QoS-aware MDA

Arnor Solberg, Knut Eilif Husa, Jan Øyvind Aagedal, Espen Abrahamsen
SINTEF, Ericsson, Simula Research laboratory
Arnor.Solberg|Jan.aagedal@sintef.no, knutehu|espabrah@ifi.uio.no

Abstract

System developers can often perceive managing QoS-requirements as complex, time consuming and abstract. As a result of not paying sufficient attention to QoS in the design phase, problems are discovered late in the development cycle, often not before the real system is set into operation. Thus, correcting QoS deficiencies often become very expensive and difficult to handle. One of the main reasons why QoS is difficult to treat in models is that there is a lack of precise relations between different models and between models and implementations with respect to QoS. Thereby, we believe the capturing and handling of QoS-requirements has to be an essential part of the MDA approach in order to gain successful system development of real time and embedded systems. Thus, the MDA approach needs to be QoS-aware. Consequently, the QoS-specifications need to be part of the PIM and PSM models, the accomplishment of model transformation code generation, configuration and deployment should be QoS-aware and ideally also the target execution platform should be QoS-aware. This paper describes some significant aspects of a QoS-aware MDA approach for system development and execution.

Introduction

OMG's Model Driven Architecture (MDA) [1] announces a framework for supporting a full lifecycle model driven development approach. *Model driven development* denotes a software systems development approach where the systems are developed by constructing abstract models of the systems from which the executable code are automatically or manually derived. Through its kernel of UML [2], MOF [3] and CWM [4] one aim is to be able to relate the set of models as well as the code and to perform model transformations and code generation, specifying different views and residing at different abstraction levels

The concepts of platform independent and platform specific models (PIMs and PSMs) are fundamental in the MDA. Thus, having specified the application logic in a platform independent model (PIM), the challenge remains to produce a platform specific model (PSM) that utilizes the platform capabilities in an optimal manner relative to the expectations of the users.

The PIM and PSM models currently tend to focus on how the system should realize the desired functionality. However, systems have additional properties that characterize the systems' ability to exist in different environments and adapt as its environment varies. These extra-functional properties, also called qualities or quality of service (QoS), address how well this functionality is (or should be) performed if it is realized. In other words, if an observable effect of the system doing something can be quantified (implying that there is more than 'done'/'not-done' effect of the behavior), one can describe the quality of that behavior. The quality requirements should be specified and these requirements should be taken into consideration during the system design as to ensure delivering design models capable of meeting both the functional and non-functional requirements.

There is a shortage of methodology for model-based development that cover complete and precise specification of QoS-requirements, such as for instance availability of computer systems which cannot be so easily captured in models. However, in many kinds of systems the consequences of QoS-failures are severe, e.g., it might be extremely important to make sure that availability requirements are implemented as specified. Examples of such system areas are: health, traffic control and banking.

This position paper discusses significant aspects for a QoS-aware MDA-based system development approach; including QoS-specification, QoS-aware model transformation and QoS-aware system deployment and execution. For the QoS deployment and execution part we refer to an execution platform called the QuA platform [5], which is currently under development as part of a research project called QoS-aware component architecture (QuA) [6].

Specifying QoS

In order to guarantee some level of QoS, the appropriate QoS-characteristics must be identified and specified. QoS offered by a component should be specified in such manner that the QoS-requirements can easily be distinguished from the specification of its functional properties. This is appropriate since: (i) the QoS of a component is dependent on its environment, (ii) there may be multiple QoS-specifications for the component, and (iii) the functional specifications of the component are independent of the particular QoS-specification that will be enforced at any point in time. This is an orthogonal separation of concerns between behavior (functional properties) and constraints (the QoS it offers).

It is important to note that it must be possible to associate multiple models of QoS with the same functionality. For example, playing a video over either 56Kbps (modem), 1Mbps (DSL or cable modem), or 100Mbps (Ethernet) Internet connections will give different levels of QoS.

The UML Profile for Schedulability, Performance, and Time Specification [7] allows for decorating UML models with QoS-requirements. Based on the meta model of the profile, the appropriate information can be extracted from the model e.g., using some kind of XML tool.

