A Distributed Continuous Quality Assurance Process to Manage Variability in Performance-intensive Software

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

Abstract

Performance-intensive software is increasingly being used on heterogeneous combinations of OS, compiler, and hardware platforms. Examples include reusable middleware that forms the basis for scientific computing grids and distributed real-time and embedded systems. Since this software has stringent quality of service (QoS) requirements, it often provides a multitude of configuration options that can be tuned for specific application workloads and run-time environments.

This paper describes the architecture of Skoll, which is a DCQA environment containing software QA processes and tools that leverage the extensive computing resources of worldwide user communities to significantly and rapidly improve software quality. It describes novel modeling tools and modeling language BGML that allow Skoll users to capture the system's axes of variability (such as configuration options, QoS strategies, and platform dependencies) to generate scaffolding code needed to conduct QA tasks on remote machines. It describes experiments that apply BGML to systematically evaluate and improve the performance of DRE component middleware on a range of platforms and configuration options. The results show that automatic analysis of QA task results can significantly improve software quality by capturing the impact of software variability on performance and providing feedback to help developers optimize performance.

1 Introduction

Emerging trends and challenges. Well-documented trends towards the expanding role of software in mission-critical systems, greater time-to-market pressures on information technology (IT) suppliers, and decreasing budgets for corporate-sponsored IT R&D are exposing deficiencies in conventional quality assurance (QA) processes. These processes have traditionally performed functional testing, code inspections/profiling, and quality of service (QoS) performance evaluation/optimization *in-house* on developer-generated workloads and regression suites. Unfortunately, in-house QA processes are not delivering the level of quality software needed for large-scale mission-critical systems 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 configuration options.

The deficiencies of in-house QA processes are particularly problematic for *performance-intensive software systems*. Examples of this type of software include high-performance scientific computing systems, distributed real-time and embedded (DRE) systems, and the accompanying systems software (*e.g.*, operating systems, middleware, and language processing tools). Reusable software for these types of systems must not only function correctly across the multiple contexts in which it is reused and customized, it must also do so efficiently and predictably.

To support the customizations demanded by users, reusable performance-intensive software often must (1) run on a variety of hardware/OS/compiler platforms and (2) provide a variety of options that can be configured at compile- and/or run-time. For example, performance-intensive middleware, such as web servers (*e.g.*, Apache), object request brokers (*e.g.*, TAO), and databases (*e.g.*, Oracle) run on dozens of platforms and have dozen or hundreds of options. While this variability promotes customization, it also creates many potential system configurations, each of which may need extensive QA to validate. Consequently, a key challenge for developers of reusable performance-intensive software involves managing variability effectively in the face of an exploding *software configuration space*. As software configuration spaces increase in size and software development resources decrease, it becomes infeasible to handle all QA activities in-house. For instance, developers may not have access to all the hardware, OS, and compiler platforms on which their reusable software artifacts will run. Moreover, due to time-to-market driven environments, developers may be forced to release their software in configurations that have not been subjected to sufficient QA. The combination of an enormous configuration space and severe development resource constraints therefore often force software developers to make design and optimization decisions without precise knowledge of their consequences in fielded systems.

Solution approach → Distributed continuous QA processes and tools. In response to the trends and challenges described above, developers and organizations have begun to change the processes they use to build and validate performance-intensive software. Specifically, they are moving towards more agile processes characterized by (1) decentralized development teams, (2) greater reliance on middleware component reuse, assembly, and deployment, (3) evolution-oriented development requiring frequent software updates, (4) product designs that allow extensive end-user customization, and (5) software repositories that help to consolidate and coordinate QA tasks associated with the other four characteristics outlined above. While these agile processes address key challenges with conventional QA approaches, they also create new challenges, *e.g.*, coping with frequent software changes, remote developer coordination, and exploding software configuration spaces.

To address the challenges with conventional and agile software QA processes, we have developed a distributed continuous quality assurance (DCQA) environment called **Skoll** (`www.cs.umd.edu/projects/skoll`) that supports around-the-world, around-the-clock QA on a computing grid provided by end-users and distributed development teams. The Skoll environment includes languages for modeling key characteristics of performance-intensive software configurations, algorithms for scheduling and remotely executing QA tasks, and analysis techniques that characterize software faults and QoS performance bottlenecks. Our feedback-driven Skoll environment divides QA processes into multiple subtasks that are intelligently and continuously distributed to, and executed by, a grid of computing resources contributed by end-users and distributed development teams around the world. The results of these executions are returned to central collection sites where they are fused together to identify defects and guide subsequent iterations of the QA process.

Our earlier publications [1] on Skoll described its structure and functionality and presented results from a feasibility study that applied Skoll tools and processes to ACE [2] and TAO [3], which are large (*i.e.*, over two million SLOC) reusable middleware packages targeted at performance-intensive software for DRE systems. Our initial work focused largely on building the Skoll infrastructure, which consisted of the languages, algorithms, mechanisms, and analysis techniques that tested the *functional correctness* of reusable software and its application to end-user systems.

This paper describes several other dimensions of DCQA processes and the Skoll environment: (1) *integrating model-based techniques with DCQA processes*, (2) *improving QoS as opposed to simply functional correctness*, and (3) *using Skoll to empirically optimize a system for specific run-time contexts*. At the heart of the Skoll work presented in this paper is **BGML** [4], which is Model-based toolsuite[1] that applies generative model-based software techniques [5] to measure and optimize the QoS of reusable performance-intensive software configurations.

[1]BGML can be downloaded from `www.dre.vanderbilt.edu/cosmic`.

BGML extends Skoll's earlier focus on functional correctness to address QoS issues associated with reusable performance-intensive software, *i.e.*, modeling and benchmarking interaction scenarios on various platforms by mixing and matching configuration options. By integrating BGML into the Skoll process, QoS evaluation tasks are performed in a feedback-driven loop that is distributed over multiple sites. Skoll tools analyze the results of these tasks and use them as the basis for subsequent evaluation tasks that are redistributed to the Skoll computing grid.

The specific contributions of the work reported in this paper include:

- Defining model-based tools to automate common Skoll QA tasks.
- Integrating QoS evaluation and optimization into the Skoll DCQA process.
- Automating benchmark generation and profiling QoS measures of highly configurable, reusable, and multi-platform performance-intensive software.
- Demonstrating the correctness and utility of model-based QA in a feasibility study involving standards-based DRE component middleware.

Paper organization. The remainder of this paper is organized as follows: Section 2 presents an overview of the Skoll DCQA architecture focusing on interactions between various components and services; Section 3 motivates and describes our model based meta-programmable tool (BGML), focusing on its syntactic and semantic modeling elements that help QA engineers to visually compose QA tasks for Skoll and its generative capabilities to the resolve accidental complexities associated with quantifying the impact of software variability on QoS; Section 4 reports the results of experiments using this model-based DCQA process on the CIAO QoS-enabled component middleware framework; Section 5 examines related work and compares it with the approaches used in Skoll and BGML; and Section 6 presents concluding remarks and outlines future work.

