Distributed Continuous Quality Assurance
to Maintain Persistent Software Attributes

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

Abstract

Time and resource constraints often force developers of highly configurable systems to assess their system’s 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

Quality of service (QoS)-intensive software must satisfy stringent requirements for latency/jitter/throughput, scalability, dependability, and security. Examples of QoS-intensive 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). This type of software is increasingly subject to the following trends:

- **Demand for user-specific customization.** To ensure that QoS-intensive software meets its requirements, it must often be fine-tuned to specific user platforms/contexts by adjusting many (i.e., 10’s-100’s) configuration options. Developers who write QoS-intensive software systems must try to ensure that their additions and modifications work across this large configuration space. In practice, however, global competition and time-to-market pressures are shrinking budgets for the development and QA of software in-house. Developers therefore can often only assess QoS on relatively few configurations and then 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. Another pitfall of in-house QA processes, stems from current day concern of ensuring high levels of security. Robust software cannot be guaranteed without extensive QA on a range of platforms. These issues directly affect reliability, portability and efficiency aspects of software.

- **Distributed and evolution-oriented development processes.** Today’s global IT economy and n-tier architectures often involve developers distributed across geographical locations, time zones, and business organizations. One of the goals of distributed development is to reduce cycle time by having developers work simultaneously, with minimal direct inter-developer coordination. Such development processes can increase churn rates in the software base, which in turn increases the need to detect, diagnose, and fix faulty changes quickly. The same situation occurs in evolution-oriented processes, where many small increments are routinely added to the base system. These issues directly affect the maintainability and usability aspects of software.

These trends are driving developers and organizations to change the processes they use to build and validate QoS-intensive software to cope with exploding software integration and configuration spaces remote developer coordination, and frequent software changes. Two promising techniques for addressing these challenges in the context of QoS-intensive software include:
• 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 [7]. This technique addresses the portability, reliability and usability attributes of software by ensuring functional correctness and QoS satisfaction across a range of hardware, OS and compiler platforms.

• Model-based software development techniques, which help to minimize the cost of QA activities by capturing the customizability of QoS-intensive software within models and automatically generating various artifacts (e.g., component interfaces, implementations, and glue code; configuration files; and deployment scripts) from these higher level models [3]. This technique minimizes the cost of maintaining and porting QoS intensive software by auto generating artifacts required for QA activities such as testing and benchmarking.

Heretofore, research on DCQA techniques and model-based software development techniques has proceeded independently. For example, the Skoll DCQA environment (www.cs.umd.edu/projects/skoll) provides a framework for executing QA tasks continuously across a grid of computers distributed around the world. Likewise, the Options Configuration Modeling language (OCML) [10] allows developers to model middleware configuration options as high-level models and the Benchmarking Generation Modeling Language (BGML) [5] allows developers to compose benchmarking experiments that observe QoS behavior by mixing and matching middleware configurations.

This article describes how the integration of DCQA environments with model-based software tools can help to monitor, safeguard, enforce, and reassert desirable properties after changes occur in QoS-intensive software. In particular, we describe model-based enhancements to Skoll that enable it to rapidly identify a small subset of highly influential QoS-related configuration options and systematically explore that subset of options empirically to estimate system QoS 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 consisting of two million lines of C++ code and regression tests contained in ~4,500 files. Our model-integrated process addresses a range of persistent software attributes ranging from functional correctness, portability to efficiency and QoS. Our results show that (1) a model-based DCQA environment and process can correctly identify a key subset of options that significantly affect system performance and (2) monitoring only these selected options improves developer insight into the effect of system changes on QoS at an acceptable cost.

The remainder of the article is organized as follows: Section 2 outlines the QA challenges associated with improving the quality of QoS-intensive software and describes how these challenges can be resolved by integrating model-based software development 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.

2 Addressing QA Challenges for QoS-intensive Software Systems

This section describes key QA challenges faced by developers of QoS-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 QoS-intensive Software Systems

Context. QoS-intensive software often provides fine-grained knobs to tune and optimized performance for particular runtime 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 QoS-intensive software systems.

