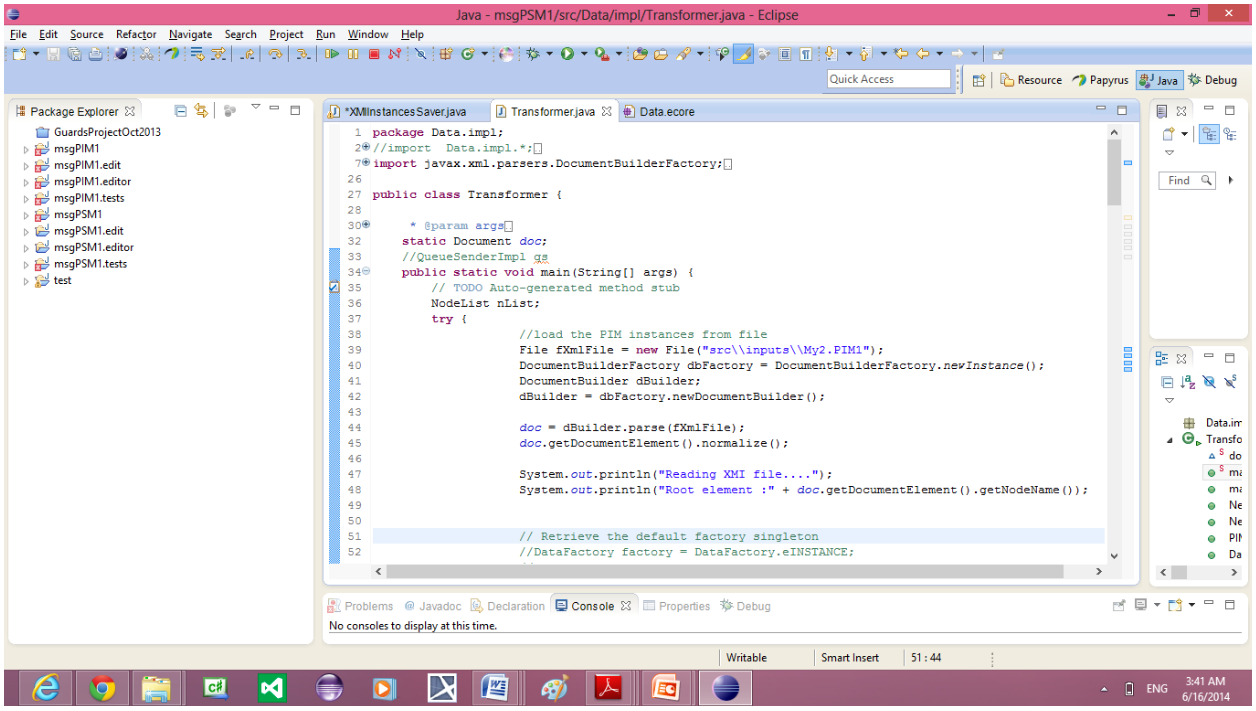
No consoles to display at this time.

