Model-Driven Techniques for Evaluating the QoS of Middleware Configurations for DRE Systems[*]

Arvind S. Krishna, Emre Turkay, Aniruddha Gokhale, and Douglas C. Schmidt
{arvindk,turkaye,gokhale,schmidt}@dre.vanderbilt.edu
Electrical Engineering and Computer Science Department
Vanderbilt University, Nashville, TN

Abstract

This paper provides two contributions to R&D on model-driven development (MDD) techniques that help codify the impact of middleware configurations on end-to-end distributed real-time and embedded (DRE) system quality of service (QoS). First, we describe how MDD techniques can help select middleware configuration parameters that satisfy key functional and QoS requirements of DRE systems. Second, we apply our MDD techniques to empirically evaluate the end-to-end QoS of representative DRE systems in the avionics and industrial manufacturing domains. Our results show how MDD techniques significantly enhance conventional ad hoc processes used by developers to configure middleware that meets the QoS needs of DRE systems.

1. Introduction

Emerging trends and challenges. Various R&D efforts [14, 2] have focused recently on *quality of service (QoS)-enabled middleware* to enhance the development, time-to-market, and reuse of distributed real-time and embedded (DRE) systems, such as avionics mission computing, industrial process automation, and total ship computing environments. An inherent characteristic of QoS-enabled middleware is its *flexibility,* which is needed to support the requirements of a wide range of DRE systems that must (1) run on many hardware/OS platforms and interoperate with many versions of related software frameworks/tools and (2) provide support for end-to-end QoS properties, such as low latency and bounded jitter; These DRE system requirements are met in part by compile- and run-time selection of middleware *configuration options*, which are used to fine-tune QoS properties of the middleware, and *customization options*, which are used to tradeoff functionality and footprint of middleware to suit DRE system requirements.

QoS enabled middleware platforms such as the the Component-Integrated ACE ORB (CIAO) [13] and PRiSm [11], have many (*i.e.*, 10's-100's) of configuration options and customization parameters that application developers can adjust to tailor the middleware to meet various functional and QoS needs. Developing mission-critical DRE systems using such highly flexible middleware can be problematic, however, due in large part to the complexity associated with configuring and customizing QoS-enabled middleware.

Model-driven techniques for configuring, customizing, and validating middleware. A promising way to address the problems outlined above is to (1) develop a rigorous process and a set of tools that simplify and automate the configuration, customization, and validation of QoS-enabled middleware and (2) apply this process and tools to DRE systems to understand how the middleware configurations affect QoS properties. This paper describes *model-driven development* (MDD) techniques we have developed and applied to help resolve key complexities related to middleware configuration, customization, and validation. We focus on an integrated set of MDD tools and an associated MDD process for (1) configuring and customizing QoS-enabled middleware and (2) generating testsuites for empirically benchmarking the configured middleware to evaluate its QoS.

This paper extends our earlier work on (1) *Options Configuration Modeling Language* (OCML) [12], which is an MDD tool that simplifies the specification and validation of complex DRE middleware and application configurations, and (2) *Benchmark Generation Modeling Language* (BGML) [6], which is an MDD tool that synthesizes benchmarking testsuites to analyze the QoS performance of OCML-configured DRE systems,[1], by illustrating how these MDD tools can be used to measure the impact of middleware configurations on end-to-end DRE system performance, nor did we evaluate how these tools help alleviate the complexities of configuring QoS-enabled middleware to support particular DRE system requirements. This paper

* This work was sponsored in part by AFRL Contract#F33615-03-C-4112 for DARPA PCES Program, Raytheon, and a grant from Siemens CT.

1 OCML and BGML are part of an open-source MDD toolchain called CoSMIC [3] that can be downloaded from `www.dre.vanderbilt.edu/cosmic/`.

therefore enhances our earlier work by describing an MDD process that leverages the generative capabilities of OCML and BGML to systematically document and validate how different configurations of QoS-enabled middleware affect DRE system QoS. We apply this process and tools to two CIAO-based DRE systems in the avionics and manufacturing domains to quantify the variation of QoS based on configuration/customization options.

The results of our research show how MDD tools and processes enhance conventional *ad hoc* processes developers use today to configure middleware that meets the QoS needs of DRE systems. In particular, our MDD tool-based process (1) identifies options that have significant influence on QoS, (2) checks for inconsistencies between options at modeling time, and (3) automatically generates code for configuration and empirical validation.

