Investigating Model Driven Architecture

Master Thesis

Atif Mashkoor

ens03amr@cs.umu.se

May 24, 2004

Department of Computing Science
Umeå University
Umeå
Sweden

Acknowledgements

I would like to thank my supervisor Magnus Eriksson for his help, valuable suggestions and timely advices, which played important role in the fulfillment of this thesis.

Table of Contents

Table of Figures

Table of Tables

Abstract

Over the last few years, computing infrastructures have spanned into every dimension. The businesses are facing new challenges. To meet these challenges, new platforms and applications, each claiming to be "the next big thing", are continually emerging on the horizon of software industry. The proliferation of new platforms has created another issue of interoperability. In this situation, Object Management Group (OMG) takes the initiative and tries to address the issue by introducing the Model Driven Architecture (MDA). MDA resolves this problem by providing the architecture which is able to communicate with the past and future applications. But another problem arises when the choice of best suitable modeling language, for MDA, has to be made. The main functionality, which is desirable to support the development process of MDA, is the ability of the language to transform the models into the code. Beside this, there are also other characteristics that should be supported by the modeling languages such as model verification, level of abstraction, tool support, etc.

This thesis, firstly, describes the process of MDA, followed by some of the outlined criteria, which will help identify the best supporting modeling language for MDA. The thesis also evaluates some best modeling languages, present today, under the derived guidelines. The vendor claimed capabilities of some tools supporting these languages will also be listed as an appendix to this thesis.

1. Introduction

Modeling is one of the most important factors in the process of computer systems development. It is the process of representing real-world concepts in the computer domains as a blueprint for the purpose of software development. Recent trends in software development have also revealed the value of developing systems at higher levels of abstraction. Evidence of this is, for example, acceptance of the Unified Modeling Language (UML) as industry standard, which is the visual modeling language and is the evolution of Booch [Booch, 1993], Object Modeling Technique (OMT) [Rumbaugh, 1991] and Object Oriented Software Engineering (OOSE) [Jacobson, 1992] object oriented analysis and design methodologies. The significance of modeling and abstraction becomes further strengthened in the process of Model Driven Architecture (MDA) [MDA, 2004], where they are considered the heart and soul of software development.

The new middleware platforms such as CORBA [CORBA, 2004], J2EE [J2EE, 2004] and .Net [Net, 2004] are continually proliferating into the market. The emergence of new platforms is a positive sign, but there is always a risk that existing platforms would no longer be required in the future. It means the applications developed for them will also have to be redesigned for new platforms. In this case, there is a threat that the investments will be drowned. In this scenario, the MDA appears on the horizon of software industry and claims to solve all of these problems. As claimed by OMG, the application architecture based on the MDA will be ready to deal with what have been built in the past, what is being built today and what will be build in the future. MDA does this using the concept of Platform Independent Models (PIMs), which are pure from any platform specific information and hence deployable on variety of platforms including the future platforms. These models should be written in such modeling languages which are capable to support the concept of MDA for its full potential.

The choice of modeling language for MDA is an important issue. [Evans, 2003] also brings this question into the consideration. OMG explicitly declares UML as key standard for MDA. So UML is an obvious candidate for this choice, especially, in the case, when it has been redesigned for MDA particularly in its next version, UML 2.0 [UML, 2004]. The other choices may be the UML 1.x and Executable and Translatable UML ($^{X}_{T}$UML) [Mellor, 2002]. Since OMG declares UML as one of key standards for MDA, so the consideration for the modeling languages will revolve around UML.

The key functionality, that makes any modeling language suitable to design artifacts for MDA development process, is the ability of the language to automatically transform models into the code. Though the transformation is achieved using translation tools, yet, the importance of the language can not be overlooked. This ability of transformation of artifacts into the code is achieved by deploying an Action Language (AL). UML 2.0 and $^{X}_{T}$UML explicitly use an AL to attain the desired transformations. This usage of AL gives them an edge over the UML 1.x and makes them strong contenders as a modeling language of choice for MDA.

All of these languages will be analyzed, in details, in order to check their strength and weaknesses to support MDA, as a part of the thesis. To check this ability, some criteria will be derived, which will help the cause. These modeling languages will be analyzed under the guidelines of these criteria.

The thesis is composed under the following hierarchy.

- Section 2 describes the process of MDA in detail.
- Section 3 outlines the criteria which will help analyze the MDA-supporting modeling languages.
- Section 4 explains the importance and functionality of Object Constraint Language (OCL) and Action Language (AL).
- Section 5 dissects the plain UML to check its ability to support the development process of MDA.
- Section 6 investigates the Executable and Translatable UML ($^{X}_{T}$UML) to test its potential to support the development process of MDA.
- Section 7 analyzes the UML 2.0 to verify its capability to support the development process of MDA.
- Section 8 concludes the thesis
- Appendix A provides a capability matrix of these aforementioned modeling languages supporting tools.

2. Model Driven Architecture

This chapter introduces the concept and process of Model Driven Architecture (MDA). It investigates the building blocks of MDA and also the way how they support the development process.

2.1. Introduction

Model Driven Architecture (MDA), defined and supported by the Object Management Group (OMG) [OMG, 2004], defines an approach to IT system specifications that separates the system functionalities from the implementation details on a particular technological platform. The MDA is a framework for model driven software development defined by the OMG which has elevated the software development to the next step. Using MDA, it is possible to have an architecture that will be language, vendor and middleware neutral.

This approach places the emphasis on models, provides a higher level of abstraction during development and enables significant decoupling between Platform Independent Models (PIMs) and Platform Specific Models (PSMs). Platform independent applications built using MDA, may range from transportation to health care industry and can be deployed on a range of open and proprietary platforms, such as CORBA, J2EE, .NET and etc (Figure 1).

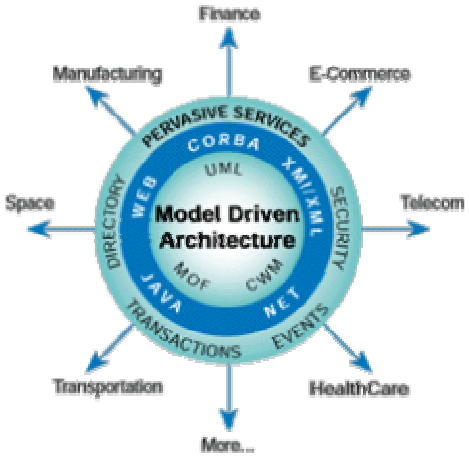

Figure 1 The Model Driven Architecture [MDA, 2004]

Projects using MDA focuses first on the functionality and behavior of a distributed application or system, leaving behind the technologies in which it will be implemented. MDA divorces implementation details from business functions. Thus, it is not necessary to repeat the process of modeling each time a new technology emerges. Mapping from a PIM to a PSM, using transformation tools, can be done, for any specific platform, at any time, once a platform independent model is built.

One benefit claimed by OMG for MDA is that it is the best way to preserve the investment, intellect and finance, spent in the development of an application. For example, if the application needs to be deployed on several platforms or migrated from one platform to another as technology changes, once the PIM is ready, the required code can be regenerated from the PIM. This, as argued, is faster and cheaper than the migration of the deployed code.

In other words this concept corresponds to cross platform interoperability, portability, platform independence and productivity. Other benefits, claimed by OMG on the behalf of MDA, are automatic generation of code using supported transformation tools and solution to documentation and maintenance problems due to the fact of auto generated code from models.

According to [Kleppe, 2003], the revolution brought into the industry by the emergence of MDA, is the paradigm shift from coding to modeling. Time and efforts spent on coding phase will now be consumed by modeling phase, making the overall business functionality further sound and authenticated. Once implemented with full potential and accepted by industry, this may obsolete the coding phase at all from the life cycles of the software development process.

2.2. Key Standards

According to [MDA, 2004], the key standards that make up the MDA include the Unified Modeling Language (UML) [UML, 2004], Meta Object Facility (MOF) [MOF, 2003], XML Metadata Interchange (XMI) [XMI, 2003] and Common Warehouse Metamodel (CWM) [CWM, 2003]. [Poole, 2001] believes that these core standards of the MDA form the basis for building coherent schemes for authoring, publishing and managing models within a model driven architecture.

Unified Modeling Language
Unified Modeling Language (UML) is a specification defining a graphical language for visualizing, specifying, constructing and documenting the artifacts for software systems. UML can be used for designing of models in PIM.

Meta Object Facility
Meta Object Facility (MOF) is an extensible model driven integration framework for defining, manipulating and integrating metadata and data in a platform independent manner. MOF based standards are in use for integrating tools, applications and data. MOF provides a standard repository for models and defines a structure that helps multiple groups to work with the model.

XML Metadata Interchange
XML Metadata Interchange (XMI) is a model driven Extensible Markup Language (XML) integration framework for defining, interchanging, manipulating and integrating XML data and objects. XMI based standards are in use for integrating tools, repositories, applications and data warehouses. It provides rules by which a schema can be generated for any valid XMI transmissible MOF based metamodel. XMI expresses UML models in XML and allows them to be exchanged around the enterprise.

Common Warehouse Metamodel
Common Warehouse Metamodel (CWM) is set of standard interfaces that can be used to enable easy interchange of warehouse and business intelligence metadata between warehouse tools, warehouse platforms and warehouse metadata repositories in distributed heterogeneous environments. CWM is the standard for data repository integration. It standardizes how to represent schema, schema transformation models and data mining models.

2.3. The MDA Process

According to [Klasse, 2004] and as shown in figure 2, the process of MDA revolves around three major steps

1. First a model with higher level of abstraction is built. This model is independent of any implementation technology and called Platform Independent Model (PIM).
2. Next, this PIM is transformed into one or more Platform Specific Models (PSMs). These PSMs are the constructs of PIMs into more detailed, platform dependent and technology oriented models for a particular platform such as .Net or Enterprise Java Beans (EJB).
3. Finally the PSM is transformed into the code. The PSM already carry all the necessary details which are required for the coding, so this step is rather trivial.

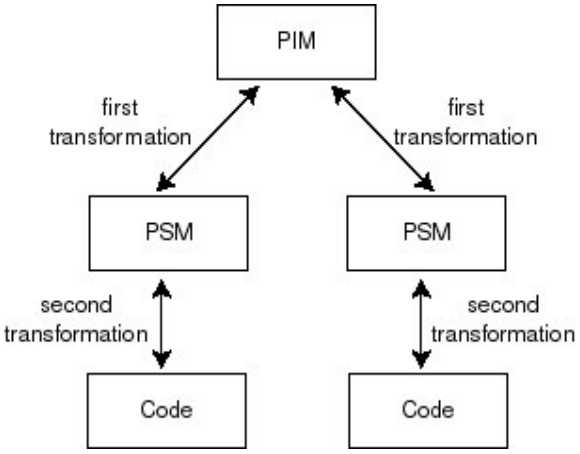

Figure 2 The MDA process [Klasse, 2004]

Step 1: Platform Independent Model
The MDA development project starts with the creation of a Platform Independent Model (PIM). According to [Kleppe, 2003], Platform Independent Model is a model with a high level of abstraction that is independent of any implementation technology. A modeling language capable of generating all the required artifacts such as the Unified Modeling Language (UML) is required at this level. According to [MDATech, 2001], PIMs may be defined using UML (preferable) or other notations where appropriate. Behavior and constraints are defined using a formal notation (UML models) or an informal notation (natural language) as appropriate.

An MDA model usually has multiple levels of PIMs. All of these PIMs include platform independent aspects of technological behavior except the base PIM. The base PIM expresses only business functionality and behavior. Business and modeling experts work together to expresses business rules and functionality as much as possible independent of any implementation technology. Although PIMs at the next level include some aspects of technology yet again platform specific details are absent e.g. persistence, transactionality, security level and even some configuration information. By adding these concepts to the second level PIMs, more precise mapping to PSM can be achieved in next step. The PIM, which is produced in the first step of an MDA development, specifies functionality and behavior of the software. The diagrams produced at this stage such as UML class and object

diagrams incorporate the structure; sequence and activity diagrams embody the behavior; class and object names, along with semantic notations, incorporate business factors; while other aspects of the model incorporate platform independent aspects of component structure and behavior [Siegel, 2001].

According to [MDATech, 2001], the Platform Independent Models provides two basic advantages. Firstly, the person responsible for defining the functionality do not have to take any platform details into the consideration while modeling, which gives him the freedom and liberty to concentrate and focus only on the business rules. Secondly, since the functionality is pure from any implementation details so it is easier to produce implementations on different platforms.

Step 2: Platform Specific Model

When the PIM is complete, it is stored in Meta Object Facility (MOF) and it serves as the input to the mapping step which will produce a Platform Specific Model (PSM). A Platform Specific Model is tailored to specify the system in terms of the implementation constructs that are available in one specific implementation technology [Kleppe, 2003].

According to [MDATech, 2001], PSMs can be described in one of two ways. One using UML diagrams (class, sequence, activity etc) and other using interface definitions in a concrete implementation technology (IDL, XML, Java etc), but in both cases behavior and constraints are specified using a formal notation (UML diagrams) or an informal notation (natural language) as appropriate.

To produce the PSMs, one or more target platforms would have to be selected for each module of the software. In contrast with the first step, here instead of technology experts, automated tools will be required, which will map the platform independent models onto specific platforms. But this is true only in the case if the tool is already aware of working environment such as on which platform the various pervasive services and domain facilities run, if it is not already known by the tool then this information would have to be supplied to allow the MDA to generate cross platform invocations.

According to [Alhir, 2003] and as shown in figure 3, there exist two ways to map PIM onto PSM. Firstly, a model type mapping that is based on the types of the model elements. It specifies how the different types of elements in the PIM are transformed to different types of elements in the PSM. Secondly, a model instance mapping, that specifies that how to map the specific model elements using marks. PIM elements are marked to indicate how to transform them. A PIM element may also be marked several times with marks from different mappings and is therefore transformed according to each of the mappings. For example if a PIM element is marked as an entity, it may show that element is to be transformed using some specific rules, perhaps it has to be converted into an EJB entity or a Java class. The same concepts are shown in figure 3. It shows that marks are used for marking a PIM to produce a marked PIM and that mappings are used for mapping a PIM or marked PIM to produce a PSM.

