Evaluating the Correctness and Effectiveness of a Middleware QoS Configuration Process in Distributed Real-time and Embedded Systems*

Amogh Kavimandan†, Anantha Narayanan, Aniruddha Gokhale and Gabor Karsai
Institute for Software Integrated Systems
Vanderbilt University, Nashville, TN
{a.kavimandan,anantha.narayanan,a.gokhale,gabor.karsai}@vanderbilt.edu

Abstract

Recent advances in software processes and artifacts for automating middleware configurations in distributed real-time and embedded (DRE) systems are starting to address the complexities faced by system developers in dealing with the flexibility and configurability provided by contemporary middleware. Despite the benefits of these new processes, there remain significant challenges in verifying their correctness, and validating their effectiveness in meeting the end-to-end quality of service (QoS) requirements of DRE systems. This paper addresses this problem by describing how model-checking and structural correspondence can be used to verify the correctness of a middleware QoS configuration process that uses model-based graph transformations at its core. Next, it provides empirical proof to validate the effectiveness of our technique to meet the end-to-end QoS requirements in the context of a representative DRE system.

1 Introduction

Contemporary component middleware technologies, such as Enterprise Java Beans (EJB) and CORBA Component Model (CCM), have helped to decouple application logic from the quality of service (QoS) configuration of distributed real-time and embedded (DRE) systems by moving the QoS configuration activity to the middleware platforms that host these systems. Middleware provides out-of-the-box support for traditional concerns affecting QoS in DRE systems including multi-threading, assigning priorities to tasks, publish/subscribe event-driven communication mechanisms, security, and multiple scheduling algorithms.

Although this component middleware simplifies application logic, the DRE system developers are now faced with the complexities of choosing the right set of middleware configuration parameters that meet the QoS demands of their systems. This problem is particularly pronounced in general-purpose middleware platforms, such as CCM and EJB, which are designed to be highly flexible and configurable to meet the needs of a large class of distributed systems.

Prior research on software processes and artifacts for QoS management in DRE systems have focused on different dimensions of the problem space. For example, configuration, analysis, optimization and adaptation techniques [9, 17] allow allocation and dynamic QoS adaptation such that end-to-end application goals are met. Another online approach [18] applies feedback control theory in conjunction with application monitors for affecting resource allocation. Several works for schedulability and timing analysis [16, 6], and behavioral analysis and verification [1] exist in the literature for calculating the exact priority schemes, time periods, and resolving functional dependencies. These related approaches often provide either runtime solutions for QoS management or their outcomes are independent of any middleware platforms and hence must be mapped onto middleware configuration options via another technique. It is only recently that design-time techniques are starting to address [7, 19, 5] some of the challenges of middleware configurations, which includes support for configuring low-level QoS properties of the middleware platform and generating test suites for benchmarking, among others.

Despite these recent research efforts in addressing the middleware configuration problem for DRE systems, techniques to evaluate the correctness of these software processes and validating their effectiveness in meeting system QoS objectives remains largely unaddressed to date. This paper focuses on this unresolved dimension of the problem space. We demonstrate our ideas on our earlier work on a design-time process for middleware QoS configuration, which includes the QUICKER [4] model-driven engineering (MDE) toolchain and its QoS mapping algorithms [5] that use graph transformations.

In this paper we verify the correctness of our QoS configuration process and validate its effectiveness in meeting the QoS requirements of DRE systems. To this end, we

*This work was sponsored in part by grant from Lockheed Martin ATL.
†Contact author

use *structural correspondence* between source and target modeling languages in QUICKER to verify the correctness of their mapping. Further, we show how the Bogor [15] model-checking tool can be seamlessly extended to employ real-time CCM (RT-CCM)-specific language constructs to ascertain that the generated configurations are appropriate. We subsequently apply our configuration in the context of a representative DRE system case study and empirically evaluate the generated QoS configurations to show how the QoS requirements are met.

The remainder of this paper is organized as follows: Section 2 gives a brief overview of our configuration process and the input and output languages used in its QoS mapping algorithms; Section 3 verifies the correctness and empirically evaluates our configuration process in the context of a DRE system case study; Section 4 compares our work with existing literature on middleware QoS configuration; and Section 5 presents concluding remarks.

