# A Model-based Distributed Continuous Quality Assurance Process to Enhance the Quality of Service of Evolving Performance-intensive Software Systems

Cemal Yilmaz[†], Arvind S. Krishna[‡], 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, such as that found in high-performance computing systems and distributed real-time and embedded systems, increasingly executes on a multitude of platforms and user contexts. To ensure that performance-intensive software meets its quality of service (QoS) requirements, it must often be fine-tuned to specific platforms/contexts by adjusting many (in some cases hundreds of) configuration options. Developers who write these types of systems must therefore try to ensure that their additions and modifications work across this large configuration space. In practice, however, time and resource constraints often force developers to assess performance on very few configurations and to extrapolate from these to the entire configuration space, which allows many performance bottlenecks and sources of QoS degradation to escape detection until systems are fielded.*

*To improve the assessment of performance across large configuration spaces, we present a model-based approach to developing and deploying a new distributed continuous quality assurance (DCQA) process. Our approach builds upon and extends the Skoll environment, which is developing and validating novel software QA processes and tools that leverage the extensive computing resources of worldwide user communities in a distributed, continuous manner to significantly and rapidly improve software quality. This paper describes how our new DCQA performance assessment process enables developers to run formally-designed screening experiments that isolate the most significant options. After that, exhaustive experiments (on the now much smaller configuration space) are conducted. We implemented this process using model-based software tools and executed it in the Skoll environment to demonstrate its effectiveness via two experiments on widely used QoS-enabled middleware. Our results show that model-based DCQA processes improves developer insight into the effect of system changes on performance at an acceptable cost.*

## 1. INTRODUCTION

While developing quality reusable software is hard, developing it for use in performance-intensive systems is even harder. Examples of performance-intensive software systems include high-performance scientific computing systems and distributed real-time and embedded (DRE) systems. The performance of these types of systems depends heavily on the underlying systems software, such as operating systems, middleware, and language processing tools. Quality of service (QoS)-enabled middleware [13, 14, 2] has become an essential abstraction layer for developing flexible and reusable performance-intensive software systems *e.g.*, its use has been cited [12] as a key factor in improving the development cost, time-to-market, and reuse of performance-intensive software systems, such as distributed scientific visualizations, 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 performance-intensive application software from lower-level infrastructure complexities, such as concurrency issues on heterogeneous platforms and error-prone network programming mechanisms. The defining characteristic of this type of middleware is its ability to specify and enforce the QoS requirements of performance-intensive systems end-to-end across heterogeneous combinations of OS, compiler, and hardware platforms. These capabilities result from the mechanisms that QoS-enabled middleware provide to provision performance-intensive systems by tuning and optimizing numerous configuration options, such as configuring policies for end-to-end priority propagation, the size of thread pools and priorities of pool threads, network connection caching policies, and type of transport protocol(s) to use, among many others.

The numerous configuration options needed to use QoS-enabled middleware effectively on heterogeneous combinations of OS, compiler and hardware platforms is exacerbating the limitations with conventional quality assurance (QA) techniques and motivating new QA processes. In particular, the explosion of the middleware configuration space is not adequately addressed by QA techniques that execute in-house on developer-generated workloads and regression tests [3]. Such processes rarely capture, predict, and recreate the run-time environment and usage patterns encountered in the field on all supported target platforms across all desired configuration options.

Other forces are also driving developers and organizations 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, and (3) evolution-oriented development requiring frequent software updates. While these new processes are beneficial in certain ways, they also exacerbate QA challenges, *e.g.*, coping with frequent software changes, remote developer coordination, and exploding software integration and configuration spaces.

Submitted to FSE'04 for Review

To address these challenges in the context of performance-intensive software, we are developing and integrating the following techniques:

- **Distributed continuous quality assurance (DCQA) techniques**, which are designed to improve software quality and performance iteratively, opportunistically, efficiently, and continuously in multiple, geographically distributed locations [8]. In prior work, we have developed a prototype DCQA environment called *Skoll* (`www.cs.umd.edu/projects/skoll`) that provides a framework for executing QA tasks continuously across a grid of computing distributed around the world.

- **Model-based software development techniques**, which help to minimize the cost of QA activities by capturing the customizability of middleware within models and automatically generating configuration files from these higher level models [4]. In prior work, we have developed prototype model-based software tools including (1) the Options Configuration Modeling language (OCML) [15] that allows developers to model middleware configuration options as high-level models and (2) model-driven benchmarking tools [6] that allow developers to compose benchmarking experiments that observe QoS behavior by mixing and matching middleware configurations.

This paper extends our prior work by focusing on how we integrated our modeling tools with the Skoll environment to support more effective DCQA processes for performance-intensive software. In particular, we describe model-based enhancements to Skoll that enable it to rapidly identify a small subset of highly influential performance-related configuration options and systematically explore that subset of options empirically to estimate system performance across the entire configuration space. We then present results of using this DCQA process on ACE+TAO (`deuce.doc.wustl.edu/Download.html`), which are widely-used production QoS-enabled middleware frameworks. Our results show that (1) model-based DCQA tools and processes can correctly identify a key subset of options that affect system performance significantly and (2) monitoring only these selected options improves developer insight into the effect of system changes on performance at an acceptable cost.

The remainder of the paper is organized as follows: Section 2 outlines the QA challenges associated with improving the quality of performance-intensive software and describes how we are resolving these challenges by integrating model-based systems engineering techniques with DCQA processes; Section 3 reports the results of experiments using this model-based DCQA process on the ACE+TAO middleware; Section related compares our research with related work; and Section 5 presents concluding remarks and outlines future work.

## 2. ADDRESSING QA CHALLENGES FOR PERFORMANCE-INTENSIVE SOFTWARE SYSTEMS

This section describes key QA challenges faced by developers of performance-intensive software and describes how DCQA environments and model-based software development techniques can help to resolve these challenges.

