Model-based Software Tools for Configuring and Customizing Middleware for Distributed Real-time and Embedded Systems[*]

Arvind S. Krishna, Emre Turkay, Aniruddha Gokhale, and Douglas C. Schmidt
{arvindk,turkaye,gokhale,schmidt}@dre.vanderbilt.edu
Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN

Abstract

Middleware is increasingly been used to develop and deploy large-scale distributed real-time and embedded (DRE) systems in domains ranging from avionics to industrial process control and financial services. To support a wide range of DRE systems with diverse QoS needs, middleware platforms often provide scores of options and configuration parameters that enable it to be customized and tuned for different use cases. Supporting this level of flexibility, however, can significantly complicate middleware and application configuration. This problem is exacerbated for developers of DRE systems by the lack of documented patterns of middleware configuration and customization.

This paper provides three contributions based on model-driven generative programming techniques that can be used by middleware developers to codify reusable patterns of middleware configuration and customization. First, we describe the Options Configuration Modeling Language (OCML), which is a Model-Driven Middleware (MDM) tool that simplifies the specification and validation of complex DRE middleware and application configurations. Second, we describe the Benchmark Generation Modeling Language (BGML), which is an MDM tool that synthesizes benchmarking testsuites to empirically analyze the QoS performance of OCML-configured DRE systems and feedback results into OCML models to help optimize domain-specific configurations and customizations. Third, we qualitatively and quantitatively evaluate our OCML and BGML MDM tools in the context of different DRE system scenarios that help the middleware developers to codify the configuration and customization patterns of reuse. Our results indicate that an MDM approach results in comparable accuracy and correctness to a handcrafted approach. Moreover, by applying MDM tools to avionics mission computing, ∼85% of the code required to configure and empirically validate middleware configurations is generated automatically, which helps eliminate accidental complexities incurred when handcrafting these configurations.

Keywords: QoS-enabled Middleware, Model-based Middleware Deployment & Configuration, Empirical Validation.

[*]This work was sponsored in part by AFRL Contract#F33615-03-C-4112 for DARPA PCES Program, NSF ITR CCR-0312859 and a grant from Siemens CT.

1 Introduction

Emerging trends and challenges. Over the past several years, many R&D efforts [1, 2, 3, 4, 5] have focused on developing *quality of service (QoS)-enabled middleware* as a means to enhance the development, time-to-market, and reuse of distributed real-time and embedded (DRE) systems, such as hot rolling mills, tele-immersion environments, fly-by-wire aircraft, and total ship computing environments. QoS-enabled middleware is systems software that resides between the applications and the underlying operating systems, network protocol stacks, and hardware and isolates DRE applications from lower-level infrastructure complexities, such as heterogeneous platforms and error-prone network programming mechanisms.

An inherent quality of QoS-enabled middleware is its high degree of *flexibility*, which stems from the need to support a wide range of DRE systems that need to (1) run on many hardware and OS platforms, (2) interoperate with many versions of related software frameworks and tools, and (3) provide support for *end-to-end QoS properties*, such as low latency and bounded jitter; fault propagation/recovery across distribution boundaries; authentication and authorization; and weight, power consumption, and memory footprint constraints. These requirements of DRE systems must be met via compile- and run-time selection and fine tuning of the middleware configuration options, which are used for fine tuning QoS properties of the middleware, and customization options, which are used for trading off functionality and footprint of the middleware to suit the DRE system requirements. For example, web servers (*e.g.*, Apache [6]), object request brokers (*e.g.*, TAO [7]), and databases (*e.g.*, Oracle [8]) have scores of configuration options and customization parameters that can be used to tailor middleware so that the desired QoS properties are delivered to the applications.

The first generation of QoS-enabled middleware (such as Real-time CORBA [9]) have been applied successfully to small- and medium-scale DRE systems, such as avionics mission computing [10, 11], dynamic mission replanning [12], distributed interactive simulation [13], and multimedia stream dissemination [5]. These successes – coupled with the rapidly expanding pressures to control physical processes with software [14] – are now motivating the use of QoS-enabled middleware to develop much larger mission-critical DRE systems, such as commercial air traffic control, electrical power grids, and network-centric defense systems.

Until recently, however, developing large-scale mission-critical DRE systems with QoS-enabled middleware was beyond the realm of serious contemplation. One major reason for this stemmed from the accidental complexities incurred in configuring and customizing QoS-enabled middleware for large-scale DRE systems. This problem is further exacerbated by the emergence of more sophisticated and flexible QoS-enabled component middleware [1]. For this middleware to be used effectively for large-scale DRE systems, R&D activities must alleviate the accidental complexities incurred when configuring and customizing QoS-enabled middleware, while also validating these configurations and customizations.

A promising way to address the problems outlined above is for middleware developers to document reusable patterns of middleware configuration and customization [15] that the DRE system developers can use in their operational context and environment. We use the term *middleware configuration and customization pattern* to mean a set of semantically compatible middleware configuration and customization options that can be used to tailor the functionality and fine-tune QoS properties of middleware. These sets are considered as patterns since they can often be used to support similar QoS needs of middleware-based DRE applications in a range of domains.

Documenting middleware configuration and customization patterns via generative technologies. This paper describes our R&D on model-driven generative programming techniques that middleware developers can use to codify middleware configuration and customization (C&C) patterns. Our R&D approach comprises the application of the following two steps:

- **Model-Driven Middleware (MDM) technologies** [16], which minimize the effort associated with developing and validating middleware by capturing key properties of middleware within higher-level models and synthesizing/generating middleware configurations and deployments from these models [17]. We have developed an MDM tool suite called CoSMIC [18], which consists of an integrated collection of modeling, analysis, and synthesis tools that address key lifecycle challenges of DRE middleware and applications. This paper focuses on MDM tools we developed for (1) configuring and customizing QoS-enabled middleware (described in Section 2.1) and (2) generating testsuites for empirically benchmarking the configured middleware to evaluate its QoS (described in Section 2.2).
- **Distributed Continuous Quality Assurance (DCQA) technologies**, which help improve software quality and performance iteratively, opportunistically, and efficiently around-the-clock in multiple, geographically distributed locations. We have developed a DCQA framework called Skoll [19], which is an environment for executing QA tasks continuously across a grid of computers distributed throughout the world. This paper describes our approach to validating C&C patterns using Skoll (Section 2.3) and experimentally evaluating a C&C pattern (Section 3).

