Automated Middleware QoS Configuration Techniques for Distributed Real-time and Embedded Systems*

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

Abstract

Quite often the modeling tools used in the development lifecycle of distributed real-time and embedded (DRE) systems are middleware-specific, where they elevate middleware artifacts, such as configuration options, to first class modeling entities. Unfortunately, this level of abstraction does not resolve the complex issues in middleware configuration process for QoS assurance. This paper describes GT-QMAP (Graph Transformation for QoS MAPping) model-driven engineering toolchain that combines (1) domain-specific modeling, to simplify specifying the QoS requirements of DRE systems intuitively, and (2) model transformations, to automate the mapping of domain-specific QoS requirements to middleware-specific QoS configuration options. The paper evaluates the automation capabilities of GT-QMAP in the context of three DRE system case studies. The results indicate that on an average the modeling effort is reduced by over 75%. Further, the results also indicate that GT-QMAP provides significant benefits in terms of scalability and automation as DRE system QoS requirements evolve during its entire development lifecycle.

Keywords: *MDE, Graph/model transformations, QoS, middleware configuration.*

1 Introduction

Component-based software engineering (CBSE) is finding wide acceptance in the development of modern distributed real-time and embedded (DRE) systems. Consequently component middleware platforms, such as Lightweight CORBA Component Model (LwCCM), are designed to be highly flexible to support a large class of DRE systems from multiple domains. These middleware platforms therefore provide a number of configuration mechanisms for (1) allocating CPU, network and OS resources *a priori*, (2) (re)configuring and (re)deploying distributed system components, and (3) (de)marshaling communication requests, component activation/deactivation and persistence services, all of which are decoupled from the functional composition aspects of DRE systems.

Assuring DRE system QoS properties involves multiple different factors. Apart from making the right decisions on deployment and functional composition, it is critical to perform the *middleware QoS configuration* activity *i.e.*, correctly mapping system QoS properties onto the underlying middleware configuration options. Such an activity requires insights about different middleware configuration options, their impact on resulting QoS, and their inter-dependencies. DRE system developers understand application-specific design and implementation issues but seldom have the necessary expertise to perform middleware QoS configuration. Failure to carefully map domain-level QoS requirements onto low-level middleware-specific configuration options can lead to a suboptimal middleware configuration degrading the overall system performance, and in worst cases cause runtime errors that are costly and difficult to debug.

Existing works in QoS assurance for DRE systems have focused on: (1) application functional specification, decomposition and analysis [6] to capture and study application structure and behavior, (2) QoS analysis, optimization and adaptation [12] to allocate resources to applications, provide for application QoS optimization and adaptation in multiple QoS dimensions, and (3) schedulability and timing analysis [16, 5] to determine exact priorities and time periods for applications. Some work has also been done in QoS specification languages [14, 2, 17] for capturing application QoS properties by elevating middleware artifacts (such as its configuration options) to first class modeling entities. We argue that this level of abstraction does not resolve the challenges involved in middleware QoS configuration, which is the focus of this paper.

Solution Approach → Model-driven Middleware QoS Configuration. Model driven engineering (MDE) has shown significant promise and success in enabling the reasoning of system properties using domain-specific notations, and automating platform-specific artifacts using generative capabilities. MDE has been successfully used in ver-

*This work was sponsored in part by Lockheed Martin Advanced Technology Laboratories.

1080-1812/08 $25.00 © 2008 IEEE
DOI 10.1109/RTAS.2008.29

ification of system correctness properties [6], and functional and QoS modeling [2], among others.

Our earlier work [9] outlined the challenges and conceptual ideas in middleware QoS configuration and described the QUICKER MDE toolchain. This paper delves into the details of the automated transformation capabilities, which are the cornerstone of tools like QUICKER. In particular, we describe GT-QMAP (Graph Transformations for Quality of service MAPping), which uses graph transformations on system models to automate the middleware QoS configuration. Our approach begins with domain-specific, platform-independent models (PIMs) of DRE system QoS requirements that are automatically transformed to more refined and detailed middleware platform-specific models (PSMs). In this paper we focus only on the automated QoS configurations for real-time (RT) request-response and publish-subscribe communication dimensions.

To describe and evaluate the algorithms developed for GT-QMAP, we use the following domain specific modeling languages (DSMLs) as input and output typed graphs for the automated QoS mapping: (1) Platform Independent Component Modeling Language (PICML) [2] used for modeling component assemblies, inter-and intra-assembly interactions and interfaces, and simplifying various activities of component-based system development such as packaging, and deployment, and (2) LwCCM QoS Modeling Language (CQML)that allows system developers to express QoS configurations at different levels of granularity using intuitive, visual representations.

Paper Organization. The remainder of this paper is organized as follows: Section 2 describes motivating DRE systems 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 evaluates GT-QMAP QoS configuration capabilities in the context of the DRE system case studies; Section 5 describes related research; and Section 6 describes concluding remarks outlining lessons learned and future work.

