QUICKER: A Model-driven QoS Mapping Tool for QoS-enabled Component Middleware[*]

Amogh Kavimandan[†], Krishnakumar Balasubramanian, Nishanth Shankaran
Aniruddha Gokhale, Douglas C. Schmidt
Dept. of EECS, Vanderbilt University, Nashville
{amoghk,kitty,nshankar,gokhale,schmidt}@dre.vanderbilt.edu

Abstract

The development and operational life-cycles of distributed real-time and embedded (DRE) systems can be improved by using component middleware. The flexibility of component middleware, however, can also complicate DRE system development since system quality of service (QoS) now depends on how the middleware is configured. This paper provides three contributions to the study of QoS configuration in component-based DRE systems. First, we describe the challenges associated with mapping the platform-independent QoS policies of an application into platform-dependent values of QoS parameters used to configure the behavior of QoS-enabled component middleware. Second, we describe a novel approach that uses model-transformation to map these QoS policies onto component middleware QoS configuration parameters. Third, we demonstrate the use of model-checking to verify the properties of the transformation and automate the synthesis of configuration parameters required to tune the QoS-enabled component middleware. Our results indicate that model-transformation and model-checking provide significant benefits with respect to automation, reusability, verifiability, and scalability of the QoS mapping process compared with conventional middleware configuration techniques.

1 Introduction

QoS configuration challenges in component middleware. The success of component middleware technologies like Enterprise Java Beans (EJB) and CORBA Component Model (CCM) has raised the level of abstraction used to develop software for distributed real-time and embedded (DRE) systems, such as avionics mission-computing and shipboard computing systems. As a result, commercial-off-the-shelf (COTS) middleware, such as application servers and object request brokers (ORBs), now provides out-of-the-box support for traditional concerns affecting QoS in DRE system development, including multi-threading, assigning priorities to tasks, publish/subscribe event-driven communication mechanisms, security, and multiple scheduling algorithms. This support helps decouple application logic from QoS mechanisms (such as portable priority mapping, end-to-end priority propagation, thread pools, distributable threads and schedulers, request buffering, and managing event subscriptions and event delivery necessary to support the traditional concerns listed above), shields the developers from low-level OS specific details, and promotes more effective reuse of such mechanisms.

Although component middleware has helped move the configuration complexity away from the application logic, the middleware itself has become more complex to develop and configure properly. To achieve the desired QoS characteristics for

[*]This work was sponsored in part by grants from AFRL/IF Pollux project, Lockheed Martin ATL and Raytheon.
[†]Contact author

DRE systems, therefore, system developers and integrators must perform *QoS configuration* of the middleware. This process involves the binding of application level *QoS policies*—which are dictated by domain requirements—onto the solution space comprising the *QoS mechanisms* for tuning the underlying middleware. Examples of domain-level QoS policies include (1) the number of threads necessary to provide a service, (2) the priorities at which the different components should run, (3) the alternate protocols that can be used to request a service, and (4) the granularity of sharing among the application components of the underlying resources such as transport level connections.

QoS configuration bindings can be performed at several time scales, including *statically*, *e.g.*, directly hard coded into the application or middleware, *semi-statically*, *e.g.*, configured at deployment time using metadata descriptors, or *dynamically*, *e.g.*, by modifying QoS configurations at runtime. Regardless of the binding time, however, the following challenges must be addressed:

- The need to translate the domain-specific QoS policies of the application into QoS configuration options of the underlying middleware.
- The need to choose valid values for the selected set of QoS configuration options.
- The need to understand the dependency relationships and impact between the different QoS configuration options, both at individual component level (local) as well as at aggregate intermediate levels, such as component assemblies, through the entire application (global).
- The need to validate the local and global QoS configurations, which include the values, the dependency relationships, and the semantics of QoS configuration options at all times throughout the DRE system lifecycle.

Without effective tools to address these challenges, the result will be QoS mis-configurations that are hard to analyze and debug. As a result, failures will stem from a new class of configuration errors rather than (just) traditional design/implementation errors or resource failures.

Solution approach → Model-driven QoS mapping: To address QoS configuration challenges, we developed the *QUality of service pICKER* (QUICKER) model-driven engineering (MDE) toolchain. QUICKER extends the *Platform-Independent Component Modeling Language* (PICML) [2], which is a domain-specific modeling language (DSML) built using the *Generic Modeling Environment* (GME) [15]. GME is a meta-programmable modeling environment with a general-purpose editing engine, separate view-controller GUI, and a configurable persistence engine. Since GME is meta-programmable, the same environment used to define DSMLs is also used to build models, which are instances of the metamodels.

QUICKER enables developers of component-based DRE systems to annotate applications with QoS policies. These policies are specified at a higher-level of abstraction using *platform-independent* models, rather than using low-level platform-specific configuration options typically found in middleware configuration files. QUICKER thus allows flexibility in binding the same QoS policy to other middleware technologies.

Before the components in a DRE system can be deployed, however, their platform-independent QoS policies must be transformed into platform-specific configuration options. QUICKER therefore uses model-transformation techniques [4] to translate the platform-independent specifications of QoS policies into a platform-specific model defined using the *Component QoS Modeling Language* (CQML), which models the QoS configuration options required to implement the QoS policies of the application specified in PICML. Unlike PICML (whose models are platform-independent), CQML models are specific to the underlying middleware infrastructure (which in our case is Real-time CCM [5]).

QUICKER subsequently uses generative techniques on the CQML model to synthesize:

- The input to the Bogor [19] model-checking framework, which validates the transformation-generated application component-specific middleware QoS configuration and identifies all permissible changes to these configuration options that can be performed at runtime, while maintaining the validity of QoS configuration across the entire application, and
- The descriptors in a middleware-specific format (such as XML) required to configure the functional and QoS properties of the application in preparation for deployment in a target environment.

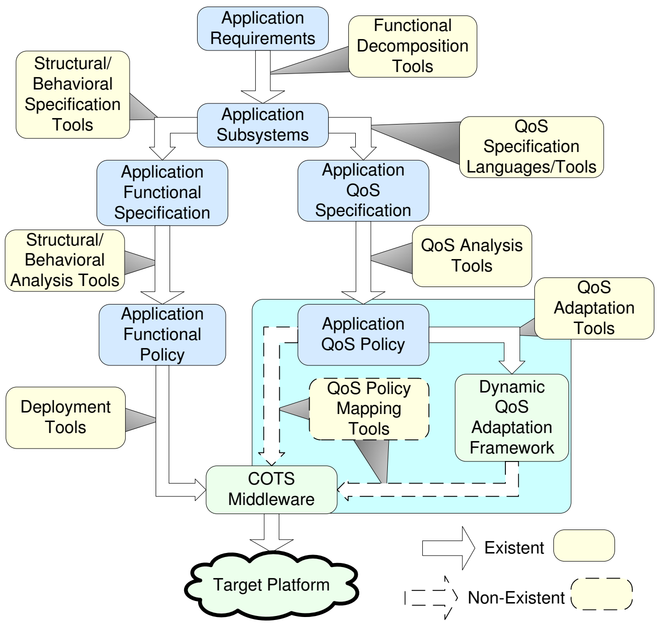

Application Requirements
Functional Decomposition Tools
Structural/ Behavioral Specification Tools
Application Subsystems
QoS Specification Languages/Tools
Application Functional Specification
Application QoS Specification
Structural/ Behavioral Analysis Tools
QoS Analysis Tools
QoS Adaptation Tools
Application Functional Policy
Application QoS Policy
Deployment Tools
QoS Policy Mapping Tools
Dynamic QoS Adaptation Framework
COTS Middleware
Target Platform
Existent
Non-Existent

Figure 1: QoS Mapping Landscape

QUICKER is designed to bridge the gap shown in Figure 1 between:

- **Functional specification and analysis tools**, such as PICML and Cadena [8], that allow specification and analysis of application structure and behavior,
- **Schedulability analysis tools**, such as TIMES [1], AIRES [12], VEST [21], that perform schedulability and timing analysis to determine the exact priorities and time periods for application components, and
- **Dynamic QoS adaptation frameworks**, such as the Resource Adaptation and Control Engine (RACE) [20] and QuO [24], that allocate resources to application components, monitor the QoS of the system continuously, and apply corrective control to modify the QoS configuration of the middleware at runtime.