Models of system functionality typically represent service offers that system components can provide, e.g., abstractions of component interfaces and component interactions. For functional specifications, this is sufficient since component behavior can be objectively defined, and an architectural model combines services to a complete functional system specification, and the consistency is typically verified by type matching. For QoS, however, one cannot merely specify the QoS offered by a component, one needs also a way to specify client satisfaction. Specifications of client satisfaction are often done by worth functions (sometimes referred to as utility or benefit functions) that evaluate a QoS-offer relative to benefit of a given client and return a worth value. QoS-aware service configuration then involves both traditional type matching and QoS-optimization using the worth functions of the QoS-offers.

QoS-characteristic might be seen as the most fundamental term required to specify QoS. A QoS-characteristic represents some aspect of the QoS of a system service or resource that can be identified and quantified, for instance time delay or availability. In the ISO QoS Framework [8], a number of QoS-characteristics are defined such as delay, throughput, etc. ISO/IEC 9126 [9] is another standard in the quality domain. It provides a consistent terminology of different aspects of quality and identifies six quality characteristics that are further subdivided into sub characteristics. The six main characteristics are functionality, reliability, usability, efficiency, maintainability and portability. Despite the lists of QoS-characteristics in standards, there is no agreed upon and exhaustive list of QoS-characteristics. In the revised submission of the UML profile for modeling QoS [10], a QoS-catalogue is defined based on ISO/IEC 9126, but it is made clear that this should not be considered as the final and exhaustive list of QoS-characteristics. Indeed, the individual modelers are free to define their own QoS-characteristics, tailored to their application, but they should then be aware the problems of interoperability with using self-defined characteristics.

A *QoS-constraint* is used to limit a set of QoS-characteristics for them to collectively represent (parts of) the QoS of a component. A QoS-constraint could for instance limit the QoS-characteristic *delay* to be less than 5ms.

QoS-constraints are combined to define *QoS-levels*. QoS-levels relate a number of QoS-constraints to system components and their environment. QoS-levels model the fact that, in general, QoS provided by a component can also be a function of how the environment performs. For example, the QoS of a multimedia presentation can depend on the QoS of a video storage service and an associated audio storage service. This dependency should be captured in an assumption/guarantee-style specification of QoS-levels, so that a QoS-level specification defines QoS-offers that may depend on QoS-offers of other services.

Some QoS-characteristics have strong interrelationship. Take for instance the availability of a service and its provided performance. Typically the higher performance constraints that are posed on the service the lower availability you will experience. It is important to be aware that it might be difficult (and very expensive) to meet QoS-constraints if the characteristics are of such nature that rising one implicates lowering the other.

QoS-requirements should in many cases preferably be associated to instances rather than to classes or types. With the introduction of UML 2.0, new language constructs are available for representing instances of types (e.g., parts, ports) and links between them (connectors). Thus, using UML 2.0 the QoS-requirements can be more easily specified for instances within the different environments. New UML profiles for QoS-specifications that are based on UML 2.0 will benefit from having these new instance constructs.

Using QoS-specifications

A modeling language to support specification of QoS may be useful in a number of phases in the software life cycle, and may be used for different purposes by a number of actors in system development. The analyst may use it to specify extra-functional user requirements and to communicate with end users. The system developer may use it to specify QoS-offers and QoS-requirements of individual system components, and different kinds of analysis of the system design may also be performed (e.g., schedulability analysis, risk analysis, performance analysis, validation analysis). Some of the QoS-properties may be given to the application after it has been implemented as a matter of configuration (e.g., by “throwing resources at it”), while other properties require a more careful approach in which conscious design decisions are called for during the development phase. One of the main purposes of QoS-specifications is to be useful for the underlying infrastructure during system operation to maintain the desired level of QoS.

Systems may manage QoS in different ways. At one extreme, QoS-requirements can be met statically during design and implementation by proper design and configuration choices (such as scheduling rules, network bandwidth allocation, etc.). This will give a well-defined behavior, but without any flexibility. At the other end is the dynamic approach that lets the systems negotiate at run-time the restrictions of the QoS-characteristics they need for their activities. This approach often involves an adaptive aspect by having monitors and corrective operations to be taken when the QoS-level drops below a certain level. This approach is very flexible as the QoS-policies can be changed at run-time, but the behavior is not well defined a priori and the performance may be degraded if costly adaptation schemes are used.