2. Key QoS-enabled Middleware Configuration and Customization Challenges

Developers of large-scale DRE systems face several challenges associated with (1) configuring and customizing QoS-enabled middleware for DRE systems and (2) evaluating the QoS performance of selected middleware configurations. This section first describes two DRE systems in the avionics and industrial manufacturing domain and describes the QoS requirement for components in these two applications. We then use these challenges to motivate the need for the OCML and BGML MDD tools described in Section 1. The OCML and BGML tools were developed using the Generic Modeling Environment (GME) [7] and form part of a broader MDD toolchain called CoSMIC [3] that supports the development, configuration, and deployment of component-based DRE systems.

2.1. Case Studies of Two DRE Systems

To motivate our MDD tools and processes, we now briefly describe two representative DRE systems from the avionics and the manufacturing domains based on our collaborations with industrial partners in the DARPA PCES program. For each scenario, we describe components that have similar QoS requirements. Section 3.2 then uses these scenarios to show how our MDD process can be applied to understand how various middleware configurations affect QoS. The middleware we use to implement both case studies is ACE v5.4.2 + TAO v1.4.2 + CIAO v0.4.2.

2.1.1. Case Study 1: Avionics Mission Computing Scenario The first case study is based on the *Basic_SP* scenario from the DARPA PCES program [11]. In this scenario, a GPS device sends out periodic position updates to a GUI display that presents these updates to a pilot. The desired data request and the display frequencies are fixed at 20 Hz. The QoS-enabled component middleware architecture uses a *push event/pull data* publisher/subscriber communication paradigm.

Component interactions The component interaction for this example is shown in Figure 1. The scenario shown in

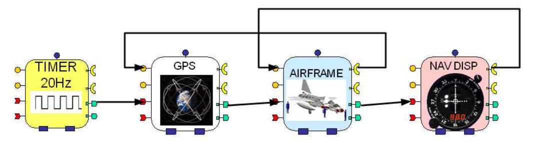

Figure 1. The *Basic_SP* Navigation Display Example

this figure begins with the `GPS` component being invoked by the TAO Real-time Event Service [4] (shown as a `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 in turn retrieves the data from the `Airframe` and updates its state and displays it to the pilot.

QoS requirements For the *Basic_SP* scenario, to satisfy the QoS requirement of ensuring display refresh rate of 20 Hz, it is necessary to configure the `Airframe` and `NavDisplay` components appropriately. To achieve this goal, we need to determine the appropriate configurations for the individual components and empirically evaluate the configurations to determine the configuration that satisfies the QoS requirement. We assume the mission computing system is configured correctly. Several characteristics of the *Basic_SP* components are important in determining the configuration space. For example, the `NavDisplay` component receives updates only from the `Airframe` component and does not send messages back to the sender, *i.e.*, it just plays the role of a client. Likewise, the `Airframe` component communicates with both the `GPS` and `Display` components, playing the role of a peer, though not concurrently processing requests since the events are handled sequentially.

2.1.2. Case Study 2: Robot Assembly Scenario The second case study is based on a manufacturing assembly line system. In this scenario, a conveyor of watches passes through a watch station where a human operator visually examines every watch for blemishes and configures (*e.g.*, sets the watch time, date, and language) the watch depending on its type. If a watch is damaged it is rejected, otherwise it is passed along the conveyor to a packaging station. The QoS-enabled component middleware architecture uses a *push event/pull data* publisher/subscriber communication paradigm. As with the *Basic_SP* example, the *robot assembly* scenario is available in the CIAO and CoSMIC releases.

Component interactions In this scenario, a pallet (controlled by a `PaletteManager` component) containing digital watches moves to a robot station (controlled by the `RobotManager` component) where its time is set using the current time provided by a periodic clock (controlled by a `WatchManager`). The management for the watch setting facility located at a remote site can send production work orders and receive response to orders, ongoing work status, inventory, and other messages. These instructions are sent to the `WatchManager` component using `ManagementWorkInstructions` (`MWI`) component. The `WatchManager` component interacts with a human operator who using the `HumanMachineInterface` (`HMI`) component accepts/rejects the watch. When the watch is accepted, the `WatchManager` component uses the `RobotManager` component to set the time. When a watch is rejected, however, the `RobotManager` component removes the watch from the assembly line. Figure 2 illustrates this assembly of components.