By combining model-transformation and generative techniques with advanced model-checking technologies, QUICKER automates the mapping of QoS policies of applications to QoS configuration options for a specific middleware technology. In particular, QUICKER's separation of platform-independent and platform-dependent concerns enables the use of PICML models to specify QoS policies that can be mapped to other types of middleware, such as Web Services and Enterprise Java

Beans (EJB). QUICKER also helps ensure the validity of the values for the QoS configuration options, both at the individual component (local) level and at the aggregate application (global) level.

The remainder of this paper is organized as follows: Section 2 uses a NASA space mission DRE system built using QoS-enabled component middleware and MDE tools as a motivating example and describes the key challenges in QoS configuration of component middleware; Section 3 describes the QUICKER toolchain itself and shows how the combination of model-transformation and model-checking technologies helps resolve QoS configuration challenges in the NASA space mission application and DRE systems in general; Section 4 compares QUICKER with related work on model-driven QoS configuration/adaptation; and Section 5 presents concluding remarks.

2 Evaluating QoS Configuration Techniques for DRE Systems

This section uses NASA's Magnetospheric Multi-scale (MMS) space mission (`stp.gsfc.nasa.gov/missions/mms/mms.htm`) as an example to motivate the need for MDE tools like QUICKER that help automate key aspects of QoS configuration in DRE systems. Below, we focus on the payload sensor, data collection, and transmission challenges of this DRE system, which NASA is developing to study the microphysics of plasma processes.

2.1 DRE System Case Study

NASA's MMS mission is a representative DRE system consisting of several interacting subsystems (both in-flight and stationary) with a variety of complex QoS requirements. The MMS mission consists of four identical instrumented spacecrafts that maintain a specific formation while orbiting over a region of scientific interest. The primary function of the spacecraft(s) is to collect data while in orbit and send it to a ground station for further processing when appropriate.

The MMS mission dictates QoS requirements in 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.

In conjunction with colleagues at Lockheed Martin Advanced Technology Center (ATC), we have developed a prototype [22] of the data processing subsystem of this DRE system using the *Component-Integrated ACE ORB* (CIAO) [5] QoS-enabled component middleware framework, the RACE [20] dynamic QoS adaptation framework and the PICML [2] MDE tool. CIAO extends our previous work on *The ACE ORB* (TAO) by providing more powerful Real-time CCM component-based abstractions using the specification, validation, packaging, configuration, and deployment techniques defined by the

OMG CCM Deployment & Configuration specifications. Moreover, CIAO also integrates the CCM capabilities outlined above with TAO's Real-time CORBA features, such as portable priorities and end-to-end priority enforcement. RACE is an adaptive resource management framework built atop CIAO that integrates multiple resource management algorithms for (re-)deploying and managing the QoS of components in DRE systems.

Figure 2 shows the instances of—and connections between—software components 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.

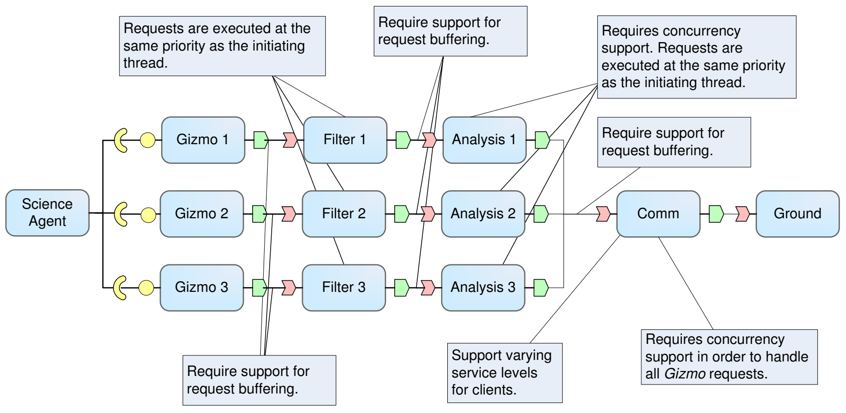

Requests are executed at the same priority as the initiating thread.
Require support for request buffering.
Requires concurrency support. Requests are executed at the same priority as the initiating thread.
Require support for request buffering.
Science Agent
Gizmo 1
Gizmo 2
Gizmo 3
Filter 1
Filter 2
Filter 3
Analysis 1
Analysis 2
Analysis 3
Comm
Ground
Require support for request buffering.
Support varying service levels for clients.
Requires concurrency support in order to handle all *Gizmo* requests.

Figure 2: MMS Mission System Components

The use of QoS-enabled component middleware and MDE tools provided several advantages during development of software components for our MMS mission prototype. For example, we modeled all components of the prototype using PICML, which (1) supported a high-level abstraction for describing the structure of the MMS scenario and (2) automated the generation of deployment metadata used to deploy the MMS components. Likewise, implementing the components with CIAO enhanced flexibility by supporting runtime component swapping that allowed runtime reconfiguration of the algorithms used by the *filter* and *analysis* components. Finally, using RACE to control the resource usage of the CIAO components allowed dynamic management of resources used by the *gizmo*, *filter*, and *analysis* components. Dynamic resource management helps our MMS prototype adapt to changes in mission goals as determined by the *science* components in response to changing conditions or as requested by explicit user commands.