2 Overview of the Structure and Functionality of Skoll

To address limitations with in-house QA approaches, the Skoll project is developing and empirically evaluating feedback-driven processes, methods, and supporting tools for *distributed continuous QA*. In this approach software quality is improved – iteratively, opportunistically, and efficiently – around-the-clock in multiple, geographically distributed locations. To support distributed continuous QA processes, we have implemented a set of components and services called the *Skoll infrastructure*, which includes languages for modeling system configurations and their constraints, algorithms for scheduling and remotely executing tasks, and analysis techniques for characterizing faults.

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. These nodes request work from a server when they wish to make themselves available.

The remainder of this section describes the components, services and interactions within the Skoll infrastructure and provides a sample scenario showing how they are used to implement Skoll processes.

2.1 The Skoll Infrastructure

Skoll QA processes are based on a client/server model. Clients distributed throughout the Skoll grid request *job configurations* (implemented as QA subtask scripts) from a Skoll server. The server determines which subtasks to allocate, bundles up all necessary scripts and artifacts, and sends them to the client. The client executes the subtasks and returns the results to the server. The server analyzes the results, interprets them, and modifies the process as appropriate, which may trigger a new round of job configurations for subsequent clients running in the grid.

At a lower level, the Skoll QA process is more sophisticated. QA process designers must determine (1) how tasks will be decomposed into subtasks, (2) on what basis and in what order subtasks will be allocated to clients, (3) how subtasks will be implemented to execute on a potentially wide set of client platforms, (4) how subtask results will be merged together and interpreted, (5) if and how should the process adapt on-the-fly based on incoming results, and (6) how the results of the overall process will be summarized and communicated to software developers. To support this process we've developed the following components and services for use by Skoll QA process designers (a comprehensive discussion appears in [1]):

Configuration space model. A cornerstone of Skoll is a formal model of a QA process's configuration space, which captures all valid configurations for QA subtasks. This information is used in planning the global QA process, for adapting the process dynamically, and to aid in interpreting results. In practice not all configurations make sense due to platform variability, *e.g.*, feature X may not be supported on operating system Y. Skoll therefore allows *inter-option constraints* that limit the setting of one option based on the setting of others. Constraints are expressed as ($P_i \longrightarrow P_j$), meaning "if predicate P_i evaluates to TRUE, then predicate P_j must evaluate to TRUE." A predicate P_k can be of the form A, $\neg A$, $A\&B$, $A|B$, or simply (V_i, C_i), where A, B are predicates, V_i is a option and C_i is one of its allowable values. A *valid configuration* is a configuration that violates no inter-option constraints.

Intelligent Steering Agent. A novel feature of Skoll is its use of an *Intelligent Steering Agent* (ISA) to control the global QA process by deciding which valid configuration to allocate to each incoming Skoll client request. The ISA treats configuration selection as an AI planning problem. For example, given the current state of the global process including the results of previous QA subtasks (*e.g.*, which configurations are known to have failed tests), the configuration model, and metaheuristics (*e.g.*, nearest neighbor searching), the ISA will chose the next configuration such that process goals (*e.g.*, evaluate configurations in proportion to known usage distributions) will be met. After a valid configuration is chosen, the ISA packages the corresponding QA subtask implementation, which consists of the code artifacts, configuration parameters, build instructions, and QA-specific code (*e.g.*, regression/performance tests) associated with a software project. This package is called a *job configuration*.

Adaptation strategies. As QA subtasks are performed by clients in the Skoll grid, their results are returned to the ISA, which can learn from the incoming results. For example, when some configurations prove to be faulty, the ISA can refocus resources on other unexplored parts of the configuration space. To support such dynamic behavior, Skoll QA process designers can develop customized *adaptation strategies* that monitor the global QA process state, analyze it, and use the information to modify future subtask assignments in ways that improve process performance.

2.2 Skoll in Action

At a high level, the Skoll process is carried out as shown in Figure 1.

1. Developers create the configuration model and adaptation strategies. The ISA automatically translates the model into planning operators. Developers create the generic QA subtask code that will be specialized when creating actual job configurations.

2. A *user* requests Skoll client software via the registration process described earlier. The user receives the Skoll client software and a configuration template. If a user wants to change certain configuration settings or constrain specific options he/she can do so by modifying the configuration template.

3. A Skoll client periodically (or on-demand) requests a job configuration from a Skoll server.

4. The Skoll server queries its databases and the user-provided configuration template to determine which configuration option settings are fixed for that user and which must be set by the ISA. It then packages this information as a planning goal

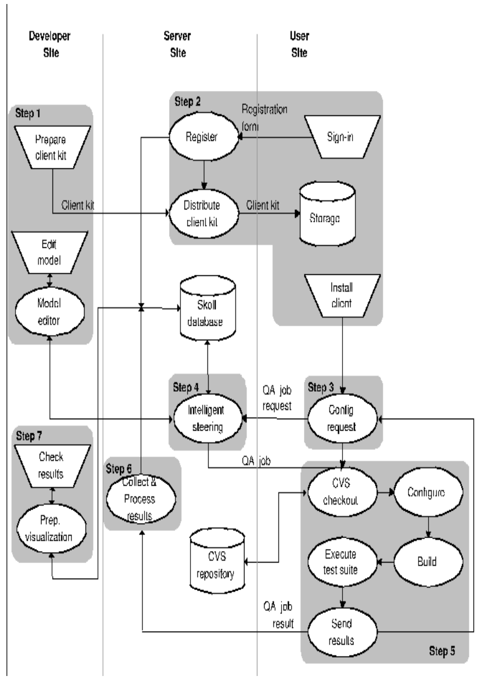

Figure 1: Skoll QA Process View

and queries the ISA. The ISA generates a plan, creates the job configuration and returns it to the Skoll client.

5. A Skoll client invokes the job configuration and returns the results to the Skoll server.

6. The Skoll server examines these results and invokes all adaptation strategies. These update the ISA operators to adapt the global process.

7. The Skoll server prepares a *virtual scoreboard* that summarizes subtask results and the current state of the overall process. This scoreboard is updated periodically and/or when prompted by developers.