2 Overview of middleware QoS configuration process

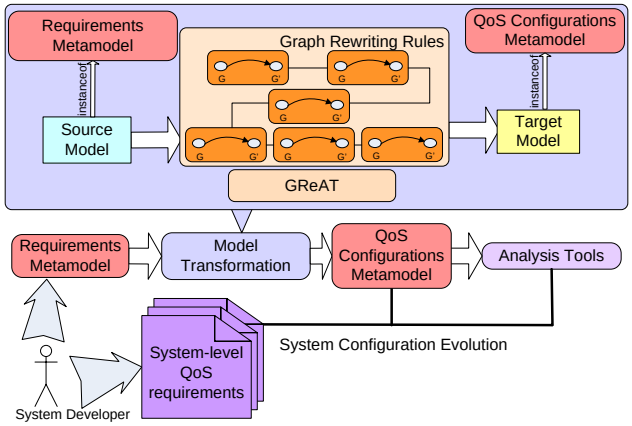


Figure 1: Model-driven QoS configuration process

Figure 1 shows our overall approach (for details please see [5]). DRE system developers use the *Requirements* domain-specific modeling language (DSML)/metamodel to specify the system QoS requirements. Using our configuration process, a specification of system QoS requirements acts as the source model of QoS mapping algorithms. As can be seen in Figure 1, our process uses model transformations for achieving QoS mapping. Middleware-specific QoS configuration options are captured as models using the *QoS Configurations* DSML which serves as the target model in the transformation process.

We have used the Generic Modeling Environment (GME) [8] toolkit for developing these source and target languages, which provides a graphical user interface that can be used to define both language semantics and system models that conform to the languages defined in it. The model-to-model transformations, on the other hand, have been developed using the Graph Rewriting And Transformation (GReAT) [3]. GReAT, which is implemented within the framework of GME, can be used to define transformation rules using its visual language, and executing these transformation rules for generating target models using the GReAT execution engine (GR-Engine).

In our configuration process developers specify QoS using the requirements language. Our QoS mapping algorithms are codified as GReAT transformation rules which use the modeled system structure and platform-specific heuristics for automatically translating the system requirements to detailed QoS configuration models. This translation is an example of a *vertical exogenous* transformation [10] that starts with an abstract type graph as the input and refines the graph to generate a more detailed type graph as the output. Finally, the generated configuration models are used for synthesizing (1) descriptors necessary for configuring functional and QoS properties of DRE system in preparation for its deployment on target platform, and (2) input to external model-checking tool for further analysis.

In our configuration process, modeling real-time requirements is simple and involves specifying the following two component-level Boolean attributes: (1) `fixed_priority_service_execution` that indicates whether or not the component modifies client service invocation priorities, and (2) `bursty_client_requests` that allows specification of the profile of service invocations made by its client components *i.e.*, whether the invocations are bursty or periodic in nature.

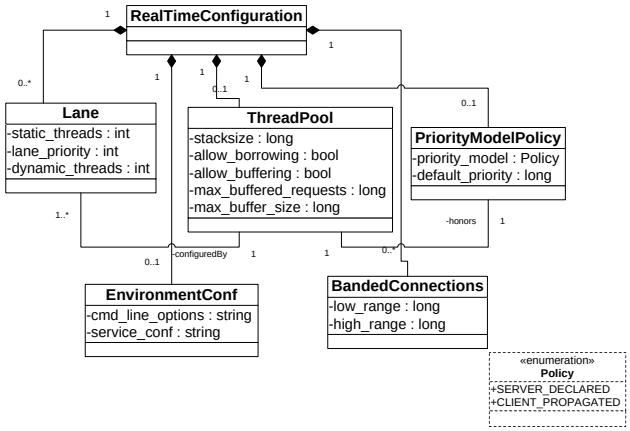


Figure 2: Simplified UML notation of real-time QoS configurations DSML

Figure 2 shows the QoS configurations metamodel which defines the following elements corresponding to several RT-CCM configuration mechanisms: (1) `Lane`, which is a logical set of threads each one of which runs at `lane_priority` priority level. Threads can be *static* or *dynamic* depending on their state with respect to the application execution lifecycle; (2) `ThreadPool`,

which controls various settings of `Lane` elements including `stacksize` of threads, whether borrowing of threads across two `Lane` elements is allowed, and resource limits for buffering requests that cannot be immediately serviced; (3) `PriorityModelPolicy`, which controls the policy model that a particular `ThreadPool` follows. It can be set to either CLIENT_PROPAGATED if the invocation priority is to be preserved end-to-end, or SERVER_DECLARED if the server component changes the priority of invocation; and (4) `BandedConnections`, which defines separate connections for individual service invocations to avoid head-of-line blocking of high priority packets by low priority packets.