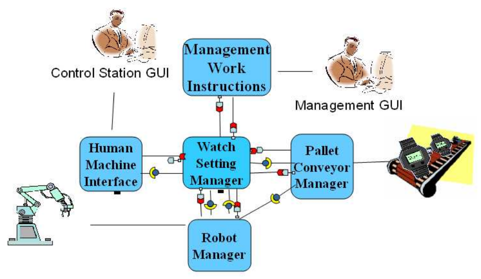

Figure 2. *Robot Assembly* Scenario

QoS requirements To maximize the production line efficiency in this scenario, the communication overhead between the `WatchManager` and `HumanOperator` must be low. The round-trip latency between messages exchanged between the two entities should therefore be small. To achieve this goal, we need to determine the appropriate configurations for the individual components and empirically evaluate theses configurations and determine the configuration that satisfies the QoS requirement (minimizing round-trip latency). One characteristic of the *robot assembly* components is important in determining the appropriate configurations. In particular, the `HumanMachineInterface` component only plays the role of a client since its only source of events is the `WatchManager` component, which also interacts with all other entities. Similar to the `Airframe` component, the `WatchManager` plays the role of a peer. It does not, however, process requests concurrently since the decision to accept/reject a watch is made sequentially by a human operator.

2.2. DRE System Configuration and Evaluation Challenges

DRE systems based on QoS-enabled middleware, such as the avionics and industrial automation scenarios described in Section 2.1, need to resolve the following challenges:

2.2.1. Challenge 1: Configuring and Customizing QoS-enabled Middleware for DRE Systems QoS-enabled middleware provides a range of configuration options that can be used to customize and tune the QoS properties of the middleware. For example, the ACE+TAO+CIAO QoS-enabled middleware provides ∼500 configuration options that can be used to tune its behavior.[2] This large number of configuration options create several problems for developers and users of middleware for DRE systems.

For example, the case studies in Section 2.1 describe how to configure the `NavDisplay` and HMI appropriately, application developers need to determine the semantically valid set of configuration options and their settings. This task is complicated, however, by the explosion of the *configuration space* of the CIAO QoS-enabled middleware. For instance, DRE system developers need to configure and tune the performance of the ACE+TAO+CIAO middleware at multiple levels, including lower-level messaging and transport mechanisms, the object request broker (ORB) itself, up to higher-level middleware services (such as event notification, scheduling, and load balancing). This problem is exacerbated by the fact that not all combinations of options form a semantically compatible set.

2.2.2. Challenge 2: Evaluating the QoS of Selected Middleware Configurations QoS-enabled middleware runs on a multitude of hardware, OS, and compiler platforms. Developers of middleware and applications often use trial-and-error methods to select the set of configuration options that maximizes the QoS attainable by the middleware. Unfortunately, the settings that maximize performance for a particular group of platforms and applications may not be suitable for different ones.

2 Examples of highly configurable middleware in other domains include (1) SQL Server 7.0, which has ∼50 configuration options, (2) Oracle 9, which has over 200 initialization parameters, and (3) Apache HTTP Server Version 1.3, which has ∼85 core configuration options.

Addressing the QoS-evaluation challenges of DRE systems fielded in a particular environment requires a suite of benchmarking tests that are customized to the system's environment. These benchmarks must test the different configurations of the fielded system and analyze the results to evaluate whether the proper QoS is actually delivered to the DRE system. An example of such a test would be one that can evaluate the configurations of the `NavDisplay` component in the Basic_SP scenario. Such an evaluation process would involve creating a benchmarking experiment to measure QoS properties, thereby requiring developers to write (1) the header files and source code that implement the functionality, (2) the configuration and script files that tune the underlying middleware and automate the task of running tests and output generation, and (3) project build files (*e.g.*, makefiles) required to generate the executable code. Our experience in earlier work [8] revealed how tedious and error-prone this process is since it requires many manual steps to generate, execute, and analyze the benchmarks.

2.3. Resolving DRE System Configuration and Evaluation Challenges using MDD Tools

The aforementioned challenges are resoloved the following MDD tools:

Resolving middleware configuration problems via OCML. OCML represents the *type system* used by middleware developers as a GME metamodeling paradigm. For example, it defines a *numeric option* type for middleware options which can have numeric values, *i.e.*, cache/buffer sizes, thread counts, etc. OCML also represents a standard set of configurations as a GME model library. It defines *standard configurations* that application developers can use to specify the set of configuration options that suit their components and enforces *dependency rules* that prevent application developers from choosing invalid combinations of configuration sets that could result in incorrect/undefined behavior. A *Configuration File Generator* (CFG) application is developed for the application developer to specify the appropriate set of option configurations and validation of this set. Middleware developers define individual models of middleware configuration options using the elements and rules of OCML metamodel to configure particular middleware platforms. For example, the CIAO ORB configuration options defined in the OCML metamodeling paradigm.

For configuring the components of the two application scenarios, we used the CFG that provides a simple wizard like interface to generate the individual configuration files. The CFG enabled the elimination of accidental complexities involved in validating semantically incompatible configurations. For example, while configuring the `Airframe` component, setting `ORBReactorType` to `select_st` precludes setting of `ORBReactorThreadQueue`. The CFG enabled detecting these kinds of inconsistencies at design time preventing undefined behavior at run-time. Additionally, for each selected option, the CFG displays the relevant documentation for that option thus helping choose the appropriate configuration settings.