### 2.1 Challenge 1: Configuration Space Explosion in Performance-intensive Software Systems

**Context.** Performance-intensive software often provide fine-grained knobs to tune QoS behavior so it can be optimized for particular run-time contexts and application requirements. For example, high-performance web servers (*e.g.,* Apache), object request brokers (*e.g.,* TAO), and databases (*e.g.,* Oracle) have hundreds of options and configuration parameters. General-purpose, one-size-fits-all solutions often have unacceptable QoS for performance-intensive software systems.

**Problem.** To support customizations demanded by users, performance-intensive software must run on many hardware and OS platforms and typically have many options to configure the system at compile- and/or run-time. Highly configurable performance-intensive software can therefore yield an explosion of the *software configuration space*. While the flexibility of many options and configuration parameters promotes customization, it also creates many potential system configurations, each of which deserves extensive QA. As software configuration spaces increase in size and software development resources decrease, it becomes infeasible to handle all QA activities in-house since developers often lack all the hardware, OS, and compiler platforms on which their reusable software artifacts will run.

**Solution approach $\rightarrow$ the Skoll DCQA environment.** To address the QA challenges caused by the explosion of the software configuration space and the limitations of in-house QA processes, we have developed the **Skoll** environment to prototype and evaluate tools necessary to perform "around-the-world, around-the-clock" DCQA processes. Our feedback-driven Skoll environment includes languages for modeling system configurations and their constraints, algorithms for scheduling and remotely executing tasks, and analysis techniques for characterizing faults. Skoll 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 DCQA process.

To support DCQA processes we have developed the following components and services for use by Skoll QA process designers (a comprehensive discussion appears in [8]):

•**Configuration space model.** The cornerstone of Skoll is its formal model of a QA process' 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 aiding in analyzing and interpreting results.

•**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. 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 choose the next configuration such that process goals (*e.g.,* evaluating configurations in proportion to known usage distributions) will be met. After a valid configuration is chosen, the ISA packages the corresponding QA subtask implementation into a *job configuration*, which consists of the code artifacts, configuration parameters, build instructions, and QA-specific code (*e.g.,* regression/performance tests) associated with a software project.

•**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.

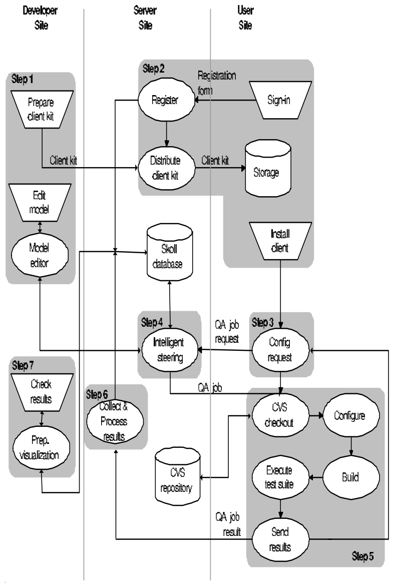


**Figure 1: An Example Skoll DCQA Process**

Skoll uses the components described above to perform DCQA processes as shown in Figure 1 and described below.
**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 a web-based registration process. 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 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* (see `www.dre.vanderbilt.edu/scoreboard` for an example) that summarizes subtask results and the current state of the overall process.

•**Evaluating Skoll.** The Skoll environment supports DCQA tasks on a range of hardware/OS/compiler platforms. The adaptation strategies in Skoll allow QA processes to accurately pinpoint the configuration space leading to compilation failures across varied platforms. This capability provides feedback to developers by helping them identify how their changes impact software quality. In the initial Skoll [8] prototype, however, QA tasks were limited to testing the *functional correctness* of software, *i.e.*, clean compilation and passing regression tests.

## 2.2 Challenge 2: Evaluating the QoS of Performance-intensive Software Systems

**Context.** Performance-intensive software systems run on a multitude of hardware/OS/compiler platforms and provide fine grained knobs to tune QoS behavior.

**Problem.** To evaluate key QoS characteristics of performance-intensive software, QA engineers today often handcraft individual QA tasks (*e.g.*, benchmarking experiments) by writing (1) interface definitions that model the data exchange format between clients and servers, (2) component implementations in the target language, *e.g.*, C, C++, Java, Ada etc., (3) client test applications that measure key performance metrics, such as round-trip request latency, jitter, and throughput, and (4) scaffolding code, such as scripts needed to startup daemons, initialize the software infrastructure and applications, run experiments, generate results, and tear down the experiment. Manually implementing these steps is tedious and error-prone since each step may be repeated many times for every QA experiment. Further, in a handcrafted approach, QA engineers visualize experiments via application source code, which provides an excessively low level of abstraction.

**Solution approach → the Benchmark Generation Modeling Language (BGML).** BGML [6] is a model-driven benchmarking tool that allows component middleware QA engineers to (1) visually model interaction scenarios between configuration options and system components using domain-specific building blocks, *i.e.*, capture software variability in higher-level models rather than in lower-level source code, (2) automate benchmarking code generation and reuse QA task code across 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, and (4) enable evaluation of multiple performance metrics, such as throughput, latency, jitter, and other QoS criteria. Below, we describe the key elements of BGML that are relevant for this paper.

•**Syntactic elements in BGML.** The BGML meta-model defines a modeling paradigm that allows the specification of QoS configurations, parameters, and constraints. To allow variation in data exchange, BGML can model a subset of OMG's IDL interface. To quantify various performance characteristics, it also models QoS metrics such as latency, throughput and jitter. The BGML modeling language consists of the following building blocks based on a QoS-enabled implementation of the CORBA Component Model(CCM) middleware:[1]

- **Object**, which in CORBA is an implementation entity that consists of an identity, an interface, and an implementation.
- **Component**, which in CCM are implementation entities that collaborate with each other via *ports*, which are explained below.
- **Server**, which export CORBA objects and/or components to clients.
- **Client**, which is the program entity that invokes an operation on an object or component implementation.

[1]We focus on CORBA and CCM in our work on BGML since it is currently the most suitable standards-compliant QoS-enabled middleware for performance-intensive software. As the QoS capabilities of other middleware platforms mature we plan to support them, as well.