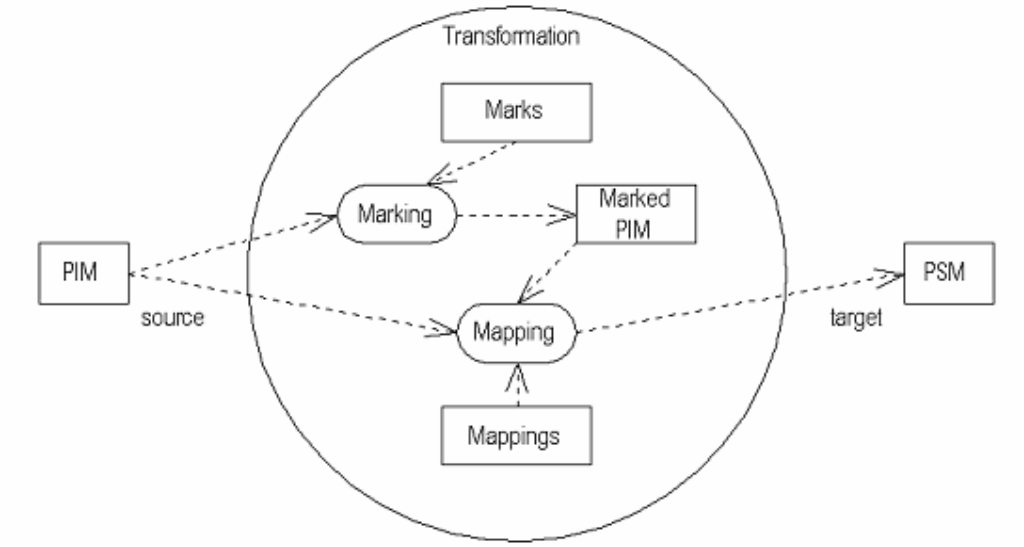

Figure 3 The MDA PIM to PSM mapping [Alhir, 2003]

When one PIM is transformed into many PSMs, there may be some relationships among these transformations. In MDA these relationships are called Bridges and it is depicted in figure 4. It is very easy to bridge the gap between two different targeted PSMs especially when all of the required information is auto generated. Cross platform interoperability can be achieved by tools that do not only produce the different PSMs but also the bridges between them and possibly to other platforms as well. These bridges can be produced by the tools. No matter how many PSMs are, there would be a bridge between each of them. The same bridges can be produced at the code level.

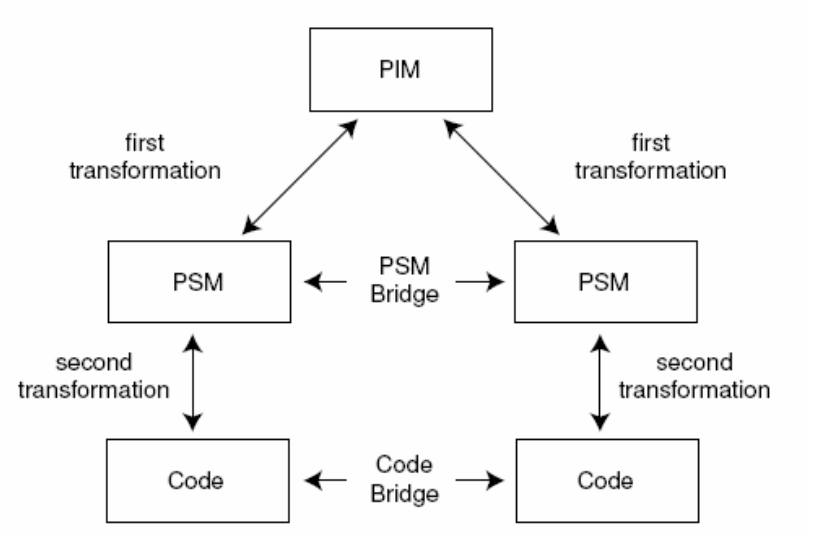

Figure 4 The detailed MDA process [Kleppe, 2003]

The run time characteristics and configuration information, which was included in PIMs in a general way at the later stages, will now be converted into the specific form. This specific form will be required by the target middleware platform. Automated tools perform as much of this conversion as possible, leaving behind only the ambiguous portions for the programming staff to resolve by hand. Early versions of the MDA supporting tools may require considerable hand adjustment but this amount is expected to be decreased as mappings mature over time and new vendors and tools arouse on the horizon. To enable hands off generation of running code from the application model, [Siegel, 2001] lists four ways to move from a PIM to a PSM. In increasing level of sophistication and automation, they are:

1. A person performs the transformation completely by hand, working each application ad hoc without reference to others.
2. A person performs the transformation using established patterns to convert from the PIM to a particular PSM.
3. The established patterns define an algorithm which is implemented in an MDA tool that produces a skeleton PSM, which is then completed by hand.
4. The tool, applying the algorithm, is able to produce the entire PSM.

Evolution here has already passed through the primitive levels 1 and 2. Even early MDA development tools started with a level somewhere around 3 and are now approaching level 4. But safely it can be said that right now the software industry is standing on the level 3 and is about to reach the verge of level 4.

Step 3: Code Generation

This step takes PSMs as input and produces the implementation of these PSMs for the particular platform using the transformation tool. It is important that the selected tool is capable of generating implementation of these PSMs for the specified platform. To make the code generation process easy to understand, [Siegel, 2001] presents an example of web service. Suppose that the PSMs are ready and the development is in the final step. Now the application server as well as programming language has to be selected for the implementation. The application server should support the proposed programming language (Java, C++ or C#) or the programming language should be deployable on the proposed application server (Apache, IIS, BEA). After this choice, an MDA development tool will generate source code for the application running on selected application server, in the chosen programming language. It will, in addition, generate files that tell the application server how to configure and deploy the application to run the way it is required, based on information that was included in the UML model. The tool will also generate Web Service Definition Language (WSDL) files and Universal Description, Discovery, and Integration registry (UDDI) entry files. To support the Extensible Markup Language (XML) messages communicated by web services, the MDA tool will also prepare Document Type Definitions (DTD), Simple Object Access Protocol (SOAP) message formats and a set of XML Style Sheet Transformations (XSLTs) that translate between them. Each MDA mapping will produce the artifacts and file types that its target platform requires. The MDA specification will require that tools trace and version all process artifacts. Immediately following code generation, the programming staff will apply any required hand coding to the output. For web services example, only the programming language code needs to be compiled, but if component-based target platform had been chosen then Interface Definition Language (IDL) files might need to be compiled into language code and then into compiled code. In any case, all of the artifacts are compiled by the system automatically to object files (for C++ and similar languages), intermediate byte code (for Java), dynamically linked libraries or other artifacts.

2.4. The MDA Development Life Cycle

MDA does not look very different from the traditional software development[1]. There are almost the same steps in traditional software development as there are in MDA. The only difference is the outcome of some phases. MDA introduces the new concept of automated transformation of PIM into more than one PSMs and ultimately into the code.

As depicted in figure 5, traditional software development mainly consists of requirements gathering, analysis, design, coding, testing and deployment phases and so does the MDA as shown in figure 6. However the outcomes of the analysis and design phases in the traditional software development are different from MDA. The outcomes of analysis and design phases in the MDA are the Platform Independent Model (PIM) and Platform Specific Models (PSMs) respectively whereas in the traditional software development it is always some text and diagrams [Kleppe, 2003].

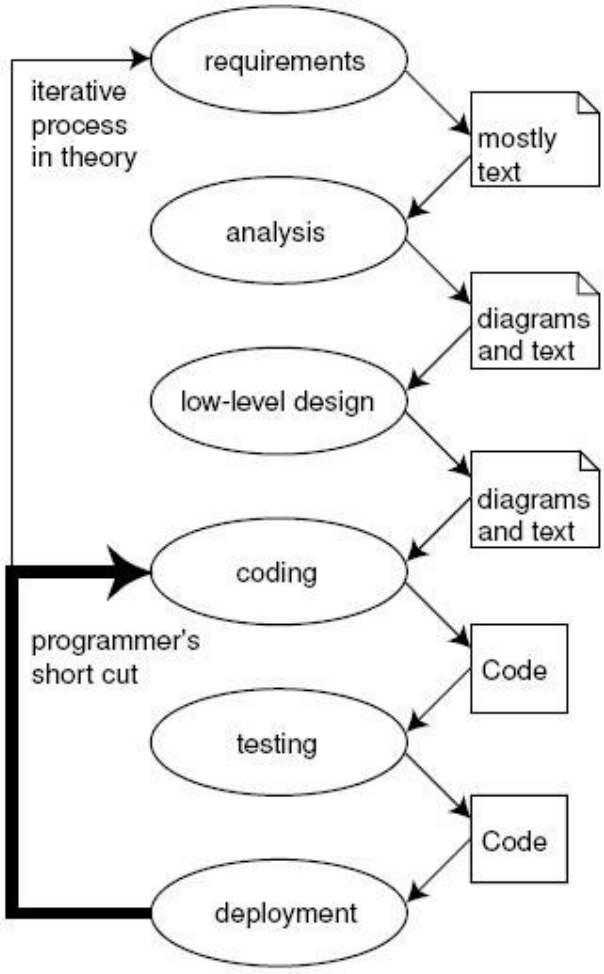

Figure 5 Traditional software development life cycle [Kleppe, 2003]

[1] Here it is assumed that traditional software development is the software development which is inspired by linear sequential water fall software development process model.

Like traditional software development, the process of MDA also starts with requirement gathering, followed by analysis of these requirements. Here, unlike traditional software development, such artifacts are produced that are not only platform independent but can also be understood by computers directly. These artifacts are known as Platform Independent Models (PIMs). Now these PIMs serve as an input to the design phase where these artifacts will be translated into the platform specific models. Again these artifacts are in the shape that can directly be understood by computers, however this time they are platform specific and hence called Platform Specific Models (PSMs). These PSMs are, in fact, more specified forms of the generalized artifacts which were produced as the PIMs. They are comprised of technological details for the specific platform. This conversion of PIM into PSM is achieved by transformation tools. As many transformations as required, each for a different platform, can be produced depending on the tool support. Using the supported tools automated implementation of PSMs into source code for target platform can also be produced. This is the next step in the process of MDA. This implementation of PSMs can be produced for more than one targeted environments. MDA does not put any restriction on the number of targeted environments, but tools may. So the choice of tools will be important. Once the PIM has been transformed into the code, the code is then left for testing and deployment.

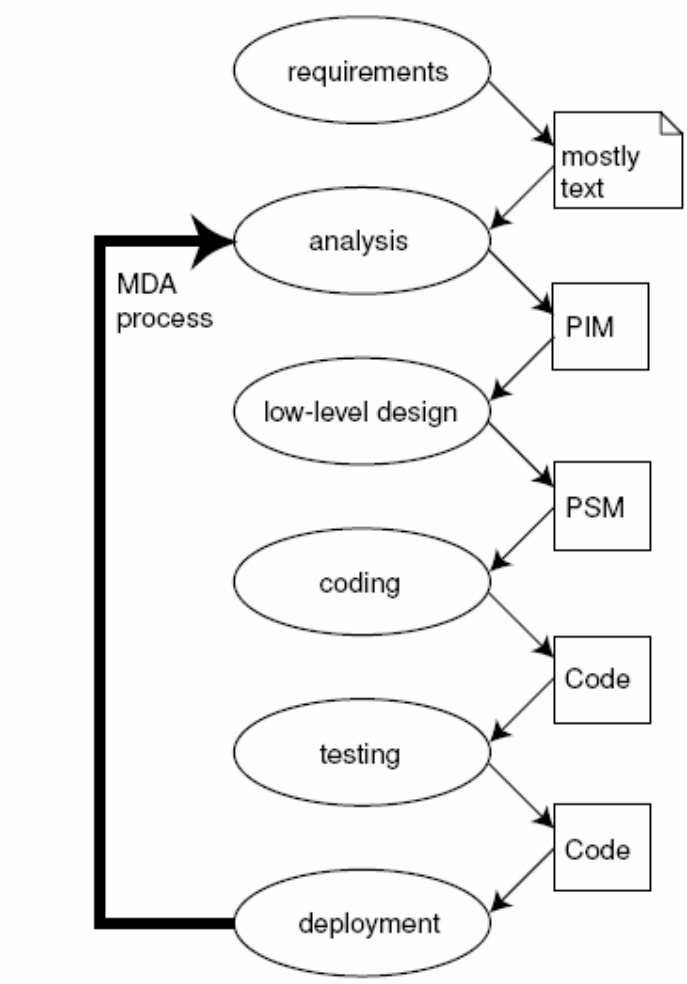

Figure 6 MDA software development life cycle [Kleppe, 2003]

In the system life cycle of MDA, a PIM can be tested and refined n times until the desired system description level is obtained. Once the PIM has been finalized then the platform specific infrastructure is taken into account and PIM is mapped onto PSM. Same thing can also be done with the PSM i.e. they can be tested and refined as many time as required. Once the PIM into PSM transformation process becomes fully automated, this process of refinement will also become obsolete.

2.5. A Typical MDA Metamodel

According to the UML Glossary [Kaushik, 1997], a metamodel is a model that defines the language for expressing a model, but in other words, it can also be said that a metamodel is a model of a model.

Figure 7 depicts a typical MDA metamodel. The main metamodel which stay at the core of the whole process is the key factor. It uses set of standards for fully description. This set mainly comprises of UML, MOF, XMI and CWM. Some other languages may also be the contenders, which may contribute in the expression of this metamodel such as $^{X}_{T}$UML or even Object Constraint Language (OCL) as stand alone or may be with plain UML. This metamodel directly interacts with PIMs, PSMs and their mapping techniques. PIM is independent of infrastructure but PSM relies on that. Both PIM and PSM are described with main metamodel and both PIM and PSM mapping techniques are also based on this metamodel.