Resolving QoS evaluation challenges via BGML. BGML is a GME-based modeling paradigm that captures key QoS evaluation concerns of QoS-enabled middleware, such as (1) modeling how distributed system components interact with each other and (2) representing metrics that can be applied to specific configuration options and platforms. Middleware/application developers can use BGML to graphically model interaction scenarios of interest. BGML automates the task of writing repetitive source code to perform benchmark experiments and generates syntactically, semantically valid source and benchmarking code. To model interactions BGML provides (1) *test elements* such as operations, return-types, latency and throughput that can be used to represent generic operation or a sequence or operation steps and associate functional QoS properties with them and (2) *workload elements* such as tasks and task-set that can be used to model and simulate background load present during the experimentation process. These workload elements are then mapped to individual platform specific code in the interpretation process.

We applied BGML to both the DRE system scenarios, in particular to resolve the evaluation concerns. The test elements, enabled us to model the operation/events associated with different components and also associate QoS parameters such as round-trip latency. For example, in the *robot assembly* scenario, we used *latency* measures for the `WorkOrderResponse` event. The interpreters then generated the benchmarking, build and glue idl files to conduct the experiment, thus eliminating the accidental complexity arising from handcrafting these files.

3. An MDD Process for Evaluating the QoS of Middleware Configurations

Developers of DRE systems commonly use *ad hoc* processes to identify the configuration settings for the individual components in their applications. These *ad hoc* processes entail (re)validation of the configuration settings for different application scenarios and domains. OCML reduces accidental complexities associated with configuring QoS-enabled middleware. Likewise, BGML reduces accidental complexities associated with evaluating QoS characteristics across different configurations of hardware, OS, compiler, and middleware platforms. Using OCML and BGML *in isolation*, however, cannot determine the most suitable configuration for a set of application QoS requirements or platform characteristics since BGML does not know what configurations resulted in the captured metric and OCML does not know what the performance was for the chosen configuration. This section describes an MDD process that integrates the OCML and BGML tools presented in Section 2 to resolve the configuration and evaluation challenges of DRE systems. We also evaluate the applicability and generality of this process by applying it to the DRE system scenarios from Section 2.1.

3.1. Enhancing Ad Hoc Processes via Integrated MDD Techniques

To aforementioned challenges, we have developed an MDD process that can be (1) used to configure DRE systems, (2) used to evaluate the QoS of different middleware configurations, and (3) applied across different domains and platforms. Section 3.2 describes how this process has been applied to *Basic_SP* and *robot assembly* scenarios to resolve key configuration and deployment concerns of those systems. To demonstrate the utility of our MDD tools and process, we describe how our approach helps in resolving the following two concerns of DRE systems:

1. How does the configuration options affect the overall end-to-end QoS of a component interaction scenario? Where a scenario defines a set of component instances interacting with each other via messages to achieve a common purpose or a goal. In particular, are there certain configuration options that influence performance more than the others?
2. How does the choice of underlying platform affect the QoS for a scenario? In particular, does the QoS vary depending on where the components are placed? Additionally, is there a way of determining the mapping between the nodes and the components for a given interaction scenario to achieve an acceptable level of QoS?

Our MDD approach consists of the following steps that are performed by application developers:

Step 1. Modeling component interaction scenarios. This first step in our process involves using PICML [1] to visually represent the interfaces of the components, their interconnections, and their dependencies on external libraries and artifacts. The PICML tool which is part of the CoSMIC tool chain supports visual modeling of components, ports, interfaces, and operations. This step is required for conducting any experiment since it generates XML-based metadata for component deployment. The system is then checked for constraint violations and XML metadata is generated.

Step 2. Determining appropriate configuration settings. For the components that need to be configured appropriately, determine the set of relevant configurations and their individual options that is expected to provide the required level of QoS, *e.g.*, expressed in terms of latency, throughput, or jitter metrics. The set of configuration and their options can be conceptualized as a *configuration space*, with each combination being a point in this space. If the configuration options for a component have been determined a priori, use the OCML MDD tool to directly generate the appropriate configuration file. Otherwise, use OCML to generate the various possible combination of configurations that are to be evaluate empirically via the MDD tool. This step involves identification and generation of the configuration space for the individual components.

Step 3. Experimental set up. Use BGML (Section 2) MDD tool to model the test, *i.e.*, associate latency/throughput characteristics with the component operations that are to be empirically evaluated to determine the right configurations. Then use BGML's model interpreters to generate the testsuite for evaluating the QoS delivered to the DRE system by the middleware configuration. This step involves the generation of the build, benchmarking and script code code from higher level models to run the experiment.