The motivation for developing these building blocks is to enable QA engineers to model the following four possible interaction scenarios in the context of CORBA and CCM: (1) a CORBA server interacting with a CORBA client, (2) a CORBA server interacting with a CCM component (playing the role of the client) , (3) a CCM component (playing role of server) interacting with a CORBA client, and (4) a CCM component interacting with another CCM component (playing both client and server roles). CCM also provides component assemblies that are hierarchical collection of interacting components. In BGML, an assembly is considered as a virtual entity that is realized by explicitly modeling the nesting of components.

BGML defines *ports*, which are well-defined interaction points through with CCM components in BGML communicate with each other. The following ports are supported in BGML:

- **Facets**, which define an interface that provides point-to-point method invocations from other components. Components and servers provide facets for other components/clients to communicate with.
- **Receptacles**, which define an interface that accepts point-to-point method invocations from other components.
- **Event sources**, which define a mechanism for exchanging typed messages with one or more components.
- **Event sinks**, which are recipients of messages generated from event sources.

The use of ports in BGML allows QA engineers to model point-to-point (via facet receptacle interconnection) and point-to-multi point (via event-source event-sink interaction) communication.

In BGML, the data exchanged between the between different interaction entities is represented using CORBA's Interface Definition Language (IDL), including:

- **Interfaces**, which represent the type of data exchanged between communicating entities. In BGML, each interface contains a set of operations.
- **Operations**, which have the following attributes associated with it: (1) *argument type*, specifies the type of input, (2) *operation name*, name for this operation and (3) *return type*, specifies type for return argument.
- **EventTypes**, which represent the typed messages that event sources and sinks exchange. Each EventType has an operation name and identifier associated with it.

Allowing QA engineers to model IDL data types facilitates synthesis of IDL files from the models directly, freeing them from the need to (1) handcraft IDL files and (2) understand the relatively low level syntax of the IDL language.

In addition to building blocks required to model the experiment, BGML provides constructs to observe QoS metrics in the experiment modeled. BGML currently supports associating QoS parameters with EventTypes and on a per operation basis with IDL interfaces. The metrics that can be gathered via BGML include:

- **Latency**, which for a given operation/event computes the mean roundtrip time at the client.
- **Throughput**, which for a given operation/event computes the mean number of invocations completed per second at the client.
- **Jitter**, which for a given operation/event computes the variance in the latency measures (jitter is always associated with a latency or throughput metric).

Each metric described above has two attributes: (1) *warmup iterations*, which indicates the number of iterations the operation under test will be invoked before start of actual measurement and (2) *sample space*, which indicates the number of sample points that will be collected to determine each metric value. Latency, throughput, and jitter metrics represent commonly used rubrics for comparing the QoS of performance-intensive software systems. By providing these building blocks, QA engineers can visually represent the metric they want to observe in the experiment and synthesize the code from the models directly.

**•Semantic elements in BGML.** BGML uses the following three modeling abstractions to help QA engineers compose benchmarking experiments:

- **Containment relation.** Every component and server can export facets to communicate with clients. Similarly, components can also provide event-sources for multi-point communication. Clients and components playing role of a client can interact with the exported facets/event sources via receptacle and event sinks. BGML uses the containment relation to depict *ports*.
- **Association relation.** Facets provide or realize an IDL interface. In BGML, this is modeled using an association relation, wherein each facet element is connected to an IDL interface. Similarly, event sources publish EventTypes that are modeled using this relation. A Facet/Event-Source in one component communicates with a Receptacle/Event-Sink in another component. This communication is modeled by associating, *i.e.,* connecting the facet with the receptacle. Similarly, QoS metrics are associated with IDL operations using association relationship.
- **Reference association.** All dependency information in BGML is modeled using references. For example, a receptacle communicating with a facet is dependent on the interface exported by the facet. Associating any other interface will be flagged as a typesystem violation by BGML.

**•BGML generative tools.** The BGML model interpreter parses the model and synthesizes the code required to benchmark the modeled configuration. The BGML model interpreter includes 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 aspects* in which end-user specify the metrics to be collected during the experiment.
- **IDL engine**, which generates the IDL files forming the contract between the client and the server. These files are compiled by the CORBA and/or CCM IDL compiler to generate the "glue-code" needed by the benchmarking infrastructure. This engine relies on the *configuration aspect* in which end-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 in which component interactions are specified.

Each of entities described above have attributes, *e.g.*, the number of threads in a component, that are specified in the building blocks as attributes.

**•Evaluating BGML.** BGML helps improve the productivity of QA engineers by resolving the accidental complexity of handcrafting tedious and error-prone source code. For each experiment, the BGML tool generates close to 90% [6] of the required IDL, configuration and implementation files. Since it is a generative tool, BGML only reduces the cost of constructing benchmarking experiment for various configurations. In its original form, however, it did not provide information on the consequences of configuration option on QoS behavior on varied hardware/OS/compiler platforms.

## 2.3 Challenge 3: Assessing QoS of Performance-intensive Software Across Large Configuration Spaces

**Context.** As developers create and modify their performance-intensive software systems, they often conduct benchmarking experiments to identify when changes negatively affect performance. Due to time and resource constraints, however, these experiments are typically executed on a very small number of default configurations. While this provides some data, it leaves substantial portions of the entire configuration space unevaluated, allowing performance problems to escape detection until the software is fielded. To close this gap, developers of performance-intensive software could use the BGML modeling language together with the Skoll DCQA environment to gather a much wider sampling of performance data.

**Problem.** Although Skoll and BGML provide an infrastructure for performing large-scale QA, the configuration spaces of performance-intensive software systems are often so large that brute force processes are still infeasible. For example, the ACE+TAO systems have $\sim$500 configuration options, with over $2^{500}$ potential combinations. To be effective for highly configurable performance-intensive software systems, therefore, DCQA processes must generally include some type of *adaptation strategy* to efficiently navigate large configuration spaces.

**Solution $\rightarrow$ Applying Experimental Design Theory for Configuration Space Reduction.** In the remainder of this section we present a new DCQA process, called *main effects screening*. The aim of this process is to efficiently improve the visibility of developers into the QoS of performance-intensive software across large configuration spaces. As described below, we have integrated BGML and Skoll to instantiate and execute the process.