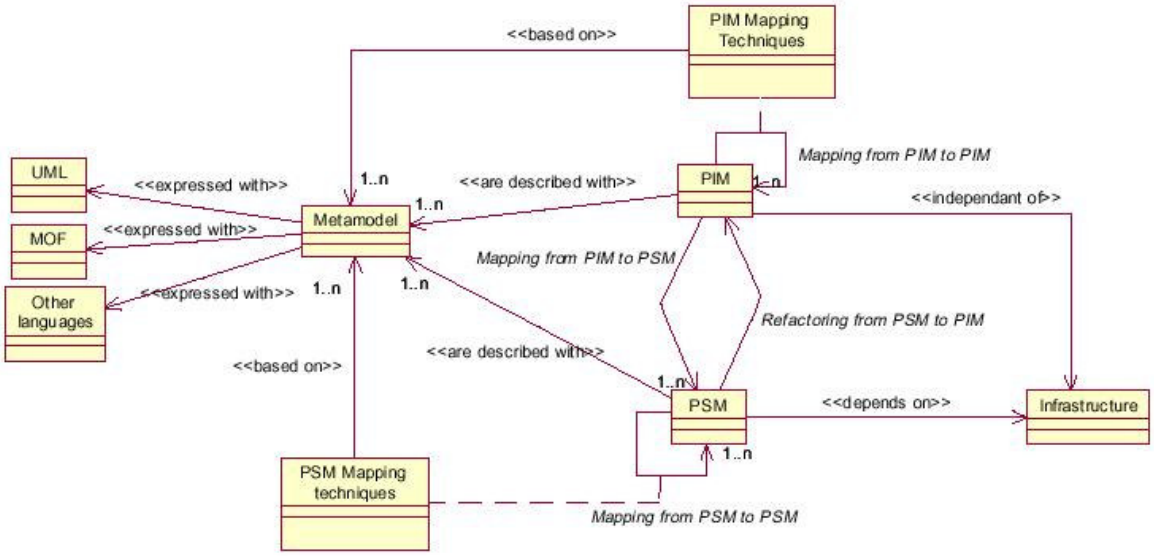

Figure 7 The MDA Metamodel [Mubin, 2004]

2.6. MDA Transformations

The concept of transformations plays a very significant role in the process of MDA. Fundamentally, transformation is the automatic process of converting one artifact into the desired artifact, using a tool, provided a rule that describe how to transform it. Transformation tools takes one artifact as input and transform it into the desired artifact according to transformation definitions and transformation rules and the overall process of conversion is known as transformation. The same process is shown by figure 8.

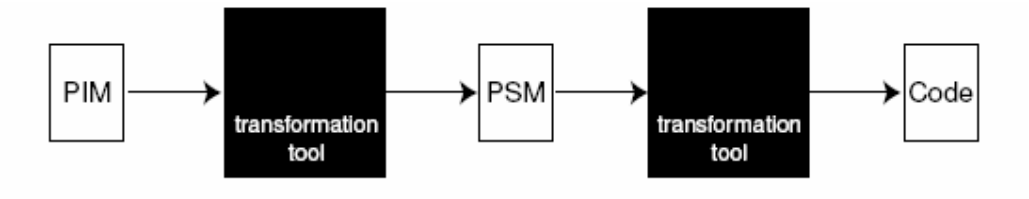

Figure 8 The MDA transformation process [Kleppe, 2003]

Transformation, transformation tool, transformation definition and transformation rule are separate entities and there is vital difference in between them which should not be mixed up. These concepts are further elaborated by [Kleppe, 2003] as:

Transformation Definition
In transformation tool, transformation definitions are required for the process of model transformation. A transformation definition, in fact, is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. The transformation tool uses transformation definitions to generate the transformations.

Transformation Rule
A transformation consists of a collection of transformation rules, which are the guidelines that how to use one model to generate other. So a transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language.

Transformation Tool
Transformation tool has the vital role in the overall process of MDA. A transformation tool performs a transformation for a specific source model according to a transformation definition. It takes one artifact as an input and transforms it into the desired artifact.

2.7. Importance of Transformation Tool

The MDA emphasizes heavily on tools, so they play the key role in the process of MDA. Once the PIM is ready, the automated process of transformation, of this PIM into PSM and subsequently into the code, starts. This is done by transformation tools. These tools take PIM as input and generate the PSM for particular platform which is then again translated into code using the same or may be different tool.

Selection of good tool is very important in the whole process. [Kleppe, 2003] describes the following characteristics that each good transformation tool should be comprised of:

- The tool should support different platforms.
- The tool should provide the flexibility to switch within one platform between different implementation strategies, application architectures and coding patterns.
- The tool should support different modeling languages and transformation definitions.
- The tool should support standard domain specific models or blueprints.
- The tool should be able to integrate with other automatic software development and maintenance tools.

2.8. Summary

The Model Driven Architecture is a framework for software development, defined by the OMG. The MDA emphasizes heavily on the usage of models in the process of software development. Summarizing the whole process of MDA, according to [Klasse, 2004], the following are the building blocks of the MDA framework:

- High level models, written in a standard, well defined language.
- One or more standard, well defined languages to write high level models.
- Transformation definitions that describe how the PIM will be transformed into the PSM and eventually into the code.
- A language in which these definitions will be written. This language must be used by the transformation tools therefore it must be a more or less formal language.
- Transformation tools that transform PIM into PSM.
- Transformation tools that transform the PSM into code.

The next chapter derives criteria for MDA supporting modeling languages. Some modeling languages, present today, will be judged under these derived guidelines in subsequent chapters.

3. The Modeling Language Dilemma

This chapter investigates the important characteristics of a modeling language which make it most desirable for the development of the models for MDA.

3.1. Introduction

Modeling is one of the most important considerations of computer systems development. It is the process of finding the most effective representation of real world concepts in the computer space for the purpose of a software project. The concept of modeling stays at the core of MDA and this fact is also obvious by the name Model Driven Architecture. The whole vision of MDA revolves around modeling. The most important consideration for modeling, in order to support the whole process of MDA, is the choice of language.

Today in the software market, there exist a few modeling languages and UML is most important and most famous of them. Unfortunately not every other modeling language may support the concept of MDA. There are some requirements which should be fulfilled before any modeling language qualifies as a modeling language for MDA.

3.2. The Criteria

In the quest of finding the best possible modeling language to design the artifacts for MDA, there should be at least some criteria or benchmarks which help find the best possible modeling language for MDA. The MDA introduces a paradigm shift in the software industry, from coding to modeling, so the execution of models directly into the code becomes the main concern of MDA and also an implicit criterion. Modeling languages have also become quite matured over the last couple of years. Some traditional criteria like flexibility, adaptability, scalability, etc. have become the prime objectives for their design. So these criteria will not be taken into account while evaluating the modeling languages for MDA, instead some other features of these languages will be evaluated, which are more crucial to support the development inspired by MDA.

To derive the criteria [Kleppe, 2003] has outlined number of guidelines. They propose few qualities that, at least, a good modeling language should possess, in order to be the suitable modeling language for MDA. These qualities are as follows:

- The language should be very expressive so that it fully specifies the system. This includes both static and dynamic aspects of the system i.e. developers should not take ordinary programming languages into the consideration for any reason.
- The language should not be application specific rather it should be general. Application specific languages like 4GLs may not be a good idea.
- The language should be at higher level of abstraction.
- The language should be suitable for n–tier application development. The number of tiers should not jeopardize the model itself, i.e. the model should have nothing to do with the tier constraint either one or n.
- The language should support distributed application development.
- The language should provide seamlessness between implementation and model.
- The language should be capable to manage large models, for instance support of aspect oriented modeling.

- The language should have adequate tool support.

3.3. The Choices

Today few choices exist which may be taken into account to figure out the best possible modeling language to support MDA. UML is at the top of them. The other contenders may be Executable and Translatable UML ($^{X}_{T}$UML) and UML 2.0, which is the latest version of UML and is still in finalizing process. Other languages such as Specification and Description Language (SDL) [SDL, 2004] or Architectural Description Languages (ADL) such as Acme [Acme, 2004] may also be the possibilities but this thesis focus only on the variants of UML. The main reason for this is that OMG, the creators of MDA, declares UML as one of the key standard for MDA. Another argument to take only the subsets of UML into consideration is that it has gained industry-wide acceptance and has become standard in the modeling. It also provides adaptability (extension mechanism), flexibility (profiling), scalability (generalization, specialization, decomposition, etc.) and etc. The following section portrays a bird eye view on selected modeling languages. The subsequent chapters will enlighten them in greater details.

UML 1.x

The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing and documenting the artifacts of software systems as well as for business modeling and other non-software systems. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas and reusable software components. UML 1.5 is the current standardized version of UML [UML, 2003].

Beside the fact that UML 1.x has been industry standard as a modeling language since 1997, the strongest point which makes the UML 1.x one of the candidate modeling languages, is its ability for modeling of the structural aspects of a system (Class diagrams). UML 1.x is one of the finest languages in this discipline but it is not as strong on the behavioral or dynamic modeling side as on structural side. [Kleppe, 2003] believe that UML 1.x has some weaknesses which prevent generating a complete PSM from PIM. The weak area is the dynamic modeling diagrams, for example, what code would be generated from an interaction diagram. Though the Action Semantics (AS) tries to bridge the gaps, yet they are not complete and concrete enough to fulfill the objective.

Executable and Translatable UML

Executable and Translatable UML ($^{X}_{T}$UML) is defined as plain UML combined with the dynamic behavior of action semantics. $^{X}_{T}$UML is, in fact, an executable profile for UML. A coherent subset has been chosen from UML for which it is possible to define the execution semantics without ambiguity. $^{X}_{T}$UML try to address the deficiencies in UML 1.5 by offering Object Action Language (OAL) that provides the necessary conditional logic and primitive actions to manipulate the UML action model.

The idea leveraged by $^{X}_{T}$UML is impressive, but it is suffering from some problems. The first and biggest of them is that $^{X}_{T}$UML is not standardized yet. Another problem with $^{X}_{T}$UML is that it is at the same level of abstraction as other procedural and object oriented languages [Kleppe, 2003].

UML 2.0

UML 2.0 is the latest version of Unified Modeling Language which is expected to be finalized later this year. OMG, the owner of UML, claims that UML 2.0 has been redesigned to support MDA exclusively; again MDA also is one of the standards of OMG [UML, 2004].

The highlight of the UML 2.0 is its both structural and behavioral modeling nature. It is equally strong, as compared to previous version, in both of the disciplines. It not only incorporates some new modeling diagrams (some call them overlapping [Berkenkötter, 2003]), but also OCL and Action Semantics (AS) as a part of specification [UML, 2004].

3.4. Summary

This chapter specifies the rules for any modeling language to be the contender for the MDA process. This chapter also discusses different contender modeling languages available today. These choices are analyzed at very brief level at this chapter however the following chapters will discuss them in greater details.

The next chapter discusses the Object Constraint Language (OCL) and Action Language (AL) which play key role to make the modeling language executable.

4. The OCL and AL

This chapter tells the significance of the Object Constraint Language (OCL) and Action Language (AL) in the context of executable modeling.

4.1. Introduction

The phenomenon of executable modeling has been in the software industry for very long period of time. [Mosses, 1992] introduced the idea of action semantics, at the same time, when object oriented methodologies, for analysis and design, were being conceived. The practical steps towards this realm was taken when the OCL and AL was introduced. The incorporation of OCL [OCL, 1997] and AL [AS, 2002] in the specifications of UML was the main factor which today help make the UML executable.

The Object Constraint Language (OCL) is a notational language which helps in analysis and design of software systems. As a part of UML, it allows to write constraints and queries over object models, which UML alone is unable to do. These constraints create a highly specific set of rules that manage the characteristics of an object and business process eventually.

An Action Language (AL) is a specification language to specify actions and activities on a state diagram and operations on a class diagram. Ideally, an AL is a language that is implementation independent and at a higher layer of abstraction above implementation.

4.2. The Object Constraint Language

The Object Constraint Language (OCL) is a standard query language and also a part of UML. It helps applying unique and complex rules on the software systems. It gives further strength to the strong structural aspects of UML by enforcing business constraints and also to the dynamic aspects by putting the triggers. According to [Kleppe, 2003], it reinforces the dynamics of the system using pre and post conditions, for example, the body of a relative simple and corresponding function can be generated by the post condition. Pre and post conditions may also help validate that the generated code conforms the PIM, but dynamics of a system can not be fully generated using the OCL.

According to [Kleppe, 1998], the OCL is a language that allows describing expressions and constraints on object oriented software models.

Expressions
An expression is an indication or specification of a value. In typical UML model expressions can be used in a many ways:

- To specify the initial value of an attribute or association end.
- To specify the derivation rule for an attribute or association end.
- To specify the body of an operation.
- To indicate an instance in a dynamic diagram.
- To indicate a condition in a dynamic diagram.
- To indicate actual parameter values in a dynamic diagram.

Constraints
A constraint is a restriction on one or more values of an object oriented software model or a system. There are four types of constraints:

- An invariant is a constraint that imposes a condition that must always be met by all instances of the class, type or interface. Invariants must be true all the time.
- A precondition is a restriction that must be true before the operation start execution. The obligations are specified by preconditions.
- A postcondition is a restriction that must be true before the operation stop execution.
- A guard is a constraint that must be true before a transition in the state takes place.

The current version of OCL is OCL 1.4. It is being revised for OCL 2.0, which will be released with UML 2.0 and is expected to be released later this year.

4.3. The Action Semantics

Action Semantics (AS) is a practical framework to describe the formal semantics of programming languages such as values, functions, bindings and storage. Using this framework the description of programming languages becomes easier. Since its appearance in 1992 [Mosses, 1992], AS has been used to describe major languages such as Pascal and Java.