2.2 QoS Configuration Challenges in the MMS mission

Although QoS-enabled component middleware like CIAO/RACE and MDE tools like PICML simplify many aspects of assembly, packaging, resource management, and deployment in DRE systems, the following key challenges remain with respect to the configuration of QoS for components (although our discussion focuses on the CIAO QoS-enabled component middleware, these challenges manifest themselves in any highly configurable component middleware including EJB and Microsoft .NET framework) that comprise the DRE systems, such as our MMS mission prototype:

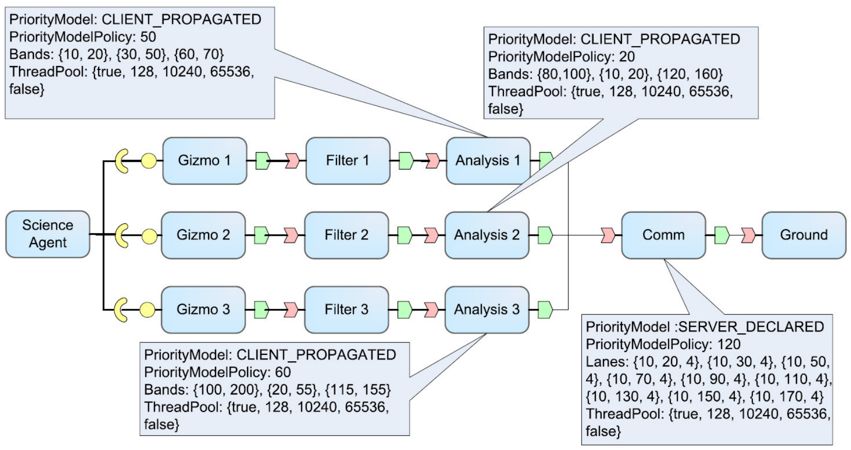
Challenge 1. Inherent complexity in translating QoS policies to QoS configuration options. Translating QoS policies into QoS configuration options is hard because it must transform semantics from the application domain to the semantics of the underlying component middleware. QoS-enabled component middleware like CIAO provides mechanisms to configure (1) *processor resources*, such as portable priorities, end-to-end priority propagation, thread pools, distributable threads and schedulers, (2) *communication resources*, such as protocol properties and explicit binding of connections, and (3) *memory resources*, such as buffering of requests. To translate the QoS policies into QoS mechanisms by configuring the QoS options, application developers need a thorough understanding of the underlying middleware platforms.

In our MMS mission prototype, for example, there is a QoS requirement that the *comm* component assign precedence to data originating from *gizmo* components that operate in burst mode. In case of a tie (*i.e.*, if more than one is operating in burst mode at the same time), requests are handled based on the importance of the originating *gizmo* component. As shown in Figure 3, one way to meet this requirement is to configure *gizmo*,*filter*, and *analysis* components with the CLIENT_PROPAGATED priority model policy, which is a Real-Time CORBA policy that ensures components execute requests at the priority determined by the origin of the request, *e.g.*, the *gizmo* component in the MMS mission. Likewise, the *comm* component can use the SERVER_DECLARED priority model with thread pool lanes. A thread pool lane corresponds to an OS priority level at which incoming requests are handled, and the SERVER_DECLARED priority model pre-allocates resources and handles requests at pre-determined priorities to ensure resource availability.

In contrast, if we choose CLIENT_PROPAGATED policy for the *comm* component, unbounded priority inversion [17] could occur since this component is shared between the different data flows originating at the *gizmo* components, each with differing importance operating in possibly different modes. We are therefore constrained to choosing the SERVER_DECLARED priority model for *comm* components. We also need to configure priority-banded connections between the *analysis* components and the *comm* components to ensure that all *analysis* component requests are routed to the appropriate priority lane defined inside the *comm* component.

While schedulability analysis might determine the right priority values for each component in the path of each control flow, the choice of QoS policies used to configure the middleware has a significant impact on the end result of satisfying QoS requirements. Without tool support, therefore, it is tedious and error-prone for a domain expert (*e.g.*, an MMS systems engineer) to translate QoS policies or analysis results to a subset of the QoS configuration options (*e.g.*, priority models, priority-bands, and thread pools) supported by the middleware that will ultimately impact the level of QoS achieved. Section 3.2 describes how the QUICKER MDE tool gathers QoS policies at a high-level of abstraction and uses model-transformation to map these policies automatically onto the QoS configuration of CIAO's Real-time CCM implementation, and also generates standard deployment descriptors.

Challenge 2. Ensuring validity of QoS configuration options. Assuming that a domain expert can translate the QoS policies into a subset of QoS configuration options, it is also necessary to understand the pre-conditions, invariants, and post-conditions of the different QoS configuration options since they affect middleware behavior. For example, to create a thread pool with lanes in the *comm* component, the following (non-exhaustive) are the pre-/post-conditions and invariants:



PriorityModel: CLIENT_PROPAGATED
PriorityModelPolicy: 50
Bands: {10, 20}, {30, 50}, {60, 70}
ThreadPool: {true, 128, 10240, 65536, false}
PriorityModel: CLIENT_PROPAGATED
PriorityModelPolicy: 20
Bands: {80,100}, {10, 20}, {120, 160}
ThreadPool: {true, 128, 10240, 65536, false}
Science Agent
Gizmo 1
Filter 1
Analysis 1
Gizmo 2
Filter 2
Analysis 2
Comm
Ground
Gizmo 3
Filter 3
Analysis 3
PriorityModel: CLIENT_PROPAGATED
PriorityModelPolicy: 60
Bands: {100, 200}, {20, 55}, {115, 155}
ThreadPool: {true, 128, 10240, 65536, false}
PriorityModel :SERVER_DECLARED
PriorityModelPolicy: 120
Lanes: {10, 20, 4}, {10, 30, 4}, {10, 50, 4}, {10, 70, 4}, {10, 90, 4}, {10, 110, 4}, {10, 130, 4}, {10, 150, 4}, {10, 170, 4}
ThreadPool: {true, 128, 10240, 65536, false}

Figure 3: MMS Mission System QoS configuration options

- *Pre-conditions.* A Real-time Portable Object Adapter created within a Real-time ORB is available, and the range of priorities (for the different lanes) and the type of priority mapping scheme chosen are compatible, *i.e.*, within the limits.
- *Post-conditions.* A thread pool with lanes corresponding to the different priorities, along with the requested number of static (pre-defined) threads is available for use.
- *Invariants.* The Real-time ORB will match incoming request priorities to the corresponding lanes, and will always handle incoming requests for higher priority lanes before incoming requests for lower priority lanes.

This problem is exacerbated by the plethora of options and choices of valid values for each option, as well as by the fact that choosing one value for a particular option may have side effects on other options. These side effects are sometimes manifested as overt failures, such as failure to perform a mapping of CORBA priority to OS priority because of insufficient priorities in the OS to support the choice of priority mapping scheme, *e.g.*, *direct* mapping. They may also be manifested, however, as hard-to-reproduce and/or debug runtime failures that only emerge during field testing, or after deployment, which are much harder to detect and fix.

In our MMS mission prototype, for example, the *analysis* components use priority-banded connections to ensure that end-to-end priority is preserved between the *gizmo* components and the *comm* components. This prioritization scheme (*i.e.*, giving precedence first to operational modes and then to importance) yields a design where separate priority-bands are defined for each mode. Within each priority-band, priorities are assigned to *analysis* components based on their relative importance.

For the *analysis* components to make invocations on the *comm* component at the right priority, the *comm* component must create thread pools with sufficient number of priority lanes. The *comm* components must also communicate the existence and availability of these lanes. Failure to configure the right set of options at both the *analysis* and *comm* components will cause requests to be handled at the wrong priority, potentially causing priority inversion, or worse, failing at the client side.

In summary, validating the values of the different QoS configuration options in isolation and together with connected components is critical to the successful deployment and ultimately the operation of DRE systems. Once again, it is hard to validate these values without automated tool support. Section 3.3 therefore describes how the QUICKER MDE tool checks the validity of the values for the different options and combinations of options according to the semantics of the underlying middleware, i.e., CIAO's Real-time CCM support.