As software systems change, developers often run regression tests to detect unintended functional side effects. In addition to functionality, developers of performance-intensive systems must also be wary of unintended effects on QoS. They will therefore periodically run performance benchmarking tests to detect such problems. As described in Section 1, however, QA efforts can be confounded in highly configurable systems due to the enormous configuration space. Moreover, time and resource constraints (and high change frequencies) severely limit the number of configurations that can be examined. For example, in our earlier experience with ACE+TAO [8], only a small number of default configurations are benchmarked routinely. As a result, ACE+TAO developers get a *very* limited view of their middleware's QoS and problems not readily seen in the single default configuration can escape detection until system based on ACE+TAO are fielded.

To address these problems, we developed the *main effects screening* DCQA process, which is performed in the following two phases:

- **Phase 1.** We execute a large-scale, formally-designed experiment across the Skoll grid. As part of this experiment, we run benchmarks on a wide-ranging, but sparsely distributed, set of configurations. These configurations are selected using a class of experimental designs called *screening designs* [17], which are highly economical and can reveal individual options that significantly affect performance (colloquially, these are referred to as first-order or "main" effects). These designs are economical since they are not intended to detect high-order interaction effects (*i.e.*, significant interactions between, *e.g.*, five different options). The choice of significance level at which to separate significant from non-significant options can be set by QA process engineers.
- **Phase 2.** Once we have identified the main effects, we only focus on them, effectively reducing the configuration space to just these few options. The process continues executing using only in-house resources. Each time the system changes, we exhaustively benchmark all combinations of the first-order options, while using default (or random) settings for the remaining options. Our intent is that by focusing only on the first-order options, we can greatly reduce the configuration space, while at the same time capture a much more complete picture of the system's QoS. This data is plotted and maintained on the system's build scoreboard (*e.g.*, `www.dre.vanderbilt.edu/Stats`). Since the main effects might change over time, the process can be restarted periodically to recalibrate the main effects options.

Although the details of computing a screening design are beyond the scope of this paper, we give a simple example to help introduce key terminology and concepts. We will assume our example system has 5 binary configuration options A, B, C, D, and E. The configuration space thus has 32 ($2^5$) configurations and the example screening design will involve only 8 configurations. We begin by creating a $2^{(3)}$ full-factorial design for options A, B, and C. This starting point ensures that the main effects will not be confounded with each other, which is referred to as a resolution III design.

Since all the options are binary, we encode their settings as either (-) for the first setting or (+) for the second setting. Non-binary options can be handled as well, but the design is more complicated. At this point, our initial design is as follows:

```
A - + - + - + - +
B - - + + - - + +
C - - - - + + + +
```

This design includes only three options. To compute the settings of the fourth option, we select a design generator, indicated 4=12, to generate the settings of the fourth option. The number on the left side of the design generator equation gives the row that will be generated, while the right side indicates *how* that row will be generated. In this particular case, therefore, the fourth row (for D) will be generated by multiplying rows 1 and 2 *i.e.*, each entry in the row for D is the same as the product of the corresponding entries in the rows for A and B. We generate row E similarly, using design generator 5=23 and the final design matrix is as follows:

```
A - + - + - + - +
B - - + + - - + +
C - - - - + + + +
D + - - + + - - +
E + + - - - - + +
```

To analyze the data, we must calculate the main effect of each option. In our simple example, the main effect of option A, ME(A), is ME(A) = z(A-) - z(A+), where z(A-) and z(A+) are the mean values of the observed variable over all runs when A is (-) and when A is (+), respectively.

## 2.4 Putting It All Together

Figure 2 presents an overview of how we have integrated BGML with the existing Skoll prototype to support the *main effects screening* DCQA process described in Section 2.3. Below we describe the enhanced Skoll QA process, referencing the steps shown in the figure.

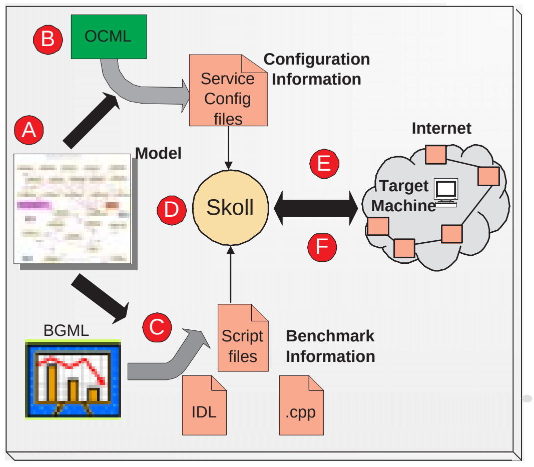


**Figure 2: Skoll QA Process View with BGML Enhancements**

**A** A QA engineer defines a test configuration using BGML models. The necessary experimentation details are captured in the models, *e.g.*, the configuration options examined during main effects screening, the IDL interface exchanged between the client and the server, and the benchmark metric performed by the experiment.
**B & C** The QA engineer then uses BGML to interpret the model. The paradigm interpreter parses the modeled CORBA middleware configuration options and generates the required configuration files to configure the underlying CORBA middleware. 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 also generate internal files used by Skoll.
**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** Clients execute steps in the main effects screening experiment and return 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.

## 3. FEASIBILITY STUDY

This section describes a feasibility study that assesses the implementation cost and the effectiveness of the main effects screening process on a large performance-intensive software system.

### 3.1 Hypotheses

In this study, we will explore the following high-level hypotheses:

1. Our proposed DCQA process can be easily instantiated for a specific system using the model-based Skoll environment described in Section 2.
2. The process's initial screening phase identifies a small subset of options whose effect on performance is significant.
3. The process's second phase produces performance data that (1) is representative of the system's QoS across the entire configuration space and (2) is more representative of the overall performance than that produced by observing a small number of randomly selected configurations.

