Automated Techniques for Mapping Domain-level QoS Requirements to Middleware-specific QoS Configurations*

Amogh Kavimandan and Aniruddha Gokhale
Dept. of EECS, Vanderbilt University, Nashville, TN
{amoghk,gokhale}@dre.vanderbilt.edu

Abstract

Contemporary middleware platforms provide a high degree of flexibility and configurability to support a large class of distributed systems found in a variety of domains, such as avionics, automotive, financials, healthcare and defense. Realizing the quality of service (QoS) properties of these systems requires mapping the domain-specific QoS requirements of these systems to the right set of configuration options of the middleware platforms. This poses a substantial challenge for the system developers who are experts in the domain but not in the middleware. Automated graph transformation techniques provide a promising approach to address these challenges.

This paper describes an automated model-driven graph transformation toolchain for QoS Mapping (GT-QMAP) that uses (1) model-driven engineering to capture system QoS requirements at domain-level abstractions, and (2) model transformations to automate the mapping of domain-specific QoS requirements to middleware-specific QoS options. We demonstrate GT-QMAP in the context of a representative NASA distributed system.

Keywords: MDE, Model transformations, Middleware.

1. Introduction

Software development processes for distributed systems found in a variety of domains such as avionics, automotive, financials, healthcare and defense are increasingly moving away from the “build from scratch” approach to a “build by composition and configuration” approach. The technology enablers fostering this new approach are contemporary component middleware solutions, such as the CORBA Component Model (CCM), Enterprise Java Beans (EJB) and .NET Web services. Since these middleware solutions are developed to support a large class of applications in multiple domains, they are designed to be highly configurable and flexible.

Despite the substantial benefits of middleware technologies, system developers are faced with significant challenges configuring the middleware platforms effectively for their domains. For example, in order to achieve the desired quality of service (QoS) characteristics for a distributed system, domain-specific QoS requirements must be mapped onto the right set of middleware-specific QoS configuration options. This requires a comprehensive knowledge of various QoS configuration options supported by the middleware, their impact on resulting QoS, and their interdependencies, all of which are critical to realize the QoS properties of the distributed system.

System developers are domain experts who understand domain-specific issues but often lack deeper insights about different middleware configuration options and their impact on resulting system QoS. Failure to carefully map domain-level QoS requirements into low-level middleware-specific configuration options can lead to a suboptimal middleware configuration degrading the overall system performance, and in worst case cause runtime errors that are costly and difficult to debug. There is a significant need therefore for research and development in automating the mapping from domain-specific, system-level QoS requirements into the right set of configurations of the hosting middleware platforms.

Graph transformation (GT) techniques [15, 9] offer a promising approach to address these requirements, particularly those that involve model transformations. GTs operate on labeled typed graphs [2], which can be augmented with attributes. The typed graph structures can be used for representing the input – in our case comprising the domain-specific QoS requirements, and the output – in our case comprising the models e.g., QoS configuration options that are represented using UML-style notations [7]. The GT rules in such model transformations are expressed in a declarative, well-understood format in terms of source and target graph patterns, i.e., subgraphs of source and target
graphs, and therefore can be validated for correctness.

Some well known examples of model transformation techniques include the Graph Rewriting And Transformation language (GReAT) [11] and VIual Automated model TRAnsformations (VIATRA) [6]. These tools provide the mechanisms and required constructs to model and implement transformation rules. The details of rewriting rules specific to an application use case i.e., the graph pattern matching that performs semantic transformations from input to output models, however, must still be specified using these transformation tools for that application.

To overcome these limitations and to realize the goals of automated QoS mapping, we have developed the Graph Transformation Quality of Service MAPping (GT-QMAP) model transformation toolchain that automates the following activities:

a. Specifying application QoS requirements – Application QoS requirements are dictated by domain-specific policies and are typically expressed by application developers in terms of what quality characteristics (i.e., quality values) are expected from the application rather than how these characteristics can be achieved at the middleware level. GT-QMAP allows developers to specify application QoS requirements including, for example, support for simultaneous service requests from client components, concurrency support such as number of threads required to provide a particular service, support for buffering service requests.

b. Identifying middleware QoS options for satisfying QoS requirements – QoS options are the mechanisms that middleware provides in order to satisfy the application QoS requirements. Examples of QoS options provided by Lightweight CCM includes Real-time CORBA features like ThreadPool to configure thread resources, and Priority Model to configure whether service requests are executed at the invocation priority or not. Correct QoS options must be chosen for a given application QoS requirement specification such that desired application QoS can be achieved by tuning these appropriately. GT-QMAP automates the process of identifying QoS options corresponding to the application QoS requirements thereby relieving the application developers from having a detailed knowledge of the low-level configuration mechanisms of the middleware platform.

