Model-Driven Architecture, the revolution of software engineering

Giovanni Piemontese, Guido Diodato
{gpe08001, gdo08001}@student.mdh.se
Università degli Studi dell’Aquila

October 30, 2008

Abstract

Nowadays, Quality of Services (QoS) attributes are often basic for the success of software application but, for taking into account QoS attributes are more costly with traditional software development. Model-Driven Architecture (MDA) is an approach for moving the development focus from programming languages code to models. This paper shows how model-driven architecture may be used to do testing and system performance. The UML 2.0 Testing Profile is considered in this paper because it provides support for UML 2.0 based model-driven testing. Moreover the paper introduces a methodology of how to use the profile to transform system design model to test design model and how to extend MDA to enable performance validation.

Keywords

Model-Driven Architecture, MOF, UML Testing Profile, Model-Driven Testing, Software Engineering, System Performance, Model-Driven Engineering

1 Introduction

Today’s software systems are complex principally due to the inherent complexity of the domain in which the software solutions are used. This complexity impacts on all the software development process and makes more complex the project, the testing, the analysis, deployment and the implementation code. The technologies for analysis, architect and implement software grows with the same velocity of the complexity so competition in the software industry leads to a steady stream of new technologies[12].

This shift among different technologies also impact on the develop process. The Model-Driven Architecture (MDA)[2] is born for resolving this problem. The MDA is a new way to software development proposed by the Object Management Group (OMG), whose main goal is to provide a way that permits, by means of model, to abstract the platform concepts. Many studies have been done in this side and we refer to [13] for an overview of these.

But the Model-Driven Architecture is not only about system modelling throughout the abstraction levels in terms of platform independent system modelling. The MDA abstraction levels can also be applied to analysis and test modelling.

This paper focuses quite in this direction and shows the state of art in testing and non-functional analy-
sis “driven” by MDA. In fact MDA, being born for software development, has to be applied to all the deployment phases in order to be useful, but often the non-functional requirement and the early test-phase are not taken into account. Goal of this paper is to present the main concepts of MDA and show how these concepts can be embedded in the software engineering process in order to perform the analysis[7] and testing phases[9] showing two existent approaches.

In this paper we also introduce same concepts from testing and performance analysis domains like the UML 2.0 testing profile [1], the UML profile for Schedulability, Performance and Time [6], the Queueing Network[11] and the Lyered Queueing Network [10] and we refer at the references material for a careful trattation.

This paper is organized as follows: in section 2 we give an overview of the MDA standard, in section 3 we show an instance of MDA for performing testing while in section 4 we show how the MDA process can be extended to perform non-functional validation. Finally in section 5 we give the conclusion.

2 Model-Driven Architecture

The Model Driven Architecture (MDA) is an approach that has its ground upon other technologies: Unified modelling Language(UML)[5], Meta Object Facility (MOF)[4], and other standard defined by the Object Management Group OMG. The core idea of the approach, is to move the development focus from programming languages code to models and to generate the implementation automatically from these models. The aim is to increase the development productivity and quality of the software system under development. But is this enough? We think no of course, indeed the complexity of the software move from the code to the UML models, so MDA cannot have the only goal to generate the implementation but have to support all the software engineering phases.

How we have mentioned above, the main goal of MDA is to provide a solution to the problem of continual challenges between companies that forces these to move, among platform, their software systems every time a new better technology appears. Now, we can give a look at the basic concept of MDA, OMG defines MDA as an approach to system development based on models. It is said to be model-driven because it provides means for using models to direct the course of understanding, design, construction, deployment, maintenance and modification.

MDA approach classifies models into three layers: Computational Independent Models (CIMs), Platform Independent Models (PIMs) and Platform Specific Models (PSMs). The CIM layer focuses on context and requirements, often we refer this model as Business or Domain model because “talks” with the same language of the business domain. The PIM layer gives the operational capability of the software in a way that abstracts the platform technical details. Finally the PSM layer extends the PIM with details about the specific platform.

Now let’s give a look to how the MDA process works and how we can combine the models above for constructing an MDA-based application. The two key concepts of MDA are models and transformations, we have just seen the models so we came to look the transformations. MDA Process is supported by only one kind of transformation: the vertical transformation that permits to switch from one model to another, this transformation is made possible by a transformation mapping that specifies, for instance, how to transform a PIM into a particular PSM. Another instrument for facilitating the transformation is the marking model which consists of a set of annotations that represent concept in the PSM and are applied to the PIM elements in order to indicate how these elements have to be transformed.