2 Challenges in Automated Middleware QoS Configuration

Section 1 outlined the need for automating the tedious and error-prone process of middleware QoS configuration. Developing a scientific approach to automate this activity poses a certain set of challenges. We discuss these challenges in the context of three case studies, which we also use in the paper for evaluating our approach.

2.1 DRE system Case Studies

We chose the following DRE systems as the application scenarios for our experiments:

BasicSP. The Basic Single Processor (BasicSP) is a scenario from The Boeing Bold Stroke component avionics computing product line. BasicSP uses a publish/subscribe service for event-based communication among its components, and has been developed and configured using a QoS-enabled component middleware platform. The application is deployed using a single deployment plan on two physical nodes.

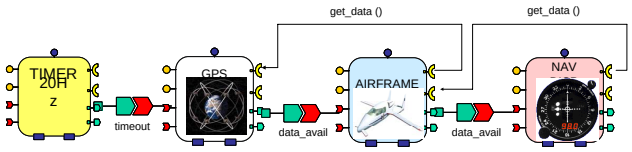

Figure 1: Basic Single Processor

A GPS device sends out periodic position updates to a GUI display that presents these updates to a pilot. The desired data request and the display frequencies are fixed at 20 Hz. The scenario shown in Figure 1 begins with the *GPS* component being invoked by the *Timer* component. On receiving a pulse event from the *Timer*, the *GPS* component generates its data and issues a data available event. The *Airframe* component retrieves the data from the *GPS* component, updates its state and issues a data available event. Finally, the *NavDisplay* component retrieves the data from the Airframe and updates its state and displays it to the pilot.

MMS. NASA's Magnetospheric MultiScale (MMS) mission is a representative DRE 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.

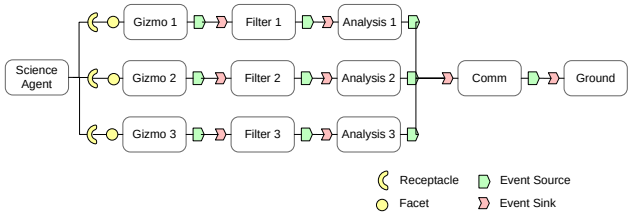

Figure 2: MMS Mission System Components

Figure 2 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 other spacecrafts 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.

SCE. The Shipboard Computing Environment (SCE) consists of a sequence of several components connected together to form multiple operational strings. Each operational string has different importance levels and these levels are used to resolve any resource contention between them.

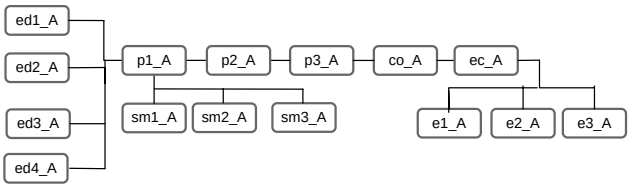

Figure 3: Shipboard Computing Environment

As shown in Figure 3, each operational string contains a number of sensor components (*e.g.*, *ed1_A*,*ed2_A*) and system monitor components (*e.g.*, *sm1_A*, *sm2_A*) that publish data from the physical devices to a series of planner components (*e.g.*, *p1_A*, *p2_A*). Once the inputs from sensors and system monitors has been analyzed, the planners perform control decisions using the effector components (*e.g.*, *e1_A*, *e2_A*). As shown in Figure 3, each operational string contains ten components altogether. SCE has ten operational strings that are deployed using ten deployment plans on five physical nodes.

Configuration complexity of scenarios. As already mentioned, in this paper we have focussed on QoS specification for request-response and publish-subscribe communication paradigms. From our past experiences with developing and configuring QoS for DRE systems [9], we chose a 3-tuple $\{C;I;D\}$ to represent configuration complexity of our application scenarios where,

- C defines the number of components of the application.
- I defines distinct number of interactions between components of the application. An interaction exists between two components if the outgoing port of one is connected to incoming port of the other.
- D defines the distinct number of dependencies between components of the application. A dependency exists between two components if a change in the QoS configuration of one necessitates a change in configuration of the other.

Table 1: Complexity of application scenarios

Application scenarios	# of components	# of component interactions	# of component dependencies
BasicSP	4	5	6
MMS	12	11	43
SCE	150	260	950

The application scenarios described above illustrate different levels of configuration complexity and can be summarized using our 3-tuple definition as shown in Table 1.

2.2 Design Challenges

In the remainder of this section, we discuss the challenges in automating the QoS configurations:

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 DRE system must be expressible in terms of *domain* concerns rather than in terms of low-level, middleware-specific mechanisms required to satisfy these concerns.

