Domain-Specific Modeling Languages for Configuring and Evaluating Enterprise DRE System Quality of Service

Stoyan Paunov, James Hill, and Douglas Schmidt
Vanderbilt University
Nashville, TN
{s.paunov,j.hill,d.schmidt}@vanderbilt.edu

Steve D. Baker and John M. Slaby
Raytheon
Portsmouth, RI
{Steve_D_Baker, john_m_slaby@raytheon.com}@raytheon.com

Abstract

The quality of service (QoS) of enterprise distributed real-time and embedded (DRE) systems can degrade under certain operating conditions and system architectures. This paper provides two contributions to research on model-driven development (MDD) tools and methods that help identify and rectify these QoS problems in component-based enterprise DRE systems. First, we show how MDD tools can be used to simplify and automate the evaluation of component-based DRE systems to identify QoS problems. Second, we show how MDD tools can be used to specify alternative QoS polices for component-based DRE systems and synthesize metadata automatically to simplify system (re)configurations that rectify QoS problems. We illustrate our MDD tools on a case study of multi-layer resource management services for shipboard computing systems that automate many aspects of power, navigation, command and control, and tactical operations.

1. Introduction

Enterprise distributed real-time and embedded (DRE) systems, such as total ship computing environments, air traffic control systems, and supervisory control and data acquisition (SCADA) systems, are growing in complexity and importance as more computing devices are being networked together to help automate tasks previously done by human operators. These types of systems are characterized by stringent quality-of-service (QoS) requirements, such as low latency and jitter, expected in real-time and embedded systems, as well as high throughput, scalability, and reliability expected in enterprise distributed systems. Therefore, it is hard enough to satisfy these QoS properties independently and even harder to satisfy them in concert.

A particularly vexing problem facing researchers and developers of large and layered enterprise DRE systems, such as major defense, aerospace, and commercial programs, is that the inadequacies of system architectures may not be ascertained until years into development. At the heart of this problem is the *serialized phasing* of layered system development, which postpones the discovery of design flaws that affect system QoS until late in the lifecycle, i.e., at integration time. A hallmark of serialized phasing is that application components are not created until *after* their underlying system infrastructure components, such as naming and discovery, event and notification, security and fault tolerance, and resource management.

In systems built using serialized phasing, the implementations, configurations, and deployments of infrastructure components are often not tested adequately under realistic workloads, which makes it hard to know how well they will satisfy key system QoS properties, such as the maximum number of clients the system can handle before it saturates and the effects of average and worst-case response time for various workloads, even after application components are completed. Moreover, the handcrafted software designs used in many enterprise DRE systems to address these concerns make it hard to conduct "what if" experiments with alternative system architectures and implementations to determine valid configurations to obtain performance goals for a particular workload. Making any significant changes to these types of handcrafted systems late in their lifecycle can be costly due to the impact on the design, implementation, deployment, and (re)validation of many application and infrastructure software/hardware components.

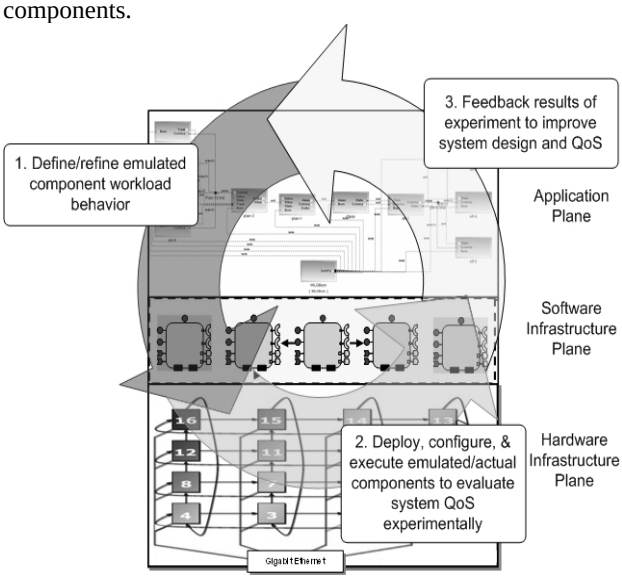


Figure 1. Evaluating the QoS of a Shipboard Computing Enterprise DRE System

To address these problems, this paper describes a methodology and associated suite of model-driven development (MDD) tools [7] shown in Figure 1 that are designed to simplify the:

1. **Emulation of application component behavior** in terms of computations, database activities, memory accesses, and network communication.
2. **Configuration, deployment, and execution** of these emulated application components atop actual infrastructure component deployments and configurations to determine their impact on QoS empirically in actual runtime environments.

3. **Process of feeding back the results** to enhance system architectures and components to improve QoS.

Over time as the actual application components mature, they can replace the emulated components, thereby providing an ever more realistic evaluation environment.

We are applying the methodology and MDD tools shown in Figure 1 in the context of the DARPA ARMS program [25], which is developing *multi-layer resource management (MLRM)* technologies to coordinate a grid of computers that manage and automate many aspects of Naval shipboard computing. This paper uses ARMS as a case study to show how our emulation-based approach yields several advantages over conventional methods for evaluating enterprise DRE system QoS. For example, it provides earlier feedback to developers, rather than forcing them to wait until the actual application components are complete to conduct performance experiments. Likewise, unlike pure simulation, models of application component configurations and deployment plans in our emulation-based methodology can be used directly in the final production system. To use emulation-based methods most effectively, however, requires the creation of MDD tools and *domain-specific modeling languages (DSMLs)* [10] to simplify and automate key aspects of QoS configuration and evaluation.

The remainder of this paper is organized as follows: Section 2 introduces the ARMS MLRM services that motivate our work on MDD tools and DSMLs and explains key challenges we faced when developing and evaluating these services; Section 3 describes the structure and functionality of two DSMLs: the *QoS Policy Modeling Language (QoSPML)*, which specifies QoS properties for component-based DRE systems and automatically synthesizes QoS configuration metadata, and the *Workload Modeling Language (WML)*, which simplifies and automates the specification of component behavior for evaluating end-to-end QoS in component-based DRE systems; Section 4 explains how we applied QoSPML and WML to our ARMS MLRM case study to resolve the challenges described in Section 2; Section 5 compares our work with related research; and Section 6 presents concluding remarks and lessons learned.