Our prior published work on Skoll and CoSMIC presented (1) the syntax and semantics of the Options Configuration Modeling Language (OCML) [20], which is an MDM tool that simplifies the specification and validation of complex DRE middleware and application configurations, and (2) the Benchmark Generation Modeling Language (BGML) [21], which is an MDM tool that synthesizes benchmarking testsuites to empirically analyze the QoS performance of OCML-configured DRE systems and feedback results into OCML models to help optimize domain-specific configurations. Our prior work also showed how these MDM tools were integrated with the Skoll framework to quantify the variability in QoS measures on a range of hardware, OS, and compiler platforms [21].

This paper enhances our earlier work by focusing on how middleware developers can use (1) the generative capabilities of the OCML and BGML MDM tools to identify and codify reusable patterns of middleware configuration and customization and (2) validate these patterns by empirically evaluating the generative capabilities using the Skoll framework in an experimentation testbed for the ACE [22], TAO [9], and CIAO [1] QoS-enabled middleware. Our results indicate that an MDM approach to middleware configuration and benchmarking results in comparable accuracy and correctness to a handcrafted approach. Moreover, by applying MDM tools to avionics mission computing, $\sim$85% of the code required to configure and empirically validate middleware configurations is generated automatically, which helps eliminate accidental complexities incurred when handcrafting these configurations.

Paper organization. This remainder of this paper is organized as follows: Section 2 describes key challenges for QoS-enabled middleware and illustrates how our OCML and BGML MDM tools can help resolve these challenges; Section 3 qualitatively and quantitatively evaluates our MDM tools; Section 4 compares our research with related work; and Section 5 presents concluding remarks.

2 Addressing Key QoS-enabled Middleware Configuration and Customization Challenges

This section describes key middleware configuration and customization challenges faced by developers of large-scale DRE systems. These challenges have been the key motivation to have middleware developers document reusable middleware

C&C patterns. To alleviate the middleware C&C challenges and provide middleware developers with the mechanisms to discover, validate and document middleware C&C patterns, this section describes how we are using our model-driven generative tools in combination with a distributed continuous quality assurance (DCQA) framework using the two step process described in Section 1.

The model driven generative tools called *Options Configuration Modeling Language (OCML)* and *Benchmark Generation Modeling Language (BGML)* we used to address the middleware C&C challenges are part of an open-source[1] Model-Driven Middleware (MDM) [18] toolchain called CoSMIC. MDM is an emerging paradigm that combines the strengths of model driven generative techniques and QoS-enabled middleware to support large-scale DRE systems.

The modeling languages and generative technologies embodied by our OCML and BGML MDM tools have been developed using the Generic Modeling Environment (GME) [23], which is meta programmable environment for creating domain-specific modeling languages and generative tools. GME is programmed via *metamodels* and *model interpreters*. The metamodels define the syntax, semantics and constraints of the modeling languages (also called paradigms). These include the modeling elements supported by the language, their properties, and their relationships. Model interpreters are software artifacts that can be associated with a modeling paradigm inside the GME environment. Interpreters can be used to traverse the paradigm's modeling elements, performing analysis and generating code.

This section outlines how OCML (Section 2.1) and BGML (Section 2.2) can be integrated with the Skoll framework (Section 2.3) to resolve key middleware configuration and validation challenges. Section 3 then presents a case-study that demonstrates the reusability of these tools in the context of several representative large-scale DRE systems, which helps in codifying and validating a C&C pattern.

2.1 Challenge 1: Configuring and Customizing QoS-enabled Middleware for DRE Systems

Context. QoS-enabled middleware often provides a range of configuration options that can be used to customize and tune the QoS properties of the middleware. For example, ACE and TAO are widely-used QoS-enabled middleware that provide ∼500 configuration options to tune middleware behavior[2]

Problems. A vexing challenge when using QoS-enabled middleware arises from the explosion of the *configuration space*. For example, DRE system developers can now configure and fine-tune the performance of middleware at multiple levels, including the Object Request Broker (ORB), reusable services, and lower-level messaging and transport mechanisms. Examples of middleware configuration options include selecting internal request buffering strategies, request demultiplexing and dispatching strategies, data marshaling strategies, concurrency strategies, end-to-end network connection management strategies, and end-to-end priority propagation strategies, among many others.

This large number of configuration options greatly complicates the activities of DRE system developers, who must carefully choose the configuration options for their applications. This problem is exacerbated by the fact that not all combinations of options form a semantically compatible set. DRE system developers today must therefore have significant expertise and understanding of the middleware to determine the appropriate set of semantically-compatible configurations to use. Today's *ad hoc* techniques that choose these configuration parameters manually are tedious and error-prone, and have no systematic basis for analytically validating the correctness of the resulting system configurations.

Solution → Resolving middleware configuration challenges via the Options Configuration Modeling Language. Addressing the challenges described above requires principled, analytical and empirical methods to configure, customize, and validate QoS-enabled middleware for DRE systems. These methods must enforce the physical constraints of DRE systems, as well as satisfy stringent end-to-end system QoS requirements. Model-based techniques are a promising way to resolve these challenges since they raise the level of abstraction used to develop and validated DRE systems and are amenable to automated model checking [24].

Our solution to these challenges is based on the generative properties of the *Options Configuration Modeling Language (OCML)* [20]. OCML is a GME-based modeling paradigm for configuring QoS-enabled middleware and alleviating various accidental complexities. OCML is designed for use by both (1) *middleware developers*, who use OCML to define the constraints and dependencies of the middleware options, and (2) *application developers*, who use OCML and its constraints to specify semantically compatible middleware configuration options. OCML provides the following benefits to its users:

- defines *standard configurations* that application developers can use to choose the configuration sets that suit their components.
- enforces *dependency rules* that prevent application developers from choosing invalid combinations of configuration sets that could result in incorrect/undefined behavior.
- generates *options documentation* so it is automatically in sync with the features provided by the middleware.
- generates *syntactically correct and semantically consistent configuration files* that can be used to customize the middleware.

[1]These tools are available at `www.dre.vanderbilt.edu/cosmic/`.

[2]A list of the options for TAO is available from `www.cs.wustl.edu/~schmidt/ACE_wrappers/TAO/docs/Options.html`.

Below we outline the OCML tool workflow, describe its novel features, and illustrate how it can be applied to resolve accidental complexities that arise when validating configuration options in middleware, such as ACE+TAO.