### 3.2 Experimental Process

We use the following experimental process to evaluate our hypotheses:

**Step 1:** Choose a middleware software system that has a representative (*i.e.*, large) configuration space.
**Step 2:** Choose an application scenario and performance criteria.
**Step 3:** Use the BGML tools (Section 2.2) to design an experiment for the application scenario.
**Step 4:** Apply the main effects screening process, *i.e.*, use the screening step to identify key options and use the second step to obtain performance variation by exhaustively exploring the key options.
**Step 5:** Explore the entire configuration space and obtain performance variation. Explore a few random configuration settings and obtain their performance variation.

Below, we compare the performance variation exposed by the main effects screening process to that obtained by examining the entire configuration space and random space.

#### *3.2.1 Step 1: Subject Applications*

We used ACE v5.4 + TAO v1.4 + CIAO v0.4 for this study. CIAO is a QoS-enabled implementation of CCM being developed at Washington University, St. Louis and Vanderbilt University to extend TAO [5] so it supports components, which simplifies the development of DRE applications by enabling developers to declaratively provision QoS policies end-to-end when assembling a system. TAO is an open-source, high-performance, highly configurable Real-time CORBA ORB that implements key patterns [11] to meet the demanding QoS requirements of DRE systems. TAO is developed atop lower-level middleware called ACE that implements core concurrency and distribution patterns [11] for communication software. ACE provides reusable C++ wrapper facades and framework components that support the QoS requirements of high-performance, real-time applications. ACE and TAO run on a wide range of OS platforms, including Windows, most versions of UNIX, and real-time operating systems such as Sun/Chorus ClassiX, LynxOS, and VxWorks.

#### *3.2.2 Step 2: Application Scenario*

Due to recent changes made to the message queueing strategy, the developers of ACE+TAO+CIAO are concerned with measuring two performance criteria: (1) the latency for each request, and (2) total message throughput (events/second) between the ACE+TAO+CIAO client and server. For this version of ACE+TAO+CIAO, the developers identified 14 run-time options they felt affected latency and throughput. Each option is binary as shown in Table 1 and the entire configuration space is $2^{14} = 16,384$. Note that since we compare our results to exhaustive configuration space exploration, this number of options was ideal for our study. In practice, however, it would be much larger.

#### *3.2.3 Step 3: BGML Tool*

Figure 3 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 the figure, the following steps were performed:

• Using the BGML modeling paradigm QA engineers composed the experiment.

• In the experiment modeled, QA engineers associate the QoS characteristic (in this case roundtrip latency and throughput) that will be captured in the experiment.

• Using the experiment modeled, BGML interpreters generate the benchmarking code required to set-up, run and tear-down the

| Option Index | Option Name | Option Settings |
|---|---|---|
| o1 | ORBReactorThreadQueue | {FIFO, LIFO} |
| o2 | ORBClientConnectionHandler | {RW, MT} |
| o3 | ORBReactorMaskSignals | {0, 1} |
| o4 | ORBConnectionPurgingStrategy | {LRU, LFU} |
| o5 | ORBConnectionCachePurgePercentage | {10, 40} |
| o6 | ORBConnectionCacheLock | {thread, null} |
| o7 | ORBCorbaObjectLock | {thread, null} |
| o8 | ORBObjectKeyTableLock | {thread, null} |
| o9 | ORBInputCDRAllocator | {thread, null} |
| o10 | ORBConcurrency | {reactive, thread-per-connection} |
| o11 | ORBActiveObjectMapSize | {32, 128} |
| o12 | ORBUseridPolicyDemuxStrategy | {linear, dynamic} |
| o13 | ORBSystemidPolicyDemuxStrategy | {linear, dynamic} |
| o14 | ORBUniqueidPolicyReverseDemuxStrategy | {linear, dynamic} |

**Table 1: The Options and Their Settings**

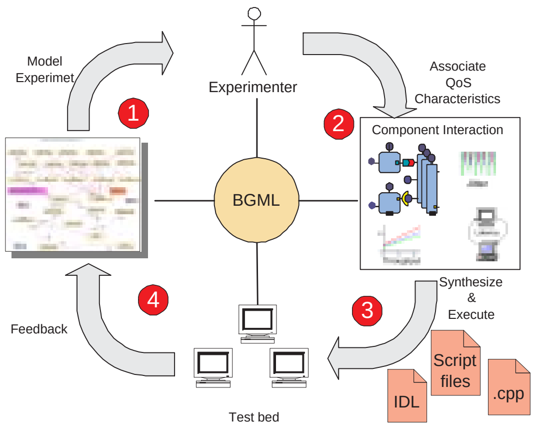


**Figure 3: BGML Use Case Scenario**

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 and QoS characteristics are measured. The execution was done both in Steps 4 and 5 described in Sections 3.2.4 and 3.2.5 respectively.

### *3.2.4 Step 4: Application of the Main Effects Screening Process*

In this step, our main concern is to find the first-order effects of configuration options; we are not interested in higher-order (interaction) effects. We designed the screening experiment using a fractional factorial design as explained below:

Based on the information we obtained from the ACE+TAO+CIAO developers on the $2^{14}$ configuration space, we decided to alias no main factors with effects involving fewer than three factors. This decision dictated a resolution IV design, in which no effect involving $i$ factors is aliased with effects involving less than $4 - i$ factors. For example, in such a design some of the main effects are aliased with three-factor interactions, but not with other main effects or two-factor interactions.

With resolution IV and run sequence efficiency requirements in mind, we designed a $2^{14-9}_{IV}$ screening experiment given in Table 2. This screening design examines 14 factors in $2^5 = 32$ runs, which is a $2^9$ fraction of the exhaustive design.

To create the final design, we started with a $2^5$ all options design (shown in the first 5 columns in Table 2. We computed the remaining 9 columns by using the following design generators: o6=123, o7=124, o8=134, o9=234, o10=125, o11=135, o12=235, o13=145, o14=245.

### *3.2.5 Step 5: Generating Variation for the Entire Configuration Space*

We also obtained the performance variation for the entire configuration space, *i.e.*, 16,384 configurations. It then took 48 hours to run all the benchmarking experiments.

## 3.3 Results

The results of the screening phase are summarized in Table 3. Looking across latency measures of "All Options" (the exhaustive design) in Table 3, we see that only options o2 and o10 have a significant effect on the latency. This result was surprising to ACE+TAO+CIAO developers since they thought that all 14 run-time options would contribute substantially to latency. The same result appears for latency variation and for throughput. Therefore, only options 02 and o10 show a significant effect on performance.