2. Case Study: Multi-Layer Resource Management for Shipboard Computing

This section describes the enterprise DRE system case study from the ARMS program that motivated our work on the MDD tools and DSMLs. This case study focuses on the ARMS *multi-layer resource management* (MLRM) framework for Naval shipboard computing systems and the challenges encountered while developing and evaluating it.

The MLRM services developed in ARMS are designed to support *total ship computing environments* (TSCEs), which form the basis for next-generation Naval programs. A TSCE is a coordinated grid of computers that manage many aspects of a ship's power, navigation, command and control, and tactical operations. To make TSCE an effective platform requires coordinated MLRM services that can support multiple QoS requirements, such as survivability, predictability, security, and efficient resource utilization.

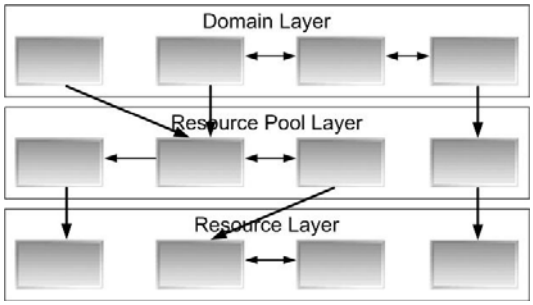


Figure 2. Component-based Architecture of the ARMS Multi-Layer Resource Manager (MLRM)

The ARMS MLRM integrates multiple resource management and control algorithms based on the CIAO [21] Lightweight CORBA Component Model (CCM) [15] and Real-time CORBA [14] mechanisms for (re)deploying and (re)configuring application components in DRE systems. As shown in Figure 2, the ARMS MLRM top *domain layer* contains infrastructure components that interact with the mission manager of TSCE by receiving command and policy inputs and passing them to the *resource pool layer*. The resource pool layer is an abstraction for a set of computer nodes managed by a *pool manager*. The pool manager is an infrastructure component that interacts with the *resource allocator* in the resource pool layer to run algorithms that deploy application components to various nodes within a resource pool. The actual computing resources reside in the third layer called the *resource layer*, which have infrastructure components called *node provisioners* that receive commands to spawn applications in every node from a pool manager. The *application string manager* is an infrastructure component that controls the resource utilization for a group of applications through the node provisioners. The ARMS MLRM services have hundreds of different types and instances of infrastructure components written in ~300,000 lines of C++ code and residing in ~750 files developed by different teams at different locations.

The component-based MLRM infrastructure for a TSCE is designed to support the highly heterogeneous environment in which long-lived shipboard computing systems operate. For example, the TSCE that provides the operational context for the ARMS MLRM services is designed to support different versions of (1) component middleware, such as CIAO and OpenCCM, (2) general-purpose operating systems, such as Linux and Solaris, (3) real-time operating systems, such as VxWorks and LynxOS, (4) hardware chipsets, such as x86, PowerPC, and SPARC processors, (5) a wide range of high-speed wired interconnects, such as Gigabit Ethernet and VME backplanes, and (6) different transport protocols, such as TCP/IP and SCTP.

In the first eighteen month phase of ARMS, we made the mistake of waiting until the integration phase of our schedule to begin benchmarking the system. Unfortunately, we quickly learned that none of the QoS requirements were met due to improperly designed and configured MLRM

infrastructure components. As a consequence, our schedule slipped and the process of reconfiguring and redeploying ARMS application and middleware components to meet the QoS requirements required significant manual effort.

To prevent the same problems in the second eighteen month phase of ARMS, we used the *Component Workload Emulator (CoWorkEr) Utilization Test Suite (CUTS)* [19], which is a system execution modeling toolkit [20] that helps systems engineers and software architects automate the steps in Figure 1 to emulate application behavior in component-based enterprise DRE systems, as well as collect and automate performance statistics provided by that emulation. We used CUTS to determine which configuration and deployment strategies (i.e., the customization and placement of components on nodes in the ARMS MLRM) could meet critical path end-to-end application QoS requirements. By emulating key properties the ARMS application components using CUTS and evaluating QoS results prior to the integration phase, we hoped to decrease the amount of time spent integrating and testing the actual components after they were completed.

While developing and evaluating the phase two ARMS MLRM services and the actual and emulated application components atop of it using CUTS and CIAO we encountered the challenges described below.

Challenge 1. Using standard Real-time CORBA APIs to configure the QoS of ARMS components. One way to ensure that ARMS application and MLRM infrastructure components exhibit the necessary QoS properties is to tightly couple the necessary QoS mechanisms into them *imperatively*. While this approach is common, it requires that developers be intimately familiar with Real-time CORBA to handle its accidental complexities. Moreover, hand-coding QoS properties into components imperatively can yield convoluted and inflexible implementations that are hard to evolve.

Challenge 2. Ensuring the right granularity of QoS. The ARMS application and infrastructure components have diverse characteristics and QoS requirements including, but not limited to, high throughput of continuously refreshed data, hard real-time deadlines associated with periodic processing, well-defined computational paths traversing multiple components, soft real-time processing of many tasks, and operator display and control requirements. Specifying the right granularity of QoS for these components imperatively using Real-time CORBA APIs is hard.

Challenge 3. Specifying system behavior to test infrastructure and resource utilization prior to system integration. While trying to assemble ARMS applications from reusable components, we ran into problems creating benchmarks that evaluated end-to-end QoS prior integration. In the first phase of ARMS we handcrafted code to test and collect benchmarking data about MLRM's infrastructure and resource utilization. As a result, much of the handcrafted benchmarking code could not be used in other contexts of the second phase of ARMS.

Challenge 4. Managing new complexities associated with CUTS. Emulation toolkits, such as CUTS, are constantly evolving to introduce new features and enhance or remove existing features. To handle these complexities, conventional methods include manually evolving existing resources, such as migrating source and descriptor files to handle new schemas. Although these methods suffice for small-scale prototypes, they scale poorly for enterprise DRE systems, e.g., managing the new complexities associated with the initial XML-based CUTS tools on the ARMS program introduced significant complexities.