For example, a requirement for the asynchronous connection between *Comm* and *Analysis* components in the MMS mission is that its access 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. Real-time publish/subscribe service provides advanced synchronization mechanisms in order to address such application requirements. It is highly desirable, however, for system developers to be able to specify these requirements at the domain-level 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 DRE systems exhibit multidimensional QoS requirements, the tool should provide clearcut separation of concerns during system QoS specification. Section 3.1 illustrates how our GT-QMAP toolchain addresses this challenge.

Challenge 2: Identifying the middleware-specific QoS configuration options for satisfying QoS requirements Although a tool may provide modeling capabilities to specify system-level QoS requirements, there remains the need to identify the right middleware-specific QoS configuration options that will satisfy the system QoS requirements. This identification process can be a challenging task because of the following factors: (1) systems evolve either

as part of the software development lifecycle, or modified domain requirements/end-goals. Naturally, the new middleware configurations would have to be identified again, which is a tedious and error-prone process, and (2) for large-scale systems this process becomes too time consuming, and in some cases infeasible.

For example in the SCE application, the planner component *p1_A* has the following requirements: (1) asynchronous connections with its client components (*i.e.*, here the system monitors) must support bursty service invocations from each of these components, and (2) service invocations from each of its client components must be prioritized. A way to satisfy the second requirement is by configuring the planner to have SERVER_DECLARED real-time CCM (RT-CCM) policy that handles invocations at pre-determined priorities. In addition, sufficient thread resources should be available to handle all client priority levels. This can be achieved by configuring *ThreadPool with Lanes* feature where a single lane corresponds to an individual priority level. Such a QoS design scheme also ensures a predictable application execution and does not exhibit unbounded priority inversions. Finally in order to satisfy the first requirement, it is prudent and economical to assign dynamic thread resources for bursty clients that reserving them in a static manner for the entire application life-cycle.

An automated QoS configuration tool should be able to codify these proven patterns and correctly identify the QoS options necessary to achieve desired system QoS from a given (semantically-correct) input model. If QoS requirements have been specified across more than one RT QoS dimensions, the tool should identify corresponding options pertaining to each of these dimensions. Section 3.2 illustrates how GT-QMAP addresses this requirement.

Challenge 3: Mapping the QoS requirements onto QoS configuration options Even if the QoS configuration options that satisfy the system QoS requirements may be identified, appropriate values for each of the configuration options must be chosen in order to correctly configure the middleware and realize system level QoS properties. Such a step would have to potentially be performed several times during the development cycle of a system and thus should be easily (and relatively quickly) repeatable.

For our MMS mission, the `Comm` component is best realized using the RT-CCM `Threadpool_with_lanes` feature so that it can provide varying levels of service to its clients. Similarly, the `Analysis` components requires the use of banded connections to prevent priority of inversions on the communication links. In both these cases it is necessary to identify how many lanes are needed in a thread pool, what priority values should be set per lane, how many bands of communication are needed, and what priority ranges are handled by each band.

Depending on the individual QoS requirements, one or more alternative QoS options may be identified in the previous step. A QoS configuration tool should choose suitable values for each of these QoS options. Additionally, it should ensure that QoS options are valid, both for the *association entity* (in the context of component middleware, an association entity would be, for example, a component, a connection between components, or an assembly to which a QoS configuration is associated), as well as for the entire component-based application. Section 3.2 illustrates how GT-QMAP addresses this requirement.

3 Design of GT-QMAP

This section describes the GT-QMAP QoS mapping toolchain for QoS-enabled component middleware. GT-QMAP is a MDE framework, which relies on DSMLs for the description of high-level, domain-specific QoS requirements that enable capturing the (platform-independent) system requirements across various QoS dimensions. Additionally, GT-QMAP uses model-driven graph transformations [8] for the translation of these QoS requirements into platform-specific QoS configuration options necessary to realize these QoS requirements on the underlying platform.

Figure 4 shows the overall GT-QMAP toolchain. DRE system developers use the *Requirements* DSML in GT-QMAP to specify the system QoS requirements. A specification of system QoS requirements acts as the source model of the GT-QMAP transformation. Similarly, middleware-specific QoS configuration options are captured as models using the *QoS Configurations* DSML which serves as the target model in the transformation process.

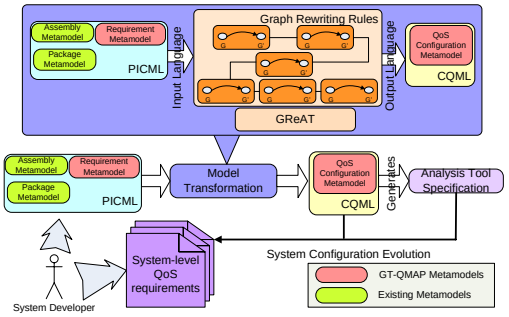

Figure 4: GT-QMAP toolchain for mapping QoS requirements to platform-specific QoS Options

GT-QMAP uses the Generic Modeling Environment (GME) [11] toolkit for developing the modeling languages used to describe the above, which provides a graphical user interface that can be used to define both DSML semantics and system models that conform to the DSMLs defined in it. *Model interpreters* can be developed using the generative capabilities in GME. The interpreters are used to traverse

the models for generating artifacts for analysis tools such as model-checking, emulation tools, etc.