Selected Object: platform:/resource/msgPSM1/model/Data.ecore


Java - msgPIM1/src/Data/XMIInstancesSaver.java - Eclipse

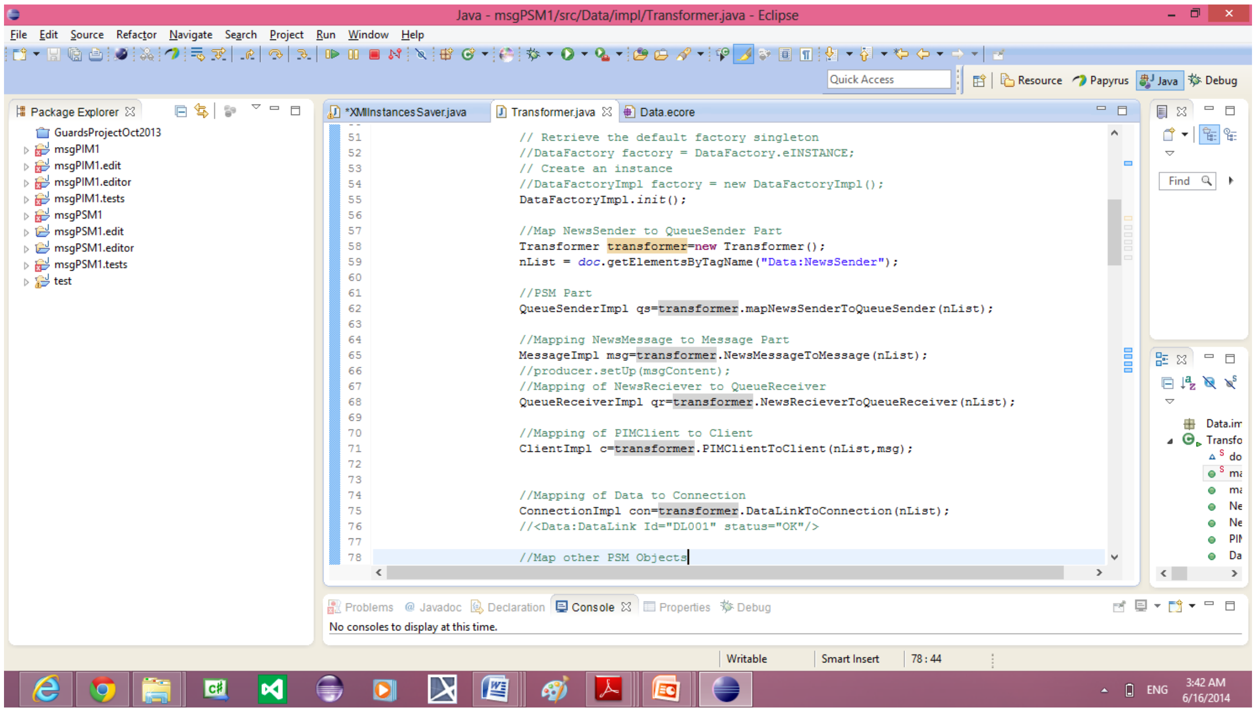
```java
package Data;
import java.io.File;

/*

public class XMIInstancesSaver {
	/**
	 * @param args
	 */

	public static void main(String[] args) {
	  // Initialize the model
	  DataPackage.eINSTANCE.eClass();
	  // Retrieve the default factory singleton
	  DataFactory factory = DataFactory.eINSTANCE;

	  // create the content of the model via this program
	  NewsSender newsSender = factory.createNewsSender();
	  newsSender.setSenderId("S001");
	  newsSender.setName("Rihab");

	  NewsReceiver newsReceiver= factory.createNewsReceiver();
	  newsReceiver.setReceiverId("R001");
	  newsReceiver.setName("ReceiverEnd");

	  DataLink dataLink=factory.createDataLink();
	  dataLink.setId("DL001");
	  dataLink.setStatus("OK");
```

No consoles to display at this time.



```
package Data.impl;
//import  Data.impl.*;
import javax.xml.parsers.DocumentBuilderFactory;

public class Transformer {

     * @param args
    static Document doc;
    //QueueSenderImpl qs
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        NodeList nList;
        try {
                //load the PIM instances from file
                File fXmlFile = new File("src\\inputs\\My2.PIM1");
                DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
                DocumentBuilder dBuilder;
                dBuilder = dbFactory.newDocumentBuilder();

                doc = dBuilder.parse(fXmlFile);
                doc.getDocumentElement().normalize();

                System.out.println("Reading XMI file....");
                System.out.println("Root element :" + doc.getDocumentElement().getNodeName());


                // Retrieve the default factory singleton
                //DataFactory factory = DataFactory.eINSTANCE;
```

No consoles to display at this time.



```
                // Retrieve the default factory singleton
                //DataFactory factory = DataFactory.eINSTANCE;
                // Create an instance
                //DataFactoryImpl factory = new DataFactoryImpl();
                DataFactoryImpl.init();

                //Map NewsSender to QueueSender Part
                Transformer transformer=new Transformer();
                nList = doc.getElementsByTagName("Data:NewsSender");

                //PSM Part
                QueueSenderImpl qs=transformer.mapNewsSenderToQueueSender(nList);

                //Mapping NewsMessage to Message Part
                MessageImpl msg=transformer.NewsMessageToMessage(nList);
                //producer.setUp(msgContent);
                //Mapping of NewsReciever to QueueReceiver
                QueueReceiverImpl qr=transformer.NewsRecieverToQueueReceiver(nList);

                //Mapping of PIMClient to Client
                ClientImpl c=transformer.PIMClientToClient(nList,msg);


                //Mapping of Data to Connection
                ConnectionImpl con=transformer.DataLinkToConnection(nList);
                //<Data:DataLink Id="DL001" status="OK"/>

                //Map other PSM Objects
```

No consoles to display at this time.