• **OCML use cases.** Figure 1 shows how OCML is used both by DRE system developers and middleware developers. Each step in this figure is explained below:

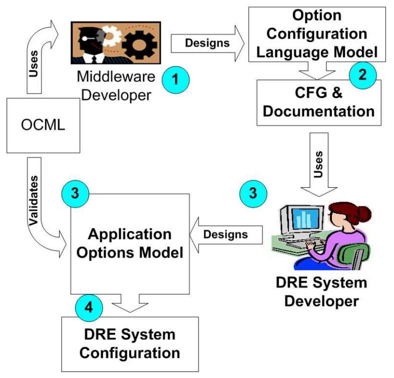

Figure 1: **OCML Workflow**

- **Step 1: Model the options using OCML.** A middleware developer uses OCML to model the options, categorize them in a hierarchical order, and define the rules governing their dependencies. The constraints on option values and rules on valid combinations of options are defined in this step by middleware developers as visual logic expressions. OCML also allows options to be categorized hierarchically into *option categories*. Each option category and its associated options contain a description attribute that can include hypertext information, which will be used in the generation of HTML documentation. At this stage, thus, OCML is used as a modeling language to create a model that is customized for a particular middleware platform.
- **Step 2: Generate the documentation and the Configuration File Generator.** When the middleware OCML model is interpreted by the OCML interpreter it produces HTML documentation and the configuration file generator (CFG) application source code. During the model interpretation process the constraints that were modeled in Step 1 are converted into C++ code, *i.e.*, each logical expression is encoded as a C++ function that is compiled into native executable code and linked with the generated CFG application. The middleware configuration layer (steps 1 and 2 in Figure 1) is hidden from DRE system developers, who only use the generated files.
- **Step 3: Use the CFG to specify desired options.** Developers of DRE application use the generated CFG application to initialize the configuration for a specific application (the generated HTML documentation can be used as a reference). The resulting designs are then checked against the constraints defined by rules encoded in OCML models by middleware developers (in Step 1), which minimizes the risk of choosing the wrong set of options.
- **Step 4: Configure the middleware.** The selected options are exported into files that contain configuration and customization metadata, which is used by the middleware to customize the system during the initialization process.

• **OCML generative tools.** As shown in Figure 1, OCML generates the following types of files:

- **Configuration files** that are parsed by the middleware to apply the requested strategy for various mechanisms (*e.g.*, concurrency strategies and component communication properties) and manage internal resources (*e.g.*, locking mechanisms and internal table and cache sizes).
- **Documentation files** that explain the proper content of the configuration files and which are used as a reference by application developers. The generated HTML documentation includes information about every option and cross-references for option dependencies. The HTML documentation contains the collection of these descriptions in a human readable format that can be rendered in an HTML browser. The documentation also displays the cross-references for the dependent objects, together with the textual representation of the rules.

• **Using CFG for middleware configuration and customization.** We now describe how the OCML generated artifacts can be used to model sets of middleware-specific valid configuration and customization options. We envision the middleware developers to use these features to codify these sets into C&C patterns. As explained earlier, the OCML model interpreter emits source code for the *configuration file generator* (CFG). CFG is a GUI-based application that application developers (or middleware developers who are documenting C&C patterns for their middleware) will use to select the desired set of options for their applications, subject to the restriction that the selected options conform to the constraints defined in the OCML model rules. These constraints ensure that application developers only select valid combinations of options.

When an application developer configures a group of options, the CFG application uses the constraint checking C++ functions to check whether the options's values conform to the constraints of the OCML model's rules. Only valid configurations are allowed thereby preventing mistakes resulting from combining incompatible options.

• **Applying OCML to ACE+TAO Middleware.** As discussed in the **context** discussion at the beginning of this section, ACE and TAO provide a large set of configuration options. During application initialization time, TAO parses an

XML configuration file (called `svc.conf`) containing values for various middleware options and uses these values to configure the ORB strategies to meet various QoS requirements, such as latency, memory footprint, and throughput. Figure 1 illustrates the following the step-by-step process of using OCML to configure TAO options:

- **Step 1: Model the TAO options using OCML.** The TAO options are expressed in OCML using a hierarchical model that allows the options to be categorized into different option categories. This design simplifies the navigation of the documentation and makes the CFG application more intuitive for application developers. For example, the `-ORBConnectionCacheMax` option used to configure the maximum value of the transport cache resides in the *Resource Factory* category and the `-ORBConcurrency` option used to specify the ORB's concurrency strategy resides in the *Strategy Factory* category. In this step the OCML rules are specified as constraints for combining various options using the visual logical expressions shown in the Figure 2.

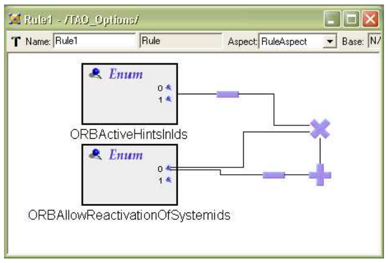

Figure 2: **Expressing OCML Option Rules Visually**

- **Step 2: Generate the TAO options documentation and the CFG.** After the OCML model of the TAO options is complete the OCML model interpreter is run to generate the documentation for the TAO options in HTML and the CFG application.
- **Step 3: Use the CFG to specify desired options.** The CFG application is used by application developers to assign values for different options required for the application they are developing. For example, the CFG allows application developers to enter an integer value for the `-ORBConnectionCacheMax` option described above. Likewise, it provides several choices for the `-ORBConcurrency` option, including *reactive* for a purely reactor-driven concurrency strategy or *thread-per-connection* to create a new thread to service each connection. Only the specified options are written to the `svc.conf` file – other options are assigned default values.
- **Step 4: Configure the middleware.** Finally, the `svc.conf` file for TAO is generated by the CFG application and used subsequently to configure the DRE middleware and application during system initialization.

2.2 Challenge 2: Evaluating the QoS of Selected Middleware Configurations

Context. QoS-enabled middleware runs on a multitude of hardware, OS, and compiler platforms. Developers often use trial-and-error methods of selecting the set of configuration options that maximizes the QoS characteristics provided by the middleware. For example, CIAO [1] is widely-used QoS-enabled middleware that implements the CORBA Component Model (CCM) and provides configuration options and policies to provision components end-to-end when assembling a DRE system.

Problems. QoS-enabled middleware is often used by applications with stringent QoS requirements, such as low latency and bounded jitter. The QoS delivered to a DRE system is influenced heavily by factors such as

- The configuration options set by users to tune the underlying hardware/software platform (*e.g.*, the concurrency architecture and number of threads used by an application significantly affects its throughput, latency, and jitter) and
- Characteristics of the underlying platform itself, (*e.g.* the jitter on a real-time OS should be much lower than on a general-purpose OS).

Managing these variable platform aspects effectively requires analysis that precisely pinpoint the consequences of mixing and matching middleware configuration options at multiple layers on various platforms.

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 empirical results for evaluating the QoS delivered to the DRE system. It is tedious and error-prone, however, to develop such benchmarking testsuites manually. In a typical analysis process, for example, creating a benchmarking experiment to measure QoS properties requires middleware developer 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 [19] revealed how tedious and error-prone this process is since it requires many manual steps to generate benchmarks.