Challenge 3. Resolving dependencies between QoS configuration options. Even with a thorough understanding of middleware QoS configuration options, manual configuration of QoS policies does not scale as the number of entities to configure increases. This lack of scalability stems from dependencies between the different QoS configuration options of each component, such as the dependency between the CORBA priority of a component, the chosen priority mapping scheme (to map CORBA priority to native OS priority), and the priority-banded connections policy (which selects the appropriate connection to route requests based on the request invocation priority). As the number of components increases, the number of intra-component dependencies increases proportionally. If the components are connected, the side effect of the connection between components may also induce an inter-component option dependency. Since these dependencies can grow quadratically, it is infeasible for developers to manage these dependencies manually.

In DRE systems with many components, the effects of changing a QoS configuration option on a component may affect many other directly connected components, their connected neighbors and so on. These dependencies can rapidly degenerate into a very large number of QoS configurations. Depending on the frequency of changes, empirically validating a change in QoS configuration options becomes time consuming at this scale, which slows down the design process considerably and permits subtle and pernicious errors to occur.

In our MMS mission prototype, for example, the priorities associated with the thread-pool lanes of the *comm* component should match the priority-bands defined on the *analysis* components. Since the *analysis* components themselves get their priority propagated from the *gizmo* components, there is a dependency between the *comm* and *gizmo* components even though they are not directly connected to each other. These dependencies are common throughout large-scale DRE systems, which makes it hard to manage the complexity manually as the number of components and number of QoS configuration options in a system grows.

Keeping track of dependencies between options and propagating the changes in one option to all options affected by that change is critical during the QoS configuration phase. What is needed is an automated tool support that can assure an application's evolution throughout its entire lifecycle. Section 3.3 therefore describes how the QUICKER MDE tool helps applications evolve by automatically (re-)calculating the dependencies between options, and can thus be used repeatedly by DRE developers during the entire lifecycle.

Challenge 4. Ensuring validity of QoS configuration options with changes in QoS policies. QoS configuration options effect the non-functional behavior of a system, and thus are affected by changes in the system environment. For a DRE system to operate effectively in hostile environments, such as space missions, component middleware and their associated QoS configuration options may need to adapt to their current conditions. Middleware that can only be configured statically (*i.e.*, at design- or installation-time)—but does not allow dynamic reconfiguration—may be of limited use in these scenarios.

While it is useful to change QoS configuration options at runtime to effect changes in behavior (such as re-prioritizing or increasing/decreasing resource usage), such dynamic reconfigurations may incur another set of challenges. In particular, not only must we handle static QoS configuration problems (such as checking validity of values and keeping track of dependencies), there is typically little leeway to accommodate misconfiguration at runtime. It is non-trivial to change a running system because the system might crash during reconfiguration due to misconfiguration of QoS options. Moreover, the reconfiguration process itself must be predictable for the reconfiguration to have the desired effect on system behavior. In a DRE system, for instance, a reconfiguration done too late may be worse than not performing a reconfiguration at all.

In our MMS mission prototype, for example, the *science* agent(s) on all spacecrafts have mission goals that represent requests from users or other *science* agents for the times and types of data to acquire. Such changes in mission goals require dynamic reconfiguration, which in turn can trigger changes in QoS configuration, such as modifying the relative importance assigned to the *gizmo* components. Depending on the nature and extent of the changes, dependent components of the *gizmo* component may be reconfigured using the available options, along with re-validating the values chosen for the options. For instance, the size of the buffers in the *comm* agent corresponding to the data collected from the different *gizmo* components, may need reconfiguration to accommodate changes in the relative importance of the *gizmo* components.

Such exhaustive evaluation of possible choices of QoS configuration options and validation of the reconfigured state is too time consuming to perform at runtime and can delay the reconfiguration process itself, rendering it useless. Once again, tools are needed to help validate and automate this reconfiguration process. Section 3.3 therefore describes how the QUICKER MDE tool helps evaluate possible choices of QoS configuration at design-time using model-checking, and shows how results of this design-time evaluation of possible choices can be used to select runtime QoS configurations by the RACE dynamic QoS adaptation framework.

Due to the challenges described above, significant manual effort is typically expended on QoS configurations for DRE systems like our MMS mission prototype. Even after much effort, it is common to identify issues like frame overruns during integration testing, which results in increasing the cost of development of DRE systems. In some cases, the problems are discovered after the system has been deployed. It is therefore critical that QoS configuration of component middleware is performed via adequate tool support. The remainder of this paper shows how our QUICKER toolchain helps address these challenges in the context of the CIAO and RACE QoS-enabled component middleware.

3 The Quality of Service Picker (QUICKER) Toolchain

This section describes the QUICKER toolchain. It explains how QUICKER addresses the QoS mapping challenges outlined in Section 2.2 by using (1) model-to-model transformations of the user QoS policies into middleware-specific configuration options and (2) the Bogor [19] model-checking framework to define composite Real-time CCM-specific validation language constructs for validating the QoS options generated by the transformation.

3.1 Overview of the QUICKER Toolchain

The architecture of QUICKER is shown in Figure 4. It enhances the *Platform Independent Component Modeling Language* (PICML) [2] with new constructs that enable developers of DRE systems to specify and analyze application QoS policies[1] at a higher level of abstraction than used by third-generation programming languages or even textual declarative notations, such as XML. Our goal was to transform these policies into the appropriate middleware-specific configuration options in a manner that satisfies the following objectives: (1) generated QoS options should be valid mappings of user-specified application QoS policies and (2) generated QoS options should themselves be models since this allows further analysis/transformations by other tools, such as model-checkers.

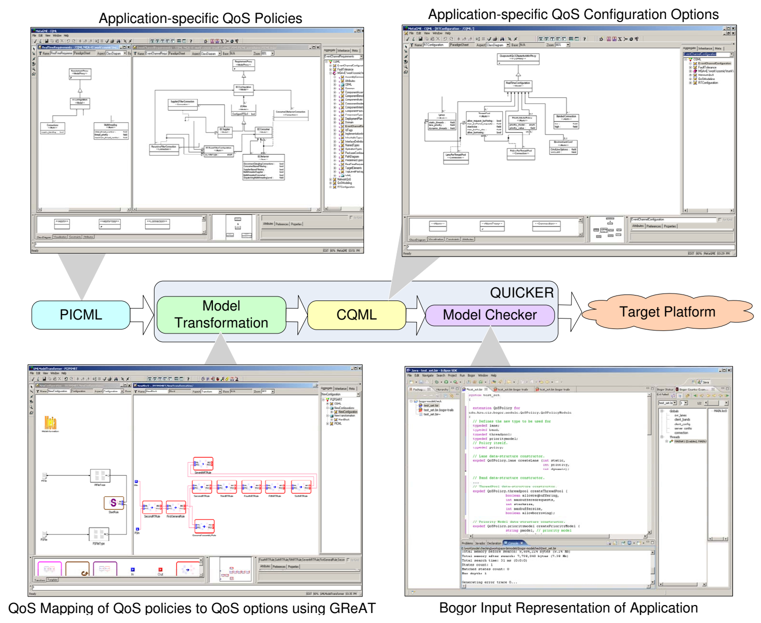

Application-specific QoS Policies
Application-specific QoS Configuration Options
PICML
Model Transformation
CQML
QUICKER
Model Checker
Target Platform
QoS Mapping of QoS policies to QoS options using GReAT
Bogor Input Representation of Application

Figure 4: QUality of service pICKER (QUICKER)

To achieve these objectives we used the *Graph Rewriting and Transformation* (GReAT) [11] tool to transform platform-independent QoS policies captured in PICML (the input) to platform-specific QoS configuration options captured in CQML (the output). PICML allows specification of application QoS policies at a high-level of abstraction, *i.e.*, focusing on desired QoS features/characteristics at the granularity of individual components and/or assemblies of application components. For example, models created in PICML represent answers to questions like:

[1]Unless stated otherwise, our use of PICML in this paper refers to its QoS policy specification capabilities.