For example, a full-screen, color, high-resolution multi-media stream can be modeled in terms of client preferences and the contribution of the environment and each component to providing the service. If something goes wrong, the action to be taken will be dependent upon what capability there is to: Fix the problem, Offer a lower-grade service, Accept a lower-grade service, Modify the application behavior

Thus, it may be possible to switch to a lower resolution, black and-white feed, or to go to a smaller screen size, or to wait a while and re-start, and so on. Any such decision is dependent upon the client preferences and on the capability for operational change.

QoS is generally pervasive across all system components, i.e., all system components contribute to the perceived QoS. This means that the underlying infrastructure, comprised of hardware, network, operating system, middleware and other generic system components, needs to be configured and managed in order to provide appropriate QoS. This QoS-management should be tailored based on the QoS-specifications and this can be achieved by model-transformations from QoS-specifications to standard QoS-management functionality.

QoS-aware model transformation and code generation

One of the key challenges of the model-driven paradigm is to define, manage, and maintain traces and relationships between different models and model views, including the code of the system. This again, is the foundation for performing model transformation and code generation. A model transformation is essentially to transform between two model spaces defined by their respective meta-models. Thus, transforming a PIM model to a PSM model is done by means of a generic transformation specification specifying how a meta-model concept of the source model should appear in the target model.

Typically in model-driven development processes, an extensive set of different interrelated models is developed. This may range from business models, requirements models, design models, code, and deployment models. An advanced MDA based framework should provide well-mannered support for modeling and interrelationship maintenance and to be able to automatically perform roundtrip model transformations as well as code generation.

A transition from for instance a PIM to a PSM covers more than a pure meta-model-based transformation of the concepts from the PIM to the PSM. To be useful, the transition process also needs to take into account and utilize the actual target technology. This includes for instance use of common patterns and use of standard mechanisms, e.g., to satisfy the required quality of services provided by the system. Thus, we believe utilization of QoS-specifications when performing model transformations and code generations are key to gain efficient and useful results.

At build time the complete requirements specification, including both the functional and extra functional requirements, establish the contract to be fulfilled by the system. To develop a useful design model, both the functional and extra functional requirements need to be considered. It is obviously not adequate to deliver the required services if the expected quality of the services (e.g., performance, security level, timeliness etc.) is far from being met. In a model driven development approach you might construct the requirements specification by developing a UML use case model including QoS-specifications associated with appropriate model elements. A model transformation of the use case model to derive an initial design model has to take into account the quality of service requirements to deliver an appropriate design. The quality of service requirements are essential for instance to be able to figure out the most appropriate design patterns to apply, as to satisfy the clients expectations with respect to both functionality and quality of the service. Thus, a QoS-aware model transformation is necessary to deliver pattern-based optimizations.

QoS-specifications should also be considered as key input in performing automatic code generation (code generation could in fact be regarded as a special kind of model transformation, as the code really is a model description in accordance to the actual programming language meta-model). Quality of service specifications delivers significant information as to be able to perform extensive code generation, and, more importantly, to deliver efficient code. The code generator might for instance use the QoS-statements to generate the appropriate code for handling the interaction with pervasive services available in the target platform (e.g., security services, resource trading services, message services). A majority of code generators today are template-based. For template-based code generators the basic idea is to use predefined templates as the vehicle for generating code. Processing the QoS-requirements will enable the code generator to choose the most efficient templates according the actual quality of service constraints.

QoS-aware deployment and execution

However, having a complete set of design models describing both the wanted functional and extra functional aspects both at the PIM and PSM level, and being able to perform QoS-aware model transformations and code generation an important aspect remain as to fully exploit the QoS-specifications. The QoS-specifications should also be utilized at deploy- and run-time. Systems perform in a dynamic environment and the system itself is dynamic, commonly having an almost infinite number of possible states. Thus, to meet the QoS-requirements component composition, configuration, instantiation, binding and execution should ideally be QoS-aware.