Challenge 5. Managing large scale system configurations. In enterprise DRE systems like ARMS with many components, manually tracking every configuration for every component and assembly of components is hard. Hand-coding QoS properties into component implementations provides a way to track component configurations, but makes it hard for developers to review component specifications quickly. Even worse, if testing and benchmarking yields weak points in the system design, or functional requirements change, developers must manually read the code, find all relevant code snippets, and update each accordingly to reconfigure the necessary components, which is tedious and error-prone.

Challenge 6. Using metadata to configure components for QoS and to define behavioral components. Many middleware platforms, such as EJB, CCM, and .NET, have chosen XML as their configuration language since it enables different (1) application developers to create interoperable subsystems and (2) middleware developers to evolve different layers of their frameworks independently. Although XML is expressive, it is hard to manually read and write due to its accidental complexities. For example, although its elements are organized in a hierarchical form specified by the schema to which they conform, XML documents have a flat structure, are highly verbose, and lack intuitive relationships to the domain they represent. Evolving and debugging XML code manually is therefore extremely cumbersome, which makes it hard to reuse XML-based configurations.

Challenge 7. Refining system QoS properties. Enterprise DRE systems inevitably evolve due to changing functional requirements and specifications, deeper understanding of the domain, or hardware/software platform refresh. As a result, the associated QoS properties defined for a particular version of the system must also evolve. Hand-coding QoS properties therefore creates systems that scale poorly and fail to evolve rapidly to reflect new requirements and specifications.

The rest of this paper shows how we developed and applied MDD tools to address these challenges.

3. The Structure and Functionality of QoSPML and WML

Previous generations of enterprise DRE systems were largely handcrafted to provide precisely the capabilities required for a specific set of requirements and operating conditions, which made them inflexible and hard to evolve. Modern enterprise DRE systems are increasingly running atop standards-based middleware frameworks that support

the composition of loosely-coupled and distributable components capable of being reused in different contexts. Although component-based DRE systems are more flexible and easier to develop, other complexities still exist, such as automatically configuring application QoS policies and evaluating the QoS of groups of application and infrastructure components early in their lifecycle, i.e., before the system integration phase.

A promising means of addressing these complexities is to use MDD tools to create DSMLs that automate key portions of QoS-enabled component middleware configuration, deployment, and evaluation. This section describes the *QoS Modeling Language (QoSPML)* and *Workload Modeling Language (WML)*, which are DSMLs developed using the *Generic Modeling Environment (GME)* [10] to model Real-time CORBA and CUTS capabilities, respectively. QoSPML is used to specify QoS for application and infrastructure components in CIAO DRE systems and WML is used to specify behavior of CoWorkErs in CUTS.

3.1 Overview of QoSPML

3.1.1 Motivation. Standard distributed object computing (DOC) middleware provides application programming interfaces (APIs) that developers use to configure infrastructure and application components *imperatively* to provide predictability, satisfy timing constraints, and preserve prioritized access to shared resources. Standards-compliant [14] Real-time CORBA DOC middleware provides standard APIs and policies that allow enterprise DRE systems to configure and control various resources, such as (1) *processor resources* via priority mechanisms, thread pools, and synchronizers, for real-time applications with fixed priorities, (2) *communication resources* via protocol properties and explicit bindings to server objects using priority bands and private connections, and (3) *memory resources* via bounding the size of request buffers and thread pools.

The standard APIs for programming QoS policies in Real-time CORBA, however, are complicated. Moreover, the imperative model for programming these features requires application developers to have detailed knowledge of the underlying semantics, implementation, and order to configure these policies correctly. Over the past several years, however, QoS-enabled component middleware, such as CIAO [21], Qedo [16], and Prism [17], has evolved to support QoS configuration via standard XML descriptors that are specified *declaratively* and processed automatically by middleware deployment and configuration runtime environments [2].

Although using XML descriptors to configure the QoS properties of the system reduces the amount of code written imperatively, it also introduced new complexities, such as verbose syntax, lack of readability at scale, and a high degree of accidental complexity and fallibility. To alleviate these complexites, we developed the *Quality of Service Modeling Language (QoSPML)*, which is a DSML that configures key QoS properties of Real-time CORBA Component Model (CCM) [21] components.

3.1.2 Structure of QoSPML. The Real-time CORBA specification provides many QoS policies for controlling application behavior. To comply with the standards, and as illustrated in Figure 3, the following QoS policy types can be modeled in QoSPML: a *priority model policy*, a *thread pool policy*, and a *connection policy*. QoSPML organizes policies into logical groups named *policy sets*, which enable the specification of alternative configurations in the same QoS model. The connection and thread pool policies are modeled as references to actual resources to permit resource sharing among separate policy sets. For example, the same thread pool policy can be shared between two different policy sets, but both policy sets define a different connection and priority policy.

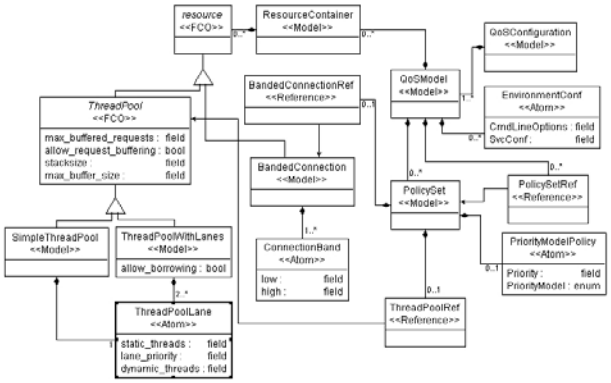


Figure 3. GME Metamodel of QoSPML

3.1.3 Functionality of QoSPML. QoSPML enables developers of enterprise DRE systems to specify and control the following Real-time CORBA QoS policies via visual models:

Propagation of priorities. Real-time CORBA defines two ways to propagate end-to-end priorities: *server-declared* and *client-propagated*. In the server-declared model the priority at which requests run is determined by the server, whereas in the client-propagated model the server honors the request priority assigned by the client. These priority propagation schemes are modeled using the *PriorityModelPolicy* element. The type of propagation scheme is selected using the *PriorityModel* enumeration attribute and the priority is specified with the priority attribute.