Solution $\rightarrow$ resolving QoS evaluation challenges using the Benchmark Generation Modeling Language. To overcome the limitations with manually developing custom benchmarking suites, we have used model-driven generative techniques to develop the *Benchmark Generation Modeling Language* (BGML). BGML is a GME-based modeling paradigm for (1) modeling how distributed system components interact with each other and (2) representing metrics that can be applied to specific configuration options and platforms. BGML uses visual representation techniques to define entities and their interactions in an application domain. Middleware/application developers[3] can use BGML to graphically model interaction scenarios of interest. Given a model, BGML generates the scaffolding code needed to run experiments, which typically includes scripts that start daemon processes, launch components on various distributed system nodes, run the benchmarks, and analyze/display the results.

Below we outline the BGML tool workflow, describe its novel features, and illustrate how it can be applied to resolve accidental complexities involved in evaluating the performance and QoS of applications that run atop CIAO.

• **BGML use cases.** Figure 3 shows how the BGML tool is typically applied. As shown in this figure, the following

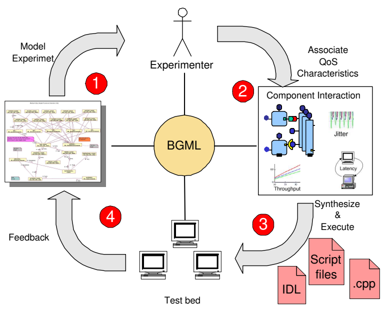

Figure 3: **BGML Workflow**

steps are required to model experiments to evaluate QoS metrics:

- **Step 1: Model the interaction scenario.** Users apply the BGML modeling paradigm to compose experiments (a detailed description on experimentation modeling using BGML appears in Section 3.2).
- **Step 2: Associate QoS metrics with model.** In the modeled experiment, users associate the QoS characteristic (*e.g.*, roundtrip latency) that is the subject of the experiment.
- **Step 3: Synthesize benchmarking code.** BGML interpreters then use the modeled experiment to generate the code required to set-up, run, and tear-down the experiment. The files generated include component implementation files (.h, .cpp), IDL files (.idl), component IDL files (.cidl), and benchmarking code (.cpp) files. The generated file is executed on a testbed (such as Emulab `www.emulab.net`) and the QoS characteristics are measured.
- **Step 4: Refine configuration models.** The results are then fed-back into the models to provide information on the consequences of mixing and matching configuration options at design time.

[3]We henceforth use the term *users* to refer to both middleware and application developers.

• **BGML generative tools.** The BGML model interpreter parses models and synthesizes the code required to benchmark modeled configurations via the following code generation engines:

- **Benchmark code engine**, which generates source code to run an experiment. The generated code includes header and implementation files to execute and profile the applications and to summarize the results. This engine relies on the *metrics aspect*, where users specify the metrics to collect during the experiment.
- **Interface definition language (IDL) engine**, which generates the IDL files forming the contract between client and server. These files are parsed by an IDL compiler that generates the "glue-code" needed by the benchmarking infrastructure. This engine relies on the *configuration aspect*, where users specify the data exchanged between the communication entities.
- **Script engine**, which generates the script files used to run the experiment in an automated manner. In particular, the scripts start and stop the server/client, pass parameters, and display the experimentation results. This engine is associated with the *interconnection aspect*, where component interactions are specified.

• **Applying BGML to CIAO.** BGML is designed to reduce the effort of conducting performance evaluation and tuning for DRE applications. Below, we describe how BGML helps alleviate key sources of complexity in a representative DRE application based on the CIAO QoS-enabled component middleware. In particular, BGML model interpreters generate the following scaffolding code that the developers traditionally had to write manually:

1. Data exchange information that forms the contract between clients and servers. The generated file adheres to the syntax of CORBA IDL and is parsed by the CIAO IDL compiler to generate the stubs and skeletons to (de)marshal data types modeled in the experiment. This information is generated from the `interface` component in the BGML models. Auto-generation of CORBA IDL eliminates need for users to understand the syntax of CORBA IDL (which is rather low-level), thereby improving productivity. Section 3.2 presents code generation metrics for the BGML tool.

2. Component description information that describe components, such as the facets/receptacles that a component exports. The format used to depict this information is CORBA

3.x IDL, which is a component-based extension of CORBA 2.x IDL. In CIAO, the generated component implementation files (.cidl) are parsed by CIAO's CCM component implementation definition language (CIDL) compiler to generate glue-code that simplifies component implementations. By generating the .cidl files for CIAO, BGML, obviates the need for users to understand the convoluted syntax of CORBA 3.x IDL since BGML generates syntactically and semantically correct files from higher-level models.

3. Component implementation information that map the CCM entities to their corresponding programming features. This step implicitly maps the component entities (*e.g.*, event-sources/-sinks) to their corresponding programming language (*e.g.*, C++, Java, or C) constructs. This mapping entails generation of the corresponding header and implementation files called *executors*. The implementation (.cpp) files generated is a template that provides a no-op implementation for all operations specified in the component description information. This information is generated from components modeled in BGML. By generating component implementation information, users are shielded from the complex mapping of CORBA IDL to programming language(s), which is delegated to the BGML model interpreters.

4. Benchmarking information that contains the code to run the experiment, measure the QoS metric, and display results. This step embellishes the component implementations (in CIAO these files correspond to *_exec.cpp) with the benchmarking code that uses the specified number of iterations to warm-up the system, high-resolution timers that timestamp pre- and post-invocation of the operations being tested, and generate the required metric. This step automates the task of writing repetitive source code to perform the benchmark. Moreover, the displayed statistics can be generated in a format parsable by tools in the CoSMIC toolsuite and fed-back to user-developed models [21].

Figure 4 shows how BGML's MDM approach enhances the component-based approach by generating all the scaffolding code required to synthesize benchmarks from higher level models. In this scenario, all the required files are auto-generated, thereby relieving the user from error-prone low-level code details, including programming languages, IDL syntax, XML syntax, and details of the underlying middleware, which are often proprietary in QoS-enabled middleware.

2.3 Challenge 3: Validating Middleware Configuration and Customization Pattern

Context. QoS-enabled middleware run on varied hardware, OS and compiler platforms. To support the needs of DRE systems, portions of the middleware may need to be customized for each platform on which it runs. For example, middleware features using asynchronous I/O are natively supported in Win32 but may not be available (or may be poorly implemented) on other platforms. In other cases, features such as thread pools might be supported on a wide range of platforms. Certain C&C patterns may be platform-specific, though still applicable to a variety of DRE domains, whereas others may be applicable across several platforms. Middleware developers will therefore need to validate the C&C patterns across different platforms.