These AS can also be deployed in modeling languages to make them executable. The OMG adopts this concept and describes its own AS for UML to make it executable. These AS are defined in details in [AS, 2002]. OMG only defines these semantics, but there is no standard language which complies with these semantics. The language, which complies these AS, is called Action Language (AL).

4.4. The Action Language

An Action Language (AL) is a specification language to specify actions and activities on a state diagram and operations on a class diagram. Ideally, an AL is a language that is implementation independent and at a higher layer of abstraction above implementation. It should also be complete and 100% executable. The AL must conform to the Action Semantics (AS) defined by OMG [AS, 2002] in order to be said as AL. Though OMG defines these AS specifications, yet, they are normative and not a standard yet. Today only few AL are available which partly conform to the AS specifications such as Action Specification Language (ASL) [Wilkie, 2001], BridgePoint Object Action Language (OAL) and Kabira Action Semantics (Kabira AS).

4.5. The Role of AL in Executable Modeling

The modeling languages do not provide the functionalities of programming languages and programming languages do not provide the essence of modeling language. By summing both of the concepts, the concept of executable modeling emerges, the models which are also executable. This is all about raising the level of abstraction.

OCL is a key concept in this terminology. It expresses the constraint and relationships within a context of a single system. The OCL allows the modeler to specify conditions on changes to the instance model, but these specifications are not executable. These are actions and OCL is not an Action Language. Actions define how state transitions will actually be implemented in order to generate the code. The typical AL defines the concrete semantics between state machines and their contexts i.e. how state machines should be started or stopped or can any object have multiple active state machines. A typical AL will also convert the incoming messages to an event in order to process that using state machine and will also do vice versa i.e. will convert the event to an outgoing message. It also removes the ambiguity while handling synchronized states [Latta, 2001].

Action Specification Language (ASL) is one of few Action Languages. It is a different language compared to programming languages. It does not provide the concept of a *main* function from where the execution of the program starts. Instead, ASL starts its execution with concurrently executing state machines which works on synchronous call stacks. These call stacks may be invoked implicitly by the system or explicitly from outside the system. Any state machine may respond by changing state on the invocation of a signal. Again the signal may be implicit i.e. from another state machine or explicit i.e. from outside the system. Once the state machine has entered on the new state, a block of processing is performed [Wilkie, 2001].

According to [Wilkie, 2001], the organization of ASL is divided into “Segments”. The segment is a sequential set of ASL statements with a local data scope. An ASL segment corresponds to a “Procedure” in [AS, 2002]. In the process of executable modeling the ASL segments can be used to define the following:

- The processing to be carried out on entry to a state
- The processing to be carried out by a method behind an operation
- The startup sequence for a system either for test or for target system purposes
- The processing to be carried out by test methods for simulation purposes
- The processing to be executed in bridges providing the mapping between domains

The following are execution rules for a segment:

- Execution starts with the first statement in the segment and proceeds successively through the subsequent lines according to the logic structures.
- Execution of the segment stops when the last statement is completed.
- The only external data available to the ASL segment is:
    - Signal data, supplied with the signal or operation call that executes the segment.
    - Attributes of classes.
- There is no “global” data other than that defined class diagram.

- The local variables created within the segment are invisible once the execution of the segment ends.

4.6. Summary

The OCL and AS are the main factors, which not only support the concept of executable modeling but also help the UML to become executable. Both of them play the key role in the executable modeling. Though the current status of OCL is satisfactory to support the development of executable modeling, yet, it is being claimed that the new version of OCL will be a full object query language, which can be used to define any type of query over an object model. On the other hand, AS are still striving to become the standard and by the passage of time, it will be fine grained and will be able to support the executable modeling more effectively.

Next chapter evaluates UML 1.x to check its capability to support the development process inspired by MDA.

5. The Unified Modeling Language 1.x

This chapter describes the Unified Modeling Language 1.x. This chapter also describes the supporting structural and behavior modeling and diagrams of UML 1.x. It also investigates the possibility of UML 1.x as modeling language for MDA.

5.1. Introduction

Modeling is the designing of software blueprints before coding. The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing and documenting these blueprints. The UML was originally derived from the object modeling languages of three leading object-oriented methods: Booch, Object Modeling Technique (OMT) and Object-Oriented Software Engineering (OOSE). It was first added to the list of OMG adopted technologies in 1997 and has since become the industry standard for modeling objects and components.

Today UML and modeling has become synonyms. There is hardly any book in the market about modeling which does not use UML as modeling notation. UML is not a programming language instead it is a notational language which provides a standard way to help the modelers in designing of software models.

UML focuses on understanding a system via the formulation of a model of the system (and its related context). The model embodies knowledge regarding the system and the appropriate application of this knowledge constitutes intelligence. As a language, UML is used for communication. That is, a means to capture knowledge (semantics) about a subject and express knowledge (syntax) regarding the system for the purpose of communication. It can be used to specify the systems that "what" is required for a system and "how" a system may be realized or implemented. It can also help in the visualization of the system because it can be used to visually depict a system before it is realized. In the construction of the systems, it can be used to guide the realization of a system similar to a "blueprint". As it applies to documenting systems, it can be used for capturing knowledge about a system throughout its life cycle [Alhir, 1999].

5.2. UML 1.x Diagrams

According to UML 1.5 standard [UML, 2003], the UML 1.5 defines the nine different graphical diagrams in terms of the views of a model. These diagrams can be grouped in two categories, structural diagrams and behavioral diagrams.

5.2.1. Structural Diagrams

The UML 1.5 defines four types of diagrams which can be used to model the static structure of the system. These diagrams are class diagram, object diagram, component diagram and deployment diagram. According to UML 1.5 standard [UML, 2003], the definitions of these diagrams can be used as follows:

Class diagram
A class diagram shows the static structure of the model. It consists of a group of classes and interfaces which reflects important entities of the system, their internal structure and the relationships between these classes and interfaces.

Object Diagram
An object diagram is an instance of a class diagram. It shows a of the detailed state of a system at a specific point in time. The object diagrams are normally used to show the examples of data structures.

Component Diagram
A component diagram is high-level diagram which shows the organization and dependencies of a set of components. Component diagrams address the static implementation view of a system.

Deployment Diagram
A deployment diagram depicts a static view of the hardware used in the system and the software components that run on the hardware. It shows the hardware of the system, the software that is installed on that hardware and the middleware that connects the different machines to one another.

5.2.2. Behavior Diagrams

The UML 1.5 defines 5 types of behavior diagrams which can be used to model the dynamic structure of the system. These diagrams are use case diagram, sequence diagram, activity diagram, collaboration diagram and state chart diagram. According to UML 1.5 standard [UML, 2003], the definition of these diagrams can be used as follows:

Use Case Diagram
A use case diagram is a diagram that shows the requirement of the target system from the perspective of the user. It provides the functional description of a system and its major processes. It further provides graphical description of the users of the system and their possible interactions with the system.

Sequence Diagram
A sequence diagram describes how the groups of objects interact in some behavior over time. It shows the objects and the messages that are passed between these objects. Sequence diagram shows the interaction between objects in timely manners.

Activity Diagram
An activity diagram typically represents the invocation of an operation, a step in a business process or an entire business process.

Collaboration Diagram
A collaboration diagram is an interaction diagram that shows the structural organization of the objects and the process of sending and receiving messages. Collaboration diagram express similar information as sequence diagram, but shown in a different way. Sequence diagrams show the interaction between objects in timely manners whereas collaboration diagrams focuses on structural relationships between objects.

State Chart Diagram
A state chart diagram depicts a detailed picture of changing state of an object. A state refers to the value associated with a specific attribute of an object and to any actions or side effects that occur when the value of attribute changes.

5.3. UML 1.x Modeling

The UML, in general, provides two types of modeling behaviors: structural modeling and behavioral modeling. The following sections describe each of these modeling techniques according to UML 1.x:

5.3.1. Structural Modeling

The structural models of UML 1.x, which addresses the structural element of a problem and solution, consists of class diagrams, which depicts the static structure of a system in general; object diagrams, which depicts the static structure of a system in particular; component diagrams, that shows the organization and dependencies of components and deployment diagrams, which shows run-time configuration of hardware nodes and the software components that run on those nodes.

According to [Alhir, 2001], a class is a general concept that has both structural and behavioral features. Structural features, includes the attributes and the relationships called associations, which define what elements constitute the class. An attribute is an element of information or data. A relationship is a connection between classes. Behavioral features, includes operations and methods, which define how the elements of the class interact to exhibit the behavior of the class. An operation is a specification of a service or processing. A method is an implementation of a service or processing. There are two types of classes: active class, whose objects own a thread of control and may initiate control activity on their own behalf and a passive class, whose objects do not own a thread of control and may only initiate control activity by a request of thread of control.

Contrary to the class, the object is a specific concept. Object is an instance of a class. A class defines the structural and behavioral characteristics of its objects. An object defines values for its structural features. The attribute values and specific relationships are called links. The behavior of an object is defined by the behavior of its class that what it can do. The state of an object is the situation throughout the life of the object during which it satisfies some condition. The state of an object is defined by the values of the structural features of the object, what an object knows [Alhir, 2001].

The class diagram and the object diagram are the main sources of the structural modeling of UML 1.x. The component diagram only shows the components that compose the system, their interrelationships, interactions and their public interfaces. The deployment diagram shows the execution architecture of systems i.e. nodes, either hardware or software and middleware which connect them.

5.3.2. Behavioral Modeling

For behavioral modeling UML 1.5 provides 4 types of diagrams, use case diagrams, interaction diagrams (sequence diagrams and collaboration diagrams), activity diagrams and state diagrams. Use case diagrams describe the services offered by the prospective system. Sequence diagrams and collaboration diagrams model the interactions between objects. Activity diagrams emphasize the flow of control from activity to activity and statechart diagrams emphasize the potential states and the transitions among those states. Both statechart diagrams and activity diagrams can be used to specify the dynamics of various aspects of a system ranging from the life cycle of a single object to complex interactions between societies of objects. Activity diagrams typically address the dynamics of the whole system including interactions between objects. Statechart diagrams are typically used to model the life cycle of an object [Aalst, 2002].

Use case diagram, introduced in the early phases of the software model, shows the services that are expected to form a system. It provides the outside view of the system. The main ingredients of the use case diagram are the use case, which is the service provided by system and actor, which is an agent who will use the system services.

Considering the interaction diagrams, the sequence diagrams are typically used to describe specific scenarios of interaction among objects. The sequence diagram is considered to be restricted to lifelines, messages, activation and concurrent branching. Collaboration diagrams are closely related to sequence diagrams. In fact, they provide a different view on the identical structures. Therefore, the results obtained from sequence diagrams can easily be transferred to collaboration diagrams. This work is trivial and is supported by many tool vendors such as Rational Rose [Aalst, 2002].

In contrast to sequence diagrams, statechart diagrams are typically not used to specify scenarios, instead they are used to model the life cycle of an object. It is the key diagram which is used for behavioral modeling. Provided with action semantics, this diagram can be used to generate the code in the process of code generation.

Activity diagrams are typically used for modeling behavior which exceeds the life cycle of a single object. It is normally used for workflow modeling. Compared to statecharts, activity diagrams allow for actions states, subactivity states, decisions and merges, object flows and concurrent transitions to model synchronization and forks [Aalst, 2002].

5.4. MDA and UML 1.x

This section lists investigations on UML 1.x under the guidelines derived in chapter 3.

Expressiveness
UML 1.x is a big language and consists of many parts. Taking the diagrammatic part into the consideration, it can be said that it is not expressive enough to describe every computable function [Rumpe, 2002]. Notations of UML 1.x are also considered to be confusing and cause problems in communication [Berkenkötter, 2003]. This is also the fact that UML 1.x diagrams alone cannot put all required constraints on the system. UML 1.x uses Object Constraint Language (OCL) [OCL, 1997] to bridge this gap, for example specifying the pre and post conditions on operations and methods.

Application Independence
[UML, 2003] defines UML 1.x as a modeling language which is not only limited to software development, but it also spans to other non-software systems and business modeling. UML is not an application specific language and this is the reason why it is an industry standard modeling language. Today UML has become the acronym for modeling and this is due to the fact that UML supports every kind of application infrastructures.

Level of Abstraction
This is the point where UML 1.x is a strong contender as a modeling language for MDA. UML 1.x is at the higher level of abstraction than ordinary programming languages and [Rumpe, 2002] also acknowledges the same fact. For example in UML 1.x, it is possible to draw an association without specifying the implementation details of this association. The tool will take care of the implementation details.

Support for N-tier Applications
UML 1.x does not provide an effective way to design the artifacts for n-tier application development. It has been often observed that when attempting to model n-tier applications with UML, some of its components don't fit nicely into standard UML 1.x modeling elements. UML 1.x often requires some customized extensions for this purpose. Time to time many of such extensions has been proposed such as [Conallen, 1999] and [Li, 2000]. UML 1.x lacks the precision and needs to be worked out further to support the n-tier application development.

Support for Distributed Applications
Currently UML is also being deployed for designing artifacts for distributed application development and there are not as such problems in this area. For example, UML has been playing an important role in the designing of artifacts for CORBA.

The Seamlessness
The issue of seamlessness between the models and their implementation depends upon the ability of the language to transform the models into the code. There are two important considerations to achieve this seamlessness: first, the executable models and second, the tools, which transform these models into the code. In case of UML 1.x, the seamlessness cannot be obtained for these two reasons. First the language does not adhere the concept of executable modeling and secondly there are no tools, available today, which explicitly takes the UML 1.x diagrams as input and fully translates them into the code. Although some tools, such as

Telelogic TAU Gen1 [TAU, 2004], provide partial implementation of the models, yet, this is not at that much detailed level which is required for MDA.