Step 4. Choosing a target deployment platform. Determine the target deployment platform, *i.e.*, the physical nodes on which the individual components will be hosted. Use the PICML MDD tool part of the CoSMIC toolsuite, to to model this target deployment platform and the component node mappings, *i.e.*, map component instances onto individual nodes in the domain. This step involves the generation of the XML meta-data describing the domain and how individual components are placed is generated.

Step 5. Navigating the configuration space. For each combination of the configuration determined in Step 2, run the benchmarking tests generated by BGML to evaluate the QoS. This step involves the actual running of the experiment on the target platform to generate data. This stage assumes that the individual components are available and does not include the phase of writing the logic for the components.

3.2. Applying Our MDD Process to the DRE System Case Studies

We applied the MDD process described in Section 3.1 to both the avionics mission computing and robot assembly scenarios. For each step described in that process, we now illustrate how the step maps to the two DRE systems. Finally, we discuss how our process helped in addressing the concerns of DRE systems described in Section 3.1.

Step 1. Modeling component interaction scenarios. Use the PICML MDD tool [1] to model each DRE system scenario. This step involves modeling the artifacts (*i.e.*, elements involved in the scenario).

Step 2. Determining appropriate configuration settings. Select a set of middleware configuration options using OCML (Section 2) that are expected to provide the necessary level of QoS. The CIAO middleware provide over 500 configuration options, though not all of these correspond to the QoS requirements for the components in our study. For example, the `NavDisplay` and `HumanMachineInterface` display-related components do not need to consider server side options as they only act as clients. We therefore narrowed down the configuration space by examining the documentation.

The display-related components are configured with a similar configuration set and common across various applications so that the standard OCML configurations library contains the *DisplayComponent* option. Setting this option for a specific application results in a CFG that narrows down the configuration space by setting appropriate values for the target middleware platform. Table 1 shows the relevant configuration options for these displayed-related components in our case studies. Further examination of this reduced con-

Option Label	Option Name	Option Settings
A	ORBReactorMaskSignals	{**0**, 1}
B	ORBInputCDRAllocator	{ **null**, thread}
C	ORBReactorType	{ **select_st**, mt}
D	ORBProfileLock	{thread, **null**}
E	ORBObjectLock	{thread, **null**}
F	ORBConnectionCacheLock	{**null**, thread}
G	ORBClientConnectionHandler	{RW, ST}
H	ORBTransportMuxStrategy	{EXCLUSIVE, MUXED}
I	ORBConnectionPurging Strategy	{LF, reactive}
J	ORBConnectStrategy	{LF, reactive}

Table 1. Configuration Space for the Display Components

figuration space reveals that some of the configurations settings can be set *a priori*, *i.e.*, without experimentation. For example, both components interact with only one source and do not need synchronization. These option settings (options A-F) can be directly determined (shown in bold) in Table 1. For the remaining configurations, where both options are suitable, the possible configuration combinations were generated using OCML and determined experimentally as described in the next step. For the `WatchManager` and `Airframe` components the configuration space was determined in a similar manner. Table 2 illustrates the configuration space. Note that apart from the (K-L) options shown in the table, options (A-E) shown in Table 1 are also relevant to these components. After determining the relevant

Notation	Option Name	Option Settings
K	ORBReactivationOfSystemIds	{0, 1}
L	ORBPOALockType	{thread, null }

Table 2. Configuration Space for Watch-Manager and Airframe Components

configurations, we use our OCML tool to generate the permutations of configuration options for all four components.

Step 3. Experimental setup. Using the BGML MDD tool, the QoS characteristic (in this case roundtrip latency and throughput) to be captured in the experiment were represented in the models. Figure 3 depicts how a latency metric was associated with the operation between the `WatchManager` and `HumanMachineInterface` components, modeled using BGML. This figure also shows three background tasks that simulated load conditions in the scenario.

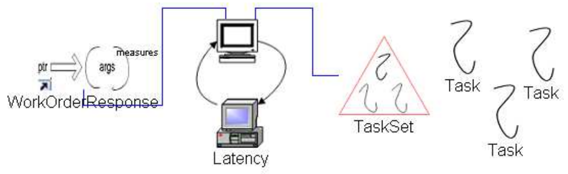

Figure 3. Associating QoS with operation in BGML

For both cases, the average latency and the required number of warmup iterations were specified in the BGML models, which then generated the scaffolding code needed to run the experiment.