Looking instead at the data labeled "screening" (*i.e.*, just the 32 runs selected by the screening design), we draw the exact same conclusion. That is the screening design gave us the same information at a fraction of the cost. Note that the time needed to run the 32 configurations was about 6 minutes.

The second phase of the process used the information that o2 and o10 are important options to generate all possible (*i.e.*, 4) configurations for the binary values of o2 and o10. Default values were assigned to the remaining options. The latency and throughput were measured for these 4 configurations.

The results of the second phase are summarized in Figure 4. These box plots show that the distributions obtained from the screening experiments are very similar to the ones obtained from the exhaustive runs. In contrast, the distributions for random configurations (4 chosen at random) were very different. Table 4 provides a detailed analysis of these box-plots. The table shows the number

| Metric | Screening | Random |
|---|---|---|
| latency | 77% | 46% |
| latency variance | 64% | 30% |
| throughput | 75% | 55% |

**Table 4: Range of Performance Metrics Covered by Screening and Random Design**

of observations for each performance metric in the entire configuration space that fall into the range of the observations obtained from screening and random designs. As this table indicates, the screening design covered a large portion of the system's range of

| Screening Design | | | | | | | | | | | | | | Observations | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| o1 | o2 | o3 | o4 | o5 | o6 | o7 | o8 | o9 | o10 | o11 | o12 | o13 | o14 | latency | $s^2$ | $ln(s^2)$ | throughput |
| - | - | - | - | - | - | - | - | - | - | - | - | - | - | 115.00 | 323.76 | 5.78 | 8572.64 |
| + | - | - | - | - | + | + | + | - | + | + | - | + | - | 95.00 | 181.27 | 5.20 | 10451.43 |
| - | + | - | - | - | + | + | - | + | + | - | + | - | + | 119.00 | 441.42 | 6.09 | 8318.61 |
| + | + | - | - | - | - | - | + | + | - | + | + | + | + | 136.00 | 555.57 | 6.32 | 7291.35 |
| - | - | + | - | - | + | - | + | + | - | + | + | - | - | 107.00 | 403.43 | 6.00 | 9233.01 |
| + | - | + | - | - | - | + | - | + | + | - | + | + | - | 81.00 | 35.87 | 3.58 | 12121.94 |
| - | + | + | - | - | - | + | + | - | + | + | - | - | + | 121.00 | 507.76 | 6.23 | 8154.60 |
| + | + | + | - | - | + | - | - | - | - | - | - | + | + | 118.00 | 365.04 | 5.90 | 8393.64 |
| - | - | - | + | - | - | + | + | + | - | - | - | + | + | 92.00 | 138.38 | 4.93 | 10701.40 |
| + | - | - | + | - | + | - | - | + | + | + | - | - | + | 80.00 | 16.95 | 2.83 | 12284.19 |
| - | + | - | + | - | + | - | + | - | + | - | + | + | - | 103.00 | 144.03 | 4.97 | 9605.94 |
| + | + | - | + | - | - | + | - | - | - | + | + | - | - | 134.00 | 639.06 | 6.46 | 7402.15 |
| - | - | + | + | - | + | + | - | - | - | + | + | + | + | 100.00 | 333.62 | 5.81 | 9918.66 |
| + | - | + | + | - | - | - | + | - | + | - | + | - | + | 84.00 | 83.10 | 4.42 | 11693.53 |
| - | + | + | + | - | - | - | - | + | + | + | - | + | - | 112.00 | 354.25 | 5.87 | 8841.91 |
| + | + | + | + | - | + | + | + | + | - | - | - | - | - | 136.00 | 685.40 | 6.53 | 7303.03 |
| - | - | - | - | + | - | - | - | - | + | + | + | + | + | 89.00 | 164.02 | 5.10 | 11041.34 |
| + | - | - | - | + | + | + | + | - | - | - | + | - | + | 92.00 | 135.64 | 4.91 | 10699.18 |
| - | + | - | - | + | + | + | - | + | - | + | - | + | - | 124.00 | 518.01 | 6.25 | 7978.12 |
| + | + | - | - | + | - | - | + | + | + | - | - | - | - | 113.00 | 314.19 | 5.75 | 8766.06 |
| - | - | + | - | + | + | - | + | + | + | - | - | + | + | 88.00 | 148.41 | 5.00 | 11192.87 |
| + | - | + | - | + | - | + | - | + | - | + | - | - | + | 98.00 | 275.89 | 5.62 | 10063.70 |
| - | + | + | - | + | - | + | + | - | - | - | + | + | - | 120.00 | 383.75 | 5.95 | 8234.28 |
| + | + | + | - | + | + | - | - | - | + | + | + | - | - | 115.00 | 365.04 | 5.90 | 8573.15 |
| - | - | - | + | + | - | + | + | + | + | + | + | - | - | 80.00 | 15.96 | 2.77 | 12373.87 |
| + | - | - | + | + | + | - | - | + | - | - | + | + | - | 91.00 | 127.74 | 4.85 | 10838.35 |
| - | + | - | + | + | + | - | + | - | - | + | - | - | + | 119.00 | 441.42 | 6.09 | 8297.93 |
| + | + | - | + | + | - | + | - | - | + | - | - | + | + | 106.00 | 172.43 | 5.15 | 9367.18 |
| - | - | + | + | + | + | + | - | - | + | - | - | - | - | 98.00 | 162.39 | 5.09 | 10059.06 |
| + | - | + | + | + | - | - | + | - | - | + | - | + | - | 106.00 | 376.15 | 5.93 | 9335.74 |
| - | + | + | + | + | - | - | - | + | - | - | + | - | + | 112.00 | 83.93 | 4.43 | 8836.80 |
| + | + | + | + | + | + | + | + | + | + | + | + | + | + | 116.00 | 437.03 | 6.08 | 8542.13 |

**Table 2: $2_{IV}^{14-9}$ Screening Experiment Design Matrix and Observed Values**

performance and covered more of the performance range than the random design did.