Large Models Management
UML 1.x leads to very complex specifications, when large models, especially, aspect-oriented models, are involved [Mikkonen, 2002]. However, despite this fact, UML 1.x is currently being used to handle the large models. Aspect-oriented modeling is a relatively new term and it is being researched that how the UML 1.x can be used to support it. Some of the extensions to UML 1.x in this regard are proposed by [Mikkonen, 2002] and [Zakaria, 2002].

Tool Support
The UML diagrams are translatable and some tools, present today, also fully translate these diagrams, efficiently, into the code. But this support is only limited to the latest version of UML i.e. UML 2.0 (See appendix 1 for detailed survey of tool). There is no specific vendor which claims to fully translate the UML 1.x diagrams into the code and provides a tool that takes diagrams of UML 1.x as input and translates them into the code. Some tools like Telelogic's TAU Gen1 [TAU, 2004] only generates partial code from models.

5.5. Summary

The fact that UML 1.x lacks a precise semantics for behavioral modeling is a serious drawback, in order to be the language of choice for MDA [Ledang, 2001]. There are also other problems which are questions on the efficiency of UML 1.x such as lack of tool support, incapability of managing large models etc. UML 1.x provides some good feature as well. Such as the structural modeling and level of abstraction but this is not enough to consider the UML 1.x as the best supporting modeling language for MDA.

Next chapter evaluates $^{X}{}_{T}$UML to check its capability to support the MDA development process.

6. Executable and Translatable UML

This chapter evaluates Executable and Translatable UML ($^{X}_{T}$UML) in details. The underneath development process and supporting notation of $^{X}_{T}$UML will be discussed. The chapter also evaluates the ability of $^{X}_{T}$UML to support the development process of MDA.

6.1. Introduction

The history of software development is all about evolution. There has always been a flux in software industry regarding the approach of software development. The problems of past are no longer there, instead industry has been facing new challenges and executable modeling is one of them.

One answer to the challenge of executable modeling may be the Executable and Translatable UML ($^{X}_{T}$UML). $^{X}_{T}$UML is a language which is at higher layer of abstraction as compared to UML 1.x. It claims to obsolete not only programming language practices but also the problems of organization of the software. If these claims are true, the artifacts built with $^{X}_{T}$UML would be deployable directly in software environments without the need of programming language.

The $^{X}_{T}$UML is a graphical specification language. It combines a subset of the Unified Modeling Language (UML) graphical notation with executable semantics and timing rules. This language can be used to build executable system specifications consisting of class, state, and action models that run just like a program. Unlike traditional specifications, an executable specification can be run, tested, debugged and measured for performance. The tested specification (models) can then be translated into target code [Projtech, 2004].

The $^{X}_{T}$UML is a profile of UML which allows describing the specification of a system into sufficient details that it can be translated into implementation. It is an effort to abstract away not only decisions about the organization of the software but also the programming languages itself so that only a specification can be deployed in various software environments without change simply neglecting the usage of any programming language [Mellor, 2002].

The $^{X}_{T}$UML is a good contender to become the supporting pillar of MDA. A PIM can be written in this language directly and leveraging the transformation tool, PSMs can also be produced. For transformations, $^{X}_{T}$UML uses transformation tools which compile the models into the source code or detailed PSM. $^{X}_{T}$UML calls them Model Compilers.

6.2. The $^{X}_{T}$UML Development Process

In the $^{X}_{T}$UML software development process application specific but implementation independent models are generated at first. These models are the specifications of the system. These models are then passed to translators for the implementation process on specific platform. This procedure is depicted by figure 9. These translators are application independent, target platform specific and target platform optimized. Theses translators produce the code for the supplied models. As claimed by tool vendors, the code generated by these translators is well commented, understandable, consistent, low defect and is free from hand coding errors [Projtech, 2004].

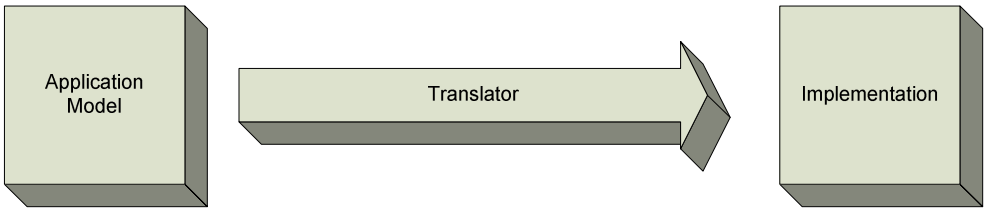

Figure 9 The $^{X}_{T}$UML development process

Executable Models
Considering software development as a manufacturing process, the models act as the blueprints. In $^{X}_{T}$UML models are built and are immediately verified for the behaviors without undergoing coding practices. To build these models requirements are gathered and are expressed as use cases. Requirement gathering and domain analysis go together. Some time the domains are already known while making use cases and some times they are identified afterwards. So the process iterates between these two activities. Once done with requirements, appropriate abstractions are invented and detail decisions are made about how the domain works. The result is an $^{X}_{T}$UML model that defines the behavioral requirements for a domain. The $^{X}_{T}$UML model for each domain comprises a set of tightly connected class, state and action models. As the models are built, their behavior is required to be verified. Formal test cases are executed against the models to verify that application requirements have been properly addressed. No design details or coding is required for the execution of these models [Mellor, 2002].

Translation
The process of translation is one of the most powerful aspects of the $^{X}_{T}$UML. This process has the ability to automatically translate application models into fully target executable and target optimized code. According to [Projtech, 2004] and as depicted by figure 10, the process of translation is comprised of three steps:

- A set of design patterns and translation rules.
- A translation engine that extracts $^{X}_{T}$UML application model information, interprets the design patterns and rules and performs the mapping.
- A run-time library comprising pre-compiled routines that support the generated code modules.

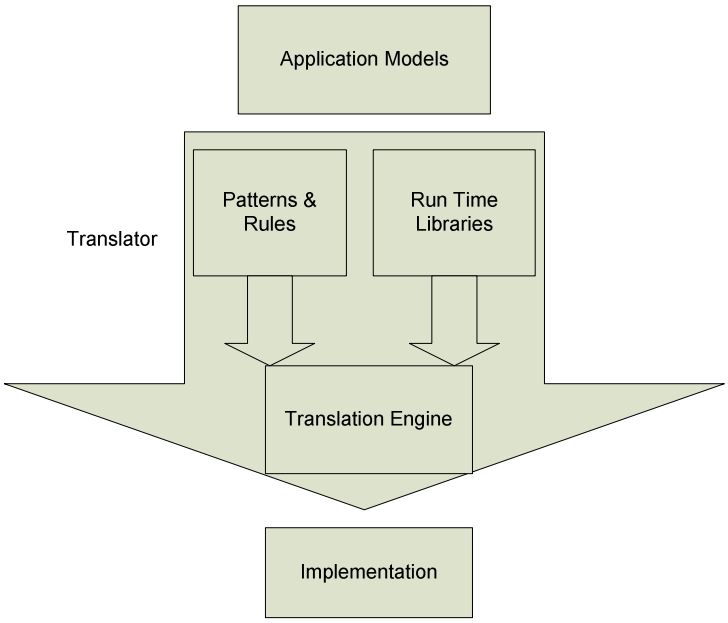

Figure 10 The $^{X}_{T}$UML translation process

In the process of code generation i.e. translation, the translator extracts information from the $^{X}_{T}$UML application model. The translator then selects the appropriate design pattern for the "to-be-translated" model component. The information extracted from this model is then used to "fill in the blanks" of the selected design patterns. As claimed by the tool vendors, the result is a fully coded model component. The robust design of translator also allows changing design patterns, translation rules or runtime libraries without having to contend with the details or code of the translation engine itself [Projtech, 2004].

The level of completeness, consistency and unambiguity of a PIM must be very high otherwise it will not be possible to generate a compatible PSM from a PIM. $^{X}_{T}$UML is one of the strong contenders as a modeling language for PIM. It not only supports MDA only in modeling but also helps afterwards.

As shown in figure 11, [KC, 2002] suggests that embodying $^{X}_{T}$UML into the process of MDA may yield into the following hierarchy:

- Build precise models
- Test these models before implementing them
- Establish an automated and well defined construction process
- Construct the product using reusable components

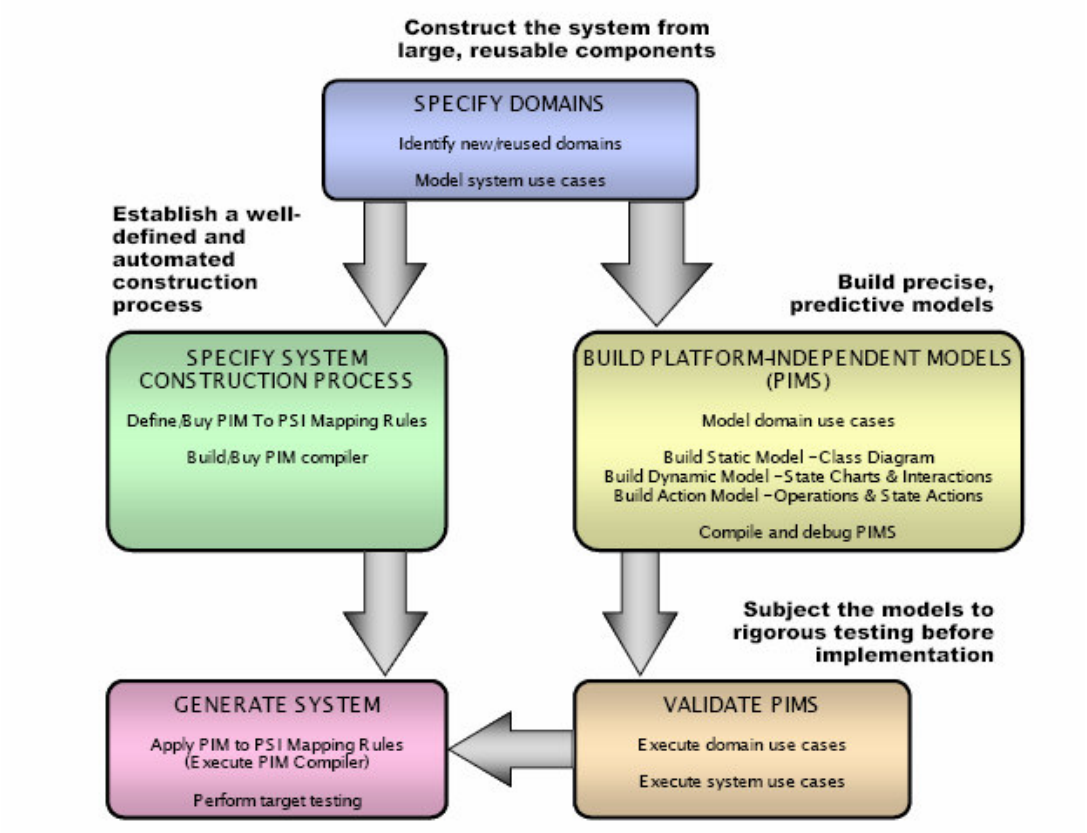

Figure 11 The MDA process using $^{X}_{T}$UML [KC, 2002]

The MDA process embodying $^{X}_{T}$UML kicks off with the partition of the system, new or used domains are identified. These domains may be application domains such as air traffic control or network management or service domains that support the application such as user interface or hardware interface. Once the domains have been identified, the process steps ahead and system use cases are defined. These system level use cases represent the various capabilities that system must deliver and modeled as interactions between the domains. These uses cases also serve as the base to test these domains. Realistic systems are generally composed of more than one domain and each domain may provide some services to other domains. So domain contracts are defined in each platform independent domain model that these domains may provide or request services from other domains. When it is required to integrate two platform independent domain models, the bridges are specified which provides the PIM to PIM mapping between the required and provided operation.

For each new domain platform independent models are created. These models are created in three stages. The first step is static model. Static model describes the conceptual entities in the problem domain in terms of classes, attributes and associations. In the second stage the dynamic behavior of the stage and state models are used in order to invoke the dynamic behavior. The state models are represented in the form of state charts and state tables. During the third stage the actions are defined. These actions are defined in two places once in state charts in response to signals and other in the methods which implement operations. These actions are expressed using Object Action Language (OAL) in $^{X}_{T}$UML.

Testing of platform independent domain models before implementation is a good feature powered by $^{X}_{T}$UML. Domain use cases are tested in order to make sure that they provide the required functionality. One interesting feature is that these domain models need not to be completed when testing take place. This phase simulates with unit testing. Once domain level testing is successful, sets of domains can be tested together to verify the bridge mappings. This is done by the execution of system level use case and is equivalent to integration testing.

After this platform independent domain models testing, the modeling of platform specific mapping can safely be started. Generic design patterns are specified and target platform is selected such as programming language, database, operating system and etc. Software design is evaluated and tested for space, performance and reliability constraints. After this testing the automatic code generation process starts. Code generators are required for this purpose. They are supposed to generate 100% fully functional code but due to any reason this code may not be 100% fully functional. Analysis models, mapping rules, choice of code generation tool or anything else may be the reason.

6.3. The $^{X}_{T}$UML Notation

The $^{X}_{T}$UML does not define its own set of notations rather it comprises of a carefully selected subset of UML that supports the needs of execution-based and translation-based development. This subset contains three primary components: class diagrams, state diagrams and procedure specifications (figure 12). The class diagram is the main diagram which helps specify the static structure of the overall application. The state diagram is the key diagram which is deployed to attain the dynamic behavior of the application specification. The procedural specifications such as Action Language (AL), provides the semantics which help model to execute. Additional diagrams (e.g., collaboration diagrams) are automatically generated from information captured in the three primary components. These generated diagrams facilitate in effective development [Projtech, 2004].