For a detailed discussion of our QoS mapping transformation algorithms, we refer the reader to [5]. In the next section, we evaluate our configuration process by applying it in the context of a representative DRE system case study.

3 Evaluation of QoS configuration process

This section evaluates our configuration process to verify the correctness of its transformation algorithms and validate its effectiveness in meeting the QoS requirements of DRE systems. First we present a representative DRE system case study we used for the evaluation. Next we discuss our structural correspondence technique for proving that the input and output models of transformations used in our process are correctly mapped. We then describe how we have applied Bogor model-checking tool for verification of the generated configurations. Finally, through empirical evaluation, we validate the generated QoS options.

3.1 DRE System Case Study

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

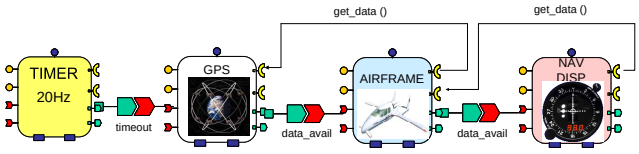


Figure 3: Basic Single Processor

A GPS device sends out periodic position updates to a GUI display that presents these updates to a pilot. The desired data request and the display frequencies are at 20 Hz. The scenario shown in Figure 3 begins with the *GPS* component being invoked by the *Timer* component. On receiving a pulse event from the *Timer*, the *GPS* component generates its data and issues a data available event. The *Airframe* component retrieves the data from the *GPS* component, updates its state, and issues a data available event. Finally, the *NavDisplay* component retrieves the data from the *Airframe* and updates its state and displays it to the pilot. In its normal mode of operation, the *Timer* component generates pulse events at a fixed priority level, although its real-time configuration can be easily changed such that it can potentially support multiple priority levels.

It is necessary to carefully examine the end-to-end application critical path and configure the system components correctly such that the display refresh rate of 20 Hz may be satisfied. In particular, the latency between *Airframe* and *NavDisplay* components needs to be minimized to achieve the desired end goal. To this end, several characteristics of the BasicSP components are important and must be taken into account in determining the most appropriate QoS configuration space. For example, the *NavDisplay* component receives update events only from the *Airframe* component and does not send messages back to the sender *i.e.*, it just plays the role of a client. The *Airframe* component on the other hand communicates with both the *GPS* and *NavDisplay* components thereby playing the role of a client as well as a server. Various QoS options provided by the target middleware platform (in case of BasicSP, it is RT-CCM) ensure that these application level QoS requirements are satisfied. In the remainder of the paper, we focus on verification and validation of the QoS options generated using our approach.

3.2 Verifying the correctness of our QoS configuration process

Verifying our model-based configuration process entails verification of correctness properties across the following two dimensions: (1) Correctness of QoS mapping algorithms *i.e.*, QoS options generated are equivalent to the QoS requirements from which these options are mapped. In our case, this translates to verification of the QoS mapping/transformations used. (2) Correctness of the generated QoS options themselves *i.e.*, whether individual values of these options are appropriate locally (*e.g.*, for a component) as well as globally (*e.g.*, for all dependent components). This section discusses verification of our process across the above two dimensions.

3.2.1 Assuring the correctness of QoS mapping algorithms

To provide an assurance that the QoS requirements specifications were correctly mapped into the QoS configuration model, we have used the transformation verification technique described in [11]. The source and target portions of the transformation are treated as typed, attributed graphs, and the correctness of the transformation is specified as a relation between these graphs. Such a relation, called a structural correspondence, is specified by identifying pivot

nodes in the metamodel and specifying what constitutes a correct transformation for these nodes.