1. Is the component primarily being used as a *service provider* (*i.e.*, a *server*), a *service consumer* (*i.e.*, a *client*), or both?
2. Would the component service requests from multiple clients simultaneously? If so, what is the minimum and maximum number of clients that it is expected to service simultaneously? Would the requests have to be prioritized?
3. In the *server* role, would the component require execution of the requests at a fixed priority or a varying priority?
4. In the *server* role, should the component buffer requests when resources (threads) are not immediately available to honor the requests?
5. In the *server* role, would the component require support for varying service levels for different clients?

In contrast, the output language (*i.e.*, CQML) of the QUICKER transformation models CCM-specific QoS configuration options for component-based applications. For example, CQML models include Real-time CCM QoS options, such as (1) *Threadpool*, which specifies the overall concurrency level in the ORB, (2) *Priority Model Policy*, which specifies the priority propagation mechanism used to execute requests at the *server* component in an end-to-end fashion, and (3) *Priority-Banded Connections*, which allow *server* components to specify explicit priorities for each connection and lets *client* components choose appropriate connections for making requests based on their invocation priorities. Each of these elements can be further divided into client- and/or server-side policies, as shown in Table 1.

Table 1: Attribute Mappings in Model Transformation

Real-time CCM Configuration Options	Option Scope	Transformation Defined?	Deduced from Input Model?
Priority Banded Connections	client		
Low		no	yes
High		no	yes
Priority Lanes	server		
Static threads		no	yes
Lane priority		no	yes
Dynamic threads		no	yes
ThreadPool	server		
Stacksize		yes	no
Allow request buffering		no	yes
Allow borrowing		yes	no
Max. buffered requests		yes	no
Max. buffer size		yes	no
Priority Model Policy	server		
Priority model		no	yes
Server priority		no	yes

To perform validity checks on the GReAT-generated CQML model, we used Bogor, which is an extensible software model-checking framework whose model-checking algorithms, visualizations, and user interface support both general-purpose and domain-specific software model-checking. The input to the Bogor model-checker is generated from the CQML model using interpreters developed as part of QUICKER. Since Bogor's input language does not directly support modeling of CCM components and QoS parameters, we extended its input language to implement these artifacts.

The remainder of this section describes QUICKER's model-transformation and model-checking capabilities in more depth.

3.2 QoS Mapping using Model Transformations

The QUICKER transformation engine uses GReAT to convert the application QoS policies in a PICML model into a CQML model. This automated conversion is an example of a *vertical exogenous* transformation [3, 11]. This type of transformation starts with an abstract type graph as the input and refines the graph by adding details to generate a more detailed type graph as the output [3, 11].

To maintain platform-independence, the QoS policy specification capabilities in PICML are defined at a higher level of abstraction than the underlying middleware-specific options. For the QoS configuration options that cannot be deduced from the input PICML model, the transformation *fills in* default values to generate a valid output model in CQML. Table 1 captures the attribute mappings in the model transformation: QoS configuration elements generated either directly or deduced from input model, and those that were provided by the transformation rules.

The steps involved (non-exhaustive) in a typical QUICKER model transformation are as follows:

a. Navigate the hierarchy of the input model and create the component assembly according to the structure read from the input model. This step is required when the configuration values are later checked for validity by Bogor.

b. For each *type* of QoS policy defined on a component, generate a corresponding element for CQML as follows: (a) if the component QoS policy specifies support for multiple clients simultaneously, create server-side configuration elements that specify concurrency level in the ORB, (b) if the execution of requests at a `server` component is required to be at a fixed priority, set *Priority Model Policy* to SERVER_DECLARED, otherwise set it to CLIENT_PROPAGATED, and (c) if varying service levels should be supported at the `server`, create appropriate number of `Priority Lanes` for that component, and additionally create *Priority Banded Connections* for the `client(s)` of that component.

c. Populate the individual attributes of the newly created elements, if not done in earlier steps, and create appropriate associations for each CQML element created in the two steps above.

Resolving Challenge 1: Translating QoS policies to QoS configuration options. QUICKER gathers the application QoS policies at the domain-level abstraction and uses model-transformation to automate the tedious and error-prone translation of QoS policies to the appropriate subset of QoS configuration options. For example, the PICML model of the MMS mission captures the following types of information about the application:

- The *Comm*, *Gizmo*, *Filter* and *Analysis* components act both as a *client* and a *server*, and all service multiple requests simultaneously
- In the *server* role, *Gizmo*, *Filter* and *Analysis* components execute requests at varying priorities such that the *Comm* component may assign precedence to the data originating at the *Gizmo* component (as mentioned in section 2.2)
- All components except *Ground* require buffering of client requests

After capturing the QoS policies for all the MMS mission components in a PICML model, QUICKER transforms this model and generates the QoS configuration options in the form of a CQML model. Figure 3 shows the generated QoS configuration options for *Analysis* and *Comm* components that satisfy the QoS policies of these components.

3.3 Validating QoS Configuration Options using Model-Checking

After the model-transformation portion of the QUICKER toolchain generates a CQML model comprising the QoS configuration options, the correctness of these options must be validated before the application assembly is deployed. We validate these options using the Bogor model-checking framework, which is a customizable explicit-state model checker implemented as an Eclipse plugin.

Validating a system using Bogor involves defining (1) a model of the system using the *Bogor Input Representation* (BIR) language and (2) the *property* (*i.e.*, specification) that the model is expected to satisfy. Bogor then traverses the system model and checks whether or not the property holds. To validate QoS configuration options of an application using Bogor, therefore, we need to specify the application model and the QoS configuration options that the application is expected to satisfy. To express this specification in the BIR input language, we leveraged Bogor's customization features to define new types and primitives that allow manipulation of the newly defined types. The remainder of this section describes QUICKER's model-checking capabilities and shows how our BIR extensions help resolve the challenges described in Section 2.2 related to validation, option dependency tracking, and providing input to dynamic QoS adaptation frameworks.

```
extension QoSOptions for
edu.ksu.cis.bogor.module.QoSOptions.QoSOptionsModule
{
  // Defines the new type to be used for
  typedef lane;
  typedef band;
  typedef threadpool;
  typedef prioritymodel;
  typedef policy;
  // Lane constructor.
  expdef QoSOptions.lane createLane (
   int static, int priority, int dynamic);
  // ThreadPool constructor.
  expdef QoSOptions.threadpool
  createThreadPool (boolean allowreqbuffering,
   int maxbufferedrequests, int stacksize, int
   maxbuffersize, boolean allowborrowing);
  // Set the band(s) for QoS policy.
  actiondef registerBands (QoSOptions.policy
   policy, QoSOptions.band ...);
  // Set the lane(s) for QoS policy.
  actiondef registerLanes (QoSOptions.policy
   policy, QoSOptions.lane ...);
  ...
}
extension Quicker for
edu.ksu.cis.bogor.module.Quicker
{
  // Defines the new type.
  typedef Component;
  // Component Constructor.
  expdef Quicker.Component
  createComponent (string component);
  // Set the QoS policy for the component.
  actiondef registerQoSOptions (Quicker.Component
    component,QoSOptions.policy policy);
  // Make connections between components.
  actiondef connectComponents (Quicker.Component
    server,Quicker.Component client);
  ...
}
```