We have used the Graph Rewriting And Transformation (GReAT) [8] language for defining model-to-model translations of QoS requirements.GReAT, which is developed using GME, can be used to define transformation rules using its visual language, and executing these transformation rules for generating target models using the GReAT execution engine (GR-Engine). The graph rewriting rules are defined in GReAT in terms of source and target typed graph (*i.e.*, metamodels). GT-QMAP transformation rules are used by the GR-Engine in order to create the QoS options model of a DRE system from its QoS requirements model.

For evaluating GT-QMAP modeling capabilities and demonstrating them through a prototypical implementation, our Requirements DSML has been superimposed on PICML. The requirements modeling abstractions however, are not tied to PICML alone and thus can be generally associated with any other structural modeling language that provides capabilities for modeling functional entities (for example, a component, an assembly, or connections thereof) of a component-based system.

3.1 Specifying QoS Requirements using GT-QMAP Modeling Capabilities

In Challenge 1 of Section 2 we motivated the need for domain-specific QoS specification for component-based DRE systems. We define modeling constructs in GT-QMAP that can be used by the DRE system developers to define models that capture QoS requirements. This section describes this capability and how it resolves Challenge 1.

3.1.1 Modeling QoS Requirements using GT-QMAP

GT-QMAP defines the `Requirement` element as a generalization of QoS requirements. As shown in Figure 5, source elements `Component`, `ComponentAssembly` or `Port` connections can be associated with a `Requirement` element. Modeling abstractions in GT-QMAP allow association of multiple source elements with the same `Requirement` as long as those source elements are of the same *type*. Moreover a `ComponentAssembly`'s `Requirement` is also associated with all the components contained in that `ComponentAssembly`. Such associations provide significant benefits in terms of flexibility in the creation of QoS requirements models and scalability of the models. The metamodels we describe below have been integrated with PICML using these associations, thus a single model of DRE system captures its entire QoS requirements specification.

Next we discuss the requirements specification across the following two RT QoS dimensions: (1) RT-CCM that is used to specify requirements for components and synchronous connections between components, and (2) RT

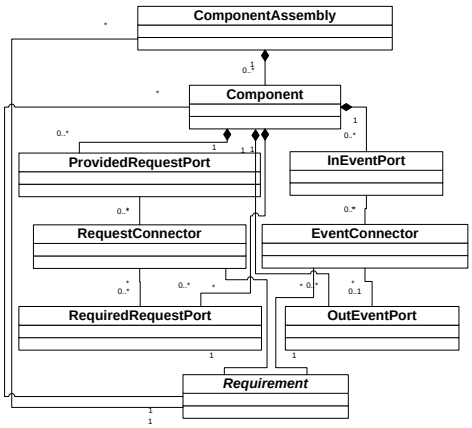

Figure 5: Simplified UML notation of QoS Requirements Associations in GT-QMAP

publish/subscribe service that is used to specify requirements for asynchronous connections between components.

Real-time QoS requirements. Real-time requirements have component-level granularity. A `RTRequirement` element which is derived from `Requirement`, captures real-time requirements of a component and may have the following two attributes:(1) `fixed_priority_service_execution`, a server component Boolean property for specifying whether or not it modifies client service invocation priorities, and (2) `bursty_client_requests`, a server component Boolean property for specifying the profile of service invocations made by its client components.

Publish/subscribe QoS requirements. We have modeled requirements for real-time publish/subscribe event service to enable specification of QoS across asynchronous and anonymous interactions in component-based DRE systems. In the context of a publish/subscribe service, a **Subscriber** component subscribes to receive events from a **Publisher** component that generates events. Publisher (subscriber) component connects to a mediator entity, an **Event Channel**, to publish (subscribe to) events.

The `ECRequirement` element is derived from `Requirement`. It models the properties of the event channel and can be used to specify the following QoS requirements: (1) `network_quality`, a connection-level property that captures the quality value of network used for running the application. (2) `connection_frequency`, a component-level property specifying the frequency at which the component (dis)connects with the publish/subscribe connection. (3) `event_distribution_ratio`, a connection-level property that specifies the ratio: $\frac{E^a{}_c}{E^s{}_c}$, where $E^a{}_c$ denotes number of events available for subscription at connection c and $E^s{}_c$ denotes average number of events subscribed to at

connection c by each subscriber component. These modeling capabilities are at a sufficiently high level of abstraction and are intuitive to be applied to a variety of publish/subscribe mechanisms. All the requirements have an enumerated data type with values `LO` and `HI`.