Step 4. Choosing a target deployment platform. To empirically evaluate the configuration and identify the recurring settings, we used the testbed shown in Table 3. The individual computers themselves then were connected via a LAN, which simulated a deployment scenario where these components are deployed on different nodes. After choosing this scenario, we used the target modeling capability in PICML MDD tool to visually represent this scenario. The model interpreters then generate the XML metadata that is used by the CIAO component runtime engine to deploy the components.

Step 5. Navigating the configuration space. For the configuration space chosen, the generated benchmarking tests were run to evaluate QoS, which involved running 32 different experiments for both the scenarios. Using the results, we identified patterns and clusters of configurations that had the most influence on QoS.

Hosts	DOC	ACE	TANGO
CPU Type	AMD Athlon	AMD Athlon	Intel Xeon
CPU Speed (GHz)	2	2	1.9
Memory (GB)	1	1	2
Cache (KB)	512	512	2048
Compiler (gcc)	3.2.2	3.3	3.3.2
OS (Linux)	Red Hat 9	Red Hat 8	Debian
Kernel	2.4.20	2.4.20	2.4.23
`Avionics`	`NavDisplay`	`Airframe`	`GPS`
`RobotAssembly`	`WatchManager`	`PaletteManager` `RobotManager` `MWI`	`HMI`

Table 3. Testbed and Deployment Summary

3.3. Evaluating the Impact of Middleware Configuration on QoS

Below we describe how our MDD process can be used to help developers of DRE systems configure their middleware and application effectively.

Experiment description. To evaluate how the configurations of the components in the *Basic_SP* and avionics scenario affected QoS, we used the process described in Section 3.2 to empirically evaluate QoS (in both the cases round-trip latency) for different combination of the configuration settings (Step 2 in Section 3.2), resulting in 64 unique experiments. 16 experiments for configurations relevant to the `NavDisplay` and HMI components and 4 experiment for configurations relevant to `Airframe` and `WatchManager` component. Based on the roles of components, client (HMI and `NavDisplay`) or server, we differentiate the configurations as client- and server-side options. This separation is possible as the options themselves are mutually exclusive.

We first vary only the client-side options, keeping the server-side options to their default values. Once we determine the best configuration at the client-side, we use it and vary the server side configurations, allowing us to narrow down the number of experiments to 32. This approach yields the same results as compared to the full 64 experiment approach.

Analysis of results. Table 4 tabulates the latency distributions for the client-side display based components. We use the notation *A1*, *B2*, etc. to identify the individual options within each category. For example, the `-ORBConnectStrategy` value of `LF` is denoted as *J1*. The top 8 configurations (out of a possible 16) are shown in increasing order of latency values. A closer look at the values reveals a clear pattern of configuration options and its effect on QoS (end-to-end) latency. For example, the option G1 has the greatest effect on performance, *i.e.*, changing its value to G2 increases latency by $\sim$4μsecs for the *robot assembly* scenario and by $\sim$ 50μsecs in the *Basic_SP* scenario. After G, the option H influences latency the most, *i.e.*, changing its value from H1 to H2 worsens latency by $\sim$2μsecs in the first case and by $\sim$30μsecs in the second case. Table 5 shows the results for the `Airframe` and `WatchManager` components. We see that the settings for G have the greatest influence on latency, *i.e.*, changing its value increases latency the most in both the cases.

HMI Component		Nav_Display Component	
Setting	Latency (μsecs)	Setting	Latency (μsecs)
(G1, H1, I2, J2)	64.70	(G1, H1, I2, J2)	504
(G1, H1, I1, J2)	65.10	(G1, H1, I2, J1)	528
(G1, H1, I2, J1)	65.40	(G1, H1, I1, J2)	529
(G1, H1, I1, J1)	65.60	(G1, H1, I1, J1)	532
(G1, H2, I2, J2)	65.80	(G1, H2, I1, J1)	536
(G1, H2, I1, J1)	66.50	(G1, H2, I2, J1)	548
(G1, H2, I1, J2)	68.11	(G1, H2, I1, J2)	552
(G1, H2, I2, J1)	68.19	(G1, H2, I2, J1)	562
(G2, H1, I1, J2)	68.30	(G2, H1, I2, J2)	568
(..)		(..)	
Scenario 1		Scenario 2	

Table 4. Latency QoS distribution for the HMI & Nav_Display Components

Discussion. Our MDD process allows application developers to understand what configurations affect performance the most, which we refer to as the "primary effects." The categorization of the latency values in decreasing order of latency values also enables use of clustering analysis to create distinct sets of option categories as tuple spaces:

$$\{(x_1, val(x_1)\}, \{x_2, val(x_2)\}, ...\{x_n, val(x_n)\} \qquad (1)$$

where x_i denotes a configuration option, $val(x_i$ its setting. Within each set then the elements of the set being closely related. The sets themselves can be visualized as being separated by a distance d (similar to the concept of Hamming distance). Where moving from a configuration in one set to another results in an improvement/degradation in the QoS measures. As shown in the Table 4, for the robot assembly scenario moving from the first to second set of latency values incurs a minimum latency of $\sim$32μsecs ($d = 32\mu secs$).