c. Mapping of application QoS requirements correctly onto the middleware QoS options – Pertinent values for a subset of QoS options (identified in (b)) must be chosen to correctly tune the middleware for a specific application. GT-QMAP provides the mapping of QoS requirements onto the middleware QoS options by selecting appropriate values for these QoS options. Further, during the above QoS mapping process, GT-QMAP ensures the validity of individual QoS options both at the component/component-interface-level, as well as at the application-level.

GT-QMAP uses existing domain specific modeling languages (DSMLs) as input and output typed graphs, respectively: (1) Platform Independent Component Modeling Language (PICML) [3] used for modeling component assemblies, inter-and intra-assembly interactions and interfaces, and simplifying various activities of component-based application development such as packaging, and deployment, and (2) Component QoS Modeling Language (CQML) [20] a platform-specific component QoS modeling language that allows application developers to express QoS configurations at different levels of granularity, in terms of intuitive, visual representations.

GT-QMAP augments PICML with a Requirements Metamodel, which enables an application model to be annotated with domain-specific QoS requirements. The QoS Configuration Metamodel in CQML on the other hand, models low-level, platform-specific QoS options. The GT-QMAP transformation rules defined in terms of input and output typed graphs automate the entire process of mapping platform-independent, system-level QoS requirements into middleware-specific, QoS configuration options. GT-QMAP thus significantly reduces the application life-cycle costs and time-to-market. The remainder of this paper is organized as follows: Section 2 describes a motivating DRE system we use to describe the challenges in QoS mapping; Section 3 describes the GT-QMAP toolchain and how it addresses the challenges outlined in Section 2; Section 4 describes related research; and Section 5 describes concluding remarks outlining lessons learned and future work.

2. Challenges in Middleware QoS Configuration

Section 1 outlined the need for automating the transformations from domain-specific QoS requirements to middleware-specific QoS configuration options. Developing a toolchain to provide these automated transformations incurs a number of challenges. In this section we discuss these challenges in the context of a case study.

2.1. Distributed system Case Study

We use NASA's Magnetospheric Multi-scale (MMS) space mission (stp.gsfc.nasa.gov/missions/mms/mms.htm) as an example to motivate the need for automated tools for mapping the domain-specific QoS requirements to middleware-specific QoS configurations.
NASA’s MMS mission is a representative distributed system consisting of several interacting subsystems with a number of complex QoS requirements. It consists of four identical spacecrafts that orbit around a region of interest in a specific formation. These spacecrafts sense and collect data specific for the region of interest and at appropriate time intervals send it to the ground stations for further analysis.

Application developers of the MMS mission must account for mission-specific QoS requirements along two separate dimensions: (1) each spacecraft needs to operate in multiple modes, and (2) each spacecraft collects data using sensors whose importance varies according to the data being collected. The MMS mission involves three modes of operation: slow, fast, and burst survey modes. The slow survey mode is entered outside the regions of scientific interests and enables only a minimal set of data acquisition (primarily for health monitoring). The fast survey mode is entered when the spacecrafts are within one or more regions of interest, which enables data acquisition for all payload sensors at a moderate rate. If plasma activity is detected while in fast survey mode, the spacecraft enters burst mode, which results in data collection at the highest data rates. A prototype of the data processing subsystem of this distributed system has been developed [19] by our collaborators at Vanderbilt University using the Component-Integrated ACE ORB (CIAO) [8] QoS-enabling component middleware framework, the RACE [18] dynamic QoS adaptation framework and the PICML [3] model-driven engineering tool. In this case study section we focus on the physical activity sensor, data collection, and transmission challenges in the MMS mission, which NASA is developing to study the microphysics of plasma processes.

Figure 1 shows the components and their interactions within a single spacecraft. Each spacecraft consists of a science agent that decomposes mission goals into navigation, control, data gathering, and data processing applications. Each science agent communicates with multiple gizmo components, which are connected to different payload sensors. Each gizmo component collects data from the sensors, which have varying data rate, data size, and compression requirements.

The data collected from the different sensors have varying importance, depending on the mode and on the mission. The collected data is passed through filter components, which remove noise from the data. The filter components pass the data onto analysis components, which compute a quality value indicating the likelihood of a transient plasma event. This quality value is then communicated to the other spacecraft and used to determine entry into burst mode while in fast mode. Finally, the analyzed data from each analysis component is passed to a comm (communication) component, which transmits the data to the ground component at an appropriate time.