3 Enhancing Skoll with a Model-based QoS Improvement Process

Reusable performance-intensive software is often used by applications with stringent QoS requirements, such as low latency and bounded jitter. The QoS of reusable performance-intensive software is influenced heavily by factors such as (1) the configuration options set by end-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 (2) 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 a QA process that can precisely pinpoint the consequences of mixing and matching configuration options on various platforms. In particular, such a QA process should resolve the following forces:

1. Minimize the time and effort associated with testing various configuration options on particular platforms,
2. Provide a framework for seamless addition of new test configurations corresponding to various platform environment and application requirement contexts,

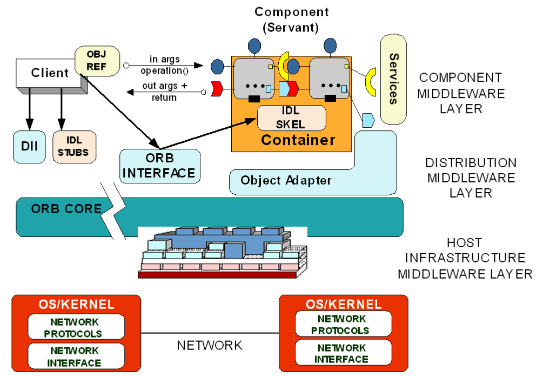

Figure 2: **Elements in the CCM Architecture**

In our initial Skoll approach, creating a benchmarking experiment to measure QoS properties required QA engineers to write (1) the header files, source code, that implement the functionality, (2) the configuration and script files that tune the underlying ORB and automate running tests and output generation, and (3) project build files (*e.g.*, makefiles) required to generate the executable code. Our experience during our initial feasibility study [1] revealed how tedious and error-prone this process was since it required multiple manual steps to generate benchmarks, thereby impeding productivity and quality in the QA process.

The remainder of this section describes how we have applied model-based techniques [5] to resolve forces 1 and 2 outlined earlier. In this context, model-based techniques involve visual representations for defining entities and their interactions in an application domain using domain-specific building blocks. These improvements are embodied in BGML [4], which is a model-based benchmarking toolsuite designed to evaluate the QoS of implementations of the CORBA Component Model (CCM), which is shown in Figure 2 and described in Sidebar 1.[2] BGML allows CCM users to:

1. Model interaction scenarios between CCM components using varied configuration options, *i.e.*, capture software

[2] We focus on CCM in our work since it is standard component middleware that is targeted for the QoS requirements of DRE systems. As QoS support for other component middleware matures we will enhance our modeling tools and DCQA processes to integrate them.

Sidebar 1: Overview of CCM

The CORBA Component Model (CCM) forms a key part of the CORBA 3.x standard [6]. CCM is designed to address the limitations with earlier versions of CORBA 2.x [7] middleware that supported a distributed object computing (DOC) model [8]. Figure 2 depicts the key elements in the architecture of CCM, which are described below.

Components. *Components* in CCM are implementation entities that collaborate with each other via *ports*. CCM supports several types of ports, including (1) *facets*, which define an interface that accepts point-to-point method invocations from other components, (2) *receptacles*, which indicate a dependency on point-to-point method interface provided by another component, and (3) *event sources/sinks*, which indicate a willingness to exchange typed messages with one or more components.

Container. A *container* in CCM provides the run-time environment for one or more components that manages various pre-defined hooks and strategies, such as persistence, event notification, transaction, and security, used by the component(s). Each container is responsible for (1) initializing instances of the component types it manages and (2) connecting them to other components and common middleware services. Developer specified metadata expressed in XML can be used to instruct CCM deployment mechanisms how to control the lifetime of these containers and the components they manage. The meta-data is present in XML files called *descriptors*.

Component assembly. In a distributed system, a component may need to be configured differently depending on the context in which it is used. As the number of component configuration parameters and options increase, it can become tedious and error-prone to configure applications consisting of many individual components. To address this problem, the CCM defines an *assembly* entity to group components and characterize the meta-data that describes these components in an assembly. Each component's meta-data in turn describes the features available in it (*e.g.*, its properties) or the features that it requires (*e.g.*, its dependencies).

Component server. A *component server* is an abstraction that is responsible for aggregating *physical* entities (*i.e.*, implementations of component instances) into *logical* entities (*i.e.*, distributed application services and subsystems).

Component packaging and deployment. In addition to the run-time building blocks outlined above, the CCM also standardizes component implementation, packaging, and deployment mechanisms. Packaging involves grouping the implementation of component functionality – typically stored in a dynamic link library (DLL) – together with other meta-data that describes properties of this particular implementation. The CCM Component Implementation Framework (CIF) helps generate the component implementation skeletons and persistent state management automatically using the Component Implementation Definition Language (CIDL).

variability in higher-level models rather than in lower-level source code.

2. Automate benchmarking code generation to systematically identify performance bottlenecks based on mixing and matching configurations.
3. Generate control scripts to distribute and execute the experiments to users around the world to monitor QoS performance behavior in a wide range of execution contexts.
4. Evaluate and compare CCM implementation performances in a highly automated way the overhead that CCM implementations impose above and beyond CORBA 2.x implementations based on the DOC model.
5. Enable comparison of CCM implementations using key metrics, such as throughput, latency, jitter, and other QoS criteria.
6. Develop a framework that automates benchmark tests and facilitates the seamless integration of new tests.

With BGML, QA engineers graphically model possible interaction scenarios. Given a model, BGML generates the scaffolding code needed to run the experiments. This typically includes Perl scripts that start daemon processes, spawn the component server and client, run the experiment, and display the required results. BGML is built on top of the Generic Modeling Environment (GME) [9], which provides a meta-programmable framework for creating domain-specific modeling languages and generative tools. GME is programmed via *meta-models* and *model interpreters*. The meta-models define modeling languages called paradigms that specify allowed modeling elements, their properties, and their relationships. Model interpreters associated with a paradigm can also be built to traverse the paradigm's modeling elements, performing analysis and generating code.

Figure 3 presents an overview of how we have integrated BGML with the Skoll infrastructure. Below we describe how

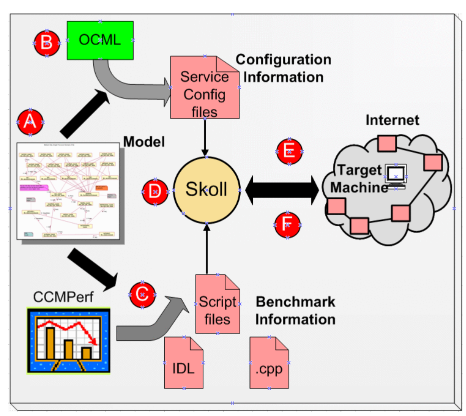

Figure 3: **Skoll QA Process View with BGML Enhancements**

our BGML modeling tools interact with the existing Skoll infrastructure to enhance its DCQA capabilities.

A. QA engineers define a test configuration using BGML models. The necessary experimentation details are captured in the models, *e.g.*, the ORB configuration options used, the IDL interface exchanged between the client and the server, and the benchmark metric performed by the experiment.