3.2 Automating QoS requirements mapping using GT-QMAP

Challenge 2 and 3 in Section 2 motivated the need for an automated toolchain for performing QoS configuration of the underlying middleware platform. In this section, we first describe CQML QoS Configuration DSML that defines middleware-specific QoS options and outline our transformation algorithm that transforms system QoS requirements.

3.2.1 Modeling Middleware QoS options in CQML

Rather than directly transforming source models of DRE system into configuration descriptors required for deploying it on the middleware, we chose to generate models of middleware-specific QoS options from these source models such that they can be used for further analysis such as model-checking QoS properties of the DRE system. We have developed interpreters for parsing CQML system models and generating deployment descriptors in preparation of deploying the DRE system on target environment.

Real-time QoS options. CQML defines the following elements corresponding to several RT-CCM configuration mechanisms: (1) `Lane`, which is a logical set of threads each one of which runs at `lane_priority` priority level. It is possible to configure *static* thread (*i.e.*, those that remain active till the system 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 threads, whether borrowing of threads across two `Lane` elements is allowed, and maximum resources assigned to buffer requests that cannot 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 options. For QoS configuration of asynchronous event communications, CQML defines the following elements: (1) `Publisher` and `Subscriber` modeling elements contain all the event source and sink settings, respectively. These include, for example, thread locks management mechanisms for publishers (subscribers) that are accessed by multi-threaded systems, and types of event filtering used, (2) `RTECFactory` element contains configurations specific to the event channel itself. These include, for example, event dispatching method that controls how events from publishers 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, and (3) `FilterGroup` element that specifies strategies to group more than one filters together for publishers (subscribers).

3.2.2 GT-QMAP Transformations for QoS Mapping

The GT-QMAP model transformation rules have been defined in GReAT and are based on our past experiences in configuring QoS for component-based DRE systems. They are applicable to any system model that conforms to the Requirements DSML, and thus can be used by the system developers repetitively during the development and/or maintenance phase(s) of the DRE system. GT-QMAP model transformations preserve the granularity specified in the source models.

Mapping real-time QoS requirements. Let ${R_p}^o$ and ${R_p}^i$ denote, respectively, the set of outgoing (required/event source) and incoming (provided/event sink) ports of component $p \in P$. Let S and C be the sets of server and client components respectively and are given by:

$p \in S \; if \; {R_p}^i \neq \emptyset$ and $p \in C \; if \; {R_p}^o \neq \emptyset$

Algorithm 1 describes (non-exhaustive) RT-CCM QoS mappings in GT-QMAP. Lines 5-13 show the thread resource allocation scheme for server components. For every incoming port of a server component, the number of interface operations and client components are counted (lines 9 and 10). These counts are used by the auxillary function *ThreadResources* to calculate the total threads required for handling all client service invocations at that server.

For handling bursts of client requests, server components should configure their thread pool to grow dynamically such that threads are created only when required.*assignThreadResources* function is used to adjust the ratio of static and dynamic threads for a server, depending on whether its `bursty_client_requests` property is set to TRUE. In addition, lane borrowing feature at the server is set to TRUE such that the thread pool lanes across various priority levels can be borrowed. Finally, *PriorityBands* are configured and the their priority values are matched with server-side lane values in line 24.

Mapping publish/subscribe QoS requirements. Let ${PC_c}^s$ denote the synchronization mechanism, ${PC_c}^t$ denote the type, ${PC_c}^i$ denote the iterator in proxy collection PC for component c, respectively. Let L_c denote the locking policy, CP_c denote control policy, SF_c denote supplier-based filtering at component c, respectively. Algorithm 2 gives the (non-exhaustive) publish/subscribe QoS mappings.

Algorithm 1: Real-time QoS Requirements Mapping

Input: set of client components C, set of server components S, set of bursty client components B, set of threadPool lanes $TPLanes$

```
1  begin
2      InterfaceOperationsCount ioc; ClientsCount cc;
3      IncomingPort ip; OutgoingPort op;ThreadCount tc;
4      Component c;set of Components CPS; Buffering bf;
5      foreach p ∈ S do
6          ioc ← 0; cc ← 0; tc ← 0; bf ← false;
7          CPS ← ClientComponents(p);
8          foreach ip ∈ R_p^i do
9              ioc ← ioc + countOperations(p,ip);
10             cc ← cc + countClientComponents(p,ip);
11         end
12         tc ← ThreadResources(ioc,cc);
13         createTPLanes(p,tc);
14         foreach c ∈ CPS do
15             if c ∈ B then
16                 bf ← true;
17                 assignThreadResources(
18                 TPLanes_p,c,tc);
19             assignTPoolAttributes(TPLanes_p,bf);
20             ioc ← 0;
21             foreach op ∈ R_c^o do
22                 ioc ← ioc + countOperations(c,op);
23             end
24             createBands(c,ioc); matchPriorities(p,c);
25         end
26     end
27 end
```