Using structural correspondence, the verification consists of two phases: the specification of the correctness conditions, and the evaluation of the correctness. In the first phase, we identify important points in the transformation, and specify the structural correspondence rules for these points. From these rules, a model traverser is automatically generated, which will traverse and evaluate the correspondence rules on the instance models. This step needs to be performed only once. The second phase involves invoking the model traverser after each execution of the model transformation. In this phase, the model instance being transformed is traversed, and the structural correspondence rules are evaluated at each relevant node. If any of the rules are not satisfied, it indicates that the model has not been transformed satisfactorily.

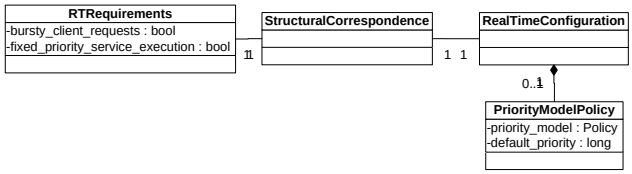


Figure 4: Structural correspondence using cross-links

Structural correspondence rules are described using (1) specification of the correspondence condition itself, and (2) the rule path expressions, which are similar to XPath queries. Figure 4 shows how we have used cross-links in GReAT as means of specifying the correspondence condition between input and output language objects such that their equivalence can later be established. `RTRequirements` is an input language object that denotes real-time requirement specification for a component. It has a correspondence relation with `RealTimeConfiguration` output language object, indicated by presence of a cross-link between them in Figure 4.

Additionally, one of the transformation rules in our QoS mapping algorithms states that if the Boolean attribute `fixed_priority_service_execution` of `RTRequirements` is set to TRUE in the input model, then `priority_model` attribute of `PriorityModelPolicy` object be set to SERVER_DECLARED in the output model. Otherwise `priority_model` should be set to CLIENT_PROPAGATED. Additionally, if `priority_model` is set to SERVER_DECLARED for a component, `Lane` values at that component and `BandedConnection` values at its clients must match. In order to complete the correspondence rule specification, the above is encoded as a rule path expression as follows:

```
(RTRequirement.
fixed_priority_service_execution = true ∧
(∀ b ∈ RTConfiguration. BandedConnection
∃ l ∈ RTConfiguration. Lanes :
    (b.low_range ≤ l.priority ≤ b.high_range)) ∧
RealTimeConfiguration.PriorityModelPolicy.
priority_model = "SERVER_DECLARED") ∨
(RTRequirement.
fixed_priority_service_execution = false ∧
RealTimeConfiguration.PriorityModelPolicy.
priority_model = "CLIENT_PROPOGATED")
```

If this expression evaluates to TRUE on an instance model, then it implies that the QoS configuration for this particular property has been mapped correctly. This applies to the `RTRequirements` and `RealTimeConfiguration` classes, and correspondence condition is added as a link between these classes in the metamodel. Similar to correspondence condition between `RTRequirements` and `RealTimeConfiguration` we described, other conditions for each of the QoS mapping rules have been specified ensuring that the transformation is verified correct if all these conditions are satisfied.

3.2.2 Verifying the generated QoS configurations using model-checking

This section illustrates how the correctness of QoS configuration mappings is verified using the Bogor model-checking framework, which is a customizable explicit-state model checker implemented as an Eclipse plugin. Verifying 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, we need to specify the application model and its QoS configurations. We use Bogor's extension features to customize the model-checker for resolving the QoS configuration challenges for component-based applications.

It is cumbersome to describe middleware QoS configuration options using the default input specification capabilities of BIR. This is because such a representation is at a much lower level of abstraction compared to domain-level concepts, such as components and QoS options, which we want to model-check. Additionally, 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 states that may be irrelevant to the properties being model-checked [15]. Therefore we have defined composite language constructs that represent functional sub-systems (such as components) and QoS options (such as thread pools) as though they were native BIR constructs.