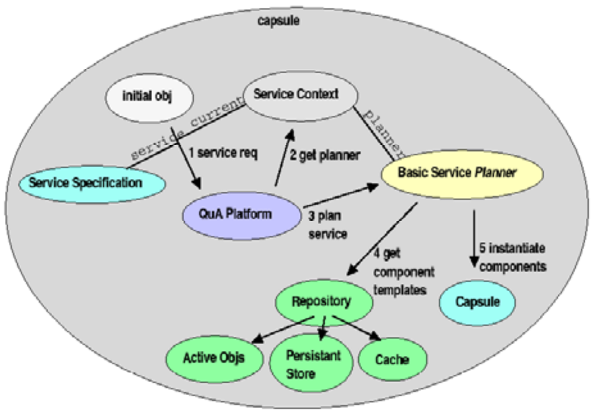

Figure 1 Collaboration diagram for QuA core object types

The QuA project [6] and the QuA platform [5] aims to define advanced QoS-awareness and deliver QoS-support throughout the system lifetime.

The QuA platform is a distributed virtual machine supporting execution of and communication between what is called QuA application objects. QuA objects use reflection to configure new application services and to reconfigure themselves. The fundamental protocol for configuring new application services is provided by a local meta-object representing the QuA platform.

The QuA platform is implemented by a network of *capsules,* each of which provide a local runtime platform for lower level implementation objects. Each capsule provides an object representing the QuA platform to support platform managed service instantiation and binding.

Figure 1 shows the current vision of the core object types in a *QuA capsule*. An initial application object acts as a client, sending a request to the singleton *QuAPlatform* to instantiate a service, which will constitute the rest of the application. The *QuAPlatform* object handles service configuration requests in a role analogous to that played by the ORB object in a CORBA middleware platform. The general form of a service request includes a *service specification, Quality specification* and *input guarantee*. The Service specification may be a composition that names a set of QuA objects, describes their initialization, references between them, and exported interfaces. A composition may describe any configuration of both new and existing objects. New objects are identified by type leaving the platform free to select an appropriate implementation.

The QuA platform satisfies service requests by invoking a service planner associated with the caller's service context. The *service context* holds meta information about the service within which the caller is currently executing. The default *BasicServicePlanner* that is built into every QuA capsule is only capable of component type resolution, instantiation and binding using services of the local *Repository* and *Capsule.* The client or other privileged agent may replace the service planner with a more sophisticated distributed and QoS-aware planner as needed. The service planner is responsible for discovering resources and implementation alternatives, and for planning the optimal configuration of the two to satisfy functional and QoS-requirements. Capsules are responsible for brokering local resource, loading software components and for manufacturing objects from object templates.

The key to enabling platform managed QoS, is the idea of a pluggable service planner, allowing to plug in specialized service planners according to the actual system and service requests. The chosen service planner encodes as much knowledge as needed to optimally satisfy the actual service requests.

The service planner uses the resource model in order to add and release resources in order to meet QoS-requirements. A *resource model* is a model of the resources in a computer system. It can be seen as an abstract representation of the real resources. A *resource metamodel* is a model of how resource models may look. It defines the underlying structure of resources and what information we need to keep about them. A *resource* is defined to be a usable system-entity, which is limited and subject to competition for its usage. A manager controls its usage.

We have used the General Resource Model (GRM) [7] as a basis for the resource model of the QuA platform and made some slight modifications on it. The current resource model consists of eight sub packages:

- The **CoreResourceModel** describes the division between *instance* and *descriptor*, as well resource *capacity.*
- The **ResourceTypesModel** is a model of the defined resource types.
- **ResourceStatusModel** is a model of the (usage-)*status* of resources.
- **ResourceReservationModel** describes how resource usage can be *reserved* by *tasks*.
- The **ResourceHierarchyModel** describes the hierarchic structure of synthetic resources being composed of other resources.
- The **ResourceCharacteristicsModel** describes different kinds of resource characteristics (e.g. how the capacity of a resource is measured).
- The **ExclusiveResourceModel** describes the structure of *exclusive resources*, and how they are acquired and released.
- The **ResourceManagementModel** is the model of how *resource managers* are associated with resources, and the functionality they offer.

All subpackages depend on the CoreResourceModel package, which contains the core resource concepts.