Algorithm 2: Publish/Subscribe service QoS Requirements Mapping

Input: set of components CPS

```
1  begin
2      Component c; ThreadPoolLaneCount lc;
3      NetworkQuality nq;
4      foreach c ∈ CPS do
5          lc = countThreadResources(c); cf = connectionFrequency(c);
           nq = networkQuality(c); dr = eventDistributionRatio(c);
6          if lc ≠ 1 then
7              PC_c^s = MT; L_c = THREAD;
8          else
9              PC_c^s = ST; L_c = NULL;
10         end
11         if cf ≠ LO then
12             PC_c^t = LIST; PC_c^i = COPY_ON_READ;
13         else
14             PC_c^t = RB_TREE; PC_c^s = COPY_ON_WRITE;
15         end
16         if nq ≠ LO then
17             CP_c = NULL;
18         else
19             CP_c = REACTIVE;
20         end
21         if c ∈ S then
22             if dr ≠ LO then
23                 SF_c = PER_SUPPLIER;
24             else
25                 SF_c = NULL;
26             end
27     end
28 end
```

A publish/subscribe service has several settings for configuring the way collections of publisher and subscriber object references are created and accessed, which must be chosen appropriately for individual applications. Lines 6-9 in Algorithm 2 show how the choice of serialization mechanism is affected by the number of thread resources configured at component c.

The choice of the *type* of collection is based on the following: (1) RB_TREE data structure exhibits faster ($O(log(n))$) insertion and removal operations. Therefore, it is more suited for connections whose components have a high (dis)connection rate; (2) LIST data structure on the other hand, should be chosen in cases where iteration is frequent (and therefore, more crucial for efficient application execution) than modifications to it.

Lines 11-14 give the steps in algorithm that configure the collection type. Finally, REACTIVE policy is chosen for applications that use low-quality value network on Lines 16-19, which ensures that (publisher/subscriber) components are periodically polled for determining their states (*i.e.*, whether or not they are connected to the event channel).

3.3 Applying GT-QMAP for Middleware QoS Configuration

The challenges described in Section 2 are resolved using GT-QMAP modeling and automated QoS configuration capabilities as follows:

Resolving Challenges 1 & 2: Target typed graph elements (*i.e.*, QoS options), are well-understood by implementation middleware experts. GT-QMAP transformation algorithms 1 and 2 are designed in terms of source and target typed graphs by these experts. System developers can describe their system QoS requirements using the modeling capabilities discussed in Section 3.1.1. By providing platform-independent modeling elements in GT-QMAP and defining representational semantics that closely follow those of the system requirements, GT-QMAP allows system developers to describe system QoS using simple, intuitive notations. Further, model transformations defined in GT-QMAP automatically identify and deduce QoS configurations that are best suited to achieve the desired QoS for DRE systems being configured.

For example, in the MMS mission, GT-QMAP automatically identifies thread safety mechanisms applicable for asynchronous connection between *Comm* and *Analysis* components as can be seen from lines 6-9 in Algorithm 2. In the SCE application, the requirement of prioritization of service invocations at *p1_A* component can be easily specified by setting `fixed_priority_service_execution` to TRUE.

Resolving Challenge 3: GT-QMAP transformation rules contain information about the semantics of the QoS options, their inter-dependencies, and how they affect the high-level QoS requirements of a DRE system and therefore are used to asssign values to the subset of options chosen earlier. Further, QoS options semantics are known precisely during transformations, and thus GT-QMAP ensures preservation

of target typed graph semantics. Component interactions defined in input typed graph instance (*i.e.*, source model), along with the user-specified QoS requirements captured in that instance are used to completely generate an instance of the output graph.

For example, in SCE application, in addition to setting `fixed_priority_service_execution` to TRUE, recall from discussion in Challenge 2 in Section 2 that sufficient thread resources should also be configured to handle all client priority levels at *p1_A*. *ThreadResources* on line 12 in Algorithm 1 calculates appropriate number of thread resources as a function of client components of *p1_A* and their interface operations.

4 Evaluating GT-QMAP Toolchain for Middleware QoS Configuration

In this section we evaluate GT-QMAP modeling (*i.e.*, using its Requirements DSML) and transformation capabilitites in the context of DRE system case studies discussed in Section 2.1. Class count metrics were used for evaluating modeling effort in using GT-QMAP. All the measurements use GME 6.11.9, GReAT 1.6.0 software packages on a Windows XP SP2 workstation. Our prototype implementation of GT-QMAP uses PICML and CQML from CoSMIC toolchain version 0.5.7.

CQML models represent detailed, middleware-specific DRE system QoS configurations that are used for generating configuration descriptors necessary for its deployment. In order to find the reduction in modeling effort using GT-QMAP, we compare its (Requirements) modeling capabilities with those of CQML.