B & C. QA engineers then use BGML to interpret the model. The OCML paradigm interpreter parses the modeled ORB configuration options and generates the required configuration files to configure the underlying ORB. The BGML paradigm interpreter then generates the required benchmarking code, *i.e.*, IDL files, the required header and source files, and necessary script files to run the experiment. Steps A, B, and C are integrated with Step 1 of the Skoll process.

D. When users register with the Skoll infrastructure they obtain the Skoll client software and configuration template. This step happens in concert with Step 2, 3, and 4 of the Skoll process.

E & F. The client executes the experiment and returns the result to the Skoll server, which updates its internal database. When prompted by developers, Skoll displays execution results using an on demand scoreboard. This scoreboard displays graphs and charts for QoS metrics, *e.g.*, performance graphs, latency measures and foot-print metrics. Steps E and F correspond to steps 5, 6, and 7 of the Skoll process.

4 Feasibility Study

This section describes the design and results of an experiment we conducted to evaluate the enhanced DCQA capabilities that stem from integrating BGML with Skoll. In this paper, we use the BGML modeling tools and Skoll infrastructure to execute a formally-designed experiment using a *full-factorial design*, which executes the experimental task (benchmarking in this case) exhaustively across all combinations of the experimental options (a subset of the configuration parameters of the CIAO QoS-enabled component middleware).

The data from our experiments is returned to the Skoll server, where it is organized into a database. The database then becomes a resource for developers of applications and middleware who wish to study the system's performance across its many different configurations. Since the data is gathered through a formally-designed experiment, we use statistical methods (*e.g.*, analysis of variance, wilcox ran sum tests, and classification tree analysis) to analyze the data. To demonstrate the utility of this approach, we present two use cases that show how (1) CIAO developers can query the database to improve the performance of the component middleware software and (2) application developers can fine-tune CIAO's configuration parameters to improve the performance of their software.

4.1 Hypotheses

The use cases we present in this section explore the following hypotheses:

1. The Skoll grid can be used together with BGML to quickly generate benchmark experiments that pinpoint specific QoS performance aspects of interest to developers of middleware and/or applications, *e.g.*, BGML allows QA process engineers to quickly setup QA processes and generate significant portions of the required benchmarking code.
2. Using the output of BGML, the Skoll infrastructure can be used to (1) quickly execute benchmarking experiments on end-user resources across a Skoll grid and (2) capture and organize the resulting data in a database that can be used to improve the QoS of performance-intensive software.
3. Developers and users of performance-intensive software can query the database to gather important information about that software, *e.g.*, obtain a mix of configuration option settings that improve the performance for their specific workload(s).

4.2 Experimental Process

We used the following experimental process to evaluate the hypotheses outlined in Section 4.1:

Step 1: Choose a software system that has stringent performance requirements. Identify a relevant configuration space.

Step 2: Select workload application model and build benchmarks using BGML.

Step 3: Deploy Skoll and BGML to run benchmarks on multiple configurations using a full factorial design of the configuration options. Gather performance data.

Step 4: Formulate and demonstrate specific uses of the performance results database from the perspective of both middleware and application developers.

4.2.1 Step 1: Subject Applications

We used ACE 5.4 + TAO 1.4 + CIAO 0.4 for this study. CIAO [10] is a QoS-enabled implementation of CCM (see Sidebar 1) developed at Washington University, St. Louis and Vanderbilt University to help simplify the development of performance-intensive software applications by enabling developers to declaratively provision QoS policies end-to-end when assembling a DRE system. CIAO adds component support to TAO [3], which is distribution middleware that implements key patterns [11] to meet the demanding QoS requirements of DRE systems.

4.2.2 Step 2: Build Benchmarks

Figure 4 describes how the ACE+TAO+CIAO QA engineers used the BGML tool to generate the screening experiments to quantify the behavior of latency and throughput. As shown in this figure, the following steps were performed:

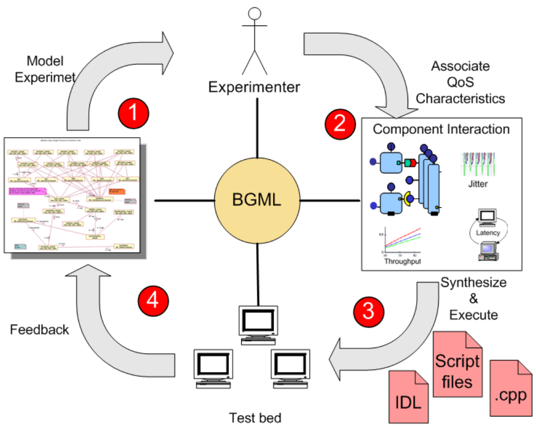

Figure 4: **BGML Use Case Scenario**

1. QA engineers used the BGML modeling paradigm to compose the experiment. In particular, QA engineers use the domain-specific building blocks in BGML to compose experiments.
2. In the experiment modeled, QA engineers associated the QoS characteristic (in this case roundtrip latency and throughput) that will be captured in the experiment. Figure 5 depicts how this is done in BGML.

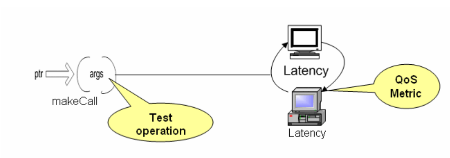

Figure 5: **Associating QoS Metrics in BGML**

3. Using the experiment modeled by QA engineers, BGML interpreters generated the benchmarking code required to set-up, run, and tear-down the experiment. The generated files include component implementation files (.h and .cpp), IDL files (.idl), component IDL files (.cidl), and benchmarking code (.cpp) files.
4. The generated file was then executed using the Skoll DCQA process and QoS characteristics were measured. The execution was done in Step 4 described in Section 4.2.4.

4.2.3 Step 3: Execute the DCQA process

For this version of ACE+TAO+CIAO, we identified 14 run-time options that could affect latency and throughput. As shown in Table 1, each option is binary, so the entire configuration space is $2^{14} = 16,384$. We executed the benchmark experiments on each of the 16,384 configurations. This is called a full-factorial experimental design. Clearly such designs will not scale up to arbitrary numbers of factors. In ongoing work we are therefore studying strategies for reducing the number of observations that must be examined. In the current example, however, the design is manageable.