Specification of threading model. Each *ThreadPool* model encapsulates data that specifies the properties of a thread pool in Real-time CORBA. For example, developers can set the stack size associated with the thread pool, allow/disallow request buffering and set the maximum number of requests to be buffered and the corresponding buffer size. A thread pool has a set number of pre-spawned *static threads* and up to a maximum limit of *dynamic threads* spawned on-demand only if all static threads are in use. QoSPML supports two types of thread pools: (1) the *SimpleThreadPool* model, which has a single priority lane and allows lower priority client-propagated requests to exhaust all the static and dynamic threads and starve higher priority requests and (2) the *ThreadPoolWithLanes* model, which creates multiple lanes for different priorities to prevent lower priority client-propagated requests from exhausting all pool's threads. If thread borrowing is enabled, higher

priority requests can temporarily promote a thread from a lower priority pool to run the request at the higher priority.

Specification of connection bands. Another Real-time CORBA feature supported by QoSPML is banded connections, which are specified by the *BandedConnections* policy element. These connections are logically divided into *ConnectionBands* and have a *low* and *high* attribute for specifying its range of priorities.

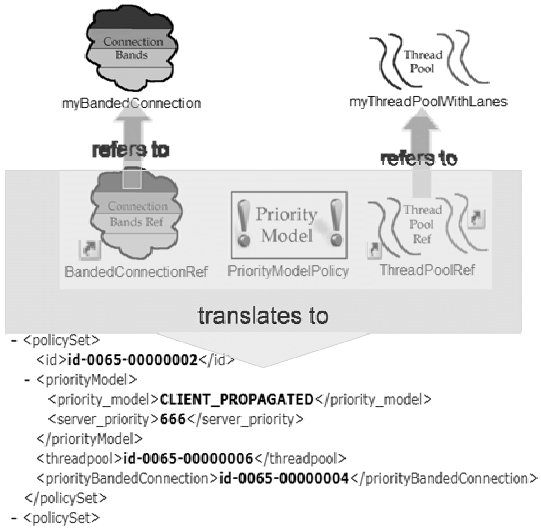


Figure 4. Example QoS Configuration QoSPML and a Snippet of the XML Interpretation

Figure 4 illustrates portions of an example QoS configuration using QoSPML. The highlighted region of the figure illustrates the priority model policy and defines references to the connection bands and the thread pool with lanes elements. The XML below the highlighted region is its interpretation in the generated XML descriptor file.

In summary, QoSPML addresses challenges 1, 2, 5, 6, and 7 from Section 2. In particular, it allows developers to avoid writing applications that use the convoluted Real-time CORBA imperative APIs directly, while still providing control over QoS policies. QoSPML also enables application developers and performance engineers to provision the QoS of applications in enterprise DRE systems via higher-level models that QoSPML converts automatically into lower-level Real-time CORBA QoS policies expressed using XML.

3.2. Overview of WML

3.2.1 Motivation. Emulation and benchmarking environments [1] provide methods of configuring the behavior of emulated components, which in many instances is done using external metadata. In prior work [19], we developed the *Component Workload Emulator (CoWorkEr) Utilization Test Suit (CUTS)*.described in Section 2. The initial version of CUTS used XML descriptor files to specify the behavior of application components in terms of their computations, database activities, memory accesses, and event-based network communication, as shown in Figure 5.

While the initial version of CUTS was useful, experience applying it to the ARMS shipboard computing enterprise DRE system revealed that it introduced new complexities related to understanding the various types of workloads and behaviors allowed in a CoWorkEr. In particular, learning the convoluted XML syntax used to specify CoWorkEr behavior was tedious, error-prone, and required detailed knowledge of its syntax and semantics. Moreover, as CUTS evolved, the concrete semantics and syntax used in its XML descriptor files changed and required low-level knowledge to transform existing descriptor files.

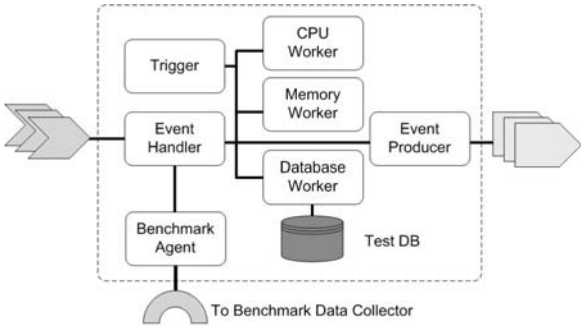


Figure 5. A CoWorkEr Assembly in CUTS

To alleviate these complexities, we developed WML *Workload Modeling Language (WML)*, which is a DSML that allows users and performance engineers to define the behavior of components in an enterprise DRE system graphically using models. WML then automatically and correctly transforms these models into XML metadata descriptor files, which can be processed automatically by CUTS to define the behavior of the emulated components being evaluated.

3.2.2 Structure of WML. Figure 6 shows the GME metamodel for WML. As illustrated in the figure, either the

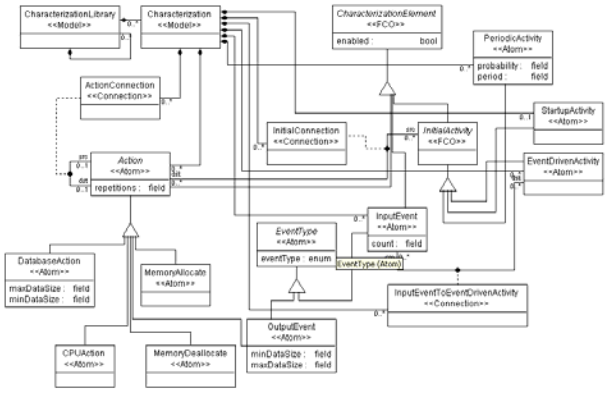


Figure 6. GME Metamodel of WML

Characterization or *CharacterizationLibrary* model is the basis for defining all behavior is WML. Each *Characterization* model contains zero or more *Action* elements, such as *DatabaseAction*, *MemoryAllocate*, *MemoryDeallocate*, *CPUAction*, and *OutputEvent*, which are interconnected using *ActionConnection* connections. *Characterization* models also contain different types of workloads, such as *PeriodicActivity*, *StartupActivity* and *EventDrivenActivity*. Lastly, characterization models contain zero or more *In-*

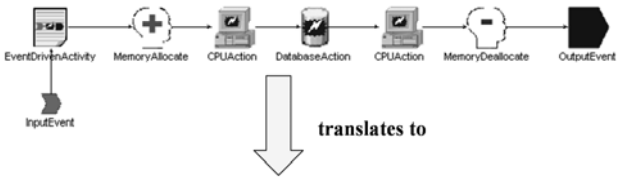
putEvent elements that specify the handling of events by CoWorkEr components in CUTS.