Source Code

```
import  Data.impl.*;

import Data.DataFactory;

import Data.Producer;



import javax.jms.JMSException;

import javax.xml.parsers.DocumentBuilderFactory;

import javax.xml.parsers.DocumentBuilder;

import javax.xml.parsers.ParserConfigurationException;



import org.w3c.dom.Document;

import org.w3c.dom.NodeList;

import org.w3c.dom.Node;

import org.w3c.dom.Element;

import org.xml.sax.SAXException;



import java.io.File;

import java.io.IOException;
```

```
public class JMSMain {


           /**
            * @param args
            */
           public static void main(String[] args) {
               // TODO Auto-generated method stub
               try {
                      File fXmlFile = new File("src\\Data\\impl\\MsgXmiPIM.xml");
                      DocumentBuilderFactory                  dbFactory                  =
DocumentBuilderFactory.newInstance();
                      DocumentBuilder dBuilder;
                      dBuilder = dbFactory.newDocumentBuilder();


                      Document doc = dBuilder.parse(fXmlFile);
                      doc.getDocumentElement().normalize();


                      System.out.println("Reading XMI file....");
                      System.out.println("Root           element           :"           +
doc.getDocumentElement().getNodeName());
```

```
                        NodeList nList;Node node;

                        //Sender Part



                        nList = doc.getElementsByTagName("Data:sender");

                        node =nList.item(0);

                        System.out.println("Sender                          :             "+
((Element)node).getAttribute("id"));

                        //JMS Part

                        // Retrieve the default factory singleton

                        //DataFactory factory = DataFactory.eINSTANCE;

                        // Create an instance

                        /*DataFactoryImpl factory = new DataFactoryImpl();

                        factory.init();*/



                        //producer.setProductId(1);

                        //System.out.println(producer.getProductId());

                        //Message Part

                        nList = doc.getElementsByTagName("Data:Email");

                        node =nList.item(0);

                        //System.out.println("Email          Message          :          "+
((Element)node).getAttribute("content") );
```

```
                        //this is not used, a msg is builtin for now

                        String msgContent= ((Element)node).getAttribute("content");

                        ProducerImpl2 producer= new ProducerImpl2("Producer2.xml");

                        producer.startStateMachine();

                        nList = doc.getElementsByTagName("Data:reciever");

                        node =nList.item(0);

                //      System.out.println("Receiver                        :            "+
((Element)node).getAttribute("id") );

/*

                        ConsumerImpl consumer=new ConsumerImpl();

                        consumer.createStateMachine();



                        if (consumer.stm.getCurrentStateId()=="setUp")

                                consumer.setData();

                        //

                        nList = doc.getElementsByTagName("Data:Inbox");

                        node =nList.item(0);

                        System.out.println("Inbox : "+((Element)node).getAttribute("id") );

*/



                } catch (ParserConfigurationException e1) {
```

```
        // TODO Auto-generated catch block

        e1.printStackTrace();

} catch (SAXException e) {

        // TODO Auto-generated catch block

        e.printStackTrace();

} catch (IOException e) {

        // TODO Auto-generated catch block

        e.printStackTrace();

} catch (Exception e) {

        // TODO Auto-generated catch block

        e.printStackTrace();

//      System.out.println(e);

}






/*ProducerImpl prod= new ProducerImpl();

try {

        prod.setUp();
```

```
               } catch (JMSException e) {

                       // TODO Auto-generated catch block

                       e.printStackTrace();

               }*/

           }




}

package Data.impl;

import Data.DataFactory;

import Data.impl.*;



/*import Data.Data;

import Data.DataPackage;

import Data.Producer;

import Data.Connection;*/

import javax.jms.*;



import java.net.URL;

import java.util.*;
import org.apache.commons.scxml.*;

//import org.apache.commons.scxml.ErrorReporter;

//import org.apache.commons.scxml.Evaluator;

//import org.apache.commons.scxml.EventDispatcher;

//import org.apache.commons.scxml.SCXMLExecutor;

import org.apache.commons.scxml.env.*;

import org.apache.commons.scxml.env.jexl.JexlEvaluator;

import org.apache.commons.scxml.io.SCXMLParser;

import org.apache.commons.scxml.model.*;

import org.xml.sax.ErrorHandler;



public class ProducerImpl2 {



           /**

            * @param args

            */

            URL url=null;

            ConnectionImpl connection;

            MessageProducer jmsProducer;
```

```
 DataImpl data;

public ProducerImpl2(String doc) {
url=getClass().getResource(doc);
}
public  void startStateMachine() throws Exception
{
    //ClientTest ct=new ClientTest("Producer2.xml");
    // TODO Auto-generated method stub
    // (1) Create a list of custom actions, add as many as are needed
    //  List<CustomAction> customActions = new ArrayList<CustomAction>();
           //      CustomAction   ca   =   new   CustomAction("http://my.custom-
actions.domain/PRODUCER",
    //                          "producer", ProducerActions.class);
     //System.out.println(ca.getClass().getName());
    //  customActions.add(ca);
      //try {
     // URL url= new URL("hello2.xml");
      //URL url = docIt("hello2.xml");
      // (2) Read the SCXML document containing the custom action(s)
      SCXML scxml = null;
```

```
ErrorHandler errHandler = null;

scxml = SCXMLParser.parse(url, errHandler);//, customActions);

    // Also see other methods in SCXMLReader API