Option Index	Option Name	Option Settings
opt1	ORBReactorThreadQueue	{FIFO, LIFO}
opt2	ORBClientConnectionHandler	{RW, MT}
opt3	ORBReactorMaskSignals	{0, 1}
opt4	ORBConnectionPurgingStrategy	{LRU, LFU}
opt5	ORBConnectionCachePurgePercentage	{10, 40}
opt6	ORBConnectionCacheLock	{thread, null}
opt7	ORBCorbaObjectLock	{thread, null}
opt8	ORBObjectKeyTableLock	{thread, null}
opt9	ORBInputCDRAllocator	{thread, null}
opt10	ORBConcurrency	{reactive, tpc}
opt11	ORBActiveObjectMapSize	{32, 128}
opt12	ORBUseridPolicyDemuxStrategy	{linear, dynamic}
opt13	ORBSystemidPolicyDemuxStrategy	{linear, dynamic}
opt14	ORBUniqueidPolicyReverseDemuxStrategy	{linear, dynamic}

Table 1: **The Configuration Space: Run-time Options and their Settings**

For a given configuration, we use the BGML modeling paradigms to model the configuration visually and generate the scaffolding code to run the benchmarking code. The experiment was run three times and for each run the client sent 300,000 requests to the server. In total, we distributed and ran ~50,000 benchmarking experiments. For each run, we measured the latency values for each request and total throughput (events/second).

The BGML modeling tool helps improve the productivity of QA engineers by allowing them to compose the experiment *visually* rather than wrestling with low-level source code. This tool thus resolves tedious and error-prone accidental complexities associated with writing correct code by auto-generating them from higher level models. Table 2 summarizes the BGML code generation metrics for a particular configuration.

Files	Number	Lines of Code	Generated (%)
IDL	3	81	100
Source (.cpp)	2	310	100
Header (.h)	1	108	100
Script (.pl)	1	115	100
Config (svc.conf)	1	6	100
Descriptors (XML)	2	90	0

Table 2: **Generated Code Summary for BGML**

This table shows how BGML automatically generates 8 of 10 required files that account for 88% of the code required for the experiment.

4.2.4 Step 4: Example Use Cases

Below we present two use cases that leverage the data collected by the Skoll DCQA process. The first scenario involves application developers who need information to help configuring CIAO for their use. The second involves CIAO middleware developers who want to prioritize certain development tasks.

Use case #1: Application developer configuration. In this scenario, a developer of a performance-intensive software application is using CIAO. This application is expected to have a fairly smooth traffic stream and needs high overall throughput and low latency for individual messages. This developer has decided on several of the option settings needed for his/her application, but is unsure how to set the remaining options and what effect those specific settings will have on application performance. To help answer this question, the application developer goes to the ACE+TAO+CIAO Skoll web page and identifies the general workload expected by the application, the platform, OS, and ACE+TAO+CIAO versions used. Next, the developer arrives at the web page shown in Figure 6. On this page the application developer inputs those option settings (s)he expects to use and left unspecified (denoted "*") those for which (s)he needs guidance. The developer also indicates the performance metrics (s)he wishes to analyze and then submits the page.

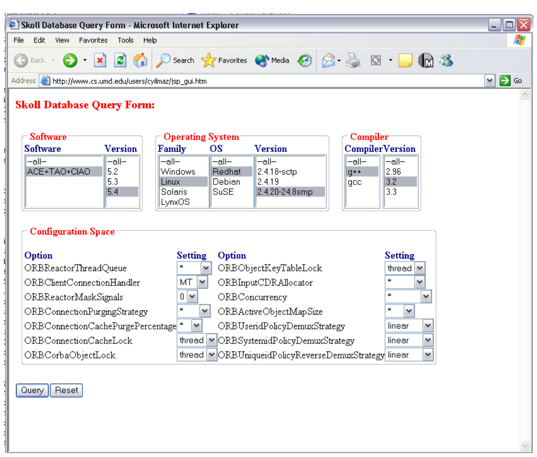

Figure 6: **Accessing Performance Database**

Submitting the page causes several things to happen. First, the data corresponding to the known option settings is located in the Skoll databases. Next, the system graphs the historical performance distributions of both the entire configuration space and the subset specified by the application developer (*i.e.*, the subset of the configuration space consistent with the developer's partially-specified options). These graphs are shown in Figure 7 and Figure 8. Last, the system presents a statistical analysis of the options that significantly affect the performance measures, as depicted in Figure 8. Together, these views present the application developer with several pieces of information. First, it shows how the expected configuration has performed historically on a specific set of benchmarks. Next, it compares this configuration's performance with the performance of other possible configurations. It also indicates which of the options have a significant effect on performance and thus should be considered carefully when selecting the final configuration.

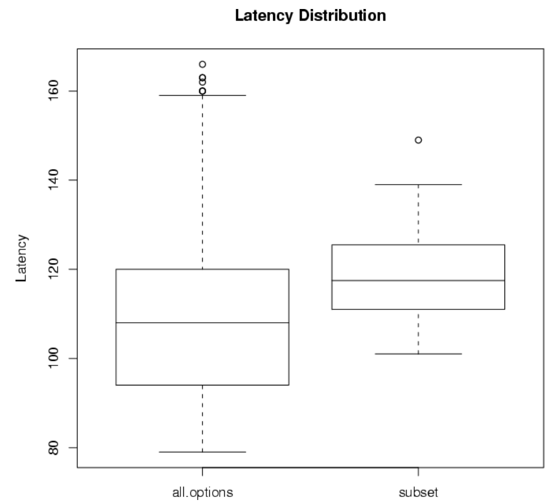

Figure 7: 1^{st} **Iteration**

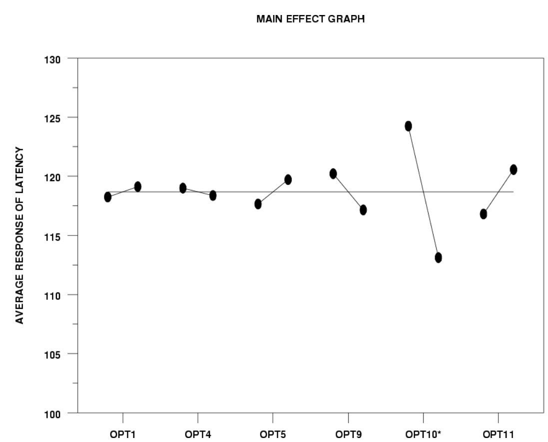

Figure 8: 1^{st} **Iteration: Main Effects Graph (Statistically Significant Options are Denoted by an *)**

Continuing our use case example, the application developer sees that option *opt10* (`ORBConcurrency`) has not been set and that it has a significant effect on performance. To better understand the effect of this option, the developer consults the main effects graph shown in Figure 8). This plot shows that setting `ORBConcurrency` to *thread-per-connection* (where the ORB dedicates one thread to each incoming connection) should lead to better performance than setting it to *reactive* (where the ORB uses a single thread to detect, demultiplex and service multiple client connections). The application developer therefore sets the option and reruns the earlier analysis. The new analysis shows that, based on historical data, the new setting does indeed improve performance, as shown in Figure 9. However, the accompanying main effects graph