QoS requirements of the MMS mission across mode (of spacecrafts) and importance (of data) dimensions are known, however the application developer still has to identify correct middleware QoS options required to meet these requirements and accurately map the requirements to QoS options by choosing pertinent values for these options. We discuss GT-QMAP and how it resolves these challenges in Section 3.

2.2. Design Challenges

Although QoS-enabling component middleware and existing MDE tools provide several advantages in software development, several key requirements need to be satisfied in order to effectively enable QoS configuration of various software components of distributed systems, such as the MMS Mission. We list these requirements below.

**Challenge 1: Specifying domain-specific QoS requirements** – System developers are domain experts who can understand and reason about various domain-level issues. Therefore, the QoS requirements of a distributed system, which are well understood by the system developers must be expressible in terms of domain concerns pertaining to that domain rather than in terms of low-level, middleware-specific mechanisms required to satisfy these concerns.

For example, in the MMS mission, in order to support multiple service invocations (from each of the Analysis components) at varying importance levels, the Comm component is configured to have ThreadPool with Lanes feature. In order to avoid deadlocks, the access to asynchronous connection between Comm and Analysis must be thread safe such that only one Comm component thread can access the asynchronous connection (for retrieving its events, for example) at any given time. It is highly desirable, however, for application developers of the MMS mission to be able to specify these requirements at the level of MMS mission instead of the middleware.

Addressing this challenge requires tool support for intuitive modeling capabilities that capture QoS concerns of a system using semantics and notations that are closer to the domain. Further, since distributed systems exhibit QoS
3. Design of GT-QMAP

This section describes the GT-QMAP QoS mapping toolchain for QoS-enabling component middleware. GT-QMAP uses model-driven engineering (MDE) [17] for the description of high-level domain-specific QoS requirements to capture the (platform-independent) system requirements across various QoS domains and model-driven graph transformations for the translation of these QoS requirements into (individual) platform-level QoS configuration options necessary to realize these QoS requirements.

We have used the Graph Rewriting And Transformation language (GReAT) [10] for model-to-model translation of QoS requirements defined by application developers (source model) to QoS configuration options (target model). GReAT which is developed using GME, can be used to define model-to-model transformation rules using its visual language, and executing these transformation rules for generating target models using the GReAT execution Engine (GR-Engine).

Figure 2 shows the overall GT-QMAP toolchain. DRE application developers use the QoS requirements modeling language defined by the Requirements meta-model to specify the application QoS requirements. Models defined using QoS requirements modeling language act as the source models of GT-QMAP transformation. Similarly middleware-specific QoS configuration options are captured using the QoS configurations meta-model in Component QoS Modeling Language (CQML), and models defined using this language are the target models. The graph rewriting rules are defined in GReAT in terms of source and target type graph (i.e., meta-models). These rules are used by the GR-Engine in order to create QoS op-

![Figure 2: GT-QMAP Toolchain for mapping QoS requirements to platform-specific QoS Options.](image-url)
tions model of a DRE system from its QoS requirements model.

Although we have superimposed the Requirements meta-model on the Platform Independent Modeling Language (PICML) [3], it is not tied PICML and thus can be associated with any other platform-independent modeling language that provides capabilities for modeling structural units (for example, a component, an assembly, or connections thereof) of a component-based system. A language such as PICML can be used to capture the structure of a DRE system in terms of its components, assemblies, their interfaces and interactions. Unless stated otherwise, our use of PICML throughout the remainder of the paper refers to its QoS Requirements modeling capabilities.

In order to be able to associate the QoS policies with structural units (for example, a component, an assembly, or connections thereof) of a component-based DRE system, the Requirements meta-model is superimposed on the Platform Independent Modeling Language (PICML) [3]. PICML can be used to capture the structure of a DRE system in terms of its components, assemblies, their interfaces and interactions. Unless stated otherwise, our use of PICML throughout the remainder of the paper refers to its QoS Requirements modeling capabilities.

![Diagram of QoS Requirements Model for Publish/Subscribe service in PICML](image)

**Figure 3: QoS Requirements Model for Publish/Subscribe service in PICML.**

### 3.0.1. Modeling QoS requirements in PICML

PICML defines `Requirement` as a generalization of QoS requirements in various domains, figure 3(b) shows that the granularity of `Requirement` can be `ComponentAssembly`- or `Port`- level and more than one such elements (of the same type) can be associated with the same `Requirement`. A particular `ComponentAssembly`'s `Requirement` is also associated with all the components contained it that `ComponentAssembly`. Such associations provide significant benefits in terms of flexibility in the creation of QoS requirements models and scalability of the models.