MDA promotes itself as solution of many problem, among them:

- Technology obsolescence and portability: how we have widely seen.
- Interoperability: through the use of models.
- Productivity and time-to-market: The maintenance and understanding of code is a difficult
and error-prone process in case of large software systems. So with a modular the understanding increases.

- Testing and simulation: model can be validated and the models can be also used for simulations.

MDA, has we have mentioned before, also aims at integrating the existing technologies standardized by OMG. Models are expressed in Unified Modelling Language (UML) and UML profiles and if another modelling language is used it should be defined using the meta-modelling language of the Meta Object Facility (MOF).

3 Model-Driven Testing

The Model-Driven Architecture is not only the system modelling throughout the abstraction levels, but it can also be applied to test modelling. The Unified Modelling Language (UML)[5] is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. UML itself provides no means to describe a test model, but with the growing system complexity the need for solid testing increases. Besides, to obtain it OMG has defined UML 2.0 profile for the testing called UML 2.0 Testing Profile (U2TP)[1]. This profile for testing is necessary to bridge the gap between designers and testers because they use different languages and tools, making difficult the communication. The UML 2.0 Testing Profile provides support for UML 2.0 based model-driven testing. The U2TP is a test modeling language that can be used with all major objects and component technologies and applied to testing systems in various application domains. The UML Testing Profile can be used stand alone for the handling of test artifacts or in an integrated manner with UML for a handling of system and test artifacts together. The U2TP provides concepts covering: test architecture, test behavior, test data and time. The set of the concepts that cover the test architecture are:

- Test context: it is a collection of test cases and allows the user to describe a corresponding test configuration (defines both test component objects and connections when a test case is started and the maximal number of test component objects and connections during the test execution) and to define the test control (execution order of test cases).

- Test component: it defines the behavior of test cases and the communication with the SUT (System Under Test) or other components.

- SUT: is a part or whole system being tested.

- Scheduler: a property of a test context used to control the execution of the different test components.

- Arbiter: a property to evaluate the result of test case and to assign a verdict (Pass, Fail, Inconc and Error).

In addition to the UML 2.0 behavioral concepts, the UML 2.0 Testing Profile specifies other concepts. The Test Behavior package describes the concepts required to specify test cases and their associated behaviors. These concepts are the follows:

- Test objective: allowing the designer to express what should be tested.

- Test case: is a specification of one case to test the system, including what to test with which input, result, and under which conditions. It is a complete technical specification of how the SUT should be tested.

- Verdicts: is the assessment of the correctness of the SUT. The possible test results are: pass, inconclusive, fail and error.

- Validation action: is an action performed by a test component to assess a test observations and/or additional characteristics/parameters.

- Log action: an action to log information in the test log.
• Test log: is an interaction resulting from the execution of a test case. It represents the different messages exchanged between the test components and the SUT.

Test data refers to the specification of types and values that are received from or sent to the SUT. Data can be specified as being static or dynamic. The U2TP adds these concepts:

• Wildcard: is very useful for the handling of unexpected events, or events containing many different values.

• Data pool: is a collection of data partitions or explicit values that are used by a test context, or test components, during the evaluation of test contexts and test cases.

• Data partition: a logical value for a parameter used in a stimulus or in an observation.

• Data selector: an operation that defines how data values or equivalence classes are selected from a data pool or data partition.

• Coding Rules: allowing the user to define the encoding and decoding of test data during the communication with the SUT.

Finally, timing concepts are provided to complete the concepts needed for completing test modeling. Two new concepts added are:

• Timers: are mechanisms that may generate a timeout event when a specified time value occurs.

• Time zones: is a grouping mechanism for test components. Each test component belongs to a certain time zone. Test components in the same time zone have the same time.

In Model-Driven Architecture there are three abstraction levels: platform independent model (PIM), platform specific model (PSM) and System Code. This distinction can be applied also to the test models which can be specified as platform independent test (PIT), platform specific test (PST) and test code. We will show how the UML 2.0 Testing Profile can be applied to obtain these three testing abstraction levels starting by the other three modeling abstraction levels through model-transformation. Before it, we want show a background of the Model-Driven Testing and explain better the distinction among the abstraction levels and the transformation applied. A model transformation is “the process of converting one model to another model of the same system”. Kleppe et al. [3] defines a transformation as the automatic generation of a target model from a source model, according to a transformation definition. As shown in Figure 1, platform independent model (PIM) can be transformed into platform specific model (PSM). The PIM describes the system at abstraction level very high, instead the PSM contains a lot of informations on the system. Later on, the PSM can be transformed into system code and the completeness of this depends on the completeness of the PSM.