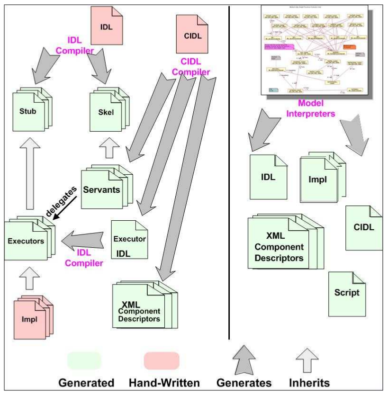

Figure 4: **Code Generation Comparison in the Handcrafted vs. MDM Approaches**

Problems. The OCML tool described in Section 2.1 reduces the accidental complexity in configuring QoS-enabled middleware. However, middleware and application developers alike must still rediscover the appropriate set of configuration options for each platform using trial-and-error. A common approach is to test configuration options across a range of relevant platforms. The BGML tool described in Section 2.2 resolves accidental complexities and costs associated with evaluating QoS characteristics across different configurations. However, testing every combination of configuration options on each platform leads to an explosion in the configuration space. For example, the ACE, TAO, and CIAO middleware provides well over $\sim$500 configuration options. If each option was binary (which is a very conservative estimate), the size of configuration space is $\sim 2^{500}$. Discovering, validating and documenting C&C patterns is thus a daunting task.

Solution $\rightarrow$ Resolving validation challenges by documenting C&C patterns using Distributed Continuous Quality Assurance (DCQA) Techniques. To address the C&C pattern validation problems described above requires capturing and documenting recurring sets of middleware configuration and customization options that maximize QoS across a range of hardware, OS, and compiler platforms. In DRE systems based on QoS-enabled middleware, these recurring configurations are tied to the operational context of objects and components. To address the challenges caused by the explosion of the software configuration space and the limitations of in-house QA processes, we have developed the **Skoll** [19] DCQA environment to prototype and evaluate tools necessary to perform "around-the-world, around-the-clock" DCQA processes to improve quality of software. The Skoll infrastructure performs its distributed QA tasks, such as testing, capturing usage patterns, and measuring system performance, on a grid of computing nodes. Skoll decomposes QA tasks into subtasks that perform part of a larger task. In the Skoll grid, computing nodes are machines provided by the core development group and volunteered by end-users and core developers. These nodes request work from a server when they wish to make themselves available.

Figure 5, depicts how the OCML and the BGML tools are used in concert with the Skoll DCQA environment to help identify recurring C&C patterns for different combinations of configuration options. Each of these steps is discussed below:

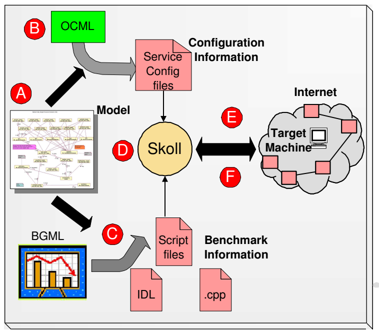

Figure 5: **Alleviating C&C Patterns Validation Challenges via DCQA Processes**

A. At the highest level, users employ the modeling environment to depict the interaction between their components and objects. It is in this model that the configuration information for the individual components and the QoS criteria to be measured are captured.

B & C. The OCML and BGML model interpreters traverse the models to synthesize the configuration and experimentation code. The OCML paradigm interpreter parses the modeled configuration options and generates the required configuration files to configure the underlying component implementation. The BGML paradigm interpreter then generates the required benchmarking code, *i.e.*, scaffolding code to set-up, run and tear down the experiment.

D. The configured experimentation code can next be fed to the Skoll DCQA environment. Remote clients register with Skoll and request periodic software to run using spare CPU cycles on local machines.

E & F. The Skoll then iteratively and continuously runs the benchmark on various clients to observe behavior across varied range of hardware, OS, and compiler platforms. A client executes the experiment and returns the result to a Skoll server, which updates its internal database. This information then can be used by the users to select configurations that maximize QoS across various platforms.

Section 3 discusses a case study detailing how the OCML and BGML tools can be integrated to identify C&C patterns.

3 Experimental Evaluation of Model-Driven Middleware Configuration and Customization

Determining whether a given set of middleware C&C options can be codified as a reusable C&C pattern requires empirical evaluation of the DRE system using these middleware C&C sets in different DRE domains. This section describes our approach to such an empirical evaluation via a case study that uses our Model-Driven Middleware (MDM) generative tools and benchmarking environment to validate QoS delivered by a selected C&C set. Our approach to validating whether a C&C options set is a C&C pattern consists of the following steps:

1. Determine the QoS expected from the middleware.
2. Select a set of middleware C&C options using the OCML tool (Section 2.1) that are expected to provide these QoS guarantees. It is assumed that the middleware developers will have the appropriate insights to select the right options.
3. Use the BGML tool (Section 2.2) to generate a testsuite for evaluating QoS delivered by the middleware.
4. Assign a value for each of the selected options in the C&C set. Once again OCML is used to select the values for each option in the set. The challenges arising from the explosion in the C&C space can be alleviated using pruning techniques discussed in our related research [19, 21].
5. For each such constrained C&C set in step 4, run the generated benchmarking tests to evaluate the QoS.
6. Repeat steps 4-5 for DRE systems in different domains and possibly different environments to determine if a particular constrained C&C option set delivers similar QoS properties. If so, the C&C set is thus a good candidate for consideration as a C&C pattern.

The remainder of this section illustrates the process outlined above via a case study. Section 3.1 describes the evaluation testbed and our example DRE system. Section 3.2 then illustrates results of running the benchmarks on a constrained C&C option set that led us to codify it as a C&C pattern.

3.1 Evaluation Platform

The QoS requirements of different DRE systems vary. For example, DRE systems involving audio streams require predictable end-to-end latencies and jitter, whereas video streams often require significant bandwidth. QoS-enabled middleware should therefore be configured appropriately to deliver the desired QoS to the target DRE systems. To demonstrate the configuration of a DRE application, we use a navigation display simulation based on the Boeing Bold Stroke avionics mission computing architecture [11] that receives the global positions from a GPS device and displays them at a GUI display in a timely manner. The desired data request and the display frequencies are fixed at 40 Hz. The BoldStroke architecture uses a *push event/pull data* publisher/subscriber communication paradigm.

The component interaction for the navigation display example is depicted in Figure 6. The scenario shown in Figure 6 begins with the GPS being invoked by the TAO Real-time Event Service [25] (shown as a `Timer` component). On receiving a pulse event from the `Timer`, the `GPS` generates its data and issues a data available event. The Real-time Event Service then forwards the event on to the `Airframe` component, which retrieves the data from the `GPS` component, updates its state and issues a data available event. The Event Service forwards the event to the `Nav_Display` component, which in turn retrieves the data from the `GPS`, updates its state and displays it.

