# 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

*This paper provides three contributions to the study of quality of service (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. 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 DRE systems, therefore, system developers and integrators must perform *QoS configuration* of the middleware.

* This work was sponsored in part by the AFRL/IF Pollux project, Lockheed Martin ATL, Raytheon, and the DARPA ARMS project.

QoS configuration involves binding application level *QoS policies*—which are dictated by domain requirements—onto *QoS mechanisms* that tune 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 different components should run, (3) the alternate protocols that can be used to request a service, and (4) the granularity at which application components share 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 between the different QoS configuration options, both at individual component level (local), as well as at aggregate intermediate levels (such as component subsystem assemblies) through the entire application (global).

Without effective tools to address these challenges, the result will be misconfigured QoS policies that are hard to analyze and debug. As a result, failures will stem from a new class of configuration errors rather than 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 Environ-*

*ment* (GME) [10].

QUICKER is designed to bridge the gap between:

- **Functional specification and analysis tools**, such as PICML and Cadena [5], that allow specification and analysis of application structure and behavior,
- **Schedulability analysis tools**, such as TIMES [1], AIRES [8], VEST [17], 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) [16] and QuO [20], 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.

## 2. Evaluating QoS Configuration Techniques for DRE Systems

This section uses NASA's Magnetospheric Multiscale (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.

### 2.1. DRE System Case Study

NASA's MMS mission is a representative DRE system consisting of several interacting subsystems (both on-board and ground systems) 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 constellation 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 interesting events are 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

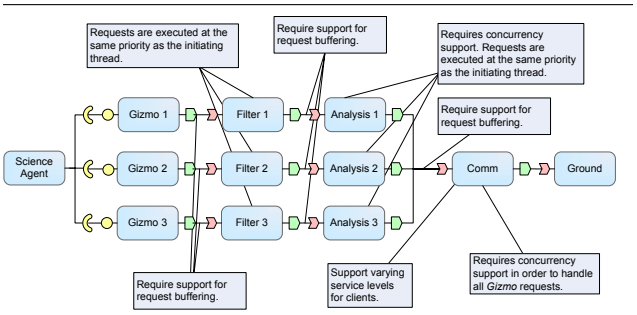


**Figure 1: MMS Mission System Components**

prototype [18] of the data processing subsystem of this DRE system using the *Component-Integrated ACE ORB* (CIAO) [3] QoS-enabled component middleware framework, the RACE [16] dynamic QoS adaptation framework and the PICML [2] MDE tool. CIAO extends our previous work on *The ACE ORB* (TAO) 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 1 shows the instances of—and connections between—software components within a single MMS 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.

## 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 that comprise the DRE systems[1], 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 semantics from the application domain must be mapped to 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, however, 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 component is operating in burst mode at the same time), requests are handled based on the importance of the originating *gizmo* component.

One way to meet this requirement is to configure *gizmo*, *filter*, and *analysis* components with the CLIENT_PROPAGATED *priority model policy*. This Real-time CORBA policy 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 [13] 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.

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, however, it is tedious and error-prone for domain experts (*e.g.*, MMS systems engineers or software architects) 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.

**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 are a (non-exhaustive) list of pre-/post-conditions and invariants:

- *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.

In our MMS mission prototype, for example, the *analysis* components use the *priority-banded connection policy* to ensure that end-to-end priority is preserved between the *gizmo* components and the *comm* components. This prioritization scheme gives precedence first to operational modes and then to importance, which 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 handle requests at the wrong priority, potentially causing priority inversion, or worse, failing at the client side.

1 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.

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. With automated tool support, however, it is hard to validate these values.

**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. 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.

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. Tracking 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, therefore, is automated tool support that can assure an application's evolution throughout its entire lifecycle.

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 and shows 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 [15] model-checking framework to define composite Real-time CCM-specific validation language constructs for validating the QoS options generated by transformations.

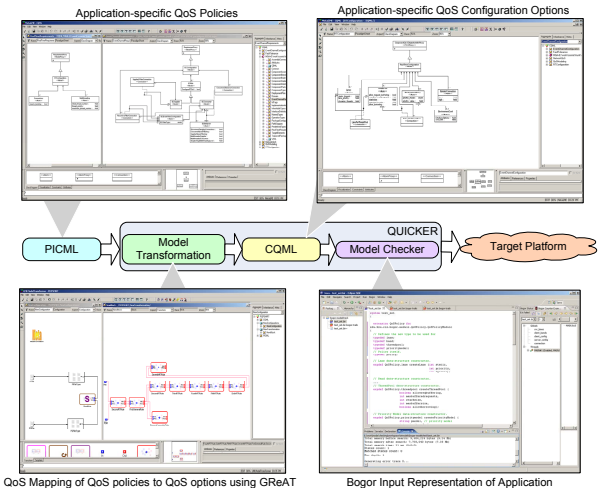


**Figure 2: QUality of service pICKER (QUICKER)**

### 3.1. Overview of the QUICKER Toolchain

The architecture of QUICKER is shown in Figure 2. QUICKER 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[2] at a higher level of abstraction than used by third-generation programming languages (such as Java or C++) or textual declarative notations (such as XML). We used the *Graph Rewriting and Transformation* (GReAT) [7] tool to transform platform-independent QoS policies captured in PICML (the input) to platform-specific QoS configuration options captured in the Component QoS Modeling Language (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 the following types of questions:

a. Is the component primarily being used as a *service provider* (*i.e.*, a *server*), a *service consumer* (*i.e.*, a *client*), or both?