![Figure 1: System Design Models vs. Test Design Models](image-url)

The same transformations can be made to test design models. The platform independent test model (PIT) can be derived by PIM through a model transformation and the same thing for the platform specific test model (PST). A test code can be obtained directly
from platform independent test model (PIT) or platform specific test model (PST).

Now, we show how use UML 2.0 Testing Profile as support to Model-Driven Testing. Having a system model design, PIM or PSM, a tester might specify tests for the system and this can be done easier by extending the system design model with U2TP concepts presented before. The first thing to do is to define a new UML package as test package and import all information, such as classes and interfaces, from system model design package. After that, the tester defines the specification of the test architecture and test behavior (test data and time are already included in test architecture). Both test architecture and test behavior are subdivided in two categories: mandatory and optional issues. The mandatory issues are essential for the test with U2TP. In test architecture there are two important mandatory issues to do:

- Assign the classes or objects you would like to test to SUT class/object.
- Specify a test context class where there are a list of test cases and test attributes and possible a test configuration.

Instead, test behavior has these mandatory issues essentially to define a test package:

- For the specification of test cases, take given Interaction Diagrams from the system design model. Change the instances and assign them the stereotypes according to their roles.
- Assign the verdicts at the end of each test case specification. The default value is set to pass.

The follow figure (Figure 2) shows the meta-model based transformation for the test design model transformation. The source model is UML meta-model and the target model is the U2TP meta-model.

To define a model-transformation, as described in [9], we need a transformation rules based on Test Directive meta-model. In this transformation rules we have to define those mechanisms to provide the tester a mean to apply a model transformation.

Figure 2: Meta-Model Based Transformation

All three meta-models are based on MOF. Transformation rules are used to describe a correspondence between patterns of elements in the source model and the elements in a target model which have to be created. These rules are defined according to the QVT (Query/View/Transformation) specification from CBOP/IBM/DSCT [4].

4 Software Performance Model-Driven Architecture

Performance Analysis, traditionally, is the process looking bottleneck and performance problem at program runtime. With the exponential growth in processor performance, the performance analysis was no longer first-class entity. The classical performance analysis was more costly, indeed the analysis was made at execution time and a discovery of software bottleneck in this stage implicates the modification of all the precedent phases.

Today’s softwares are more often based on mobility device with limited computational capacity, so the performance analysis becomes again first-class entity. In this optical, the integration of non-functional activities to take into account QoS attributes, from the beginning of the software deployment process, became fundamental. In fact, addressing performance
issues as early as possible permits to reduce design failure and reduces the necessary process backtracking.

In this section we show a framework, presented in [7], that embeds software performance analysis in the MDA process in order to automatically generate performance models in the early stage of software deployment. Moreover we present, in the next subsection, an approach, SAP-ONE [8], that implements this methodology.

![MDA Diagram]

**Figure 3: Software Performance Model-Driven Architecture**

How we have mentioned in the introduction section, the typical three-steps structure of MDA are not able to take into account the non-functional requirements. For this reason, the authors in [7], have proposed an extension of MDA named SPMDA (Software Performance Model-Driven Architecture) (see Figure 3) that defines three new MDA parallel layer and transformation/relationship for managing performance issues. These three layers are:

- The CIPM layer which is the CPM parallel layer and in the same manner of the last one, represents the system requirements through use-case model, but in CIPM context the use-case are annotated with performance requirements that the system has to meet.

- The PIPM is the PIM parallel model and have to be computationally and platform independent. This model, being platform independent, can't give indices that match the requirements value because they could take very different value on different platforms. In this level, the indices of performance are no-measurable unit. Moreover the early performance indices can be utilized for verifying if the business architecture has some problem on performance and for finding bottleneck component or high level resource with unexpected utilization. Moreover the PIPM can be used for obtaining upper and lower bounds and for comparing alternative PIM architectures, so that, we can choose the best one.

- Finally, there is the PSPM layer that merges the PIPM model upon the platform adopted to run the logic. In the PSPM model, differently from the PSM, the platform is not only a set of interface and artefacts but also embeds low level hardware which is indispensable for the performance analysis.

The main feature introduced with SPMDA is the concept of horizontal transformation which permits, through model-transformation, to generate performance models from architectural models. This process is called 2-steps process, indeed, at first the models have to be annotated and only after, the annotated models, are transformed in a performance model.