    SCXMLExecutor exec = null;

    //try {

        exec = new SCXMLExecutor ();

        JexlEvaluator ev= new JexlEvaluator() ;

        exec.setEvaluator(ev);

        exec.setEventdispatcher(new SimpleDispatcher());

        SimpleErrorReporter er= new SimpleErrorReporter();

        exec.setErrorReporter(er);

        exec.setStateMachine(scxml);

        exec.addListener(scxml,new SimpleSCXMLListener() );

        Context rootCtx=ev.newContext(null);

        exec.setRootContext(rootCtx);

        exec.go();

        while (exec.getCurrentStatus().getStates().iterator().hasNext())

        {

             State                   CurrentState         =         (State)
exec.getCurrentStatus().getStates().iterator().next();
```

```
                                        if(CurrentState.isFinal())

                                               break;

                                        else

                                        {

                                                      String stateId=CurrentState.getId();

                                               System.out.println("In state: "+stateId);

                                                   switch (stateId)

                                                   {

                                                              case  "region1":  display1("region1");
break;

                                                              case  "begin":      display1("begin");
break;

                                                              case   "ready":        getReady(exec);
break;

                                                              case "sending": sending(exec); break;

                                                              case  "close":      display1("close");
connection.jmsConnection.close(); break;

                                                              case               "raiseExceptions":
display1("raiseExceptions"); break;

                                                   }

                                        }//else

                                   }//while
```

```
}



public void setUp(String msg)throws JMSException

{

            // Name of the queue we will be sending messages to

    String subject = "TESTQUEUE";

    // Retrieve the default factory singleton

            DataFactoryImpl factory = new DataFactoryImpl();

            factory.init();

            // Getting JMS connection from the server and starting it

    connection = new ConnectionImpl();//factory.createConnection();

        connection.start();

            System.out.println("connection started");

        // JMS messages are sent and received using a Session. We will

        // create here a non-transactional session object. If you want

        // to use transactions you should set the first parameter to 'true'

        SessionImpl session = (SessionImpl)connection.childSession;



        // Destination represents here our queue 'TESTQUEUE' on the

        // JMS server. You don't have to do anything special on the
// server to create it, it will be created automatically.

BufferImpl queue = new BufferImpl(session,subject);



// MessageProducer is used for sending messages (as opposed

// to MessageConsumer which is used for receiving them)

jmsProducer = session.jmsSession.createProducer(queue.jmsQueue);



// We will send a small text message saying 'Hello' in Sudanese

data = new DataImpl (session,msg);



// we can call

//String newMsg="am so tired and having headache ";

data.setMsg(session, msg);

}



      public  void display1(String name)

      {

             System.out.println("In state: "+name);

      }



        public  void getReady( SCXMLExecutor ex)throws ModelException,
JMSException

                         {

                                 setUp("Melbourne  named  world's  most  liveable  city  for
fourth straight year");

                                System.out.println("Message is ready...");

                                TriggerEvent event = new TriggerEvent("callSendEvent",



TriggerEvent.SIGNAL_EVENT);

                                System.out.println("event..."+event.getName());

                             ex.triggerEvent(event);

                         }



                             public   void  sending(SCXMLExecutor  ex)throws  ModelException,
JMSException

                         {

                                String nextEvent;

                                System.out.println("Producer is sending...");

                                                      //String            passing=(String)
ex.getRootContext().getVars().get((Object) "passing");

                                //System.out.println("Executing...has it "+passing);

                             //call the object send method
```

```
                                // Here we are sending the message!
                                 jmsProducer.send(data.message);
                                                System.out.println("Sent   message   '"   +
data.message.getText() + "'");
                                // if sent nextEvent="MsgSentEvent";
                                 nextEvent="closeEvent";
                                //failed  nextEvent="failToSendEvent";
                                //close nextEvent="closeEvent";
                                   TriggerEvent   event   =   new   TriggerEvent(nextEvent,
TriggerEvent.SIGNAL_EVENT);
                                System.out.println("event..."+event.getName());
                             ex.triggerEvent(event);
                        }


            public void callState(String name){
                 // this.invoke(name);
                 }
                 /**
                  * Get current state ID as string
                 */
            /* public String getCurrentStateId() {
                  Set states = getEngine().getCurrentStatus().getStates();

                  State state = (State) states.iterator().next();

                  return state.getId();

                 } */

}
```