Characterization libraries are models in GME that allow grouping of characterization models for organizational purposes. For example, characterization libraries are ideal for hierarchically grouping characterizations that have similar traits.

3.2.3 Functionality of WML. WML enables users and performance engineers who evaluate enterprise DRE systems to specify and control the following CoWorkEr policies via visual models:

Specification of worker actions. As shown in Figure 5, each CoWorkEr assembly-based component in CUTS contains four monolithic worker components: *DatabaseWorker*, *MemoryWorker*, *CPUWorker*, and *EventProducer*. These components can be assigned work and the number of repetitions to repeat the specified work using the following action types: (1) a *DatabaseAction* for specifying work completed by a *DatabaseWorker* component and the associated minimum and maximum size of data inserted to and deleted from the test database, (2) a *MemoryAllocate* and *MemoryDeallocate* for specifying the *MemoryWorker's* memory allocation and deallocation, respectively, (3) a *CPUAction* for specifying work to be completed by the *CPUWorker*, and (4) an *OutputEvent* for specifying the transmission of a user-defined event and the associated minimum and maximum size of data transmitted, which is handled by an *EventProducer*.

Specification of workloads. The *ActionConnection* connector element is used to connect the previously described actions to one another. When a one of more action elements is connected sequentially, a characterization string is defined. Figure 7 illustrates an example characterization string for an *EventDrivenActivity* in WML and a snippet of the generated XML metadata. This example shows that receiving an event causes the following sequence of actions: (1) allocate memory, (2) perform CPU operations, (3) perform database operations, (4) perform CPU operations, (5) deallocate memory, and (6) transmit an event.




```
<InputEvent eventType="CommandSimEvent" count="1" />
<Workload>
  <MemoryAction repetitions="35" operation="ALLOCATE" />
  <CPUAction repetitions="15" />
  <DatabaseAction repetitions="30" />
  <CPUAction repetitions="30" />
  <MemoryAction repetitions="35" operation="DEALLOCATE" />
  <PublicationAction repetitions="1" eventType="TrackSimEvent" dataSize="16" />
</Workload>
```

Figure 7. Example of a Workload Specification in WML and a Snippet of the XML Interpretation

The execution of a characterization string by a worker can occur at three stages of execution: *InitialActivity*, *PeriodicActivity*, or *EventDrivenActivity*. Initial activity is specified by connecting an *InitialActivity* element with the first action element of the characterization string. Work of this type is executed in the **`ccm_activate()`** hook method of a CoWorkEr during the startup time of CUTS. Periodic activity is specified by connecting a *PeriodicActivity* element to the first action in the characterization string. Work of this type contains a probability factor to define non-deterministic behavior to perform at user-defined intervals. Event-driven activity is specified in two steps: (1) an *EventType* element is connected to an *EventDrivenActivity* element and (2) the *EventDrivenActivity* is connected to the first action in a characterization string. Since the *EventType* element must be connected to an *EventDrivenActivity*, users can associate multiple *EventTypes* with an *EventDrivenActivity*. Work of this type is processed after receiving a user-defined event, and the number of events needed to perform work.

Figure 8. Example WML Model with Disabled Actions

Temporarily disabling actions from workload specification. The *CharacterizationElement* is the base type for all action elements in WML. Since all actions derive from this single element, we defined an attribute called *enabled* to allow inclusion or exclusion of actions in the generated metadata descriptors, depending on whether enabled is set to true or false, respectively. Moreover, because an enabled element has the same image as one that is not enabled, we created a GME decorator that masks a cancellation icon on top of elements that are not enabled, which allowed us to temporarily disable portions of the workload for certain kinds of analysis without losing the original definition and without having to make copies of the workload model for minor variations in the behavior. Figure 8 illustrates the same workload specification in Figure 7 with the *MemoryAllocate* and *CPUAction* elements disabled.

In summary, WML addresses challenges 3, 4, and 6 from Section 2. In particular, it removes the complexity of developers having to learn the syntax and semantic of the XML metadata descriptor files for defining behavior in CUTS, which is especially useful as CUTS evolves. WML also allows developers to capture behavior in models, which can be archived and used in other domains or projects that utilize CUTS.

4. Resolving ARMS MLRM Challenges with QoSPML and WML

We now examine how the QoSPML and WML DSMLs described in Section 3 can be applied to address the challenges discussed in Section 2 that arose when developing, evolving, and evaluating the ARMS MLRM case study for shipboard computing enterprise DRE systems.

4.1 Configuring MLRM Infrastructure and Application Components for QoS

Challenge 1 in Section 2 described the difficulties associated with writing applications using the Real-time CORBA API imperatively. QoSPML provides a more scalable and robust approach to configuring the QoS properties of the CCM components being developed, or reused, by enabling developers to specify these properties declaratively and visually. Developers use QoSPML to specify the QoS policies that determine the threading, connection, and priority propagation mechanisms used for a particular component and group these policies into policy sets. The specified mechanisms are modeled in terms of the actual system resources that implement them, which enables different policy sets to share the same instance of a resource at the middleware layer. QoSPML also enables developers to verify the correctness of their models by providing constraint checking mechanisms embedded in the language. The QoS models can be interpreted by means of a model interpreter that generates correct metadata descriptors understood by the Real-time CCM middleware runtime.