Class counts is an important metric for model-based quantitative software measurementsand has been applied and adopted in industrial contexts [4]. For our measurements, we use the following counts from the (meta)models: (1) modeling elements, which includes all the concrete modeling objects, (2) connections between modeling elements, (3) constraints that provide design-time type and/or dependency checks for enforcing language semantics, and (4) attributes of modeling elements. The counts were measured for both real-time and publish/subscribe QoS dimensions.

A comparison of CQML and GT-QMAP metamodels in terms of class counts given above is tabulated in Table 2. The configuration space in this table simply refers to all of CQML's modeling elements each of which corresponds to RT-CCM and publish/subscribe options as explained in Section 3.2.1. Using GT-QMAP, the number of modeling elements are reduced by an average of ∼58% while the number of attributes are reduced by an average of ∼81%.

The results from class count measurments for BasicSP, MMS and SCE application scenarios are shown in Table 3. From these results it is observed that the modeling elements and number of attributes required for QoS specification for the publish/subscribe QoS dimension reduced by an average 54.55% and 76.4%, respectively. Reductions for RT-CCM were considerably higher *i.e.*, modeling elements reduction was 86.53% while number of attributes were reduced by 88.47%.

Table 2: Comparing Requirements DSML against configuration space

Effort measured on	# of modeling elements	# of attributes
CQML		
publish/subscribe	9	22
RT-CCM	6	14
GT-QMAP		
publish/subscribe	4	5
RT-CCM	1	2

Connections defined in GT-QMAP are simple associations between modeling elements. For instance, recall from Section 3.1.1 that real-time and publish/subscribe QoS requirement elements have component- or connection-level granularity. In contrast, modeling elements in CQML exhibit more complex dependency relationships. For example, *e.g.*, a REACTIVE event dispatching method at an event channel necessitates that `ProxyCollection` at corresponding publisher and subscriber components be either MT or ST.

It is easier to evolve DRE system QoS using GT-QMAP owing to its automated requirements mapping capabilities. For example, an additional requirement in the SCE scenario during its development cycle necessitates that similar to *p1_A*, component *ec_A* must prioritize its service invocations. In GT-QMAP this is achieved simply by setting `fixed_priority_service_execution` property at *ec_A* (to TRUE). For the entire SCE application since it contains 10 such application strings (and therefore, 10 *ec_A* components), this additional requirement requires modification of 10 attributes in its GT-QMAP model. In CQML, on the other hand, such an additional requirement would require the following modifications: (1) Configuring the *PriorityModelPolicy* to SERVER_DECLARED, and assigning sufficient *Lanes* at *ec_A* for handling all of its client service invocations. (2) Assigning *PriorityBands* at client components (*e1_A* etc.) such that a separate connection is used for each request priority level. This configuration further requires that these band priority values match with some lane values at *ec_A* component. Even if smallest possible number of *Bands* and *Lanes* are chosen at respective components, this requires specifying ∼10 modeling elements, ∼4 connections, and ∼16 attributes for each of the 10 application strings in SCE.

Table 3: Reduction in modeling effort using GT-QMAP

Effort measured on	# of modeling elements	# of connections	# of constraints	# of Boolean	# of int/long	# of string	# of enum
BasicSP in CQML							
publish/subscribe	27	9	21	0	3	1	54
RT-CCM	18	9	9	4	28	6	2
BasicSP in GT-QMAP							
publish/subscribe	12	3	0	3	0	0	12
RT-CCM	3	3	0	4	0	0	0
MMS in CQML							
publish/subscribe	101	35	77	0	35	11	210
RT-CCM	87	44	21	14	163	20	7
MMS in GT-QMAP							
publish/subscribe	46	11	0	11	0	0	46
RT-CCM	10	10	0	18	0	0	0
SCE in CQML							
publish/subscribe	1100	390	840	0	390	120	2270
RT-CCM	980	390	360	240	1000	160	120
SCE in GT-QMAP							
publish/subscribe	510	120	0	120	0	0	510
RT-CCM	120	120	0	240	0	0	0

5 Related Work

The COMQUAD project [15] discusses extensions to model driven architecture (MDA) of OMG in order to allow application developers to refine non-functional aspects of their application from an abstract point of view to a model closer 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 that design languages should allow for appropriate levels of platform-independence to be defined at each development steps.

GT-QMAP differs from the above projects as follows: 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 a system 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 model and modify only the high-level requirements models, 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 (*e.g.*, response time, delay, memory usage).

Ritter *et.al.* [14] describe CCM extensions for generic QoS support and discusses a QoS metamodel that supports domain-specific multi-category QoS contracts. The work in [7], focuses on capturing QoS properties in terms of *interaction patterns* amongst system components that are involved in executing a particular service and supporting run-time monitoring of QoS properties by distributing them over components (which can be monitored) to realize that service. An approach that uses an aspect-oriented specification techniques for component-based distributed systems is discussed in [3].

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 [10] 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.