**Real-time QoS requirements.** Real-time requirements have component-level granularity and the meta-model is shown in Figure 3(a). Depending on the implementa-

### 3.0.2. Modeling CCM QoS options in CQML

While PICML models capture platform-independent, domain-
specific QoS requirements, QoS Configuration meta-model defined in CQML is used to represent platform-specific QoS options. In our case, for example, we have defined meta-models for CCM QoS options. In this section we describe the meta-models publish/subscribe QoS domain in detail.

![Diagram](image)

**Figure 4: Simplified UML notation of QoS options for CCM publish/subscribe service in CQML.**

Real-time QoS configuration options. As shown in Figure 4(b), RealTimeConfiguration modeling element is a concrete implementation of QoSCharacteristic and contains RT-CCM options. RealTimeConfiguration contains the following elements: (1) Lane, which is a logical set of threads each one of which run at lane_priority priority level. It is possible to configure static thread(i.e., those that remain active till the application is running and dynamic thread (i.e., those threads that are created and destroyed as required) numbers using Lane element. (2) ThreadPool, which controls various settings of Lane elements, or a group thereof. These setting include, stacksize of thread(s), whether borrowing of threads across two Lane elements is allowed, and maximum resources assigned to buffer requests that can not be immediately serviced. (3) PriorityModelPolicy, which controls the policy model that a particular ThreadPool follows. It can be set to either CLIENT_PROPAGATED if the invocation priority is preserved, or SERVER_DECLARED if the server component changes the priority of invocation. (4) BandedConnections, which defines separate connections for individual (client) service invocations.

**Publish/Subscribe QoS configuration options.** Figure 4(a) shows a simplified UML diagram of publish/subscribe services in CCM. Similar to RealTimeConfiguration element, RTECConfiguration is a concrete implementation of QoSCharacteristic-ic. We discuss some of the publish/subscribe service settings below: (1) Publisher and Subscriber element covers all the event source and sink settings, respectively. These include, managing thread locks for publishers/subscribers accessed by multi-threaded applications, and types of filtering set. (2)m RTECFactory element contains configurations specific to the event channel itself. These include, event dispatch method that controls how publisher events are forwarded to the respective subscribers, scheduling of events for delivery and other scheduler-related coordination, and handling of timeout events in order to forward them to respective subscribers. (3) Strategies to group more than one filters together for publishers/subscribers can be specified using FilterGroup element.

**Resolving Challenge 1. Specifying domain-specific QoS requirements.** By providing platform-independent modeling elements in PICML and defining representational semantics that closely follow those of the application requirements, GT-QMAP allows DRE developers to describe application QoS using simple, intuitive notation. Thus, in the MMS mission prototype, there is a QoS requirement that all the asynchronous connections between Gizmo and Comm components should preserve priorities end-to-end. Such a requirement can be represented easily by (1) enabling Real-time scheduling for each of these connections using configure_rt_info, and (2) ensuring that dispatching of events at each publisher component honors priorities by using prioritize_event_dispatch at each, Gizmo, Filter, Analysis and Comm components. Thus, in the MMS mission prototype, the QoS requirement that that the Comm component should be able to support service requests having varying levels of importance (and invoked from all its client components), can be easily represented by: (1) setting the varying_service_levels requirement value to TRUE and configuring MultipleServiceRequests at the Comm component, and (2) setting fixed_priority_service_invocations to FALSE at Gizmo, Filter, Analysis, and Comm components such that importance of the sensor data is propagated and preserved as it is sent from each of these components.

**3.1. Identifying the QoS options and Automating the mapping of QoS requirements to QoS options using GT-QMAP**

The GT-QMAP model transformations for mapping of the publish/subscribe service requirements have been defined in GReAT and contain ~150 rules. In this section we briefly outline our transformation algorithm, which is written in terms of PICML and CQML DSMLs, and discuss a few of these GT rules in publish/subscribe service domain. These rules are applicable to any application model that
confirms to PICML meta-model, and thus can be used by the application developers repetitively during the development and/or maintenance phase(s) of the application. Note that GT-QMAP model transformations preserve the granularity specified in the input PICML models, i.e., the concrete QoSCharacteristic elements in output CQML models as shown in Figure 4 have the same granularity as the corresponding concrete Requirement element in input PICML models as shown in Figure 3. We refer the reader to [] for a detailed discussion of transformation algorithm for Real-time QoS domain.