Listing 1 shows an example of our QoS extensions in Bo-

gor to represent QoS configuration options in middleware, which 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`.

```
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

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. In our case, property simply means whether or not the QoS configurations are verified correct. Thus, since we need to verify values of various QoS options as means to check whether application property is satisfied, we define *rules* that capture values of these QoS options. BIR primitives are used to express these rules in the input specification of DRE system. Primitives are also used to capture component interconnections in BIR format which are required for populating the dependency structure for the specified input application. They are also used later during verification of options for connected components.

QoS extensions are also helpful in maintaining and resolving dependencies between application components. For example, consider a real-time configuration of BasicSP scenario in which each of the *GPS*, *AirFrame*, and *NavDisplay* components are configured to have `priority bands` for separate service invocation priorities and the *Timer* component is configured to support multiple priority levels during generation of pulse events. Given such a configuration, we have that `priority band` values at *GPS* (client) component must match `ThreadPoolLanes` at *Timer* (server) component *i.e.*, a direct configuration dependency exists between these two components.

Further, since the pulse events are subsequently reported to *AirFrame* and *NavDisplay* components there is a similar indirect dependency between `band` values at these components and `lanes` at *Timer* component. The dependency structure of BasicSP scenario is maintained in QoS extensions to track such dependencies between QoS options. Figure 5 represents the dependency structure generated using QoS extensions with the given configurations for our BasicSP scenario. Occurrences of change in configurations of either of the dependent components are followed by detection of potential mismatches such that all dependencies are exposed and resolved during application QoS design iterations.

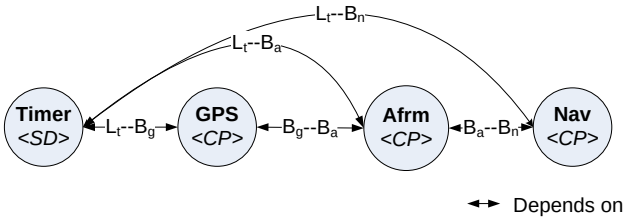
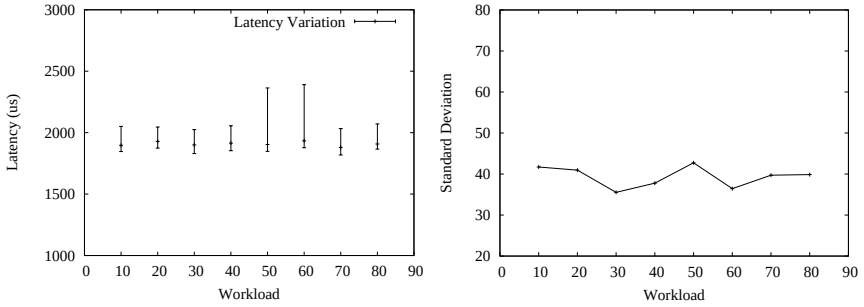


Figure 5: Dependency structure of BasicSP. L_c denotes threadpool lane and B_c denotes priority bands at component c. SD and CP indicate the SERVER_DECLARED and CLIENT_PROPAGATED priority models, respectively.

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.*, avionics engineers) since configuring application QoS policies is typically done iteratively. Depending on the number of components and 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 QoS configurations model. This generative process is done in GME using a model interpreter that traverses the QoS configurations model and generates a BIR file that captures the application structure and its QoS properties. Our 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



(a) End-to-end latency measurements (b) Statistical dispersion of latency

Figure 6: Evaluating BasicSP QoS configurations against increasing workload at a constant 20Hz invocation rate.

CCM-specific descriptors required to configure functional and QoS properties of an application and deploy it in its target environment. In the next section we empirically validate these generated QoS configurations.

3.3 Empirically evaluating BasicSP QoS configurations

In this section we empirically validate the effectiveness of the generated QoS configurations for the BasicSP case study.

Experiment Configuration. We have used ISISLab (`www.dre.vanderbilt.edu/ISISlab`) for evaluating observed QoS properties of DRE systems based on middleware QoS configurations generated using our configuration process. Each of the physical nodes used in our experiments was a 2.8 GHz Intel Xeon dual processor, 1 GB physical memory, 1 GHz network interface, and 40GB hard disks. Version 0.6 of our RT-CCM middleware CIAO was used running on Redhat Fedora Core release 4 with real-time preemption patches. The processes that hosted BasicSP components were run in the POSIX scheduling class SCHED_FIFO, enabling first-in-first-out scheduling semantics based on the priority of the process.

As the first step, we modeled BasicSP QoS requirements using the requirements DSML described in Section 2. `bursty_client_requests` was set to FALSE for all components and `fixed_priority_service_execution` attribute was set to FALSE for every component except *Timer*. Secondly, we applied our model transformation algorithm to the requirements model above for generating detailed application configurations. Table 1 captures some of the important QoS configurations generated in our process. These configurations are represented as an application model. In the final step, we apply model interpreters for synthesizing descriptors required to configure the functional and QoS properties of the application during deployment.