Listing 1: QUICKER BIR Extension

• **BIR input extensions.** Bogor provides the BIR language to specify input to its model-checker. It is cumbersome, however, to describe middleware QoS configuration options using the default capabilities of BIR since (1) this representation is at a much lower level of abstraction compared to domain-level concepts, such as components and QoS options, that we want to model-check, and (2) specifying middleware QoS configuration options using BIR's low-level constructs can yield an unmanageably large state space since representing domain-level concepts with a low-level BIR specification requires additional auxiliary state that may be irrelevant to the properties being model-checked [19]. To specify and model-check properties more closely to the domain of component middleware QoS configuration options, therefore, we used BIR's input extension feature to define composite constructs that represent concepts (such as components) and QoS options (such as thread pools) as though they were native BIR constructs.

Listing 1 shows an example of our BIR extensions to represent QoS configuration options in middleware. The extensions shown in the listing define two new data types: `Component`, which corresponds to a CCM component, and `QoSOptions`, which captures QoS configuration options, such as `lane`, `band`, and `threadpool`. Each `Component` maintains a single `QoSOptions` instance internally. Depending on a component's role (*i.e.*, `client` or `server`), various individual QoS options may need to be set. Since these BIR extensions allow representation of middleware QoS options, we refer to them as *QoS extensions*.

• **BIR primitives.** In addition to defining constructs that represent domain concepts, such as components and QoS options, we also need to specify the *property* that the application should satisfy. Since a *property* can be denoted by multiple QoS options, we define *rules* to capture valid option values. BIR primitives are used to express these rules in the input specification of MMS mission prototype, as shown in Listing 2.

```
// Declaration of the extensions
// Declare all data-type variables to be used in this
// file
loc loc0: live {} //
  do
  {
    // Instantiate the components with QoSPolicies;
    // Register policies with components; Register
    // components as per assembly structure in CQML model
    ...
  } goto loc1;
loc loc1: live {pm}
  when ! Quicker.hasQoSOptions (Comm)
  do
  {
    pm := Quicker.getPriorityModelPolicy (Comm);
  } goto loc2;
loc loc2: live {pm}
  do
  {
    when (pm == pmodel.SERVER_DECLARED) do {} goto loc3;
    when !(pm == pmodel.SERVER_DECLARED) do {} goto loc12;
  } goto loc30;
loc loc3: live {lr1}
  do
  {
    lr1 := QosOptions.getLowerBound (Analysis_1);
  } goto loc4;
    ...
loc loc8: live {hr3}
  do
  {
    lr1 := QosOptions.getUpperBound (Analysis_3);
  } goto loc9;
  ...
```

Listing 2: Application-specific BIR Primitives

Primitives are the language constructs we defined to access and manipulate data types for Real-time CCM. For example, instances of `QoSOptions` can be created and manipulated using various primitives defined in Listing 2. The rule defined in this listing first determines the *Priority Model* of a particular component using the `getPriorityModelPolicy` primitive. If the *Priority Model* policy of a component is SERVER_DECLARED, the rule calculates the range of connected component's *PriorityBand* values using `getLowerBound` and `getUpperBound` primitives. Finally, this rule also checks if the range calculated above matches the priority associated with *Comm* component's thread pool (not shown in the listing).

Primitives are also used to capture component interconnections in BIR format. These interconnections are needed to construct the dependency structure for the specified input application, such as the MMS mission prototype. They are also used later during validation of options for connected components.

• **Generative capabilities.** Applications that need to be model-checked by Bogor must be represented in BIR format. Writing and maintaining BIR manually can be tedious and error-prone for domain experts (*e.g.*, spacecraft engineers) since configuring application QoS policies is typically done iteratively. Depending on the number of components and the number of available configuration options, manual processes do not scale well.

To automate the process of creating BIR specification of applications, we therefore used the generative capabilities in GME to automatically generate BIR specification of an application from its CQML model. This generative process is done in GME using a model interpreter that traverses the CQML model and generates a BIR file that capture the application structure and its QoS properties. The QUICKER toolchain therefore automates the entire process of mapping application QoS policies to middleware QoS options, as well as converting these QoS options into BIR. A second model interpreter is used to generate the Real-time CCM-specific descriptors required to configure functional and QoS properties of an application and deploy it in its target environment.

Resolving Challenge 2: Ensuring validity of QoS configuration options. A priority-banded connection between *Analysis* and *Comm* components must have matching policies and values, as discussed in Section 2.2. QUICKER uses BIR primitives to ensure the validity of QoS configuration options for MMS mission components. To show how the primitives (and rules defined using those primitives) can ensure valid QoS configuration options, we deliberately introduced a QoS misconfiguration by specifying the concurrency model of *Comm* component to use *Thread Pool* without *Lanes* in the PICML model of MMS mission prototype. The same configuration is contained in the (transformation-generated) CQML model of the application, which is used to generate the BIR specification.

One of the rules specified in Listing 2 uses BIR primitives to validate that the banded connection between *Comm* and *Analysis* components have matching priority values. The change to QoS option (of *Comm* component) outlined above therefore causes a misconfiguration, which is detected by the rule in Listing 2. Rules defined using BIR primitives are thus used in QUICKER to provide an automated tool support for ensuring the validity of QoS configuration options.

Resolving Challenge 3: Resolving dependencies between QoS configuration options. There is a dependency between *Gizmo* components and *Comm* component in the MMS mission prototype, as explained in Section 2.2, *i.e.*, the *Gizmo* component invocation priority values should match the thread pool lane priority values. The dependency structure of the MMS mission prototype is maintained in QoS extensions to track such dependencies between QoS options. When a change occurs to either of the dependent QoS options (*i.e.*, the thread pool lanes of *Comm* component and the invocation priority of the *Gizmo* component), the QoS extensions detect mismatches between the priority values.

Even with a thorough understanding of middleware QoS configuration options, it is tedious and error-prone to track these dependencies and detect mis-configurations. As the number of components and component interconnections increases, this problem is exacerbated. QUICKER helps developers automatically track these dependencies throughout the evolution of an application.

Resolving Challenge 4: Ensuring validity of QoS configuration options with changes in QoS policies. A dynamic reconfiguration that triggers changes in QoS options of *Gizmo* components necessitates reconfiguration of components dependent on *Gizmo* and revalidation of the configuration, as discussed in Section 2.2. Our QoS extensions explore the possible states of an application and generate a set of valid application states. Since we are only concerned with the QoS configuration options, we use the term "application states" to represent just the set of QoS configuration option values for all the components in an application.