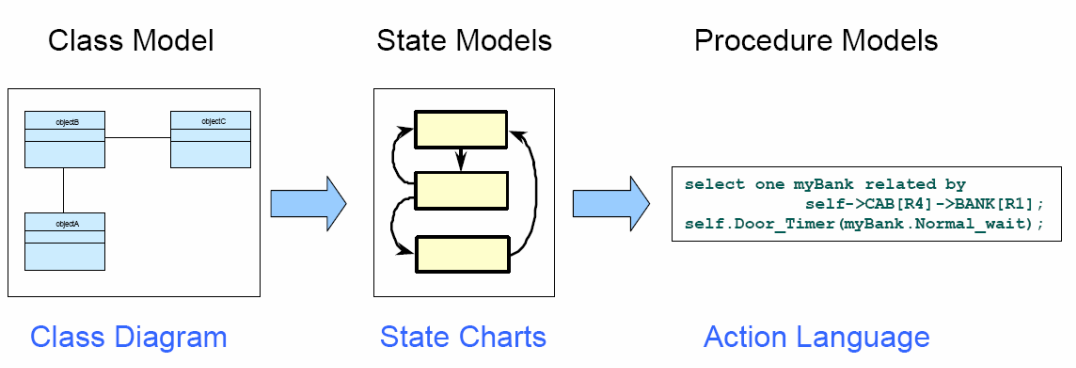

Figure 12 The $^{X}_{T}$UML notational hierarchy [Projtech, 2004]

6.4. The Object Action Language

The $^{X}_{T}$UML uses Object Action Language (OAL) to implement the AS. It defines the semantics for the processing that occurs in an action. According to [OAL, 2003], actions can be associated with the following list of modeled elements:

- States
- Bridge operations
- Functions
- Class and instance based operations
- Mathematically dependent attributes

The OAL provides five types of action processes which help to transform the model into the code. These action processes are:

- Data access
- Event generation
- Test
- Transformation
- Bridge and function

The underlying methodology which OAL adopts to support these actions is comprised of:

- Control logic
- Access to the class data
- Access to the data supplied by events initiating actions
- The ability to generate events
- Access to timers and to the current time and date

The Object Action Language is used to define the processing executed during the action. The execution rules are as follows:

- Execution starts at the first statement in the action and as directed by control logic structures, it proceeds successively through the subsequent lines.
- Execution of the action stops when the final statement is completed.

These rules also apply to actions defined for bridge operations, functions, class and instance based operations and mathematically dependent attributes.

6.5. MDA and $^{X}_{T}$UML

This section evaluates $^{X}_{T}$UML under derived guidelines of chapter 3 to see the ability of $^{X}_{T}$UML to support MDA.

Expressiveness
As aforementioned, the $^{X}_{T}$UML is defined leveraging the strength of UML for its structural part and action semantics for its dynamic behavior. So it inherits the same problems as UML have for its structural part such as confusion in communication and incapability to define every computable function. However the action semantics, included in $^{X}_{T}$UML, try to make the $^{X}_{T}$UML process more expressive. They enhance the dynamic part of the language and also abstract away the need for intermediate programming languages for coding. But the problem is that there is no standard action specifying language, present today, which can be used for this purpose.

Application Independence
The $^{X}_{T}$UML rely heavily on state diagrams to specify the complete behavior. This approach is good but only in specific domains such as embedded software development. In others, more administrative domains, the usage of state diagrams in order to define the complete behavior is too cumbersome to be used in practice [Kleppe, 2003]. This thing gives an impression that $^{X}_{T}$UML may not be suitable for all kind of applications.

Level of Abstraction
The $^{X}_{T}$UML uses Object Action Language (OAL) [OAL, 2003], to implement the Action Semantics (AS) [AS, 2002]. OAL is not a very high level language. In fact, it is at the same level of abstraction as current procedural and object oriented languages. The PIM is at the higher level of abstraction than PSM and to design the artifacts of PIM, more abstract language is required. Therefore, using $^{X}_{T}$UML has little advantage over writing the dynamics of the system in the PSM directly. Same amount of code would have to be written at the same level of abstraction. The OAL should provide more abstraction, so that the amount of code to be written reduces with the passage of time and becomes obsolete eventually.

Support for N-tier Applications
The $^{X}_{T}$UML is equipped with some smart concepts which help in variety of scenarios such as development of n-tier applications. The $^{X}_{T}$UML helps in case of n-tier application development by providing the model testing facility before implementation. The “divide and conquer” rule provided by the concept of project partitioning in $^{X}_{T}$UML also helps in problem solving for n-tier applications. The provided dynamics of the language also helps in the automated generation of code.

Support for Distributed Applications
The $^{X}_{T}$UML also helps in case of distributed application development. As aforementioned, the facility of testing the models before actual implementation of these models is a significant help. The interactions of different remote objects can be mapped and can be verified for the correctness of the design. The process of automated code generation using different tools saves time and avoids possible coding errors.

The Seamlessness
The $^{X}_{T}$UML provides seamlessness between the models and the implementation. It has the power to execute the models into the code but it is supported only by limited number of vendors. Carefully modeled artifacts can be easily transformed into the running code using the limited number of translation tools.

Large Models Management
The $^{X}_{T}$UML provides the concept of project partitioning to support the large models. The larger system is partitioned into many small subsystems. This concept provides an ease to handle large models. Each subsystem may focus on different behavioral aspects of the system.

Tool Support
The process of translation is one of the most powerful aspects of the $^{X}_{T}$UML. This process has the ability to automatically translate application models into target executable and target optimized code. This is done using the supporting tools. Since $^{X}_{T}$UML is not a standard so it is not backed up by large number of companies. Today only few vendors support the $^{X}_{T}$UML. So it has only limited tool support such as Kennedy Carter's iUML or iCCG [KC, 2004] and Projtech's BridgePoint [Projtech, 2004]. See appendix 1 for detailed analysis of these tools.

6.6. Summary

The emergence of $^{X}_{T}$UML is an effort to raise the level of abstraction of the modeling languages. Carefully drawn models can easily be translated into the code using the adequate $^{X}_{T}$UML tool support. The $^{X}_{T}$UML provides some good feature such as project partitioning, platform independence, reduced defect rates due to automated coding and reusability, but these arguments are not enough. The biggest problem with $^{X}_{T}$UML is of standardization. It is not a standardized modeling language. It may be useful in limited domain, but not on a mass level.

Next chapter evaluates UML 2.0 to check its capability to support the development process of MDA.

7. Unified Modeling Language 2.0

In this chapter the UML 2.0 will be discussed in details. This chapter also evaluates its provided features to check its ability to support the development process of MDA.

7.1. Introduction

The UML 2.0 is the latest version of UML which introduces many new features. These new features include the improved old diagrams and incorporation of some new diagrams which help specifying the models more effectively. The specification of UML 2.0 also includes the updated AS which make the translation process more effective. The UML 2.0 RFP was released in September 2000 consisting of 4 parts Infrastructure, Superstructure, OCL and Diagram Interchange (XMI). It was originally due on August 20, 2001 but actually adopted in June 2003. It is currently in finalization process and expected to become a released standard later this year.

As shown in figure 13, UML 2.0 is divided into a language core (infrastructure) that is compliant to Meta Object Facility (MOF), Common Warehouse Metamodel (CWM) and other metamodels supported by the OMG and modeling elements (superstructure) that build up on the core and provide functionality that is needed for constructing models. This distinction helps identifying fundamental parts of the language that are needed as a basis for building models on the one hand and elements that are really used in models on the other hand. Furthermore, compliance between metamodels and model interchange is improved in this way. Nevertheless, UML 2.0 has not a formal specification but an informal one which mostly consists of natural language in addition with some constraints given in OCL [Berkenkötter, 2003].

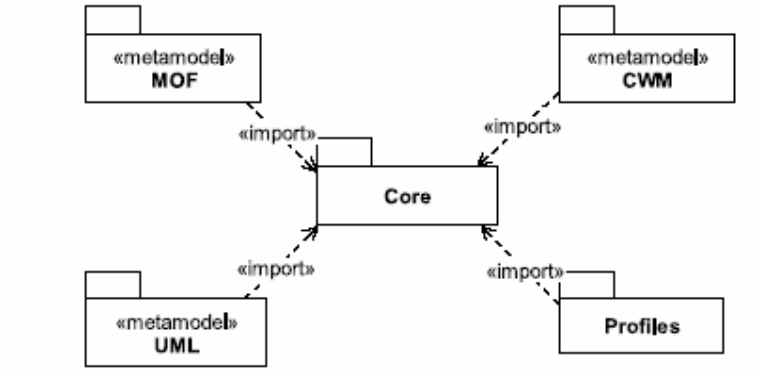

Figure 13 UML 2.0 core concept [Berkenkötter, 2003]

7.2. Improvements in UML 2.0

UML 2.0 provides many improvements over UML 1.x. Some of these improvements are listed here:

Improved communication
The role of communication between different teams is inevitable. UML 2.0 improves this communication between different teams by incorporating Diagram Interchange Specification. UML 2.0 includes a formal interchange specification using XML, which helps the electronic interchange of interface specifications or complete models. It minimizes the possibility of errors in formal communication.

Improved Structural Modeling
According to [Berkenkötter, 2003], the UML 2.0 provides several new concepts that offer a wide variety of possibilities in modeling software systems including e.g. hierarchical composition of models, communication structures, and components. Probably the most important of them is the hierarchical structuring of classifiers. Each classifier that may (classes, components) or must (collaborations) be composed by other elements therefore may or must contain an internal structure which specifies its inside. The internal structure describes how a containing element is composed by other elements called parts or connectable elements. These are never classifiers themselves, but instances or instance sets of classifiers. The relationship between the containing element and its internal structure is a kind of aggregation, but of a strict form as the internal elements belong to their containing element and cannot exist without it. In this way, a model can be described in different levels of abstraction as its elements can be nested, e.g. components that consist of other components and so on. Hiding the nested parts gives an overview of the system under consideration while showing them allows detailed information. All these features are important for the component concept. A component owns an internal structure that shows how it is composed and interacts with its environment exclusively over interfaces or more often, ports. Therefore a component can be replaced by another one which offers at least the same provided and required interfaces or ports as these are the only parts of the component which are accessible by its environment. Subsystem is a standard stereotype of component in UML 2.0. Physical instances of software are called artifacts which may be implementations of components. They can be deployed on nodes, if needed with an additional specification that describes the deployment.

Improved Behavioral Modeling
According to [Berkenkötter, 2003], the UML 2.0 is supported by a fine grained version of an upgraded action semantics known as action model. It increases the possibilities of modeling behavior. The action model defines more than twenty different action types. But actions are the smallest kind of behavior in fact activities are required for processing and coordinating them. Activity diagrams consist of nodes and edges where nodes are either object nodes or control nodes like fork nodes, join nodes, merge nodes, etc. They also support interruptible regions or loops to improve the possibilities of activity modeling. Another feature in behavioral modeling is UML concerns interactions, which are used for describing interactions between parts, respectively connectable elements. This concept is similar to the one depicted by sequence diagrams and collaboration diagrams. Indeed, sequence diagrams and communication diagrams (just a new name for collaboration diagrams) are also used for depicting interactions, but these are of a more profound nature. Instead of just visualizing message flow, interactions (more precisely interaction fragments) can be grouped to combined fragments that support further modeling possibilities like alternatives, options,

breaks, loops or critical regions. Interaction flow can be modeled by an interaction overview diagram, i.e. a specific activity graph whose activities are all interactions.

According to [Gurd, 2003], the sequence diagram can now describe complex behavior in a concise and powerful way, which is effective for describing functional requirements. For example figure 14 shows the detail of the interaction of a *ProcessTrack* use case of a missile program. Since all the systems have states, so the scenario sequence diagrams can include this state information when required. The implication of the two state symbols labeled *armed* is that both the *RadarDataProcessor* and the *FireController* instances must be in the *armed* state before the described sequence can happen. This provides essential information exactly where it is needed to understand the requirements. Figure 14 also shows how the reuse of behavior is made, by including a *ref* symbol containing the name of another interaction, in this case *CheckIFF*. Of course, several references may be made in one diagram, and a referenced sequence may itself contain further references.

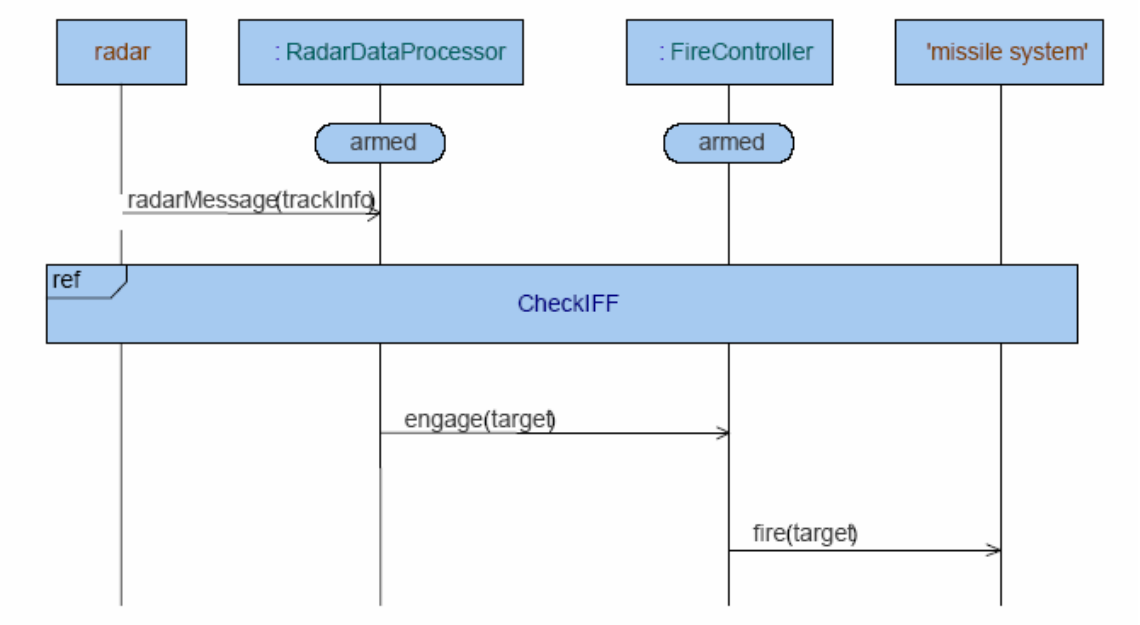

Figure 14 UML 2.0 sequence diagram [Gurd, 2003]

Use cases are also regarded as behavior even if strictly speaking, a use case is a classifier and not a behavior itself. Nevertheless, use cases describe the behavior that is offered by the system under consideration, i.e. they visualize the requirements of this system. If the main emphasis shall be on reasoning about time, timing diagrams can be used to visualize change in states or other conditions of structural elements. There are two types of state machines: behavioral and protocol state machines. The first ones describe intra object behavior and the second ones are used for defining protocols. The main difference between these two types of state machines is that behavioral ones describe the specific behavior of a classifier while protocol ones describe an abstract behavior and may be associated to interfaces or ports [Berkenkötter, 2003].