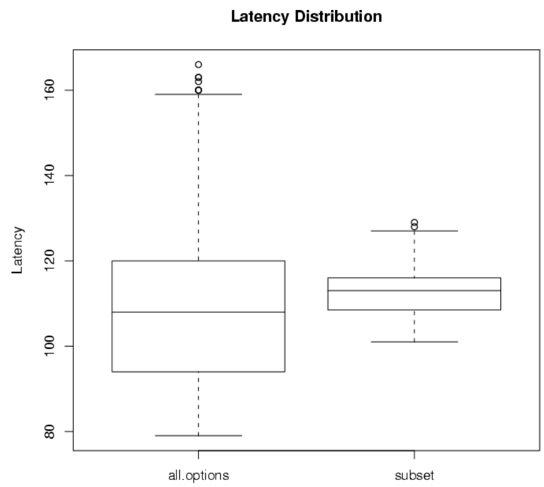

Figure 9: 2^{nd} **Iteration**

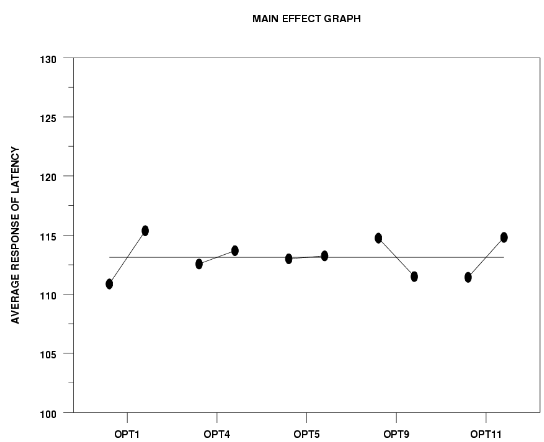

Figure 10: 2^{nd} **Iteration: Main effects graph (Statistically Significant Options are Denoted by an *)**

shown in Figure 10 shows that the remaining unset options are unlikely to have a substantial effect on performance. At this point, the application developer has several choices, *e.g.*, (s)he can stop here and set the remaining options to their default settings or (s)he can revisit the original settings. In this case, our developer reexamines the original settings and their main effects (See Figure 11) and determines that changing the setting of *opt2* (`ORBClientConnectionHandler`) might greatly improve performance.

Using this setting will require making some changes to the actual application, so the application developer reruns the analysis to get an idea of the potential benefits of changing the option setting. The resulting data is shown in Figure 12. The results in this figure show that the performance improvement from setting this option would be substantial. The developer would now have to decide whether the benefits justify the costs of changing the application.

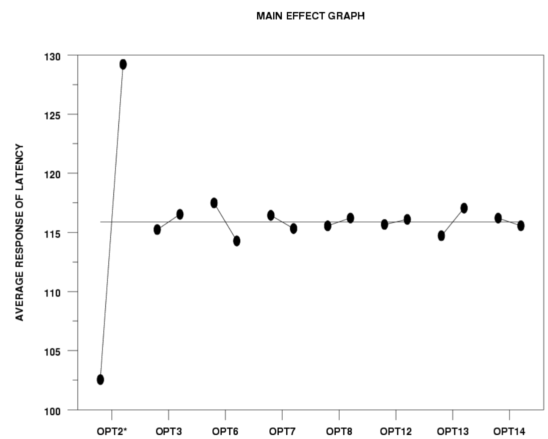

Figure 11: 3^{rd} **Iteration: Main Effects Graph (Statistically Significant Options are Denoted by an *)**

Use case #2: Middleware developer task prioritization. In this scenario, a developer of CIAO middleware itself wants to do an exploratory analysis of the system's performance across its configuration space. This developer is looking for areas that are in the greatest need of improvement. To do this (s)he accesses the ACE+TAO+CIAO and Skoll web page and performs several tasks. First, (s)he examines the overall performance distribution of one or more performance metric. In this case, the middleware developer examines measurements of system latency, noting that the tails of the distribution are quite long (the latency plots are the same as those found in the "all.options" subplots of Figure 7). The developers wants to better understand which specific configurations are the poor performers.[3]

[3]For latency the worst performers are found in the upper tail, whereas for throughput it is the opposite.

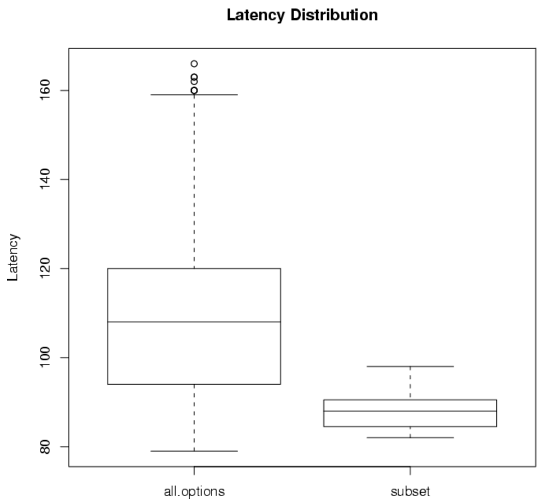

Figure 12: 3^{rd} **Iteration: Step 3**

Our DCQA process casts this question as a classification problem. The middleware developer therefore recodes the performance data into two categories: those in the worse-performing 10% and the rest. From here out, (s)he considers poor performing configurations as those in the bottom 10%. Next, (s)he uses classification tree analysis [12] to model the specific combinations of options that lead to degraded performance.

For our current use case example, the middleware developer uses a classification tree to extract performance-degrading option patterns, *i.e.*, (s)he extracts the options and option settings from the tree that characterize poorly performing configurations. Figure 13 shows one tree obtained from the CIAO data (for space reasons the tree shown in the Figure gives only a coarse picture of the information actually contained in the tree). By examining the tree, the middleware developer

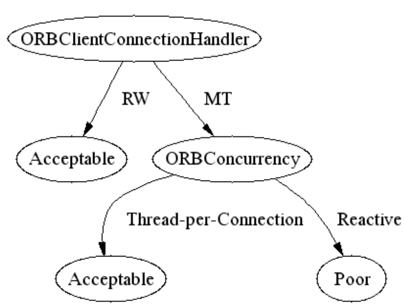

Figure 13: **Sample Classification Tree Modeling Poorly Performing Configurations**

notes that a large majority of the poorly performing configurations have `ORBClientConnectionHandler` set to *MT* and `ORBConcurrency` set to *reactive*. The first option indicates that the CORBA ORB uses separate threads to service each incoming connections. The second option indicates that the ORB's reactor [11] (the framework that detects and accepts connections and dispatches event to the corresponding event handlers when events arrive) are executed by a pool of threads.