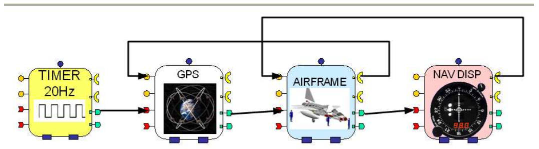

Figure 6: **Navigation Display Collaboration Example**

Each of the components in the scenario of Figure 6 has the following QoS requirements:

- The `GPS` component serves as the source for multiple components requiring position updates at a regular interval. This component's concurrency mechanism should therefore be tuned to serve multiple requests simultaneously in parallel, thereby reducing the overhead of processing these requests.
- The `Airframe` component serves as the recipient of `GPS` component's position updates and feeds these updates to the `Nav_Display` component. The role played by this component therefore does not require any concurrency since `Airframe` pulls data from the `GPS` component and pushes this data onto `Nav_Display`, which can be serialized.
- The `Nav_Display` component receives updates events from the `Airframe` component and refreshes its display. The `Nav_Display` component also receives other updates, such as changes in altitude, flying speed, and pressure[4]. The concurrency mechanism for this component therefore needs to handle multiple concurrent refreshes from a predefined set of components.

To discover and document the middleware C&C pattern applicable in this scenario, the middleware developer may decide to vary the concurrency configuration mechanism provided by the middleware so that the desired QoS, *e.g.*, maximizing throughput or minimizing latency and jitter, is achieved. Thus, each application QoS for the components can be mapped to several well-defined concurrency mechanisms provided by QoS-enabled component middleware, such as in our Component Integrated ACE ORB (CIAO) [1], including:

- The **thread pool** concurrency mechanism, where a pool of threads are used to service requests in parallel. This approach is most suitable for the GPS component.
- The **single-threaded** concurrency mechanism, where a single-thread for servicing requests. This approach is best used for components that have no concurrent jobs to perform, which makes it most suitable for the Airframe component.
- The **thread-per-connection** concurrency mechanism, where the server uses a separate thread to service requests from each client. This approach is most suitable for the Navigation component since it needs to obtain refreshes from a predefined set of components.

The desired data request frequency for the `Airframe` component is 40 Hz, which is the same value with the desired update frequency of the `Nav_Display` component. We use the modeling paradigms OCML and BGML to define various configurations, model experiments and generate the benchmarking applications, and compare the results of the benchmarks and decided the configuration that suits the performance requirements. The step-by-step process is explained next.

3.2 Study Execution

We now present the results of experiments that quantitatively evaluate OCML and BGML in the context of DRE system scenarios that help the middleware developers to codify the configuration and customization patterns of reuse. In particular we present the following analyses:

- **Code generation analysis**, which illustrates the number of lines of code auto-generated by the Model-interpreters that otherwise would have to be handwritten,
- **Configuration pattern analysis**, which explores the configuration space for the `Navigation` display component and empirically validates the configuration that minimizes the end-to-end latency for the scenario and

[4]These components are not explicitly defined in the scenario.

- **Code execution analysis**, which compares our generative approach with that of a handcrafted approach for execution efficiency and accuracy.

For all these experiments, we used ACE v5.4, TAO v1.4, and CIAO v0.4 as the middleware platforms.

Code generation analysis. Table 1 summarizes the code generation metrics for the individual tools.

Files	Number	Source Lines of Code (SLOC)
IDL (*.idl)	5	125
Executor IDL (*E.idl)	4	40
Comp. IDL (*.cidl)	4	56
Source (.cpp)	4	525
Header (.h)	4	230
Config (svc.conf)	4	28
Benchmark (.cpp)	2	90

Table 1: **Generated Code Summary for MDM Approach**

Table 1 shows how our MDM approach auto-generates the required scaffolding code required to start up and run the experiment, capture the QoS metric, and tear down the set-up when the experiment is complete. For the testbed scenario the SLOC generated was ∼1000, which shows that to compose a simple five component experiment incurs a non-trivial overhead in terms of handcrafting boiler-plate code to run an experiment. The use of our MDM tools resolves this complexity by generating both syntactically and semantically correct code, thereby increasing user productivity. As users compose various experiments for testing and performance evaluation, the gain in terms of generated code become even more significant. Although a highly efficient handcrafted code could reduce the SLOC needed to run the experiment, it would require a laborious process for every interaction scenario.

Configuration pattern analysis. In this experiment we empirically show how our MDM tools work together to empirically validate C&C patterns. Table 2 describes the testbed used for the generation of results along with the deployment information for the `Trigger`, `BMDevice`, `BMClosed` and `BMDisplay` components described in Section 3.1.

	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
Deployment	`Trigger` `BMDisplay`	`BMDevice`	`BMClosed`

Table 2: **Testbed and Deployment Summary**

This configuration simulates a deployment scenario where these components will be deployment on different nodes or embedded within processor boards. For this configuration, we choose the `Navigation Display` component and identify the configuration space based on its operational context defined in the aforementioned section. This selection enables the ensuing discussion to be intuitive, however, our discussion can be generalized to any of the components in the testbed scenario. In the testbed application, the `Display` component acts a "thick client" retrieving GPS updates from the `Airframe` component. For this role played by the component, Table 3 summarizes the configuration space provided by the CIAO middleware to tune the QoS of this component.

Notation	Option Name	Option Settings
o1	ORBProfileLock	{Thread, Null}
o2	ORBClientConnectionHandler	{RW, ST}
o3	ORBTransportMuxStrategy	{EXCLUSIVE, MUXED}
o4	ORBConnectStrategy	{reactive, LF}

Table 3: **Configuration Space for CIAO Client Component**

As shown in Table 3, we use the notation *o11*, *o12*, etc. to identify the individual options within each category. For example, the `Thread` value for the `ORBProfile` lock option is denoted as *o11* and null value as *o12*. For a comprehensive discussion of the semantics of the configuration options appears in `www.cs.wustl.edu/~schmidt/ACE_wrappers/TAO/docs`. For this configuration space, our goal is to identify the configuration, *i.e.*, options category along with its value, that maximizes the end-to-end latency for the scenario. The following steps were used to identify the candidate configuration:

1. Model each permutation of this configuration space using the OCML tool to generate the configuration files for the `Navigation` component.
2. Use default configuration for all other components to isolate QoS improvements accrued from refining configuration for the `Navigation` component.
3. Use the scaffolding code generated via BGML along with the configuration file to run the experiment. Note, modeling the same scenario enables us to use the code generated by BGML for all configurations.
4. Capture the latency information for each case and use benchmark data to analyze results and identify candidate solution.