Figure 5 shows the generated states for the MMS mission prototype. For simplicity, this figure only shows the generated states for *Analysis* and *Comm* components. For our explanation purposes, it is assumed that the rest of the application state is valid and does not change. The server QoS options in Figure 5 are denoted by an *n*-tuple consisting of: (1) `Priority Model` (SERVER_DECLARED, or CLIENT_PROPAGATED), (2) `Priority Value` (RTCORBA::Priority), (3) set of *l* `Lane` configurations (4) `ThreadPool` configuration. Similarly, client QoS options are denoted as *m*-tuple consisting of all the priority `Band` configurations for a client.

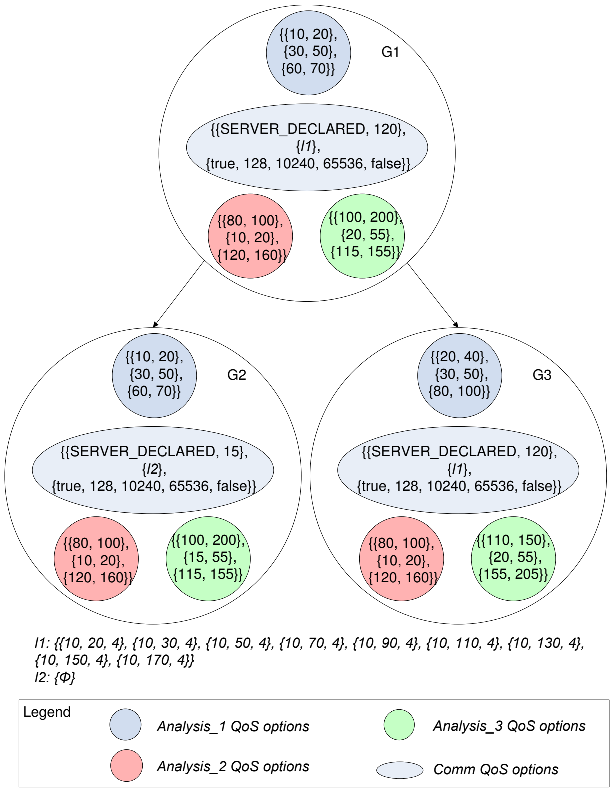


Figure 5: Snapshot of QoS Configuration States

In the MMS mission prototype specification, if new mission goals require changes in QoS options of the *Comm* and *Analysis_3* components the application state could change from initial state G1 to G2. Likewise, the state could change to G3, when new mission goals dictate that priority `band` configuration for *Analysis_1* and *Analysis_2* components should change. From each of these states, the system could enter a number of possible states depending on further changes in QoS configuration. Each of G1, G2, and G3 (along with other generated states) are valid application states. By exploring all the possible application states, QoS extensions generate validated states of an application at design-time, which can be used to select runtime QoS configurations by the RACE QoS adaptation framework.

By exploring all the possible states of an application, QoS extensions identify both the set of valid and invalid application states. We can then construct an automaton using the chain of configuration modifications (generated by the model-checker) from the application state corresponding to the initial QoS configuration that lead to the invalid state. This automaton can guide the behavior of the RACE controller that adapts configuration options dynamically, to ensure that reconfiguration will not yield an invalid application state.

4 Related Work

This section compares our work on QUICKER with related work on applying model-driven engineering techniques for QoS configuration and adaptation of DRE systems. We categorize this work into four categories: (1) tools that support analysis of functional behavior, including control and data dependencies in component-based systems, (2) tools that model QoS adaptations, (3) tools that allow specification of application QoS, and (4) tools that analyze QoS properties like schedulability, timing, and power.

4.1 Functional Specification and Analysis Tools

Cadena [8] is an integrated environment built using Eclipse for building and analyzing CCM based systems. Cadena provides a framework for lightweight dependency analysis (including both intra-component and inter-component) of behavior of components. Cadena also supports an integrated model-checking infrastructure (using Bogor) dedicated to checking global system properties using event-based inter-component communication via real-time middleware [6]. QUICKER is similar to Cadena in terms of usage of Bogor for model-checking. The difference is that Cadena applies model-checking to verify functional behavior of components, whereas QUICKER applies model-checking to verify QoS configuration options of component middleware in the presence of dynamic adaptation of these options via RACE.

4.2 QoS Adaptation Modeling Tools

The *Distributed QoS modeling environment* (DQME) [23] is a DSML that enables the design of QoS adaptive applications in combination with using QoS provisioning frameworks, such as QuO [24]. DQME uses a hierarchical representation for modeling QoS adaptation strategies and supports design of controllers based on state machines. The primary difference is that DQME focuses on a high-level design of QoS adaptation strategies, whereas QUICKER's emphasis is more fine-grained and focuses on the runtime configuration options of the underlying middleware. Operating at a high-level of abstraction with respect to QoS adaptation strategies ultimately requires mapping of the design adaptation strategies to implementation-specific options. QUICKER focuses on translating high-level QoS adaptation design intent into actual QoS configuration options that exists in tools like DQME. QUICKER also helps configure QoS adaptation strategies dynamically at runtime by feeding RACE information about valid QoS configuration states from the analysis results obtained using model-checking.

4.3 QoS Specification Tools

Other related work focuses on the specification of application-specific QoS properties. For example, Ritter *et.al.* [18] describe CCM extensions for generic QoS support and discusses a QoS metamodel that supports domain-specific multi-category QoS contracts. The QML QoS specification language [7] specifies component-level QoS properties and is used in [9] to capture user requirements that are translated into corresponding network and system parameters. The work in [10] 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. In contrast to the projects and tools described above, QUICKER focuses on automating the error-prone activity of mapping platform-independent QoS policies to middleware-specific QoS configuration options. Representing QoS policies as model elements allows for a unified (with functional aspects of the application) and flexible QoS specification mechanism, while automating evolution of the QoS policies with application evolution; the platform-independent QoS policies also allow configurable re-targeting of the QoS mapping to support other types of middleware technologies. Since QUICKER is integrated with a model-checking tool, it also enables evaluation of dynamic QoS adaptation in an application-specific manner as well as passing the results to QoS adaptation frameworks. Although [10] supports model-based verification techniques, their focus is on monitoring deadline violations in executable specifications.

4.4 Schedulability Analysis Tools

Research presented in [16] maps application models captured in the *Embedded Systems Modeling Language* (ESML) to UPPAAL timed automata [14] using graph transformation to verify properties like schedulability of a set of real-time tasks with both time- and event-driven interactions, and absence of deadlocks in the system. Other related efforts include the *Virginia Embedded Systems Toolkit* (VEST) [21] and the *Automatic Integration of Reusable Embedded Systems* (AIRES) [12], which are model-driven analysis tools that evaluate whether certain timing, memory, power, and cost constraints of real-time and embedded applications are satisfied. As shown in Figure 1, QUICKER focuses on a different level of abstraction (*i.e.*, QoS policy mapping tools) than [16, 21, 12] (which are QoS analysis tools). Our approach in QUICKER is thus complementary to these efforts since QUICKER places emphasis on mechanisms to (1) translate design-intent into actual configuration options of underlying middleware and (2) verify that both the transformation and subsequent modifications to the configuration options remain semantically valid.

5 Concluding Remarks

With the trend towards implementing key DRE system infrastructure at the middleware level, achieving the desired QoS is increasingly becoming more of a configuration problem than (just) a development problem. The flexibility of configuration options in QoS-enabled component middleware, however, has created a new set of challenges. Key challenges include determining the correct set of values for the configuration options, understanding the dependency relations between the different options, and evolving the QoS configurations with changes to application functionality.