Another interesting observation one can make from both the tables is that there is a similarity in the configurations that maximizes QoS for both the scenarios. For both the scenarios, the client-side primary effects are the same (G,H,I,J) in that order. Understanding and identifying similar patterns of configuration across multiple domains, would allow direct generation of the configuration information based on the QoS eliminating the need to (re) validate these across different configurations.

Airframe Component

Setting	Latency (μsecs)
(K0, L0)	452
(K1, L0)	459
(K0, L1)	462
(K1, L1)	467

Scenario 3

Watch Manager Component

Setting	Latency (μsecs)
(K0, L0)	55.5
(K1, L0)	56.8
(K0, L1)	59.6
(K1, L1)	60.2

Scenario 4

Table 5. Latency QoS distribution for Airframe & Watch_Manager Components

3.4. Evaluating the Impact of the Deployment Platform on QoS

Apart from evaluating the impact of configuration on performance, our MDD process can also be used to evaluate the performance impact of the underlying platform. For example, an application developer might be interested in knowing how does the latency vary in the robot assembly scenario if the `Airframe` component was placed on the node DOC and the `NavDisplay` component was placed on node TANGO.

Experiment description. To answer the question of how deployment platform affects QoS, we used the process described in Section 3.2 to change how components were deployed on the target platform. In particular, the only change required was in the model, *i.e.*, just changing the node mapping aspect[3]. All the required XML metadata are then generated by the PICML and BGML model interpreters. For this experiment, the configurations for the `Display` component was (G1, H1, I2, J2) and for Airframe (K0, L0). The respective configurations lead to minimal latencies in the earlier experiment.

Analysis of results. Table 6 tabulates the effect of changing the node mapping in the avionics scenario. As shown in the table, the choice was a bad one as the average roundtrip latency increased sharply to $\sim$1200μsecs whereas our default mapping, *i.e.*, one used in the earlier study is significantly better. The reason being that the `GPS` component that generates requests is on a slower node (ACE) than the default case (TANGO).

Discussion. Information such as this helps the application developers to make intelligent choices regarding how the underlying platform influences QoS and how best to optimize deployment decisions. Our MDD approach helped in quickly identifying how the underlying platform affects performance of the scenario. As described earlier, all the required changes were in the the MDD tools and the required

3 *Aspects* in GME are a mechanism to provide different views of the same model, where each aspect captures a concern of the same system

DOC	ACE	TANGO	latency μsecs
`Display`	`Airframe`	`GPS`	504
`Airframe`	`GPS`	`Display`	1206

Table 6. Deployment Analysis

XML meta-data was synthesized eliminating the accidental complexities involved in handcrafting low level source and XML meta-data files.

Our MDD process can also be used to provide feed-back to developers on how their deployment decisions (component node mappings) affect QoS. Many real-time systems are designed in a manner where the node mappings are determined *a priori*, which stems from the fact that the software tends to be closely tied to the hardware on which it runs. When nodes failure, however, it may be necessary to determine where to place the backup components such that the system still delivers an acceptable level of performance. In this case, the acceptable level of performance could be a certain latency threshold below which the DRE system cannot function in a proper manner.

4. Related Work

This section compares our work on MDD techniques in OCML and BGML with related research efforts, including middleware configuration techniques and generative techniques for empirical QoS evaluation.

Techniques for middleware configuration. A number of ORBs (such as VisiBroker, ORBacus, omniORB, and CIAO) provide mechanisms for configuring and customizing the middleware. For example, CIAO uses the ACE Service Configurator framework [10], which can be used to statically and dynamically configure components into middleware and applications. Likewise, Java ORBs provide an API for configuring applications based on *XML property files*. Key/value pairs for specific options are stored in an XML-formatted files and read by applications using XML parsers or a provided API.

Similar to our OCML approach, the Micro-QoSCORBA [5] middleware provides a GUI based tool to determine the type of architecture on which the system is being built. Application developers run the configuration tool to configure the application. Code generators then store this information in an application specific configuration file. These files along with the IDL specification are then used to generate customized client-side stubs and skeletons, ORB configuration files and build files. However, this tool is tied to its middleware and cannot be applied to other middleware solutions like OCML.

Empirical QoS evaluation. Several efforts use generative techniques similar to BGML to generate testcases and benchmarks for performance evaluation.