Table 4 depicts the variation of latency across the entire configuration space.

As shown in Table 4, the topmost configuration minimizes the average latency for the scenario and is our candidate solution that corresponds to setting `ORBProfile=null`, `ORBClientConnectionHandler=RW`, `ORBTransportMuxStrategy=EXCLUSIVE`, and `ORBConnectStrategy=Reactive`. The table also

Option Configuration	Latency QoS (msecs)
(o12 o22 o31 o42)	**35.58**
(o12 o22 o31 o41)	77.13
(o11 o22 o32 o41)	173.03
(o11 o21 o32 o41)	214.6
(o12 o21 o31 o41)	280.18
(o12 o21 o31 o42)	559.0
(o11 o21 o31 o42)	585.003
(o11 o21 o32 o42)	597.5
(o12 o21 o32 o42)	684.7
(o12 o21 o32 o41)	771.047
(o12 o22 o32 o42)	833.6
(o12 o22 o32 o41)	1077.725
(o11 o22 o32 o42)	1217.3
(o12 o21 o31 o41)	1258.8
(o11 o22 o31 o42)	1506.4
(o11 o22 o31 o41)	1589.4

Table 4: **Variation of QoS Across the Configuration Space**

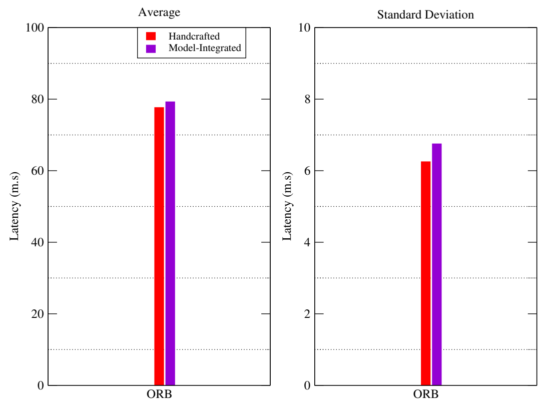

Figure 7: **Execution Comparison**

presents an hierarchical classification of the latencies from best to worst. Each configuration therefore represents a one-step, *i.e.*, successive refinement over the configuration below it. A similar technique can be used for the other components to identify the configuration space corresponding to their operation context and using successive refinement techniques to identify target configurations that maximize the QoS.

The empirical identification and validation of the solution permits its reuse across similar operations contexts, in our case, a component playing a role of a client. This configuration then represents a C&C pattern that is reusable across domains. As described above, identifying C&C pattern requires exhaustive profiling and testing across varied hardware/OS/compiler platforms. DCQA frameworks like Skoll minimize the cost of capturing and validating these patterns across diverse platforms. Moreover, Skoll enables automation of these experiments. For example, our experiment can be mapped to a Skoll task that is executed across a range of user machines. By combing DCQA frameworks with our model-based tools, we can minimize the effort required to identify and document new patterns, while simultaneously validating existing ones.

Code execution analysis. Figure 7 compares the execution correctness of our MDM approach vs. a handcrafted approach. For this experiment, the configuration (*o12 o22 o31 o41*) (from Table 4) was chosen for the `Navigation` component. As shown in this figure, the latency measures from both the approaches are comparable. In particular, a deeper analysis from the individual samples also showed that the variation within the two data sets is also comparable. Though we present only a particular case, a generalization can be made for all the cases as (1) BGML tool only generates the implementation templates while the logic is still written by the end-user and (2) OCML tool generates the exact XML configuration files as the handcrafted approach, as in CIAO configurations are (name, value) pairs. The results show that an MDM approach does not incur any additional overhead than a handcrafted approach.

4 Related Work

This section compares our work on model-driven techniques in OCML and BGML with other related research efforts including Distributed Continuous Quality Assurance (DCQA) environments, middleware configuration techniques, and generative techniques for empirical QoS evaluation.

DCQA environments. The VTK project uses a DCQA environment called Dart [26], which supports a continuous build and test process that starts whenever repository check-ins occur. In earlier work [19, 21], we developed a prototype of a DCQA environment called Skoll that overcomes the limitations of earlier in-the-field quality assurance approaches. Skoll provides an *Intelligent Steering Agent* that controls the QA process across configuration spaces by decomposing anomaly detection, QoS evaluation, and integration testing QA processes into multiple tasks and then distributing/executing these tasks continuously on a grid of computing resources contributed by end-users and distributed developers around the world.

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 [22], which can be used to statically and dynamically configure components into middleware and applications. Likewise, Java provides 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.

The Microsoft .Net platform provides a similar approach to the Java XML property files named *.Net configuration files*. The `System.Configuration` API can be used to read the configuration. Using this API, .Net provides access to three different information: (1) *machine configuration files*, which control machine-wide assembly binding and remoting channels, (2) *application configuration files*, which control application-specific configurations, such as assembly binding policies and remoting objects, and (3) *security configuration files*, which contain security information for applications.

Editing text configuration files formatted with XML is common for .Net, Java, and various CORBA and CCM implementations. OCML enhances the configuration of various middleware platforms by providing an MDM methodology. From these OCML models, documentation about the configuration of the middleware and a GUI interface for middleware configuration can be generated automatically.

Empirical QoS evaluation. There have been several initiatives that use generative techniques similar to BGML to generate testcases and benchmarks for performance evaluation. ForeSight [27] uses an empirical benchmarking engine to capture QoS information for component middleware. The results are used to build mathematical models to predict performance. ForeSight uses a three-pronged approach of (1) creating a performance profile of how components in a middleware affect performance, (2) constructing a reasoning framework to understand architectural trade-offs, *i.e.*, know how different QoS attributes interact with one another, and (3) feeding this configuration information into generic performance models to predict the configuration settings required to maximize performance.

SoftArch/MTE [28] provides a framework for system architects to define higher level abstractions of their system by specifying characteristics such as middleware, database technology, and client requests. SoftArch/MTE then generates an implementation of the system along with the performance tests that measure these system characteristics. These results are then displayed back, *i.e.*, annotated in the high level diagrams, using tools such as Microsoft Excel, thereby allowing architects to refine the design for system deployment.

BGML is related to the ForeSight and SoftArch tools described above. However, both these tools lack DCQA environments to accurately capture QoS variations on a range of varied hardware, OS and compiler platforms. Rather than using a generic mathematical models to predict performance, the BGML tools use feedback-driven approach [21], wherein the DCQA environment is used to empirically evaluate the QoS characteristics offline. This information can then be used to provide modelers with accurate system information. Moreover, platform-specific optimization techniques can be applied to enhance system performance.