According to [Gurd, 2003], UML 2.0 provides a state-centric state diagram notation, which is ideal for use during the early stages of analysis. When the system specifications are being derived, main problem is to discover the states. At this initial level the state diagram can be deployed to help identify the initial requirements (Figure 15). Later in the analysis, when specifications get matured, these state machine diagrams can be elaborated partially details (Figure 16). Finally, when all of the states are known and the problem becomes one of clearly specifying the detail of what happens during the transitions, full state machine diagram can be drawn.

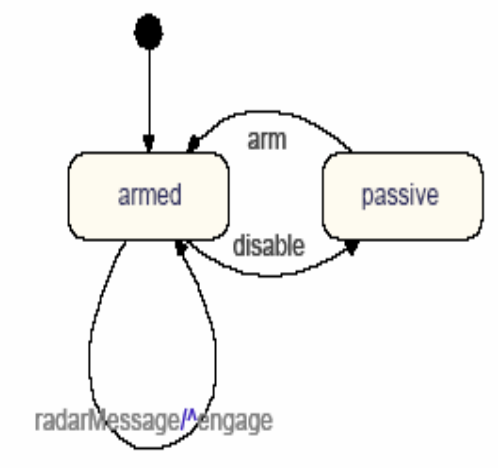

Figure 15 UML 2.0 initial state diagram [Gurd, 2003]

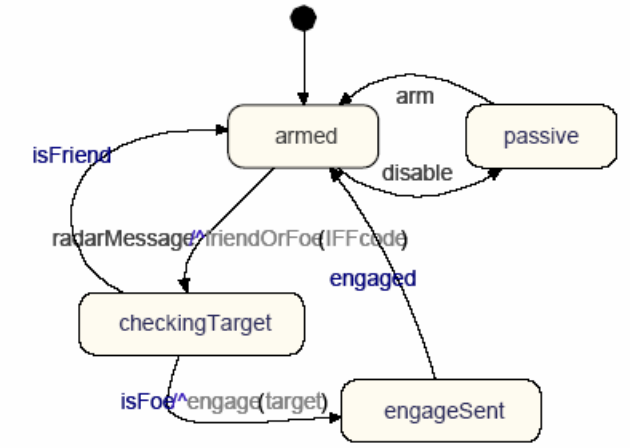

Figure 16 UML 2.0 partially detailed state diagram [Gurd, 2003]

Project Partitioning

Dividing the large system into many subsystems has many advantages e.g. different teams can work on individual parts of the system at the same time. These subsystems can later be integrated to make the complete system. The subsystems are easy to develop, reusable and make the system rapidly. To attain this approach, UML 2.0 provides architecture diagrams and interface classes. Architecture diagrams are completely new in UML 2.0. They allow subdividing a system into a hierarchy of subsystems, known as components.

Testing Facility
The test profile of UML 2.0 allows full test specifications to be modeled. This facility allows the testers to define the architecture of test systems and to define tests. This is a good incentive.

7.3. UML Diagrams

According to [UML, 2004], UML 2.0 constitutes of twelve types of diagrams, divided into three categories. Four diagram types represent static application structure, five represent different aspects of dynamic behavior and three represent the ways how application modules can be organized and managed. Structural Diagrams include the Class Diagram, Object Diagram, Component Diagram and Deployment Diagram. Behavior Diagrams include the Use Case Diagram, Sequence Diagram, Activity Diagram, Collaboration Diagram and State Chart Diagram. Model Management Diagrams include Packages, Subsystems, and Models. Some other sources ([Berkenkötter, 2003], [Ambler, 2004]) contradicts with [UML, 2004] and says that UML 2.0 is constitute of thirteen diagrams and diagrams which they claim is part of UML 2.0 are timing diagram which is a member of diagrams group that exhibit the behavioral structural of application and composite structure diagram that is the part of structural diagrams.

7.3.1. Structural Diagrams

Two new structural diagrams included in UML 2.0 are:

Composite Structure Diagram
Composite Structure Diagram depicts the internal structure of a classifier such as a class, component or use case. It also includes the interaction points of the classifier to other parts of the system.

Package Diagram
Package Diagram depicts that how the model elements are organized into packages as well as the dependencies between packages.

7.3.2. Behavioral Diagrams

The new behavioral diagrams of UML 2.0 are:

State Machine Diagram
State Machine Diagram describes the states of an object or interaction and their transitions between states. It was formerly referred to as a state diagram, state chart diagram or a state transition diagram

Communication Diagram
Communication Diagram depicts the instances of classes, their interrelationships and the message flow between them. Communication diagrams typically focus on the structural organization of objects that send and receive messages. They formerly used to be called Collaboration Diagram.

Interaction Overview Diagram
Interaction Overview Diagram is an alternative of an activity diagram which overviews the control flow within a system or business process. Each activity within the diagram can represent another interaction diagram.

Timing Diagram
Timing Diagram depicts the change in state or condition of a classifier instance or role over time. It is typically used to show the change in state of an object over time in response to external events.

7.5. MDA and UML 2.0

This section evaluates UML 2.0 under derived guidelines of chapter 3 to see the ability of UML 2.0 to support MDA.

Expressiveness
UML has not been expressive enough until the time when OCL became the formal part of its specification. OCL is a rich textual language to describe properties of the software systems. UML, coupled with an executable sub-language of OCL, will be expressive enough to describe each possible computation function. The question of how to access graphical user interfaces and operating systems and other issues of this kind will probably be solved in the same way as in ordinary programming languages. This means, it is required to provide modeling libraries especially suited for these issues that have a hard-coded implementation. Providing modeling libraries for reusable code would, of course, always be useful [Rumpe, 2002].

New set of diagrams, introduced in UML 2.0, has also further improved the expressiveness of UML. For example, timing diagram is a new diagram and is used to explore the behaviors of one or more objects throughout a given period of time. But [Berkenkötter, 2003] comments that the introduction of some of the new diagrams in UML 2.0 is dubious because there are some concerns that if they were really needed or if they are just unnecessary overlapping.

Application Independence
As discussed for UML 1.x, the UML 2.0 also, is not an application specific modeling language. UML 2.0 can be used for any kind of software modeling. It provides the mechanism and set of diagrams which are good for defining the software models as well as non-software models. Currently UML is being deployed for every kind of software modeling ranging from real-time systems to standalone desktop applications.

Level of Abstraction
UML 2.0 is at higher level of abstraction than UML 1.x. The ability to execute models and the improved expressions of OCL to define constraints, are few arguments which take it at the higher level of abstraction. The action model, included in UML 2.0, is the enhanced version of old action model, in which, there is possibility to express actions with precise procedural semantics [Björkander, 2003]. The revised version of OCL [OCL, 2003] is also at higher level of abstraction than its successors and it is expected to be even more in OCL 2.0, which will be released with UML 2.0.

Support for N-tier Applications
The executable semantics provided by UML 2.0 will definitely be a great help to support the n-tier application development. They will provide the facility to test the design artifacts before going further into the development. OMG is also trying to extend the UML using the profile concept. Profiles are used to extend UML with domain specific elements. So it will also be possible to the make custom profiles, tailored specially to meet the need for any particular criteria, such as n-tier application development.

Support for Distributed Applications
Component based modeling technique and the ability to test design artifacts before actual implementation process make the UML suitable for distributed application development as well, for example, better support for CORBA has also been an important consideration while designing the UML 2.0.

The Seamlessness
The important issue of seamlessness between the model and the code can be addressed properly using UML 2.0. The concept of seamlessness depends upon the ability of language to produce the executable models which can be transformed into the code using the supported tool. Drawing of precise models and selection of right tool will be the main key for this problem. In case of UML 2.0, it should not be a problem, because UML 2.0 is supported by adequate tool support. Carefully drawn models will easily be converted into the efficient code.

Large Models Management
[Björkander, 2003] says that component based development is one of the main feature of UML 2.0. The UML 2.0 decomposes the larger models into the smaller ones. The interaction diagrams provide interaction between these models. UML 2.0 composite structure diagrams also provide an effective way to describe complex architectures using the concept of subsystems [Gurd, 2003]. Thus the interaction and composite structure diagram provides a methodology to manage large models in UML 2.0. The UML 2.0 ability to test the models for errors before actual implementation also supports the large models management.

Tool Support
Using the transformation tools, the UML 2.0 models can be translated directly into the code. Being a standard, UML 2.0 is also supported by large number of companies and its tool support is also adequate such as Telelogic TAU Gen2 [TAU, 2004] and I-Logix Rhapsody [Rhapsody, 2004]. See appendix 1 for detailed evaluation of these tools.

7.6. Summary

UML 2.0 is the latest version of industry standard modeling language UML. It has been revised to improve many of its shortcomings. It provides lots of improvements such as better communication, better structural modeling, better behavioral modeling, project partitioning and etc. It provides better support of MDA as compared to its successors and the OCL and AS play an important role in order to achieve this betterment. Being an industry standard, UML 2.0 is most likely to be deployable on mass level. UML 2.0 also enjoys adequate tool support. There are some flaws which come by revision. But UML 2.0 meets most of the derived criteria.

8. Summary & Conclusions

Software industry is facing new challenges everyday. The problem of interoperability between past, present and future platforms is one of them. MDA is a promising approach to solve this problem. MDA addresses this issue by introducing the concept of PIMs, the platform independent models that are executable. To attain the concept of executable modeling MDA relies on UML.

The UML is the industry standard for modeling. To avoid the problems such as communication and learning curves, using the standards has always been the best approach to design the software models. The UML 1.x, indeed, has many shortcomings, which OMG tries to overcome in UML 2.0. This increase in strength of UML comes with some known flaws introduced by revision. This is the price, which is often to be paid for standards and for having different models and tools which can interoperate. The $^{X}_{T}$UML is a nice effort to bridge the gaps remained in UML. But the biggest problem is that it is not a standardized way to model the artifacts. It can be used as a profile for UML in limited scope, where it may be more productive. But in the longer run, relaying on the standards is the best strategy.

The investigations, made regarding the MDA-supporting modeling languages to judge them under the derived guidelines, are summarized in the following table:

Criteria	**UML 1.x**	**$^{X}_{T}$UML**	**UML 2.0**
Expressiveness	+	++	+++
Application Independence	++	++	+++
Level of Abstraction	++	++	+++
Support for N-tier Applications	+	++	+++
Support for Distributed Applications	+	++	+++
The Seamlessness	+	+++	+++
Large Models Management	+	+++	+++
Tool Support	+	++	+++

Tabel 1 A comparison between UML 1.x, $^{X}_{T}$UML and UML 2.0

\+ Not Suitable
++ Suitable
+++ Well Suited

By considering all the arguments which have been made in the favor or against all of the discussed modeling languages, it can be concluded that the usage of UML 2.0, as a modeling language for MDA, is comparatively a better choice. Deployment of UML 2.0, as a modeling language to design the artifacts for the development inspired by MDA, may produce better results compared to others in general scenarios.

Transformation tools play an important role in the overall process of MDA. A good transformation tool may significantly improve the productivity of a development team. After selection of modeling language, the next step is to select the best transformation tool for that language. UML 1.x does not fully adhere to the concept of executable modeling, so there is a little choice of tools for that. The $^{X}_{T}$UML is also supported by limited number of software vendors. But UML 2.0 has adequate tool support. Despite the fact that it has not been finalized yet, there are lots of tools available in the market which support the development of MDA using UML 2.0. All of the available tools claim themselves the best tool. In this situation it is very difficult for the software modelers to choose the right tool for their needs.

The appendix A lists some of the functionalities which should be present in a good tool for each modeling language. The appendix also evaluates some of the tools for the vendor claimed qualities of each tool, but due to the time constraint it was not possible to evaluate them in details. According to the derived guidelines in appendix A, the best transformation tools to support the development process of MDA using UML 2.0, are Rational Rose and InnoQ iQgen. The reason, why they seem to be promising, is their capability to support more programming languages and platforms than other transformation tools. But these are only initial thoughts on these tools by just looking at their claimed capabilities. The practical evaluation of these tools to identify the best supporting tool for UML 2.0 is an important issue and needs to be worked out in future.

