# Model-driven Software Tools for Configuring and Customizing Middleware for Distributed Real-time and Embedded Systems

Arvind S. Krishna[‡], Emre Turkay[‡], Cemal Yilmaz[†],
Douglas C. Schmidt[‡], Aniruddha Gokhale[‡], Atif Memon[†], Adam Porter[†]
[†]*Dept. of Computer Science, University of Maryland, College Park, MD 20742*
[‡]*Dept. of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, TN 37203*

## ABSTRACT

Middleware is increasingly being 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 quality of service (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 variability, 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 (C&C) that resolve key variability challenges. This paper describes how domain-specific languages and model-driven generative programming techniques can be used by middleware developers to codify reusable patterns of middleware configuration and customization.

## 1. EMERGING TRENDS AND CHALLENGES

Large-scale DRE systems increasingly execute on a range of middleware platforms, user contexts, and different application domains. To ensure that the applications and middleware meet their quality of service (QoS) requirements in the face of multi-dimensional *variability* (such as different workloads, operating systems, middleware, and application feature sets, compiler flags, and/or run-time optimization settings), the software infrastructure for DRE systems must be highly tunable, *i.e.*, provide 10's-100's of configuration options to customize itself and applications for specific environments and requirements. Quality assurance (QA) of these customizations have traditionally been performed in-house. Unfortunately, using in-house QA processes for these DRE systems has the following limitations:

*Limited testing capability..* In-house QA processes often fail to assure software quality since they do not manage software variability effectively. For example, in-house QA processes can rarely capture, predict, and recreate the run-time environment and usage patterns that will be encountered in the field on all supported target platforms across all desired system configuration options.

*Lack of proper QoS validation due to variability in operational contexts..* Assessing QoS on a few possible configurations/platforms and then extrapolating these to the entire configuration space may be invalid since configuration options that maximize QoS for a particular set of hardware, OS, and compiler platforms may not produce optimal QoS for a different configuration. As a result, many performance bottlenecks and QoS degradation escape detection until systems are fielded.

To address software variability and its impact on QoS, we need software *processes* that aid in systematically and efficiently evaluating system QoS as well as *tools* that synthesize artifacts (such as benchmarking regression suite) for estimating system QoS on various configuration options. Our ultimate goal is to identify patterns of recurring configurations of software that satisfy systems QoS for various types of DRE systems in different domains. We are validating our approach on a range of hardware, OS, and compiler platforms by integrating the following technologies:

*Distributed continuous quality assurance (DCQA) techniques [**?**]..* These techniques are designed to improve software quality and performance iteratively, opportunistically, efficiently, and continuously in multiple, geographically distributed locations [**?**]. We have developed a prototype DCQA environment called *Skoll* (`www.cs.umd.edu/projects/skoll`) that help resolves challenges associated with ensuring software attributes (such as functional correctness and performance) across diverse platforms by executing QA tasks continuously and intelligently across a grid of computers distributed around the world.

*Model-driven software development techniques..* These techniques are designed to minimize the cost of QA activities by capturing the configuration and customization (C&C) variability of middleware within models and automatically generating configuration files from these higher level models [**?**]. We have developed prototype model-driven software tools in the CoSMIC project [**?**], including (1) the Options Configuration Modeling language (OCML) [**?**] that allows developers to model middleware configuration options as high-level models and (2) model-driven test and benchmarking tools (BGML) [**?**] that allow developers to compose benchmarking experiments that observe QoS behavior by mixing and matching middleware configurations.

## 2. ADDRESSING C&C CHALLENGES USING MODEL-DRIVEN DCQA

Our experience to date [**?**, **?**] indicates that model-driven DCQA techniques help capture configurations that can achieve end-to-end QoS on diverse hardware, OS, and compiler platforms. These techniques have not yet been shown to reduce the time required to identify configuration options that maximize the QoS for a given appli-


ACM 0-89791-88-6/97/05.

cation, however, due to the following unresolved challenges:

- **Lack of effective configuration space exploration techniques**, which increase the number of potential configurations that must be evaluated empirically to identify candidate configurations that maximize QoS in a particular context.
- **Lack of reusable configurations applicable across domains**, which require DRE application developers to (re)explore the configuration space and expend effort that would otherwise have been minimized had there been proven and reusable documented configurations.

To help resolve these challenges, we are applying the following two-pronged approach:

*The Skoll DCQA environment..* We use Skoll to evaluate configuration options and identify/validate configuration solutions that are applicable across a wide range of hardware, OS, and compiler platforms. The Skoll environment includes languages for modeling system configurations and their constraints, algorithms for scheduling and remotely executing tasks, and analysis techniques for characterizing results.

*Capture software variability as C&C patterns..* A design pattern presents a solution to a common software problem within a particular context [**?**]. A C&C pattern is similar to a design pattern in that it represents a recurring solution to a *configuration and customization* problem arising within a particular context, *e.g.*, for a certain domain, such as middleware, web services, or database systems. C&C patterns are expressed as tuples consisting of configuration parameters and their settings for each individual platform that help developers identify options that significantly influence QoS and provide feedback that can be used to choose configurations that assure QoS.

For example, Table 1 illustrates a configuration for ACE+TAO middleware that maximizes throughput for applications that do not need concurrency. These options and their settings represent a

| Notation | Option Name | Option Settings |
|---|---|---|
| o1 | ORBProfileLock | {Null} |
| o2 | ORBClientConnectionHandler | {RW } |
| o3 | ORBTransportMuxStrategy | {EXCLUSIVE } |
| o4 | ORBConnectStrategy | {reactive } |

**Table 1: A C&C Pattern for Single-threaded Applications**

C&C pattern that can be reused for similar contexts across several domains and have a significant influence on performance, irrespective of other configuration options and their settings.

Below, we briefly describe how we have used Skoll and its model-driven tool suite on a range of platforms (including Windows, Linux, Solaris, and several real-time operating systems) to identify configurations that impact performance significantly and document them as C&C patterns:

*Step 1: Define the application scenario.* At the highest level, users employ the model-driven OCML and BGML tools in CoSMIC to depict the interaction between their components and objects.

*Step 2: Create benchmarks using the model-driven BGML tool.* The model interpreters in OCML and BGML traverse the models to synthesize the configuration and experimentation code. The OCML interpreter generates the configuration files while the BGML interpreter generates the required benchmarking code, *i.e.*, code to set-up, run, tear-down the experiment.

*Step 3: Evaluate configurations using DCQA processes.* The configured experimentation code can next be fed to the Skoll DCQA environment which does exhaustive testing to run each candidate configuration.

The data from the configurations are stored in an internal database and analyzed to observe behavior applicable across a wide range of platforms. The configuration that caused this behavior is documented as a C&C pattern.

## 2.1 Concluding Remarks

Software for DRE systems needs to be fine-tuned to specific user platforms/contexts to meet end-to-end QoS requirements. This variability can yield an explosion in the software configuration space, which places enormous demands on the developers who must must ensure that their decisions and modifications work across this large (and often changing) configuration space. Our codification of reusable configurations and customizations as C&C patterns is helping to formalize QoS design expertise and is enabling their reuse across several application domains, thereby minimizing unnecessary effort expended in rediscovering these C&C patterns for each application domain.