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

b. Would the component simultaneously service requests from multiple clients? If so, what is the minimum and maximum number of clients it is expected to service simultaneously and would the requests be prioritized?

c. In the *server* role, would the component require execution of the requests at a fixed priority or a varying priority?

d. In the *server* role, should the component buffer requests when resources (threads) are not immediately available to honor the requests?

e. In the *server* role, would the component require support for varying service levels for different clients?

The output language (*i.e.*, CQML) of the QUICKER transformation captures CCM-specific QoS configuration options for component-based applications. For example, CQML models include Real-time CCM QoS options, such as (1) *Threadpools*, 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 enables *client* components to choose appropriate connections for making requests based on their invocation priorities.

To perform validity checks on the GReAT-generated CQML model, we used Bogor [15], 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. Since Bogor's input language does not directly support modeling of CCM components and QoS parameters, we extended its input language to support these abstractions.

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 [12] that 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.

The steps involved 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 Bogoar checks configuration values for validity.

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 the ORB's concurrency level, (b) if the execution of requests at a `server` component must be 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 an appropriate number of `Priority Lanes` for that component and 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 components simultaneously service multiple requests.
- 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.

QUICKER automatically transforms this high-level description of QoS policies of MMS mission components specified in a PICML model and generates the detailed QoS configuration options in the form of a CQML model.

### 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.

To validate QoS configuration options of an application using Bogor, we specify the application model and the QoS configuration options that the application is expected to satisfy. To express this specification in the *Bogor Input Representation* (BIR) 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 and option dependency tracking.

• **BIR input extensions.** Bogor provides the BIR language to specify input to its model-checker. It is cumbersome and error-prone, however, to describe middleware QoS configuration options using the default capabilities of BIR [15]. 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.

The BIR extensions we have developed 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. It may be necessary to set various individual QoS options, depending on a component's role (*i.e.*, `client` or `server`). Since these BIR extensions can represent 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 1.

Primitives are the Bogor input 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 1. 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 *Priority-Band* values using `getLowerBound` and `getUpperBound` primitives.

```
// Declaration of the extensions
// Declare data-type variables 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 1: Application-specific BIR Primitives**

The rule in Figure 1 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.

**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 misconfigured the QoS by specifying the concurrency model of *Comm* component to use *Thread Pool* without *Lanes* in the CQML model of MMS mission prototype.

One of the rules specified in Listing 1 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 1. Rules defined using BIR

primitives are thus used in QUICKER to provide automated tool support that helps ensure 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.

## 4. Related Work

This section compares our work on QUICKER with related work that applies model-driven engineering techniques for QoS configuration and adaptation of DRE systems.

**Functional specification and analysis tools.** Cadena [5] is an integrated environment built using Eclipse for building and analyzing CCM based systems. Cadena provides a framework for lightweight dependency analysis of behavior of components. Cadena also supports an integrated model-checking infrastructure dedicated to checking global system properties using event-based inter-component communication via real-time middleware [4]. 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.

**QoS adaptation modeling tools.** The *Distributed QoS modeling environment* (DQME) [19] is a DSML that enables the design of QoS adaptive applications in combination with using QoS provisioning frameworks, such as QuO [20]. 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.

**QoS specification tools.** 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 [6], on the other hand, 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; the platform-independent QoS policies also allow configurable re-targeting of the QoS mapping to support other types of middleware technologies.

**Schedulability analysis tools.** Research presented in [11] maps application models captured in the *Embedded Systems Modeling Language* (ESML) to UPPAAL timed automata [9] 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) [17] and the *Automatic Integration of Reusable Embedded Systems* (AIRES) [8], which are model-driven analysis tools that evaluate whether certain timing, memory, power, and cost constraints of real-time and embedded applications are satisfied. QUICKER focuses on a different level of abstraction (*i.e.*, QoS policy mapping tools) than [11, 17, 8] (which are QoS analysis tools). QUICKER is complementary to these efforts since it emphasizes 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 a development problem. To address these challenges, we have developed the *QUality of service pICKER* (QUICKER) toolchain, which uses (1) *model-transformation* to automate the mapping of application QoS policies into middleware-specific QoS configuration options and (2) *model-checking* to ensure that the QoS configuration options are valid at the individual component level and at the global application level. To demonstrate the use of QUICKER, we applied it to address configuration chal-

lenges in a prototype of NASA's MMS space mission and 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.

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] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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`.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] T. Mens, P. V. Gorp, D. Varro, and G. Karsai. Applying a Model Transformation Taxonomy to Graph Transformation Technology. In *Lecture Notes in Computer Science: Proceedings of the International Workshop on Graph and Model Transformation (GraMoT'05)*, volume 152, pages 143–159, Tallinn, Estonia, Sept. 2006. Springer-Verlag.

[13] 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.

[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* $36^{th}$ *Hawaii International Conference on System Sciences, Software Technology Track, Distributed Object and Component-based Software Systems Minitrack, HICSS 2003*, Honolulu, HW, Jan. 2003. HICSS.

[15] Robby, M. Dwyer, and J. Hatcliff. Bogor: An Extensible and Highly-Modular Model Checking Framework. In *Proceedings of the* $4^{th}$ *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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.