## 3.4 Discussion

From this study, we observed that over the course of several hours, we used BGML to generate scaffolding code that ran all the benchmarks automatically on all configurations of the subject applications and collected results. In the past, we have written this code by hand and manually sent it to the Skoll client sites. Many times, the code contained editing bugs, causing tests to fail unexpectedly, requiring manual intervention and leading to delays in the experiments.

As a result of this work, the ACE+TAO+CIAO developers learned that only 2 of the 14 run-time options by themselves had a significant affect on performance. These two options were identified for them automatically using the new main effects screening process. We learned that examining only 4 configurations exposed about 75% of the entire range of the system's performance across all 16,000+ valid configurations. Given the small number of important options, ACE+TAO+CIAO developers can incorporate the benchmark execution on the 4 configurations whenever they change the code. Incorporating this into their regular build cycle provides them with rapid feedback on the effects of their changes (the benchmarks run on these 4 configurations in just a few seconds, so compile time is the main bottleneck).

The ACE+TAO+CIAO developers can also use the main effects screening process to track their performance distributions over time. For example, as changes are made to the code, the screened options can be recalibrated periodically by executing phase one of our process, which not only gives developers a more accurate view of their software's performance, it also provides a valuable defect detection aid. If the screened options change unexpectedly when recalibrated, the developers can re-examine the software to identify possible problems.

## 4. RELATED WORK

This section compares our work on model-driven performance evaluation techniques in Skoll and BGML with other related research efforts including large-scale testbed environments that provide a platform to conduct experiments using heterogeneous hardware, OS, and compiler platforms, feedback-based optimization techniques, and generative techniques for synthesizing benchmarks.

*Large-scale benchmarking testbeds.* EMULab [16] 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 [9] to ∼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 is not limited by resources of a single testbed, but instead can leverage the large amounts of 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.* Feedback-driven techniques involve using feedback control loops [7] to adapt adapt QoS measures. Our approach in BGML and Skoll uses screening experiments to determine first order configuration parameters

Latency

| Opt. | Effect |
|---|---|
| o2 | -25.634 |
| o10 | 13.183 |
| o3 | -1.495 |
| o6 | 1.077 |
| o12 | -0.367 |
| o9 | 0.361 |
| o11 | 0.327 |
| o13 | -0.241 |
| o1 | -0.171 |
| o4 | 0.155 |
| o8 | 0.142 |
| o14 | 0.098 |
| o5 | -0.018 |
| o7 | -0.005 |

All Options

Latency

| Opt. | Effect |
|---|---|
| o2 | -25.500 |
| o10 | 12.500 |
| o5 | 4.125 |
| o11 | -4.00 |
| o4 | 3.875 |
| o14 | 3.750 |
| o13 | 2.875 |
| o12 | 2.625 |
| o9 | 1.875 |
| o7 | -1.500 |
| o3 | -1.500 |
| o8 | -1.00 |
| o6 | -0.125 |
| o1 | -0.125 |

Screening

Latency Var.

| Opt. | Effect |
|---|---|
| o10 | 0.652 |
| o2 | -0.584 |
| o3 | -0.101 |
| o14 | -0.044 |
| o9 | 0.028 |
| o8 | 0.023 |
| o12 | -0.017 |
| o6 | 0.016 |
| o1 | -0.013 |
| o11 | 0.013 |
| o4 | 0.007 |
| o7 | -0.004 |
| o13 | -0.002 |
| o5 | 0.001 |

All Options

Latency Var.

| Opt. | Effect |
|---|---|
| o2 | -1.007 |
| o10 | 0.731 |
| o4 | 0.460 |
| o9 | 0.373 |
| o11 | -0.319 |
| o3 | -0.305 |
| o12 | 0.282 |
| o6 | -0.201 |
| o8 | -0.149 |
| o5 | 0.127 |
| o13 | -0.125 |
| o14 | 0.124 |
| o7 | -0.094 |
| o1 | 0.058 |

Screening

Throughput

| Opt. | Effect |
|---|---|
| o2 | 2201.744 |
| o10 | -1138.725 |
| o3 | 110.168 |
| o6 | -88.326 |
| o9 | -38.973 |
| o12 | 27.472 |
| o11 | -16.630 |
| o13 | 14.270 |
| o1 | 14.127 |
| o4 | -12.949 |
| o8 | -12.003 |
| o14 | -5.156 |
| o7 | 2.766 |
| o5 | -0.413 |