In the context of ARMS, QoSPML facilitates the seamless configuration of components for QoS in each layer of MLRM because it allows application developers to bypass the tedious tasks of hard-coding the Real-time CORBA code or hand-crafting the XML descriptors that can be used to describe the QoS configuration.

4.2 Using QoSPML to Meet the QoS Needs of the Various MLRM Subsystems

Challenge 2 in Section 2 discussed the diversity of services and QoS requirements supported by the ARMS MLRM infrastructure. Each of these QoS requirements is hard to achieve separately and even harder to achieve in combination. Fortunately, QoSPML detaches configuration developers from the inherent complexities of the configuration code and allows them to concentrate on the business logic of application components.

In the context of ARMS, a major cause of missed deadlines is *priority inversions*, where lower priority requests access a resource at the expense of higher priority requests. Priority inversions must be prevented or bounded since they can cause the ARMS applications to miss their deadlines. QoSPML's *ThreadPoolWithLanes* element can be used in conjunction with the *BandedConnection* and the *PriorityModelPolicy* elements to configure MLRM properly and reduce priority inversions.

The *ThreadPoolWithLanes* feature of QoSPML can be used to meet some of the QoS needs of ARMS. By using this feature, the MLRM will be configured so that lower priority requests cannot exhaust threads allocated for higher priority requests when a request is executed. Long-running requests in MLRM can also exhaust the maximum number of static threads, causing the system to miss deadlines. QoSPML therefore allows ARMS MLRM developers to specify the maximum number of dynamically spawned threads to better manage long running requests and periodic high loads.

The *BandedConnection* element in QoSPML allows MLRM developers to control network resources effectively by separating lower and higher priority requests so they do not share the same multiplexed connection. In multiplexed connections, requests are queued and serviced on a FIFO basis, where low priority requests could be scheduled first. By using priority bands, developers can partition the communication links between application and MLRM components based on a range of priority values. This QoS policy ensures that low priority requests travel separate paths from high priority requests, therefore preventing priority inversions. A beneficial side-effect of this partitioning mechanism is that it decreases latency and improves response time.

It is also important to ensure the portability of priorities in cases when ARMS application and MLRM component run atop different OS platforms with different priority ranges. Once the necessary priority mappings have been defined, QoSPML's *PriorityModelPolicy* feature can be used to preserve the end-to-end priorities and to define the priority propagation scheme used to configure Real-time CORBA policies. As discussed in Section 3.1.3, there are currently two types of policies: server declared and client propagated.

4.3 Specifying System Behavior to Test Infrastructure and Available Resources Prior to System Development and Integration

Challenge 3 in Section 2 discussed the complexities of specifying application component behavior to evaluate the QoS of enterprise DRE system infrastructure implementations, deployments, and configurations. In many instances, behavioral code is handcrafted and hard-coded, which makes it confined to the current project. Fortunately, WML provides an alternative for specifying system behavior using its high-level representation of workload behavior, which makes it easy to apply and reusable in other projects and areas.

In the context of ARMS, one of the main capabilities of the MLRM is determining optimal (re)deployment and (re)configuration strategies of a component-based system depending on its current behavior. Unfortunately, many of the application components that the ARMS MLRM will manage are not available for extended periods of time during MLRM development, so we used CUTS to emulate their behavior and guide the MLRM design and implementation. Although CUTS allows the MLRM to be tested prior to receiving the actual components on which it will act, we needed to create behavior specifications that will drive CUTS's QoS experiments. WLM therefore provides testers of MLRM with the necessary tools to (re)define and (re)configure the behavior of components in CUTS without having to handcraft or hardcode the behavior of emulated components.

4.4 Managing New Complexities Associated with CUTS

Challenge 4 in Section 2 described the challenge of managing new complexities of CUTS as features are added or removed. WML addresses this challenge evolving in parallel with CUTS. As features are added and removed

from CUTS, therefore, WML reflects these changes in its modeling language and interpreter, which removes the complexity of managing syntactic changes in the underlying XML metadata descriptors from end users.

In the context of ARMS, the MLRM software has evolved continuously over several years, during which time CUTS has also evolved. Using WML to model the behavior specification for CUTS to test the MLRM therefore minimizes the complexity of manually managing the XML metadata descriptors as CUTS evolves. For example, as features are added, enhanced, and removed from CUTS, the existing WML models in use by MLRM will need to reflect these modifications.

4.5 Using MDD Tools to Generate XML Metadata

Challenge 6 in Section 2 discussed the complexities introduced by applying XML metadata to configure DRE systems. We used QoSPML and WML to bypass the XML coding necessary to configure application and middleware components, which raised the level of abstraction by means of visual modeling and DSMLs. We used these MDD tools to formally model the configuration space and enable the automatic generation of configuration code. QoSPML and WML therefore allow developers to concentrate on the actual design of the enterprise DRE system, while shielding them from the accidental complexities of the configuration artifacts. They also make rapid (re)configuration possible, thus allowing developers to evolve their systems more conveniently.

In the context of ARMS, we had initially used validation tools, such as XML SPY, to verify the syntactic correctness of the XML metadata against the schema to which they conform. Unfortunately, these validation tools miss many problems with handcrafted XML. In contrast, QoSPML and WML provide a more effective solution since they use GME's powerful constraint-checking facility to ensure that models are correct-by-construction. The XML descriptors they generate are therefore correct as long as the QoSPML and WML interpreters conform to the XML schemas that describe the documents.

4.6 Managing and Refining the System Configuration Space

Challenge 5 in Section 2 described how managing a large amount of XML metadata is cumbersome and hard to extract information from it without significant effort. Likewise, challenge 7 in Section 2 described how it is even harder to modify XML-based configuration files in response to (1) changing system requirements, (2) better understanding of the QoS needs to the application, or (3) uncovered design weaknesses. For example, even a single typo in an XML file can compromise the document structure and cause the parsers to fail, which make handcrafted XML files extremely hard to manage and evolve.

In the context of ARMS, by using QoSPML and WML ARMS developers no longer have to deal with XML metadata directly. Instead, they can use visual models to perform their tasks from a domain-centric perspective. After making the necessary changes to the system configuration, they can regenerate the descriptors quickly and correctly, which scales much better for enterprise DRE systems like ARMS.