To address these challenges, we have developed the *QUality of service pICKER* (QUICKER) toolchain, which (1) uses model-transformation to automate the mapping of application QoS policies into middleware-specific QoS configuration options and (2) applies model-checking to ensure that the QoS configuration options are valid at the individual component level as well as at the application level. To demonstrate the use of QUICKER, we applied it to address configuration challenges in a prototype of NASA's MMS space mission developed using the CIAO and RACE QoS-enabled component middleware. Using QoS requirements of the MMS mission as a motivating example, we showed how QUICKER's QoS mapping capabilities and validation of QoS options using model-checking enabled the successful configuration and deployment of the MMS space mission components. The following is a summary of lessons learned from our experience using QUICKER to develop this prototype:

• **QoS mapping is critical to successful deployment of systems built using component middleware.** With the increase in configuration complexity, the QoS mapping capabilities provided by QUICKER are essential to managing the complexity. By employing MDE tools, QUICKER not only simplifies the QoS mapping process for DRE system developers, it also preserves the rich semantics associated with the mapping between the QoS policies and QoS configuration options at this level.

• **Integration of QoS mapping with runtime entities like runtime QoS controllers essential to ensure dynamic configuration.** In addition to QUICKER toolchain capabilities described in this paper, our ultimate goal is to provide inputs to runtime QoS controllers, such as those in RACE. The current version of the RACE controller [20] performs coarse-grained control of CCM components by changing component priorities to effect control. To enable fine-grained control of CCM components, therefore, we are extending the QUICKER toolchain to incorporate a cost model for dynamic resource adaptation and automatic generation of a RACE controller based on results of the Bogor model-checker.

• **Horizontal mapping of QoS is as important as vertical QoS mapping.** QUICKER currently focuses on mapping application QoS policies onto a single underlying middleware technology: the CIAO and RACE RT-CCM platform. Large-scale DRE systems—particularly systems requiring dynamic resource management [13]—are often composed of heterogeneous technologies. It is therefore essential for QoS mapping tools to not only support *vertical mapping* (*i.e.*, the mapping of policies and validation onto a single technology) but also *horizontal mapping* (*i.e.*, the mapping of QoS policies onto multiple heterogeneous technologies, while reconciling the differences between these technologies). Until such mapping is performed, QoS configuration and associated tools will remain as islands, which significantly complicates integration efforts for large-scale DRE systems.

QUICKER is available as open-source from `www.dre.vanderbilt.edu/CoSMIC/`.

References

[1] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES: A Tool for Schedulability Analysis and Code Generation of Real-Time Systems. In K. G. Larsen and P. Niebert, editors, *Formal Modeling and Analysis of Timed Systems: First International Workshop, FORMATS 2003, Marseille, France, September 6-7, 2003. Revised Papers*, volume 2791 of *Lecture Notes in Computer Science*, pages 60–72. Springer, 2003.

[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 *Proceedings of the 11th Real-time Technology and Application Symposium (RTAS '05)*, pages 190–199, San Francisco, CA, Mar. 2005. IEEE.

[3] L. Baresi, R. Heckel, S. Thöne, and D. Varró. Style-Based Refinement of Dynamic Software Architectures. In *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture*, Oslo, Norway, June 2004. IFIP.

[4] K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Syst. J.*, 45(3):621–645, 2006.

[5] G. Deng, C. Gill, D. C. Schmidt, and N. Wang. QoS-enabled Component Middleware for Distributed Real-Time and Embedded Systems. In I. Lee, J. Leung, and S. Son, editors, *Handbook of Real-Time and Embedded Systems*. CRC Press, 2007.

[6] X. Deng, M. B. Dwyer, J. Hatcliff, G. Jung, Robby, and G. Singh. Model-Checking Middleware-Based Event-Driven Real-time Embedded Software. In *FMCO*, pages 154–181, 2002.

[7] S. Frolund and J. Koistinen. Quality of Service Specification in Distributed Object Systems. *IEE/BCS Distributed Systems Engineering Journal*, 5:179–202, Dec. 1998.

[8] 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.

[9] C. Hesselman, I. Widya, A. van Halteren, and B. Nieuwenhuis. Middleware Support for Media Streaming Establishment Driven by User-Oriented QoS Requirements. In *Lecture Notes in Computer Science: Interactive Distributed Multimedia Systems and Telecommunication Services (IDMS'00)*, volume 1905, pages 158–171, Enschede, The Netherlands, Oct. 2000. Springer-Verlag.

[10] 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.

[11] 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`.

[12] S. Kodase, S. Wang, Z. Gu, and K. G. Shin. Improving Scalability of Task Allocation and Scheduling in Large Distributed Real-time Systems using Shared Buffers. In *Proceedings of the 9th Real-time/Embedded Technology and Applications Symposium (RTAS)*, Washington, DC, May 2003. IEEE.

[13] P. Lardieri, J. Balasubramanian, D. C. Schmidt, G. Thaker, A. Gokhale, and T. Damiano. A Multi-layered Resource Management Framework for Dynamic Resource Management in Enterprise DRE Systems. *Journal of Systems and Software: special issue on Dynamic Resource Management in Distributed Real-time Systems*, 2007 (to appear).

[14] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct. 1997.

[15] 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.

[16] G. Madl, S. Abdelwahed, and G. Karsai. Automatic Verification of Component-Based Real-time CORBA Applications. In *The 25th IEEE Real-time Systems Symposium (RTSS'04)*, Lisbon, Portugal, Dec. 2004.

[17] I. Pyarali, D. C. Schmidt, and R. Cytron. Techniques for Enhancing Real-time CORBA Quality of Service. *IEEE Proceedings Special Issue on Real-time Systems*, 91(7), July 2003.

[18] 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, Software Technology Track, Distributed Object and Component-based Software Systems Minitrack, HICSS 2003*, Honolulu, HW, Jan. 2003. HICSS.

[19] Robby, M. Dwyer, and J. Hatcliff. Bogor: An Extensible and Highly-Modular Model Checking Framework. In *Proceedings of the 4th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003)*, Helsinki, Finland, September 2003. ACM.

[20] N. Shankaran, D. C. Schmidt, Y. Chen, X. Koutsoukous, and C. Lu. The Design and Performance of Configurable Component Middleware for End-to-End Adaptation of Distributed Real-time Embedded Systems. In *Submitted to the 10th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing*, Santorini Island, Greece, May 2007.

[21] 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 *Proceedings of the IEEE Real-time Applications Symposium*, pages 58–69, Washington, DC, May 2003. IEEE.

[22] D. Suri, A. Howell, N. Shankaran, J. Kinnebrew, W. Otte, D. C. Schmidt, and G. Biswas. Onboard Processing using the Adaptive Network Architecture. In *Proceedings of the Sixth Annual NASA Earth Science Technology Conference*, College Park, MD, June 2006.

[23] J. Ye, J. Loyall, R. Shapiro, R. Schantz, S. Neema, S. Abdelwahed, N. Mahadevan, M. Koets, and D. Varner. A Model-Based Approach to Designing QoS Adaptive Applications. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 221–230, Washington, DC, USA, 2004. IEEE Computer Society.

[24] J. A. Zinky, D. E. Bakken, and R. Schantz. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems*, 3(1):1–20, 1997.