References

[Aalst, 2002] Aalst W, Inheritance of Dynamic Behavior in UML, Research paper, Proceedings of the Second Workshop on Modelling of Objects, Components and Agents (MOCA 2002), volume 561 of DAIMI, pages 105-120, Aarhus, Denmark, tmitwww.tm.tue.nl/staff/wvdaalst/Publications/p161.pdf, August 2002

[Acme, 2004] The Acme Architectural Description Language, www-2.cs.cmu.edu/~acme, Carnegie Mellon University, Lastly accessed: May 13, 2004

[Alhir, 1999] Alhir S, Understanding the Unified Modeling Language (UML), Methods & Tools: An international software engineering digital newsletter published by Martinig & Associates, April 1999

[Alhir, 2001] Alhir S, Extending the Unified Modeling Language, UML World 2001 Conference, NY, USA, June 2001

[Alhir, 2003] Alhir S, Understanding the Model Driven Architecture, Methods & Tools: An international software engineering digital newsletter published by Martinig & Associates, home.earthlink.net/~salhir/UnderstandingTheMDA.PDF, October 2003

[Ambler, 2004] Ambler S.W, The Object Primer 3rd Edition: Agile Model Driven Development with UML 2.0, Cambridge University Press, 2004

[AS, 2002] Action semantics for UML, Action semantics specification document, www.omg.org/docs/ptc/02-01-09.pdf, Object Management Group, August 2002

[Berkenkötter, 2003] Berkenkötter K, Using UML 2.0 in Real-Time Development: A Critical Review, Research paper, SVERTS, workshop hold in conjunction with UML, October 2003

[Björkander, 2003] Björkander M, Kobryn C, Architecting Systems with UML 2.0, Research paper, Published by the IEEE Computer Society, Telelogic AB, August 2003

[Booch, 1993] Booch G, Object-oriented Analysis and Design with Applications, 2nd edition. Benjamin Cummings, ISBN 0-8053-5340-2, 1993

[Conallen, 1999] Conallen J, Modeling Web Application Architectures with UML, White paper, Rational Software, June 1999

[CORBA, 2004] Common Object Requested Broker Architecture, The official homepage of CORBA, www.corba.org, Object Management Group, Lastly accessed May 9, 2004

[CWM, 2003] Common Warehouse Metamodel, Specification v1.1, Document - formal/2003-03-02, www.omg.org/docs/formal/03-03-02.pdf, Object Management Group, March 2003

[Evans, 2003] Evans A, Maskeri G, Sammut P, Willans J, Building families of languages for model-driven system development, Research paper, Workshop in Software Model Engineering, San Francisco, USA, October 2003

[Gurd, 2003] Gurd A, Using UML 2.0 to Solve Systems Engineering Problems, White Paper, Telelogic AB, July 2003

[Jacobson, 1992] Jacobson, I, Christerson M, Jonsson P, Övergaard G, Object-Oriented Software Engineering - A Use Case Driven Approach, Addison-Wesley, 1992

[J2EE, 2004] Java2 Enterprise Edition, The official homepage of J2EE, java.sun.com/j2ee, Sun Microsystems, Lastly accessed May 9, 2004

[Kaushik, 1997] Kaushik V, UML Glossary Version 1.0, swiki.hfbk-hamburg.de:8888/MusicTechnology/24, 1997

[KC, 2002] Supporting the Model Driven Architecture with Executable UML Ref: CTN 80 V2.2, www.kc.com, 2002

[KC, 2004] Kennedy Carter, The official homepage of iUML and i-CCG, www.kc.com/products, Kennedy Carter Inc, Lastly accessed: May 17, 2004

[Klasse, 2004] Klasse Objecten, www.klasse.nl/english/mda/mda-introduction.html, Klasse Objecten Inc, Lastly accessed: May 15, 2004

[Kleppe, 1998] Kleppe A, Warmer J, The Object Constraint Language: Precise Modeling with UML, Addison-Wesley, 1998

[Kleppe, 2003] Kleppe A, Warmer J, Bast W, MDA Explained - The Model Driven Architecture: Practice and Promise, Addison-Wesley, 2003

[Latta, 2001] Latta M, Tronstad Y, Executable Enterprise Modeling with UML Presented in OMG Workshop on UML in the Enterprise, www.omg.org/news/meetings/workshops/presentations/uml2001_presentations/03-2_Latta-Executable_Enterprise_Modeling_with_UML.pdf, December 2001

[Ledang, 2001] Ledang H, Souquieres J, Formalizing UML Behavioral Diagrams with B, Research paper, Proceedings of Tenth OOPSLA Workshop on Behavioral Semantics: Back to Basics, Tampa Bay, Florida, USA, www.loria.fr/~souquier/publications/oopsla01.pdf, October 2001

[Li, 2000] Li J, Chen J, Chen P, Modeling Web Application Architecture with UML, Research paper, 36th International Conference on Technology of Object–Oriented Languages and Systems (TOOLS-Asia'00), November 2000

[MDATech, 2001] Model Driven Architecture: A Technical Perspective, Architecture Board MDA Drafting Team, Document Number ab/2001-02-04, xml.coverpages.org/OMG-tech-01-02-04.pdf, Object Management Group, February 2001

[MDA, 2004] Model Driven Architecture, The official homepage of MDA, www.omg.org/mda, Object Management Group, 2004

[Mellor, 2002] Mellor S, Balcer M, Executable UML: A Foundation for Model Driven Architecture, Addison-Wesley, 2002

[Mikkonen, 2002] Mikkonen T, Katara M, Refinements and Aspects in UML, Research paper, Second International Workshop on aspect oriented modeling with UML, Dresden Germany, September 2002

[MOF, 2003] Meta Object Facility, Document - formal/02-04-03, www.omg.org/docs/formal/02-04-03.pdf, Object Management Group, 2003

[Mosses, 1992] Mosses P, Action Semantics, Number 26 in Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1992

[Mubin, 2004] Mubin A, Constructing MDA-based Application using Rational XDE for .NET, March 2004

[Net, 2004] Microsoft .NET Framework, The official homepage of Microsoft .NET Framework, www.microsoft.com/net, Microsoft Corporation, Lastly accessed: May 9, 2004

[OAL, 2003] Object Action Language, Object Action Language Manual V 1.4, www.projtech.com/pdfs/bp/oal.pdf, Projtech Inc, 2003

[OCL, 1997] Object Constraint Language, OCL 1.1 specification document, umlcenter.visual-paradigm.com/umlresources/obje_11.pdf, Object Management Group, September 1997

[OCL, 2003] Object Constraint Language, OCL 1.6 specification document ad/2003-01-07, www.omg.org/docs/ad/03-01-07.pdf, Object Management Group, January 2003

[OMG, 2004] Object Management Group, The official homepage of OMG, www.omg.org, Lastly accessed: May 17, 2004

[Poole, 2001] Poole J, Model-Driven Architecture: Vision, Standards and Emerging Technologies, Position Paper Submitted to ECOOP 2001: Workshop on Metamodeling and Adaptive Object Models, April 2001

[Projtech, 2004] The official homepage of Project Technology Inc, www.projtech.com, Lastly accessed: May 14, 2004

[Rhapsody, 2004] I-Logix Rhapsody, The official homepage of I-Logix Rhapsody, www.ilogix.com/rhapsody/rhapsody.cfm, I_Logix Inc, Lastly accessed: May 17, 2004

[Robinson, 2003] Robinson P, Extreme Architecture: Amplifying the UML, presented at WA Oracle User Conference, www.extremearchitecture.org, 2003

[Rumbaugh, 1991] Rumbaugh J, Blaha M, Premerlani W, Eddy F, Lorensen W, Object-Oriented Modeling and Design, Prentice-Hall, 1991

[Rumpe, 2002] Rumpe B, Executable Modeling with UML - A Vision or a Nightmare? Research paper, Issues & Trends of Information Technology Management in Contemporary Associations, Seattle, Idea Group Publishing, Hershey, London,
pp. 697-701, 2002

[SDL, 2004] Specification and Description Language, www.sdl-forum.org, SDL Forum Society, Lastly accessed: May 13, 2004

[Siegel, 2001] Siegel J, Developing in OMG's Model-Driven Architecture: White Paper Revision 2.6, Object Management Group, November 2001

[TAU, 2004] Telelogic TAU, www.telelogic.com/products/tau/index2.cfm, Telelogic AB, Lastly accessed: May 13, 2004
[UML, 2003] Unified Modeling Language, UML 1.5 specification document: formal/03-03-01, www.omg.org/docs/formal/03-03-01.pdf, Object Management Group, March 2003

[UML, 2004] Unified Modeling Language, The official homepage of UML, www.uml.org, Object Management Group, Lastly accessed: May 17, 2004

[Wilkie, 2001] Wilkie I, King A, Clarke M, Weaver C, Rastrick C, UML ASL Reference Guide, ASL Language 2.5, Manual revision B, ccc.inaoep.mx/~labvision/doo/xuml/ASL_Reference_Guide.pdf, Kennedy Carter Ltd, 2001

[XMI, 2003] XML Metadata Interchange, Specification document v2.0 - formal/03-05-02, www.omg.org/docs/formal/03-05-02.pdf, May 2003

[Zakaria, 2002] Zakaria A, Hosny H, Zeid A, A UML Extension for Modeling Aspect-Oriented Systems, Research paper, Second International Workshop on aspect oriented modeling with UML, Dresden Germany, September 2002

Appendix A
Investigation of Tools

This appendix provides a survey of some particular modeling language-supporting tools. This section investigates each tool according to its vendor claims for some derived qualities which are important from the perspective of MDA and show them in the form of traceability matrix.

A.1. The Functionalities

Following are the functionalities, for which each tool was investigated:

- Project partitioning, it denotes to the ability of the tool to support the large model management.
- Model verification, it denotes to the ability of the tool to test the models before going to implementation phase.
- Translation, it denotes to the ability of the tool to execute the models directly into the code.
- Multi-user & network support, it denotes to the ability of the tool to support more than one user over the network.
- Central repository, it denotes to the ability of the tool to provide the features such as databases, version management, saving models and code etc.
- Testing and debugging, it denotes to the ability of the tool to test and debug the code.
- Supported platforms, it denotes to the ability of the tool to support major platforms such as Windows, Solaris, Linux etc.
- Online help, it denotes to the ability of the tool to provide help over the internet.
- Third party products integration, it denotes to the ability of the tool to provide integration with other tools such as requirements management (Telelogic DOORS), configuration management (PVCS Version Manager) and development environments (IBM VisualAge, Microsoft Visual Studio) etc.
- Supported languages, it denotes to the ability of the tool to provide code in more than one language such as C++, Java, Ada, etc

A.2. Unified Modeling Language 1.x

For UML 1.x, following two supporting tools were evaluated for their provided functionalities:

- IBM Rational Rose Enterprise (www.rational.com)
- Telelogic TAU Generation 1 (www.telelogic.com)

They provide following functionalities:

Functionality	TAU	Rose
Project partitioning	No	No
Models verification	No	No
Translation	No	No
Multi-user & network support	Yes	Yes
Central repository	Yes	Yes
Testing & debugging	Yes	Yes
Supported platforms	Windows	Windows
Online help	Yes	Yes
Third party products integration	Yes	Yes
Supported languages	C++, Ada, Java, Smalltalk	Java, C++, VB, Ada

Tabel 2 A capability matrix of UML 1.x supporting tools

A.3. Executable and Translatable UML

For $^{X}_{T}$UML, following two supporting tools were evaluated for their provided functionalities:

- Project Technology BridgePoint (www.projtech.com)
- Kennedy Carter iUML (www.kc.com)

They provide following functionalities:

Functionality	BridgePoint	iUML
Project partitioning	Yes	Yes
Models verification	Yes	Yes
Translation	Yes	Yes
Multi-user & Network support	Yes	Yes
Central repository	Yes	Yes
Testing & debugging	Yes	Yes
Supported platforms	Windows, Solaris	Windows
Online help	Yes	Yes
Third party products integration	Yes	Yes
Supported languages	C, C++	C, C++, Java

Tabel 3 A capability matrix of $^{X}_{T}$UML supporting tools

A.4. Unified Modeling Langauge 2.0

For UML 2.0, following ten supporting tools were evaluated for their provided functionalities:

- Telelogic TAU Generation 2 (www.telelogic.com)
- ILogix Rhapsody (www.ilogix.com)
- Rational Rose Technical Developer (www.rational.com)
- Interactive Objects ArcStyler (www.io-software.com)
- Pathfinder Solutions PathMATE (www.pathfindermda.com)
- ObjectFrontier FrontierSuite (www.objectfrontier.com)
- InnoQ iQgen (www.innoq.com)
- Artisan Software Real Time Studio (www.artisansw.com)
- Codagen Architect (www.codagen.com)
- MIA Software Generation (www.mia-software.com)

They provide following functionalities:

Functionality	**TAU**	**Rhapsody**	**Rose**	**ArcStyler**	**PathMATE**
Project partitioning	Yes	Yes	Yes	Yes	Yes
Models verification	Yes	Yes	Yes	Yes	Yes
Translation	Yes	Yes	Yes	Yes	Yes
Multi-user & Network support	Yes	Yes	Yes	Yes	Yes
Central repository	Yes	Yes	Yes	Yes	Yes
Testing & debugging	Yes	Yes	Yes	Yes	Yes
Supported platforms	Windows	Windows	Windows, Linux, Solaris	Windows, Linux	Windows
Online help	Yes	Yes	Yes	Yes	Yes
Third party products integration	Yes	Yes	Yes	Yes	Yes
Supported languages	C, C++, Java	C, C++, Ada	C, C++, Java, Ada	Java, C#	C, C++, Java

Tabel 4a A capability matrix of UML 2.0 supporting tools

Functionality	FrontierSuite	iQgen	Real Time Studio	Architect	Generation
Project partitioning	Yes	Yes	Yes	Yes	Yes
Models verification	Yes	Yes	Yes	Yes	Yes
Translation	Yes	Yes	Yes	Yes	Yes
Multi-user & Network support	Yes	Yes	Yes	Yes	Yes
Central repository	Yes	Yes	Yes	Yes	Yes
Testing & debugging	Yes	Yes	Yes	Yes	Yes
Supported platforms	Windows, Solaris, Linux	Windows, Linux, Mac OS	Windows	Windows	Windows
Online help	Yes	Yes	Yes	Yes	Yes
Third party products integration	Yes	Yes	Yes	Yes	Yes
Supported languages	Java	Java, C, C++, C#	C, C++, Java, Ada	Java, C++, VB, C#	Java

Tabel 4b A capability matrix of UML 2.0 supporting tools