5. Related Work

This section compares our work on using DSMLs and MDD tools for configuring and evaluating QoS for enterprise DRE system with related research efforts.

DSMLs for Modeling System Behavior. Several DSMLs exist that can be used to model system behavior. For example, KLAPER [4] is a modeling language for specifying system behavior of component-based systems. Similar to WML, KLAPER allows the specification of workloads, such as resource utilization, but it does not specify handling of events. In KLAPER, component behavior is based on the provided and required services, whereas WML defines behavior based on resource utilization within a component. WML thus enhances KLAPER by allowing sequential specification of resource utilization, specification of event transmission, and receipt, classification of workload types, such as event-driven, periodic, or startup.

The *Action Language* [13] is another modeling language that can be used to define system behavior using semantics similar to statecharts [5]. Similar to WLM, the Action Language allows the specification of event handling, actions and workloads based on event-handling, but not periodic or system startup workloads as an *apparent* language feature. In contrast, WLM focuses on resource utilization in DRE systems, places guards and conditions only at the beginning of a workload specification instead of on each individual action in a workload specification, and is tailored to the functionality of CoWorkEr components in CUTS.

WinFX Workflow [26] is a modeling language developed by Microsoft Corporation that is part of the Windows Workflow Foundation [27]. It allows developers to express programs as declarative, long-running processes called *workflows*. Similar to WinFX, WML permits the expression of *workflows* for a system and its processes. WinFX, however, is tightly coupled to the Microsoft .NET framework, whereas WML can be applied across multiple SOAs, including Microsoft .NET, Real-time CCM, and EJB.

DSMLs for Configuring QoS. Several DSMLs are defined to capture QoS requirements in DRE systems. The *Distributed QoS Modeling Environment (DQME)* [22] is a DSML for modeling QoS, which is designed for runtime the adaptive capabilities provided by the Quality Objects *(QuO)* adaptive QoS middleware framework. The *Object Constraint Modeling Language (OCML)* [9] is another domain-independent modeling language that simplifies the specification and validation of complex DRE middleware and application configurations and their QoS requirements. Although DQME and OCML can be used to specify QoS for DRE system, QoSPML augments and focuses these types of capabilities by capturing the specifications and

capabilities of Real-time CORBA and integrating them into CIAO to provide QoS for Real-time CCM.

QoS-UniFrame [11] is a QoS modeling language that uses Petri-nets to model QoS requirements by producing reachability graphs to determine which QoS requirements are capable of being meet. Although QoS-UniFrame has a powerful constraint manager to verify that various QoS requirements will not conflict, and modeling which combinations of QoS properties will be feasible for the DRE system, it is designed for AspectJ [8]. In contrast, QoSPML's constraint management is based on valid QoS configurations in Real-time CORBA in C++ and Java.

Performance evaluation techniques using RT-UML. [18], [23], and [24] propose component-based software performance engineering *(CB-SPE)* techniques for modeling and evaluating the behavior of component-based systems. These CB-SPE techniques use RT-UML to define services and QoS policies for components, though modeling system behavior is future work. The authors state that this technique is designed to be supported by external *simulation* tools, which are still under development. WML extends these efforts by providing a complete DSML that allows developers to specify a component-based system behavior, which is *emulated* by an existing MDD toolsuite called CUTS. QoSPML also extends related work on CB-SPE by allowing system developers to specify QoS policies that are configured into the underlying middleware and emulated components.

6. Concluding Remarks

Our prior work focuses on the design, optimization, and configuration of Real-time CORBA and Real-time CCM [21] middleware and the development of the CUTS application emulation toolkit [19]. This paper focused on our experience gained integrating and applying the QoSPML and WML DSMLs to the DARPA ARMS *multi-layer resource management (MLRM)* services for Naval shipboard computing enterprise DRE systems. The benefits we have observed by applying our MDD tools and DSMLs to the component-based ARMS applications and infrastructure services thus far include:

- Using highly configurable component middleware, such as CIAO [21] and DAnCE [2], enhances software development quality and productivity. Unfortunately it also introduces extra complexities, which are hard to handle in an *ad hoc* manner for enterprise DRE systems.
- Using DSMLs expedites application development and system QoS evaluation by providing proper integration and tuning of MDD tools with the underlying component middleware infrastructure. In our ARMS MLRM case study, the QoSPML and WML DSMLs simplify the evaluation of many different system configurations and facilitate QoS-related "what if" scenarios prior to the integration or even the development phase. These DSMLs also play an important role in enterprise DRE system evolution because they provide a way to evaluate alternative system configurations visually and empirically.
- The QoSPML and WML DSMLs complement each other in various ways. For example, developers can use QoSPML to configure the QoS properties of CoWorkEr and infrastructure components. Likewise, developers to use the results from WML-based CUTS QoS evaluations to drive the evolution of QoSPML models. Developers can also experiment with various QoS properties captured in QoSPML models to determine which ones perform better with a specific WML-based CUTS QoS configuration.
- QoSPML and WML can help to reduce the learning curve for end users. For example, in the ARMS MLRM case study, application developers needed little knowledge of the Real-time CORBA QoS policy APIs and the CIAO XML descriptors that declaratively configure these policies in Real-time CCM. Instead, they used higher-level models of QoS policy provisioning mechanisms provided by QoSPML and avoided the need to learn the low-level XML-related details of CUTS by using WML to specify the desired application component behavior.

Although our MDD tools solve many hard problems encountered in the ARMS program, they also leave room for improvement and future work:

- Despite the fact that QoSPML facilitates the QoS configuration of enterprise DRE systems based on Real-time CORBA, developers are still faced with the question of what constitutes a "good" configuration. While WML and CUTS simplify the effort required to address this question experimentally, developers are still ultimately responsible for determining the appropriate configurations.
- Although QoSPML and WML remove many complexities associated with handcrafted solutions, developers are still faced with the challenge of evolving existing models when the respective domain evolves. Although model evolution tools, such as GREAT [6], exist they are hard to use and only provide partially automated solutions.