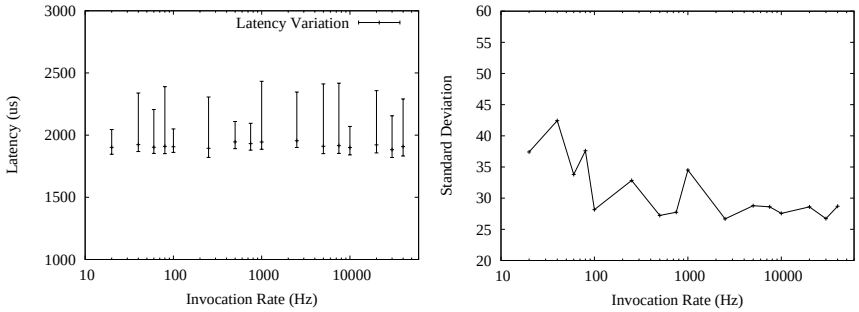
In evaluating effectiveness of our configuration process, we collected end-to-end latency measurements between *Timer* and *NavDisplay* components. Earlier in Section 3.2.2 we discussed how correctness of QoS options can be verified using our process, the first experiment discussed below empirically evaluates the effectiveness of these options in meeting 20Hz operational display refresh rate of BasicSP from low to high workload conditions. Further, operational conditions of DRE system might change (unfavorably) during its execution. In order to evaluate the tolerance of our generated configurations under such conditions, in the second experiment we measure the metrics discussed above when invocation rate is steadily increased. Each of these experiments were performed for a constant time period and after executing 10,000 warmup iterations.

Table 1: Generated QoS Configuration for BasicSP

QoS configuration	Timer	GPS	Airframe
PriorityModel	SD	CP	CP
ThreadPool			
stacksize	1024	1024	1024
max_buff_reqs.	–	20	20
allow_borro.	FALSE	FALSE	FALSE
allow_req_buff	FALSE	TRUE	TRUE
Lane			
static_thrds	4	8	8
dyna_thrds	0	0	0

Experiment 1: Increasing System Workload. Figure 6 plots the latency measurements under increasing system workload. The workload is characterized [13] as a function performed with every client invocation. The signature of the function is given as: *void work*(*int units*); where *units* argument specifies the amount of processor intensive work performed per call. The experiment was run for workload values of 10 through 80. End-to-end latency was observed to be at an average value of $\sim$1925 as can be seen in Figure 6a. Further, successive event-driven computations in the scenario exhibit an almost constant time complexity, indicated by relatively small dispersion in latencies as plotted in Figure 6b.

Experiment 2: Increasing System Invocation Rate. Performance of the generated configurations for BasicSP is given in Figure 7. Throughout this experiment the rate of invocation was increased from a normal operational value of



(a) End-to-end latency measurements (b) Statistical dispersion of latency

Figure 7: Evaluating BasicSP QoS configurations against increasing invocation rate: All the plots use logarithmic X axis and linear Y axis.

20Hz to a maximum of 40000Hz. Latency results are shown in Figure 7a which plots maximum, mean and minimum delay measurements for each invocation rate data point. Even with increasing rate the mean latency did not change significantly and was observed to be consistently just above 1900 μs for the entire range of invocation rates. Note that this is a desirable characteristic since even with an unfavorable change in operational conditions (*i.e.*, change in invocation rate) the latency was observed to be constant. Jitter in latencies for each invocation rate is plotted in Figure 7b which shows that the deviation is bound between a high value of 42.44 (at 40Hz) and low value of 26.68 (at 2500Hz). At frequencies 2500Hz and higher the jitter values became quite stable showing a maximum variation of only 2.11. Overall, our results indicate that even under increased rate of invocation, the configurations perform effectively in achieving BasicSP latency requirements.

4 Related Work

Validation and Analysis Techniques. Model-driven techniques in [7, 19] rely on a visual interface to help developers select a wide array of middleware QoS options for their applications. Such information is later used for generating test suites for purposes of empirical evaluation. In contrast, our configuration process does not expose the developers to all of the configuration space of underlying middleware and relies on platform-specific heuristics for generating QoS configurations. Further, using our process, the correctness of generated configurations is established in the design time. We argue that since our transformation algorithms codify best practices and patterns in middleware QoS configuration, QoS design and evolution throughout the system lifecycle using our approach is faster.

Analysis tools such as VEST [16], Cadena [1] and AIRES [6] evaluate whether certain timing, memory, power, and cost constraints and functional dependencies of real-time and embedded applications are satisfied. Our configuration process can be used as a complementary QoS design and analysis technique to these efforts since it emphasizes 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.