The main quality of this framework is that the user has not to know about how the transformation are implemented by the framework and neither what kind of models the framework utilizes for the performance analysis, anyhow for the completeness of the discussion we come to introduce a methodology that implements the SPMDA framework: SAP-ONE introduced in [7-8]. But after introduce the SAP-ONE approach we have to introduce the Queue Network
System and the Layered Queue Network, in fact these two are respectively the PIPM model and the PSPM model in SAP-ONE methodology.

Queuing Networks (QNs)[11] are characterized by three main components: the service center, the customer and the queue. The customers make requests to a service center, the service center, usually, can process only a request in a determined time, so the other requests have to wait in the service center queue and they are going to be processed in accordance with the queue policies (FIFO, LIFO and so on). The multi-chain QN adds to QN model the capacity to distinguish a number of different classes of users which have different behaviors in the interaction with the system. Instead the Layered Queue Networks (LQNs)[10] differs by the QN because they take into account contention for software servers and devices and models both software tasks and devices. Moreover the LQN has a hierarchical composition, so the components are organized as client-server.

4.1 The SAP-ONE Methodology

The SAP-ONE methodology finds pre-determined pattern in the software models and replace it, in the performance model, with the corresponding performance model pattern. Prerequisite of SAP-ONE approach is to model the requirements through use-case UML 2.0 diagram, the software architecture through component and sequence diagrams and the software-to-hardware mapping through deployment diagram. All these diagrams are annotated through stereotype defined in
the UML profile for Schedulability, Performance and Time (UML SPT)[6].

The UML STP, in a similar manner to UTP, defines stereotype for taking into account the performance characteristics. We have the Scenario stereotype which define a response path of the system. The Scenario is composed by various Steps that define performance specifications (workload intensity, resource usage and so on). Now, we can give a look to how the SAP-ONE approach works.

The Platform Independent Performance Model, how we have said before, is performed by a multi-chain QN model. SAP-ONE generates a service center for each component (that is annotated with the stereotype < <<P Ahost>> ) in the PIM component model and the approach associates the QN customer with the service demand provided for each components. Indeed the user has to provide also the scheduling policy for the “queue” of each components and the follow data for the Sequence diagram: workload for each sequence, probability on the branch points and average number of loop iterations.

The <<PAstep>> stereotype is used upon the provided service in the component diagram, for indicating the service demand and in the sequence diagram to annotate the probability in alternative system behaviours.

The software annotations are expressed in high-level units of measure, lower and upper bound analysis, software bottleneck identification and comparison of alternative software architectures can be carried out basing on this analysis results.

The Platform Specific Performance Model is performed by Layered Queuing Networks. The SAP-ONE approach, starting from the PIPM model, generates a task for each service center in the PIPM model, instead from the PSM the SAP-ONE takes new tasks according to the platform utilized (these new tasks are placed in accord with the code which is visited by the approach), finally LQN devices are extracted by the Deployment diagram.

At this level many different types of performance analysis can be done, in fact, the LQN model made by the SAP-ONE approach has all the software and hardware parameters need for a careful performance analysis like: end-to-end response time, utilization and through put of any platform device. So, solving the LQN model, we can compare the performance indices with those on the CIPM in order to validate the software. Moreover, if we execute the measurements by PSPM, we can take into account workload domain that are unfeasible and we can study the system under different workloads, resources and platforms only by changing some annotations and/or parameters in the PSM model.

The outputs of the PSPM evaluation represent the feedback on the PSM, which needs to be modified if performance does not satisfy the requirements.

5 Conclusions

In this paper we have tried to move the attention from the “crude” implementation to the analysis and testing. These two, how we have widely shown during all the paper, are fundamental for quality and costs, indeed the discovery of bug and failure has a cost often much expensive when the deployment process comes in late phase. In non-functional analysis, we have the same situation, often mismatch in non-functional requirement is due to wrong implementation choices. Indeed, it is well known that in both analysis and testing, when these processes begin in the end of the deployment process, they become more costly and are the first budget “cut”.

In the last years, the main reason why non-functional analysis is rarely taken into account in the deployment process is because it requires high skills and specific models.

The MDA process has given a big opportunity to testing and non-functional analysis, indeed, how we have seen in sections three and four, the MDA supplies, to the developer, a easy way to perform these analyses, and specially the developer couldn’t have knowledge in depth of the test and performance domain. This work has tried to link two collateral deployment process edges and we have tried to show how they can go with the MDA in a parallel way with the deployment process.
References