Performance Patterns [9] and *Performance Pattern Languages* (PPL) provide an automatable script-based framework within which extensive ORB endsystem performance benchmarks can be described efficiently and executed automatically. These patterns are embodied in the NetSpec tool developed at the Kansas University. The patterns themselves are written using PPL. Examples of such patterns include Cubit Tests (measuring (de) marshaling overhead), Client-Server benchmarks (simple client server two node approach for benchmarking) and Proxy benchmarks (introducing a proxy that acts as an intermediary between the client and server). The patterns approach can also be expressed using higher-level modeling constructs, *e.g.*, using modeling elements in the BGML modeling paradigm. These constructs can be used to denote patterns at the modeling level. Interpreters can then mimic the same functionality of scripts that parse the PPL language to generate the benchmarking scaffolding code. Providing modeling constructs reduces accidental complexities associated with PPL, as both middleware and application developers are not required to grapple with language syntax and semantics.

5. Concluding Remarks

This paper describes an MDD process that leverages the Options Configuration Modeling Language (OCML) and Benchmark Generation Modeling Language (BGML), which are MDD tools we developed to alleviate complexities associated with (1) choosing syntactically- and semantically-compatible sets of middleware configurations for specific application use cases and (2) evaluating the specified configurations and assisting middleware and application developers in deciding appropriate configurations for their QoS requirements, respectively. We qualitatively and quantitatively apply the process to two DRE systems in the avionics and the industrial manufacturing domains to evaluate (1) how middleware configurations affect system QoS, and (2) how deployment platforms influence QoS.

References

[1] 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 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 190–199, San Francisco, CA, Mar. 2005.

[2] M. A. de Miguel. QoS-Aware Component Frameworks. In *The* 10^{th} *International Workshop on Quality of Service (IWQoS 2002)*, Miami Beach, Florida, May 2002.

[3] A. Gokhale, K. Balasubramanian, J. Balasubramanian, A. S. Krishna, G. T. Edwards, G. Deng, E. Turkay, J. Parsons, and D. C. Schmidt. Model Driven Middleware: A New Paradigm for Deploying and Provisioning Distributed Real-time and Embedded Applications. *The Journal of Science of Computer Programming: Special Issue on Model Driven Architecture*, 2004.

[4] T. H. Harrison, D. L. Levine, and D. C. Schmidt. The Design and Performance of a Real-time CORBA Event Service. In *Proceedings of OOPSLA '97*, pages 184–199, Atlanta, GA, Oct. 1997. ACM.

[5] O. Haugan. *Confi guration and Code Generation Tools Targeting Middleware for Embedded Devices*. PhD thesis, Washington State University, Pullman, WA 99164, 2001.

[6] A. S. Krishna, D. C. Schmidt, A. Porter, A. Memon, and D. Sevilla-Ruiz. Improving the Quality of Performance-intensive Software via Model-integrated Distributed Continuous Quality Assurance. In *Proceedings of the 8th International Conference on Software Reuse*, Madrid, Spain, July 2004. ACM/IEEE.

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

[8] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. C. Schmidt, and B. Natarajan. Skoll: Distributed Continuous Quality Assurance. In *Proceedings of the 26th IEEE/ACM International Conference on Software Engineering*, Edinburgh, Scotland, May 2004. IEEE/ACM.

[9] S. Nimmagadda, C. Liyanaarnchchi, A. Gopinath, D. Niehaus, and A. Kaushal. Performance patterns: Automated scenario-based ORB performance evaluation. In *Conference on Object Oriented Technologies and Systems*, pages 15–28, San Diego, CA, 1999.

[10] D. C. Schmidt and S. D. Huston. *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Addison-Wesley, Reading, Massachusetts, 2002.

[11] D. C. Sharp and W. C. Roll. Model-Based Integration of Reusable Component-Based Avionics System. In *Proceedings of the Workshop on Model-Driven Embedded Systems in RTAS 2003*, May 2003.

[12] E. Turkay, A. Gokhale, and B. Natarajan. Addressing the Middleware Confi guration Challenges using Model-based Techniques. In *Proceedings of the 42nd Annual Southeast Conference*, Huntsville, AL, Apr. 2004. ACM.

[13] N. Wang, C. Gill, D. C. Schmidt, and V. Subramonian. Confi guring Real-time Aspects in Component Middleware. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'04)*, pages 1520–1537, Agia Napa, Cyprus, Oct. 2004.

[14] N. Wang, D. C. Schmidt, A. Gokhale, C. Rodrigues, B. Natarajan, J. P. Loyall, R. E. Schantz, and C. D. Gill. QoS-enabled Middleware. In Q. Mahmoud, editor, *Middleware for Communications*, pages 131–162. Wiley and Sons, New York, 2003.