Real-time requirement transformation rules. The transformation algorithm for Real-time QoS requirement mapping is given below:

a. For every RTRequirement element in the input PICML model, create a corresponding RealTimeConfiguration element in the output CQML model. Associate the same component with the newly created RealTimeConfiguration in CQML model as was associated with the corresponding RTRequirement in PICML model.

b. For every RTRequirement element in PICML, if prioritize_service_invocations is set in ServiceConsumer, count the number of outgoing ports for the associated component (i.e., Out-EventPort and ProvidedRequestPort) and create as many BandedConnections elements in the corresponding RealTimeConfiguration element in CQML model.

c. For every RTRequirement element in PICML, if simultaneous_service_execution is set in MultipleServiceRequests element, create ThreadPool and Lane elements in the corresponding RealTimeConfiguration element in CQML and associate them with each other. Additionally, minimum and maximum service levels get directly mapped onto static and dynamic thread counts in Lane element, respectively.

d. If fixed_priority_service_execution is set in a ServiceProvider, create a PriorityModelPolicy element with SERVER_DECLARED priority model in the corresponding RealTimeConfiguration element in CQML model. Otherwise, configure the priority model to CLIENT_PROPAGATED for the newly created PriorityModelPolicy element. Finally associate PriorityModelPolicy with ThreadPool created in earlier step.

e. If varying_service_levels is set for the ServiceProvider in RTRequirement of a component, count the number of its client components. Ensure that the same number of Lane elements are present in the RealTimeConfiguration of that component in the CQML model. Populate the static and dynamic thread count as per rule (c) above.

Publish/subscribe service requirement transformation rules. The transformation algorithm for event channel QoS requirement mapping is given below:

a. For every ECRequirement in PICML model, create a corresponding RTECConfiguration element in CQML model. Similar to ECRequirement that contains all the publish/subscribe service requirements, RTECConfiguration element contains all the configuration options for this service. The newly created RTECConfiguration in CQML model has the same granularity as that of the ECRequirement in PICML model. Also, for every RTECConfiguration element created, create a new RTECFactory element (containing configuration options specific to the event channel itself) and associate it with that RTECConfiguration element.

b. For a ECPublisher(ECSubscriber) element in PICML model, create a Publisher(Subscriber) element in RTECConfiguration of CQML model, which will contain all the publisher(subscriber)-related configuration options. If a filter type is configured for a ECPublisher (ECSubscriber), create the same filter type for the corresponding Publisher(Subscriber).

c. If disconnect_dangling_connections is set to TRUE in ECRequirement, configure the control policy (that handles invalid object reference(s)) in both Publisher and Subscriber to REACTIVE. If REACTIVE control policy is set at the Publisher/Subscriber, the object references of these components are polled at regular intervals in order to ensure that they are valid.

d. If publisher_filtering is set in ECRequirement, in the CQML model configure the filtering in Publisher to per_publisher. Otherwise, configure filtering to NULL. Similarly, if subscriber_filtering is set, following two choices are available: (1) if configure_rt_sched is set to TRUE for the corresponding ECSubscriber indicating that real-time scheduling is being used, in the CQML model configure the filtering in RTECFactoy to PRIORITY such that subscriber filtering honors priorities of events being filtered, (2) otherwise, configure the filtering to BASIC.

e. If multithreaded access is configured for ECPublisher (ECSubscriber), in the CQML model configure the lock in Publisher(Subscriber) to REACTIVE ensuring that the access to the event channel is
If \(1 \leq \alpha \leq 3.1\) and apply the GT-QMAP transformation elements. For example, in our MMS mission prototype, since (1) Real-time scheduling is enabled, and (2) event dispatch is prioritized, at all asynchronous connections between Gizmo and Comm components, as per publish/subscribe transformation algorithm step (f), a value of PRIORITY is set for dispatching at each event channels.

4. Related Work

We discuss the related work in the area of middleware QoS configuration.

Model Driven Architecture (MDA) [14] development centers around defining platform-independent model (PIM) of an application and applying (typed, and attribute augmented) transformations to PIM to obtain platform-specific model(s)(PSM). The COMQUAD project [16] discusses extensions to MDA in order to allow application developers to refine non-functional aspects of their application from an abstract point of view to a model close to the implementation. Model transformations are defined between different non-functional aspects and are applied to QoS characteristics (i.e., measurement of quality value) definitions to allow for such a refinement.