6 Concluding Remarks

In this paper we introduced an automated, reusable model-driven QoS mapping toolchain that (1) raises the level of specification abstraction for system developers (who lack a detailed understanding of these QoS mechanisms and their inter-dependencies) such that system QoS requirements can be expressed intuitively, and (2) correctly maps these QoS specifications to middleware-specific QoS configuration options. In the future, we plan to extend our GT-QMAP toolchain for other QoS dimensions, for example, security and fault-tolerance, and for other middleware technologies, such as EJB and Web Services. To this goal, UML profile discussed in [13] can be used as a common modeling abstraction for various QoS dimensions and platforms. We are also investigating the use of advanced model-checking techniques to correctly perform middleware QoS configuration simultaneously across various QoS dimensions. GT-QMAP is available as open-source from `www.dre.vanderbilt.edu/CoSMIC/`.

7 Acknowledgement

The authors would like to thank Reviewer 4 whose insightful and constructive comments and suggestions improved readibility and overall quality of the paper.

References

[1] J. P. Almeida, R. Dijkman, M. van Sinderen, and L. F. Pires. On the Notion of Abstract Platform in MDA Development. In *Proceedings of the 3rd IEEE International Enterprise Distributed Object Computing Conference (EDOC 2004)*, pages 253–263, Sept. 2004.

[2] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt. A Platform-Independent Component Modeling Language for Distributed Real-Time and Embedded Systems. In *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 190–199, Los Alamitos, CA, USA, 2005.

[3] L. Blair, G. S. Blair, A. Anderson, and T. Jones. Formal Support For Dynamic QoS Management in the Development of Open Component-based Distributed Systems. *IEEE Software*, 148(3), Nov. 2001.

[4] B. H. C. Cheng, R. Stephenson, and B. Berenbach. Lessons learned from automated analysis of industrial uml class models (an experience report). In *Model Driven Engineering Languages and Systems, 8th International Conference (MoDELS 2005)*, volume 3713, pages 324–338, 2005.

[5] Z. Gu, S. Kodase, S. Wang, and K. G. Shin. A Model-Based Approach to System-Level Dependency and Real-time Analysis of Embedded Software. In *Proceedings of the IEEE Real-time and Embedded Technology and Applications Symposium (RTAS'03)*, pages 78–85, Washington, DC, May 2003. IEEE.

[6] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad. Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems. In *Proceedings of the 25th International Conference on Software Engineering*, Portland, OR, May 2003.

[7] Jaswinder Ahluwalia and Ingolf H. Krüger and Walter Phillips and Michael Meisinger. Model-Based Run-Time Monitoring of End-to-End Deadlines. In *Proceedings of the Fifth ACM International Conference On Embedded Software*, Jersey City, NJ, Sept. 2005. ACM.

[8] G. Karsai, A. Agrawal, F. Shi, and J. Sprinkle. On the Use of Graph Transformation in the Formal Specification of Model Interpreters. *Journal of Universal Computer Science*, 9(11):1296–1321, 2003. `http://www.jucs.org/jucs_9_11/on_the_use_of`.

[9] A. Kavimandan, K. Balasubramanian, N. Shankaran, A. Gokhale, and D. C. Schmidt. Quicker: A model-driven qos mapping tool for qos-enabled component middleware. In *ISORC '07: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 62–70, Washington, DC, USA, 2007. IEEE Computer Society.

[10] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture(MDA^{TM}): Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, Apr 2003.

[11] A. Ledeczi, A. Bakay, M. Maroti, P. Volgysei, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing Domain-Specific Design Environments. *IEEE Computer*, pages 44–51, November 2001.

[12] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, and J. Hansen. A Scalable Solution to the Multi-Resource QoS Problem. In *Proceedings of the IEEE Real-time Systems Symposium (RTSS 99)*, pages 315–326, Phoenix, AZ, Dec. 1999.

[13] Object Management Group. *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms Joint Revised Submission*, OMG Document realtime/03-05-02 edition, May 2003.

[14] T. Ritter, M. Born, T. Unterschütz, and T. Weis. A QoS Metamodel and its Realization in a CORBA Component Infrastructure. In *Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS'03)*, Honolulu, HI, Jan. 2003.

[15] S. Röttger and S. Zschaler. Model-Driven Development for Non-functional Properties: Refinement Through Model Transformation. In *Proceedings of the 7th International Conference on Unified Modelling Language: Modelling Languages and Applications (UML 2004)*, pages 275–289, Oct. 2004.

[16] J. A. Stankovic, R. Zhu, R. Poornalingam, C. Lu, Z. Yu, M. Humphrey, and B. Ellis. VEST: An Aspect-Based Composition Tool for Real-Time Systems. In *RTAS '03: Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, page 58, Washington, DC, USA, 2003. IEEE Computer Society.

[17] D. Wichadakul. *Q-Compiler: MetaData QoS-Aware Programming and Compilation Framework*. PhD thesis, University of Illinois at Urbana Champaign, 2003.