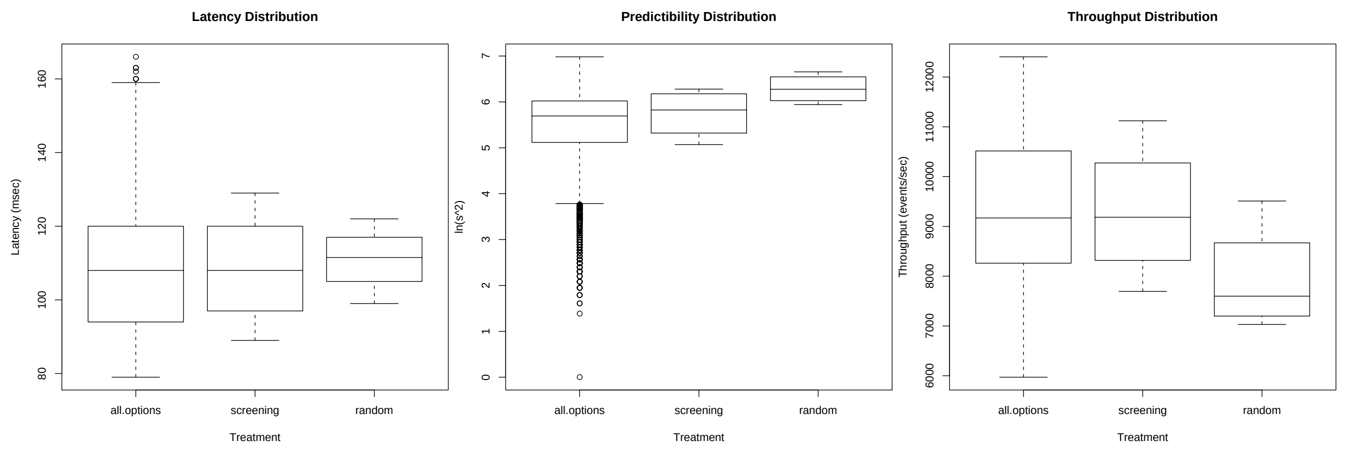
All Options

Throughput

| Opt. | Effect |
|---|---|
| o2 | 2292.127 |
| o10 | -1142.989 |
| o4 | -394.747 |
| o14 | -319.152 |
| o12 | -310.049 |
| o11 | 307.577 |
| o9 | -305.431 |
| o5 | -244.483 |
| o3 | 218.231 |
| o13 | -201.548 |
| o1 | -110.357 |
| o6 | 69.324 |
| o7 | 69.319 |
| o8 | 45.943 |

Screening

**Table 3: Main Effect Estimations of Latency, Variance of Latency, and Throughput for All Options and Screening Designs**



Performance (latency) Predictibility (latency) Performance (throughput)

**Figure 4: Performance and Predictibility Distributions**

impacting performance. These results are then fed-back into performance models to provide information on the consequences of mixing and matching configuration options at model building time.

The continuous compilation strategy [1] combines aspects of offline (use program analysis to improve compiler-generated code) and online analysis (where feedback control is used to dynamically adapt QoS measures). This strategy 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 techniques for synthesizing benchmarks.* There have been a several initiatives that use generative techniques similar to BGML for generating test-cases and benchmarking for performance evaluation. The MODEST [10] tool provides a generative approach for producing (1) test cases, *i.e.*, test-code that is used to test the system and (2) test-harness, *i.e.*, the scaffolding code required for test setup and tear down. In this system, test cases are generated in parallel with the actual system to provide users with the system and the test-code to reduce maintenance costs.

Our BGML modeling tool focuses on empirical performance evaluation rather than just test cases generation since generating functional test cases may be amenable only to systems that can capture all inputs to the system. The MODEST approach captures this information via a domain specification supplied as XML input. In MODEST, all artifacts conform to the same architecture, with variability only in the domain specification given by the user. This assumption rarely holds, however, for performance-intensive software, which often runs on heterogeneous combinations of OS, compiler and hardware platforms. In contrast, our model-based DCQA approach to benchmarking allows QA engineers to fully characterize the inputs to the system and observe QoS behavior and variations across a wide range of platforms.

## 5. CONCLUDING REMARKS

This paper described a model-based distributed continuous quality assurance (DCQA) process called main effects screening. The process is designed to improve performance assessment across the large configuration spaces found in performance-intensive software. We quickly implemented this process using BGML, executed it on Skoll, and demonstrated its effectiveness via a feasibility study involving ACE+TAO middleware, which are two large-scale performance-intensive software frameworks consisting of well over one million lines of C++ code and regression tests contained in ∼4,500 files.

The main effects screening process leverages formally-designed screening experiments to isolate the most significant individual options in the configuration space. This screening experiment is executed by users around the world using the Skoll infrastructure. After this reduced option set has been identified it can then be examined exhaustively (using local resources) as often as desired. Our feasibility study showed that this process could automatically reduce an original set of 14 options down to 2 main effects and that the information of these main effects provides much of the information that could have been gained from exhaustive testing (even though exhaustive testing is infeasible). We will continue to explore new DCQA processes, *e.g.*, we are examining how to prioritize parts of the configuration model based on end-user usage patterns, developer priorities, or other economic justifications.

The results of the work presented in this paper have also motivated research in several new directions. We are working closely with the ACE+TAO developers to generalize Skoll's processes to cover a broader range of QA activities, in particular new end-to-end QoS measures on heterogeneous DRE systems. One immediate application is to start to refactor ACE to shrink its memory footprint and enhance its run-time performance. The DCQA process will then be used to measure ACE's footprint and QoS at every check-in across different configurations, while simultaneously ensuring correctness via Skoll's automated and intelligent regression testing environment [8].

## 6. REFERENCES

[1] 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.

[2] 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, May 2004. IEEE.

[3] M. Harman and B. F. Jones. Search-based software engineering. *Information & Software Technology*, 43(14):833–839, 2001.

[4] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-Integrated Development of Embedded Software. *Proceedings of the IEEE*, 91(1):145–164, Jan. 2003.

[5] A. S. Krishna, D. C. Schmidt, R. Klefstad, and A. Corsaro. Real-time CORBA Middleware. In Q. Mahmoud, editor, *Middleware for Communications*. Wiley and Sons, New York, 2003.

[6] A. S. Krishna, N. Wang, B. Natarajan, A. Gokhale, D. C. Schmidt, and G. Thaker. CCMPerf: A Benchmarking Tool for CORBA Component Model Implementations. In *Proceedings of the 10th Real-time Technology and Application Symposium (RTAS '04)*, Toronto, CA, May 2004. IEEE.

[7] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Feedback Control Real-Time Scheduling: Framework, Modeling, and Algorithms. *Journal of Real-Time Systems, Special Issue on Control-Theoretical Approaches to Real-Time Computing*, 2002, to appear.

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

[9] Robert Ricci and Chris Alfred and Jay Lepreau. A Solver for the Network Testbed Mapping Problem. *SIGCOMM Computer Communications Review*, 33(2):30–44, Apr. 2003.

[10] M. Rutherford and A. L. Wolf. A Case for Test-Code Generation in Model-Driven Systems. In *International Conference on Generative Programming and Component Engineering (GPCE) 2003, Erfurt Germany*. ACM SIGPLAN SIGSOFT, Sept. 2003.

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

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

[13] D. C. Sharp. Avionics Product Line Software Architecture Flow Policies. In *Proceedings of the 18th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, Oct. 1999.

[14] 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.

[15] 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, Apr. 2004. ACM.

[16] 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*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.

[17] C. F. J. Wu and M. Hamada. *Experiments: Planning, Analysis, and Parameter Design Optimization*. Wiley, 2000.