Authors in [1] attempt to clearly define platform-independent modeling in MDA development by introducing an important architectural notion of Abstract Platform that captures an abstraction of infrastructure characteristics for models of an application at some platform-independent level in its design process. An important observation of the authors is the requirement of design languages that they should allow for appropriate levels of platform-independence to be defined.

GT-QMAP differs from the above projects in the following two ways: (1) COMQUAD allows for specification and transformation of non-functional aspects at different levels of abstraction, as the application itself evolves. For example, response time of a function call may be expressed more clearly as the time between reception of a request and sending the corresponding response, or time between reception of a request and reception of the corresponding response. Successive refinement models in COMQUAD are exposed to the application developers, such that more details can be added. Similarly, work discussed in [1] proposes that design languages should support platform-independence at each abstract platform levels. GT-QMAP, on the other hand, deals with mechanisms to translate QoS requirements an application places on the implementation platform onto QoS configuration options of that platform. Output models of GT-QMAP can be treated as read only models. Application developers thus, model and modify the high-level PICML models only, and are thus shielded from the low-level details about the middleware platform. Finally, we focus on QoS requirements (and mappings thereof) of an application at the middleware level while COMQUAD focuses on QoS...
characteristics for an application (i.e., response time, delay, memory usage).

The Adaptive Quality Modeling Language (AQML) [13] provides QoS adaption policy modeling artifacts. AQML generators can (1) translate the QoS adaption policies (specified in AQML) into Matlab Simulink/Stateflow models for simulations using a control-centric view of QoS adaptation and (2) generate Contract Definition Language (CDL) specifications used in QuO [21] from AQML models. GT-QMAP differs with AQML in several ways, including the application of QoS adaption and the precision of the middleware modeling. For example, GT-QMAP models the configuration of standards-based QoS-enabled component middleware technologies, whereas AQML targets QuO. Moreover, GT-QMAP’s middleware model precisely abstracts the implementation so it does not need a two-level declarative translation (from AQML to CDL to potentially CCM using QuO delegates [21]) to achieve QoS adaptation.

Another approach that uses an aspect-oriented specification technique for component-based distributed systems is discussed in [4]. This work deals with specification of functional behavior, non-functional behavior, QoS management policies, and requirements of the application and synthesis of QoS management components for that supporting application-level adaptation strategies. A declarative approach is used to specify the system, e.g., real-time temporal logic and timed automata notations are used to describe the application requirements and QoS management policies, respectively. This aspect-oriented technique is similar to QuO [21], which uses several high-level languages to capture different aspects of QoS support. In contrast to the above works, GT-QMAP focuses on automating the error-prone activity of middleware QoS configuration, i.e., mapping QoS requirements to QoS configuration options. Such an automation along with a flexible and intuitive QoS requirement specification mechanism naturally supports application QoS evolution during its development cycle. An interesting side-effect of using model transformations for QoS configuration is that since the changes to application QoS are made only at QoS requirement specification time, the implementation platform details (i.e., middleware QoS options) always remain in-sync with the application QoS requirements, thereby addressing the productivity problem [12] at the middleware level. Finally, since the specification of the QoS requirements itself is platform-independent, it allows for reconfiguring the QoS mappings to suit other middleware platforms.

5. Concluding Remarks

Large-scale distributed systems are increasingly built using middleware technologies that provide reusable building blocks and services to support rapid software development. In order to configure them correctly for particular application needs, these middleware platforms provide highly customizable QoS mechanisms. There is a need, however, to (1) raise the level of specification abstraction for application developers (who lack a detailed understanding of these QoS mechanisms and their inter-dependencies) such that application QoS requirements can be expressed in a intuitively, and (2) correctly map these QoS specifications to middleware-specific QoS options.

In this paper we introduced a model-driven QoS mapping toolchain that addresses these challenges in middleware QoS configuration. We showed the use of the toolchain for Real-time and CORBA publish/subscribe service QoS domains for correctly translating QoS requirements in these domains onto respective QoS options. Since the transformation rules are written in terms of typed graphs (i.e., metamodels), the toolchain can be reused for any application QoS requirement specification confirming to the input typed graph.

In the future, we plan to extend our GT-QMAP toolchain for other QoS domains, for example, security and fault-tolerance, and for other middleware technologies, such as EJB and Web Services. GT-QMAP is available as open-source from www.dre.vanderbilt.edu/CoSMIC/.

References