The information gleaned by the classification tree is then used to guide exploratory data analysis. To help middleware developers organize and visualize the large amount of data, we employed the Treemaps data visualizer (`www.cs.umd.edu/hcil/treemap`), which allows developers to explore multidimensional data. The performance data described in the previous paragraph is shown in Figure 14. This figure shows poorly performing configurations as dark tiles and the acceptably performing configurations as lighter tiles. The layout first divides the data into two halves: the left for configurations with `ORBClientConnectionHandler` set to *RW* and the right for those set to *MT*. Each half is further subdivided, with the upper half for configurations with `ORCConcurrency` set to *thread-per-connection* and the lower half for those set to *reactive*. The data can be further subdivided to arbitrary levels, depending on how many options the middleware developer wishes to explore. The treemap shown in Figure 14 depicts how almost all the poor performers are in the bottom right quadrant, which suggests that the options discovered by the classification tree are reasonably good descriptors of the poorly performing configurations.

The middleware developer continues to explore the data, checking whether the addition of other options would further isolate the poor performers, thereby providing more information about the options that negatively influence performance. After some exploration, the middleware developer find no other influential options. Next, (s)he examines the poor performing configurations that are not part of the n group, *i.e.*, those with `ORBCurrency` set to *thread-per-connection* rather than *reactive*. The middleware developer determines that nearly all of the latency values for these configurations are quite close to the 10% cutoff. In fact, lowering the arbitrary cutoff to around 8% leads to the situation in which nearly every poor performer has `ORBConnectionClientHandler` set to *MT* and `ORBConcurency` set to *reactive*. Based on this information, the middleware developer can conduct further studies to determine whether a redesign might improve performance.

4.3 Discussion

The experiments reported in this section empirically explored how integrating BGML and Skoll allowed us to quickly implement specific DCQA processes to help application and middleware developers understand, use, and improve highly-variable

Figure 14: **Treemap Visualization**

performance-intensive systems. To accomplish this, we used BGML and Skoll to implement a DCQA process that conducted a large-scale, formally-designed experiment across a grid of remote machines. This process quickly collected performance data across all combinations of a set of system configuration options, thereby allowing application and middleware developers to conduct sophisticated statistical analyses.

We found that the BGML modeling approach allowed us to specify the relevant configuration space quickly and to automatically generate a large fraction of the benchmark code needed by the DCQA process. In our previous efforts [1] we performed these steps manually, making numerous errors. Overall, it took around 48 hours of CPU time to run the ∼50,000 experimental tasks dictated by the experimental design. Calendar time is effectively dictated by the number of end-users participating in the process. We see no problem conducting these types of experiment several times a day, which is particularly useful for ACE++TAOCIAO developers (whose middleware infrastructure changes quite frequently), since this will help keep the performance data in synch with the evolving middleware.

Although this paper focused on experiments over a single platform, we can run our Skoll DCQA process over many platforms. This cross-platform portability is extremely important to ACE++TAOCIAO developers because their middleware run over dozens of compiler/OS platforms, though individual middleware developers often have access to only a few platforms. Our DCQA process therefore gives individual developers virtual access to all platforms. Moreover, our approach makes performance data accessible to application developers and end-users, which helps extend the benefits of DCQA processes from the core to the periphery.

Despite the success of our experiments, we also found numerous areas for improvement. For example, we realize that exhaustive experimental designs can only scale up so far. As the number of configuration options under study grows, it will become increasingly important to find more efficient experimental designs. Moreover, the options we studied were binary and had no inter-option constraints, which will not always be the case in practice. Additional attention therefore must be paid to the experimental design to avoid incorrect analysis results.

We also found that much more work is needed to support data visualization and interactive exploratory data analysis. We have included some tools for this in Skoll, but they are rudimentary. More attention must be paid to characterizing the workload examined by the benchmark experiments. The one we used in this study modeled a constant flow of messages, but obviously different usage scenarios will call for different benchmarks. Finally, we note that our use cases focused on middleware and applications at a particular point in time. Time-series analyses that study systems as they evolve may also be valuable.

5 Related Work

This section compares our work on model-driven performance evaluation techniques in Skoll and BGML with other related research efforts including (1) large-scale testbed environments that provide a platform to conduct experiments using heterogeneous hardware, OS, and compiler platforms, (2) evaluating the performance of middleware layers, and (3) feedback-based optimization techniques that use empirical data and mathematical models to identify performance bottlenecks.

Large-scale benchmarking testbeds. EMULab [13] is a testbed at the University of Utah that provides an environment for experimental evaluation of networked systems. EMULab provides tools that researchers can use to configure the topology of their experiments, *e.g.*, by modeling the underlying OS, hardware, and communication links. This topology is then mapped [14] to $\sim$250 physical nodes that can be accessed via the Internet. The EMULab tools can generate script files that use the Network Simulator (NS) (`http://www.isi.edu/nsnam/ns/`) syntax and semantics to run the experiment.

The Skoll infrastructure provides a superset of EMULab that it is not limited by resources of a particular testbed, but instead can leverage the vast end-user computer resources in the Skoll grid. Moreover, the BGML model interpreters can generate NS scripts to integrate our benchmarks with experiments in EMULab.

Feedback-driven optimization techniques. Traditional feedback-driven optimization techniques can be divided into the following categories:

• **Offline analysis**, which has been applied to program analysis to improve compiler-generated code. For example, the ATLAS [15] numerical algebra library uses an empirical optimization engine to decide the values of optimization parameters by generating different program versions that are run on various hardware/OS platforms. The output from these runs are used to select parameter values that provide the best performance. Mathematical models are also used to estimate optimization parameters based on the underlying architecture, though empirical data is not fed into the models to refine it.

Our approach on BGML enhances ATLAS by feeding back platform-specific information into the models to identifying performance bottlenecks at model construction time. This information can be used to select optimal configurations ahead of time that maximize QoS behavior.

• **Online analysis**, where feedback control is used to dynamically adapt QoS measures. An example of online analysis is the ControlWare middleware [16], which uses feedback control theory by analyzing the architecture and modeling it as a feedback control loop. Actuators and sensors then monitor the system and affect server resource allocation. Real-time scheduling based on feedback loops has also been applied to Real-time CORBA middleware [17] to automatically adjust the rate of remote operation invocation transparently to an application.

Though online analysis enables systems to adapt at run-time, the optimal set of QoS features are not determined at system initialization. Using the model-based techniques provided by BGML, QoS behavior and performance bottlenecks on various hardware and software configurations can be determined offline and then fed into the models to generate optimal QoS characteristics at model construction time. Moreover, dynamic adaptation can incur considerable overhead from system monitoring and adaptation, which may be unacceptable for performance-intensive DRE systems.