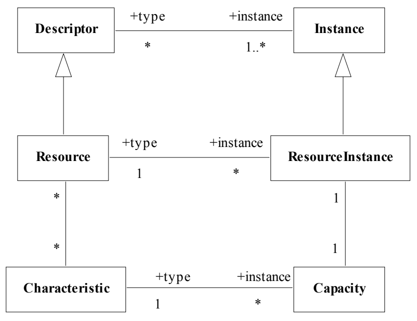

Figure 2 Core resource model

The *core resource model* is shown in *figure 2*. It is a slightly modified and simplified version of the core resource model in GRM. It makes a clear division between *descriptor* and *instance*, where an instance is the instantiation of descriptor. The *capacity* of a resource instance is some value that must comply with the characteristic of that instance. The *characteristic* is the specification of how something (in this case, the capacity) is measured.

Conclusion and further work

By using an MDA approach one will mainly carry out the development using models at different levels of abstraction and with different viewpoints. In this paper we have emphasized the importance of comprising QoS-specifications as part of the models and to support the QoS aspect within MDA frameworks. The introduction of UML 2.0 is a step forward in this respect. UML 2.0 introduces the concept of instance modeling, which is suitable for specifying QoS-requirements. Mechanisms and techniques supporting precise modeling of QoS and mechanisms for tracking and utilizing QoS specifications are essential as to gain the vision of MDA. In order to specify QoS-characteristics the current submission of the UML profile for

Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms RFP [11] look promising. We are involved in this standardization process and will continue to do research and contribute in this field.

The QuA project is developing a platform for QoS-aware execution and deployment. This platform will be able to execute services according to QoS-specifications given in the models. A key to enable QoS-aware execution is the notion of a service planner. The service planner is responsible for discovering and trading resources and deduce implementation alternatives, and to figure out the optimal configuration to satisfy functional and QoS requirements.

Two of the authors have just started on their PhD and the topics of the PhD's explore aspects of QoS-aware MDA. Some problem areas that will be further investigated as part of the QuA project as well as part of the PhD work are:

- QoS-specification, aiming to investigate, understand and improve how to specify Quality of Service at the model level in a convenient and standardized way.
- Architecture assessment and taxonomies enabling to reason about properties of architectural patterns with special attention to QoS, e.g., to be able to predict how a specific architectural pattern support different Quality of Service requirements
- Model transformation in general and QoS-aware model transformation in particular. Transformations might be from one PIM to another, from PIM to PSM and from PIM/PSM to code (code generation)
- QoS-aware reasoning, e.g., to consider if some of the QoS-requirements contradict and how to deal with such situations (what to reason about and how to reason).
- QoS-aware deployment and execution e.g., dynamic adaptation; how might a system adapt to changes of the environment dynamically.

References

[1] Soley, R.M, Frankel, D.S.,Mukerji, J.,Castain, E.H., Model Driven Architecture - The Architecture Of Choice For A Changing World, OMG 2001. http://www.omg.org/mda/

[2] OMG, *Unified Modelling Language (UML) 1.4 Specification*, Object Management Group, Document formal/01-09-67, 2001

[3] Meta-Object Facility ((MOF™)), version 1.4, www.omg.org

[4] Common Warehouse Metamodel™ (CWM™) Specification, v1.1, www.omg.org

[5] Richard Staehli, Frank Eliassen. QuA: A QoS-aware Component Architecture. Technical Report Simula 2002-13, Simula Research Laboratory

[6] http://www.simula.no:8888/QuA

[7] Object Management Group, *UML Profile for Schedulability, Performance, and Time Specification* (March 2002), www.omg.org

[8] [ISO/IEC JTC1/SC21, 1995b], *QoS-- Basic Framework*, ISO, Report: ISO/IEC JTC1/SC 21 N9309.

[9] [ISO/IEC JTC1/SC7, 1999a], *Information Technology - Software product quality - Part 1: Quality model*, ISO/IEC, Report: 9126-1, pp. 25.

[10] UML™ Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms, Revised submission, May 4 2003, www.omg.org (members only)

[11] UML™ Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms Request for Proposal OMG Document: ad/2002-01-07 www.omg.org

[12] UML profile for QVT