5 Concluding Remarks

QoS-enabled middleware addresses key aspects of the DRE application development and deployment and also provides policies and mechanisms for specifying and enforcing large-scale DRE application QoS requirements. Historically, it has been hard to customize and tune QoS-enabled middleware due to accidental complexities associated with validating and optimizing middleware configurations. This paper describes a pair of generative Model-Driven Middleware (MDM) tools – the Options Configuration Modeling Language (OCML) and the Benchmarking Generation Modeling Language (BGML) – that help alleviate the complexity of (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. These MDM tools help decrease accidental complexity by automatically generating various parts of the middleware, application, and benchmarking code, in addition to configuration data.

The concept of configuration and customization (C&C) patterns represent common configuration schemas shared across various application domains. By using the OCML and BGML tools, C&C patterns can be defined, validated, and stored for reuse in similar scenarios. OCML and BGML are a part of the CoSMIC toolsuite. The latest information and source code can be obtained from CoSMIC website at `www.dre.vanderbilt.edu/cosmic`. The CoSMIC tool suite is developed in association with the CIAO QoS-enabled component middleware, which is available at `www.dre.vanderbilt.edu/CIAO`.

We have used the MDM approach presented in this paper to design and evaluate C&C patterns for a specific set of middleware, *i.e.*, ACE, TAO, and CIAO. In future work we plan to generalize these tools to support a broader range of middleware platforms and C&C patterns using techniques from an Integrated Concern Modeling and Manipulation Environment (ICMME) [29]. The ICMME brings a higher level of abstraction to DRE application design by separating concerns (such as remoting, communication and processor resource usage, and persistence) from component middleware and application design. ICMME is designed to integrate MDM and aspect-oriented software development generative techniques.

References

[1] 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 *Middleware for Communications* (Q. Mahmoud, ed.), New York: Wiley and Sons, 2003.

[2] T. Ritter, M. Born, T. Unterschütz, and T. Weis, "A QoS Metamodel and its Realization in a CORBA Component Infrastructure," in *Proceedings of the* 36th *Hawaii International Conference on System Sciences, Software Technology Track, Distributed Object and Component-based Software Systems Minitrack, HICSS 2003*, (Honolulu, HW), HICSS, Jan. 2003.

[3] D. Conan, E. Putrycz, N. Farcet, and M. DeMiguel, "Integration of Non-Functional Properties in Containers," *Proceedings of the Sixth International Workshop on Component-Oriented Programming (WCOP)*, 2001.

[4] M. A. de Miguel, "QoS-Aware Component Frameworks," in *The* 10th *International Workshop on Quality of Service (IWQoS 2002)*, (Miami Beach, Florida), May 2002.

[5] J. Loyall, J. Gossett, C. Gill, R. Schantz, J. Zinky, P. Pal, R. Shapiro, C. Rodrigues, M. Atighetchi, and D. Karr, "Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications," in *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21)*, pp. 625–634, IEEE, Apr. 2001.

[6] R. Thau, "Design Considerations for the Apache Server API," in *Proceedings of the Fifth International World Wide Web Conference on Computer Networks and ISDN Systems*, pp. 1113–1122, Elsevier Science Publishers B. V., 1996.

[7] D. C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible ORB Middleware," *IEEE Communications Magazine*, vol. 37, Apr. 1999.

[8] B. Laskey and D. C. Kreines, *Oracle Database Administration: The Essential Reference*. Sebastopol, CA: O'Reilly, 1999.

[9] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.

[10] D. C. Sharp, "Reducing Avionics Software Cost Through Component Based Product Line Development," in *Proceedings of the 10th Annual Software Technology Conference*, Apr. 1998.

[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] C. D. Gill, J. M. Gossett, D. Corman, J. P. Loyall, R. E. Schantz, M. Atighetchi, and D. C. Schmidt, "Integrated Adaptive QoS Management in Middleware: An Empirical Case Study," in *Proceedings of the 10th Real-time Technology and Application Symposium (RTAS '04), Embedded Applications Track*, (Toronto, CA), IEEE, May 2004.

[13] R. Noseworthy, "IKE 2 – Implementing the Stateful Distributed Object Paradigm ," in *5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*, (Washington, DC), IEEE, Apr. 2002.

[14] C. on Networked Systems of Embedded Computers, *Embedded, Everywhere: A Research Agenda for Networked Systems of Embedded Computers*. Washington, D.C.: National Academies Press, 2001.

[15] S. Berczuk and B. Appleton, *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Addison-Wesley, 2003.

[16] A. Gokhale, D. C. Schmidt, B. Natarajan, J. Gray, and N. Wang, "Model Driven Middleware," in *Middleware for Communications* (Q. Mahmoud, ed.), New York: Wiley and Sons, 2003.

[17] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-Integrated Development of Embedded Software," *Proceedings of the IEEE*, vol. 91, pp. 145–164, Jan. 2003.

[18] A. Gokhale, K. Balasubramanian, J. Balasubramanian, A. 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.

[19] 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), IEEE/ACM, May 2004.

[20] E. Turkaye, A. Gokhale, and B. Natarajan, "Addressing the Middleware Configuration Challenges using Model-based Techniques," in *Proceedings of the 42nd Annual Southeast Conference*, (Huntsville, AL), ACM, Apr. 2004.

[21] 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), ACM/IEEE, July 2004.

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

[23] A. Ledeczi, A. Bakay, M. Maroti, P. Volgysei, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing Domain-Specific Design Environments," *IEEE Computer*, Nov. 2001.

[24] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad, "Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems," in *Proceedings of the 25th International Conference on Software Engineering*, (Portland, OR), May 2003.

[25] 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*, (Atlanta, GA), pp. 184–199, ACM, Oct. 1997.

[26] "public.kitware.com." http://public.kitware.com.

[27] Y. Liu, I. Gorton, A. Liu, N. Jiang, and S. Chen, "Designing a Test Suite for Empirically-based Middleware Performance Prediction," in *40 th International Conference on Tenchnology of Object-Oriented Languages and Systems, Sydney Australia*, Australian Computer Society, Aug. 2002.

[28] J. Grundy, Y. Cai, and A. Liu, "Generation of Distributed System Test-beds from High-level Software Architecture Description," in *16 th International Conference on Automated Software Engineering, Linz Austria*, ACM SIGSOFT, Sept. 2001.

[29] A. Nechypurenko, D. C. Schmidt, T. Lu, G. Deng, A. Gokhale, and E. Turkay, "Concern-based Composition and Reuse of Distributed Systems," in *The 8th International Conference on Software Reuse*, (Madrid, Spain), ACM/IEEE, July 2004.