• **Hybrid analysis**, which combines aspects of offline and online analysis. For example, the continuous compilation strategy [18] constantly monitors and improves application code using code optimization techniques. These optimizations are applied in four phases including (1) *static analysis*, in which information from training runs is used to estimate and predict optimization plans, (2) *dynamic optimization*, in which monitors apply code transformations at run-time to adapt program behavior, (3) *offline adaptation*, in which optimization plans are actually improved using actual execution, and (4) *recompilation*, where the optimization plans are regenerated.

BGML's model-based strategy can enhance conventional hybrid analysis by tabulating platform-specific and platform-independent information separately using the Skoll framework. In particular, Skoll does not incur the overhead of system monitoring since behavior does not change at run-time. New platform-specific information obtained can be fed back into the models to optimize QoS measures.

Generative Benchmarking Techniques. There have been a several initiatives that use generative techniques similar to BGML for generating test-cases and benchmarking for performance evaluation. The ForeSight [19] tool uses empirical benchmarking engine to capture QoS information for COTS based component middleware system. The results are used to build mathematical models to predict performance. This is achieved using a three pronged approach of (1) create a performance profile of how components in a middleware affect performance, (2) Construct a reasoning framework to understand architectural trade-offs, *i.e.*, know how different QoS attributes interact with one another and (3) Feed this configuration information into generic performance models to predict the configuration settings required to maximize performance.

The SoftArch/MTE [20] tool provides a framework for system architects to provide higher level abstraction of the system specifying system characteristics such as middleware, database technology, and client requests. The tool then generates a implementation of the system along with the performance tests that measure system characteristics. These results are then displayed back, *i.e.*, annotated in the high level diagrams, using tools such as Microsoft Excel. This allows architects to refine the design for system deployment.

BGML closely relates to the aforementioned approaches. However, both ForeSight and SoftArch 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 a feedback-driven approach [4], wherein the DCQA environment is used to empirically evaluate the QoS characteristics offline. This information can then be used to provide the modeler with accurate system information. Further, platform specific optimization techniques can also applied to maximize QoS characteristics of the system.

6 Concluding Remarks

Reusable software for performance-intensive systems increasingly has a multitude of configuration options and runs on a wide variety of hardware, compiler, network, OS, and middleware platforms. The distributed continuous QA techniques provided by Skoll play an important role in ensuring the correctness and quality of service (QoS) of performance-intensive software.

Skoll helps to ameliorate the variability in reusable software contexts by providing

- Domain-specific modeling languages that encapsulate the variability in software configuration options and interaction scenarios within GME modeling paradigms.
- An Intelligent Steering Agent (ISA) to map configuration options to clients that test the configuration and adaptation strategies to learn from the results obtained from clients and
- Model-based interpreters that generate benchmarking code and provide a framework to automate benchmark tests and facilitate the seamless integration of new tests.

Our experimental results showed how the modeling tools improve productivity by resolving the accidental complexity involved in writing error-prone source code for each benchmarking configuration. Section 4.2.3 showed that by using BGML, $\sim$90% of the code required to test and profile each combination of options can be generated, thereby significantly reducing the effort required by QA engineers to empirically evaluate impact of software variability on numerous QOS parameters. Section 4.2.4 showed how the results collected using Skoll can be used to populate a data repository that can be used by both application and middleware developers. The two use case presented in our feasibility study showed how our approach provides feedback to (1) application developers, *e.g.*, to tune configurations to maximize end-to-end QoS and (2) middleware developers, *e.g.*, to more readily identify configurations that should be optimized further.

In future work, we are applying DCQA processes to a grid of geographically decentralized computers composed of thousands of machines provided by users, developers, and organizations around the world. We are also integrating our DCQA technologies into the DRE software repository maintained by the ESCHER Institute (`www.escherinstitute.org`), which is a non-profit organization[4] established to preserve, maintain, and promote the technology transfer of government-sponsored R&D tools and frameworks in the DRE computing domain.

The ESCHER repository contains over 3 million lines of reusable, quality-controlled C++ and Java software tools and frameworks for the DRE system developer and user communities. Tools and frameworks enter the repository based on certain criteria (*e.g.*, maturity, reliability, interoperability, applicability to DRE system development) and use of quality development standards (*e.g.*, documentation, defect tracking, source code management, testing, and metrics). The ESCHER repository therefore provides an ideal environment for integrating and evaluating DCQA technologies in the context of performance-intensive software.

References

[1] 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.

[2] D. C. Schmidt and S. D. Huston, *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*. Boston: Addison-Wesley, 2002.

[3] 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.

[4] 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.

[5] 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.

[6] Object Management Group, *CORBA Components*, OMG Document formal/2002-06-65 ed., June 2002.

[7] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.6.1 ed., May 2002.

[8] A. Gokhale, D. C. Schmidt, B. Natarajan, and N. Wang, "Applying Model-Integrated Computing to Component Middleware and Enterprise Applications," *The Communications of the ACM Special Issue on Enterprise Components, Service and Business Rules*, vol. 45, Oct. 2002.

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

[4]The initial sponsors of the ESCHER Institute are General Motors, Raytheon, Boeing, NSF, and DARPA.

[10] 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.

[11] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. New York: Wiley & Sons, 2000.

[12] A. Porter and R. Selby, "Empirically Guided Software Development Using Metric-Based Classification Trees," *IEEE Software*, Mar. 1990.

[13] B. White and J. L. et al, "An Integrated Experimental Environment for Distributed Systems and Networks," in *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, (Boston, MA), pp. 255–270, USENIX Association, Dec. 2002.

[14] Robert Ricci and Chris Alfred and Jay Lepreau, "A Solver for the Network Testbed Mapping Problem," *SIGCOMM Computer Communications Review*, vol. 33, pp. 30–44, Apr. 2003.

[15] Kamen Yotov and Xiaoming Li and Gan Ren et.al, "A Comparison of Empirical and Model-driven Optimization," in *Proceedings of ACM SIGPLAN conference on Programming Language Design and Implementation*, June 2003.

[16] R. Zhang, C. Lu, T. Abdelzaher, and J. Stankovic, "Controlware: A Middleware Architecture for Feedback Control of Software Performance," in *Proceedings of the International Conference on Distributed Systems 2002*, July 2002.

[17] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son, "Feedback Control Real-Time Scheduling: Framework, Modeling, and Algorithms," *Real-Time Systems Journal*, vol. 23, pp. 85–126, July 2002.

[18] B. Childers, J. Davidson, and M. Soffa, "Continuous Compilation: A New Approach to Aggressive and Adaptive Code Transformation," in *Proceedings of the International Parallel and Distributed Processing Symposium*, Apr. 2003.

[19] 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.

[20] J. Grundy, Y. Cai, and A. Liu, "Generation of Distributed System Test-beds from High-level Software Architecture Description," in *16 ^{t}h International Conference on Automated Software Engineering, Linz Austria*, IEEE, Sept. 2001.