QoS Design and Specification Techniques. The Adaptive Quality Modeling Language (AQML) [12] provides QoS adaptation policy modeling artifacts. AQML generators can (1) translate the QoS adaption policies (specified in AQML) into Matlab Simulink/Stateflow models for simulations using a control-centric view of QoS adaptation and (2) generate Contract Definition Language (CDL) specifications from AQML models to be used in target middleware. Our work differs with AQML since its middleware model precisely abstracts the actual real-time CORBA implementation and does not need a two-level declarative translation (from AQML to CDL to target middleware) to achieve QoS configuration.

Ritter *et.al.* [14] describe CCM extensions for generic QoS support and discuss a QoS metamodel that supports domain-specific multi-category QoS contracts. The work in [2] focuses on capturing QoS properties in terms of *interaction patterns* among 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, our work 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.

5 Concluding Remarks

In this paper we discussed our approach to evaluating correctness and effectiveness of a QoS configuration process in the context of a representative DRE system. We showed how structural correspondence between input and output languages in our model-driven approach can be used to establish that initial system requirements are correctly mapped to middleware QoS options. We verified the correctness of generated QoS options using a model-checker and empirically showed that they are effective in satisfying system requirements.

In the future in order to show its scalability we plan to apply and evaluate our technique to complex and large-scale DRE systems. Our current approach is one-dimensional *i.e.*, both the QoS requirements mapping and configuration validation is done for a single dimension (such as real-time request-response or publish-subscribe communication dimensions). In the future we plan to investigate and develop configuration techniques under simultaneous requirements across distinct QoS dimensions. An effort is underway in extending our process for other component middleware platforms that exhibit the same level of configurability. As part of this effort, we are looking at development of parameterized model transformations that allow specification of templatized QoS mappings and later generation of platform-specific QoS mapping instances by specializing these templatized mappings.

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

References

[1] 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.

[2] 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.

[3] 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. `www.jucs.org/jucs_9_11/on_the_use_of`.

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

[5] A. Kavimandan and A. Gokhale. Automated Middleware QoS Configuration Techniques using Model Transformations. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2008)*, St. Louis, MO, USA, April 2008.

[6] 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 2003)*, Washington, DC, May 2003. IEEE.

[7] A. S. Krishna, E. Turkay, A. Gokhale, and D. C. Schmidt. Model-Driven Techniques for Evaluating the QoS of Middleware Configurations for DRE Systems. In *Proceedings of the 11th Real-time Technology and Application Symposium (RTAS '05)*, pages 180–189, San Francisco, CA, Mar. 2005. IEEE.

[8] 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.

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

[10] 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.

[11] A. Narayanan and G. Karsai. Verifying Model Transformations by Structural Correspondence. Technical Report ISIS-07-809, Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN, Dec 2007.

[12] S. Neema, T. Bapty, J. Gray, and A. Gokhale. Generators for Synthesis of QoS Adaptation in Distributed Real-time Embedded Systems. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)*, Pittsburgh, PA, Oct. 2002.

[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):1070–1085, 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 36th Hawaii International Conference on System Sciences (HICSS'03)*, Honolulu, HI, Jan. 2003.

[15] 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.

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

[17] X. Wang, Y. Chen, C. Lu, and X. Koutsoukos. FC-ORB: A robust distributed real-time embedded middleware with end-to-end utilization controlstar, open. *Journal of Systems and Software*, 80(7):938–950, 2007.

[18] R. Zhang, C. Lu, T. F. Abdelzaher, and J. A. Stankovic. ControlWare: A Middleware Architecture for Feedback Control of Software Performance. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, Vienna, Austria, July 2002.

[19] L. Zhu, N. B. Bui, Y. Liu, and I. Gorton. MDABench: Customized benchmark generation using MDA. *Journal of Systems and Software*, 80(2):265–282, Feb. 2007.