This experience motivates further research on automated QoS configuration and deployment techniques to uncover effective heuristics to guide us in the complicated process of enterprise DRE system QoS evaluation, as well as further research on model migration to simplify the process evolving DSMLs as the understanding of their respective domains matures.

The open-source CIAO QoS-enabled component middleware can be downloaded from www.dre.vanderbilt.edu/CIAO. The GME domain-specific modeling toolkit is available in open-source format and can be downloaded from www.isis.vanderbilt.edu/projects/GME. QoSPML and WML are being integrated with the open-source CoSMIC (www.dre.vanderbilt.edu/cosmic) and are available from the contact author upon request.

8. References

[1] Denaro, G., Polini, A., Wolfgang Emmerich, W. "Early Performance Testing of Distributed Software Applications," *4th International Workshop on Software and Performance*, Jan 2004.

[2] Deng, G., Balasubramanian, J., and Otte, W., Schmidt, D. and Gokhale, A. "DAnCE: A QoS-enabled Component Deployment and Conguration Engine," *Proceedings of the 3rd Working Conference on Component Deployment*. Grenoble, France, Nov 2005.

[3] Gokhale, A., Balasubramanian, K., Balasubramanian, J., Krishna, A., Edwards, G., Deng, G., Turkay, E., Parsons, J. , and Schmidt, D. (2005). "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.* [in press].

[4] Grassi, V., Mirandola, R., and Sabetta, A., "From Design to Analysis Models: A Kernel Language for Performance and Reliability Analysis of Component-based Systems," *Fifth International Workshop on Software and Performance*, Jul 2005.

[5] Harel, D., "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming*, 1987.

[6] Karsai G. and Agrawal A. and Shi F. and Sprinkle J., "On the use of Graph Transformations in the Formal Specification of Computer-Based Systems," *Proceedings of IEEE TC-ECBS and IFIP10.1 Joint Workshop on Formal Specifications of Computer-Based Systems*, Huntsville, AL, Apr 2003.

[7] Karsai, G., Sztipanovits, J., Ledeczi, A. and Bapty, T. "Model-Integrated Development of Embedded Software," *Proceedings of the IEEE*, Jan 2003.

[8] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G., "Getting started with AspectJ", *Communications of the ACM*, 2001.

[9] Krishna, A., Turkay, E., Gokhale, A., and Schmidt, D., "Model-Driven Techniques for Evaluating the QoS of Middleware Configurations for DRE Systems", *11th IEEE Real-Time and Embedded Technology and Applications Symposium*, Mar 2005.

[10] Ledeczi, A., Maroti, M., Karsai G., and Nordstrom G., "Metaprogrammable Toolkit for Model-Integrated Computing*", Proceedings of the IEEE International Conference on the Engineering of Computer-Based Systems Conference*, Mar 1999.

[11] Liu, S., Bryant,B., Gray, J., Raje, R., Olson, M., and Auguston, M., "QoS-UniFrame: A Petri Net-Based Modeling Approach to Assure QoS Requirements of Distributed Real-Time and Embedded Systems", *12th IEEE International Conference on the Engineering of Computer-Based Systems (ECBS 2005)*, Apr 2005.

[12] Nordstrom, G.,*"Formalizing the Specification of Model Integrated Program Synthesis Environments"*, IEEE Aerospace Conference, Mar 2000.

[13] Nordstrom, S., Shetty, S., Yao, D., Ahuja, S., Neema, S., Bapty, T., "The Action Language: Refining a Behavioral Modeling Language", *12th IEEE International Conference on the Engineering of Computer-Based Systems*, Apr 2005.

[14] Object Management Group (2002, Aug). Real-time CORBA Specification. Ed. OMG Document formal/02-08-02.

[15] Object Group Management (2003, May). Light Weight CORBA Component Model Revised Submission, Ed. OMG Document realtime/03-05-05.

[16] Ritter, T., Born, M., Unterschutz, T., and Weis, T., "A QoS Metamodel and its Realization in a CORBA Component Infrastructure," *Proceedings of the 36th Hawaii International Conference on System Sciences*, Honolulu, Hawaii, Jan 2003.

[17] Roll, W. "Towards Model-Based and CCM-Based Applications for Real-Time Systems," *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, IEEE/IFIP, Hakodate, Hokkaido, Japan, May 2003.

[18] Grassi, V. and Mirandola, R., "Towards Automatic Compositional Performance Analysis of Component-based Systems," *Proceedings of the 4th International Workshop on Software and Performance*, Jan 2004.

[19] Slaby, J., Baker, S., Hill, J., Schmidt, D., "Applying System Execution Modeling Tools to Evaluate Enterprise Distributed Real-time and Embedded System QoS," ISIS Technical Report ISIS-05-604, Oct 2005.

[20] Smith, C., *Performance Engineering of Software Systems*, Addison Wesley, 1990.

[21] Wang, N., Gill, C., Schmidt, D. and Subramonian, V. "Configuring Real-time Aspects in Component Middleware," *Proceedings of the International Symposium on Distributed Objects and Applications*. Agia Napa, Cyprus, Oct 2004.

[22] Ye, J., Loyall, J., Shapiro, R., Schantz, R., Neema, S., Abdelwahed, S., Mahadevan, N., Koets, M., Varner, D., "Model-Based Approach to Designing QoS Adaptive Applications", *25th International Real-Time Systems Symposium*, May 2004.

[23] Bertolino, A. and Mirandola, R., "Software Performance Engineering of Component-based Systems," *Proceedings of the 4th International Workshop on Software and Performance*, Jan 2004

[24] Bertolino, A. and Mirandola, R., "Towards Component-Based Software Performance Engineering" *Proceedings of the 6th ICSE Workshop on Component-Based Software Engineering*, May,2003.

[25] ARMS DARPA Website, dtsn.darpa.mil/ixodarpatech/ixo_FeatureDetail.asp?id=6, Jan 2006.

[26] Box, D. and Shukla, D., "WinFX workflow: Simplify Development with the Declarative Model of Windows Workflow Foundation," *MSDN Magazine, 21*, Jan 2006.

[27] Andrew, P., Conard, J, Conrad, J., Flandars, J., & Woodgate, S., *Presenting Windows Workflow Foundation*, Sams Publishing, 2005.