Problem. To support customizations demanded by users, QoS-intensive software must run on many hardware and OS platforms and typically have many options to configure the system at compile- and/or runtime. Highly configurable QoS-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 – 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 [7]):

- **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, their results are returned to the ISA. By default, the ISA ignores these results. Often, however, we want to learn from incoming results. For example, when some configurations prove to be faulty, why not refocus resources on other unexplored parts of the configuration space. When such dynamic behavior is desired, process designers develop customized adaptation strategies, that monitor the global process state, analyze it, and use the information to modify future subtask assignments in ways that improve process performance.

In Skoll, adaptation strategies as independent programs executed by the Skoll server when subtask results arrive. This loosely couples Skoll and the adaptation strategies and allows us to develop, add, and remove adaptation strategies at will. As they must process subtask results, adaptation strategies must be tailored for each QA process. We describe several concrete adaptation strategies in a later feasibility study.

- **Automatic Characterization of Subtask Results.** It will often be necessary to interpret subtask results during the execution of a Skoll process. This is useful both for adapting the process (as shown in the previous section) and for providing developers with feedback. Given the amount and complexity of the data, we need to automate this process.

To this end we have included implementations of Classification Tree Analysis (CTA) [1] in the Skoll infrastructure. CTA approaches are based on algorithms that take a set of objects, \( O_i \), each of which is described by a set of features, \( F_{ij} \), and a class assignment, \( C_i \). CTA's output is a tree-based model that predicts object class assignment based on the values of a subset of object features.

- **Visualization.** Since Skoll processes are expected to generate large amounts of data, developers will likely need techniques for organizing and visualizing process results. We employ web-based scoreboards that use XML to display job configuration results. The server scoreboard manager provides a web-based query form allowing developers to browse Skoll databases for the results of particular job configurations. Visualizations are programmable with results presented in ways that are easy to use, readily deployed, and helpful to wide range of developers with varying needs. We have also incorporated a multi-dimensional, hierarchical data visualizer called Treemaps http://www.cs.umd.edu/hcil/treemap to display the findings of the automatic subtask results characterization described in the previous section.

![Figure 1: An Example Skoll DCQA Process](image-url)

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 [7] 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 Quality and Performance of QoS-intensive Software Systems

**Context.** QoS-intensive software systems run on a multitude of hardware/OS/compiler platforms and provide fine grained knobs to tune QoS behavior. Correctness and performance problems can and do show up in different subsets of the software configuration space.

**Problem.** To evaluate key performance and correctness characteristics of QoS-intensive software, QA engineers today often handcraft individual QA tasks (e.g., benchmarking experiments and test cases) 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, throughput, and correctness, 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 [5] 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. The scaffolding, script and task code generate can also be used to evaluate functional-correctness of software though the actual test-code needs to be handcrafted.

**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% [5] 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 the Performance and Correctness of QoS-intensive Software Across Large Configuration Spaces

**Context.** As developers create and modify their QoS-intensive software systems, they often conduct test/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 num-
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 QoS-intensive software systems are often so large that brute force processes are still infeasible. For example, the ACE+TAO middleware platforms have \( \sim 500 \) configuration options, with over \( 2^{500} \) potential combinations. To be effective for highly configurable QoS-intensive software systems, therefore, DCQA processes must generally include some type of adaptation strategy to efficiently navigate large configuration spaces.

**Solution approach → Techniques for configuration space reduction.** As QoS-intensive software systems change, developers often run regression tests to detect unintended functional side effects. In addition to functionality, developers of QoS-intensive systems must be wary of unintended effects on QoS and will often run performance benchmarking tests periodically 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 [7], only a small number of default configurations are benchmarked routinely. As a result ACE+TAO developers got a very limited view of their system’s QoS, which means that problems not readily seen in the few tested configurations can escape detection until the systems 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 [12], 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.

### 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.

1. A QA engineer defines a test configuration using BGML models. This step corresponds to Phase 1 of main effect screening process, where the experiment is visually composed and 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.

2. The QA engineer then uses BGML to interpret the model. The OCML paradigm interpreter parses the modeled CORBA middleware configuration options and generates the required configuration files to configure the underly-
ing 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. This step reduces the accidental complexity involved in handcrafting error-prone source code for potentially large set of configurations.

3. When users register with the Skoll infrastructure they obtain the Skoll client software and configuration template. As a part of the QA process, the main effects screening code is scheduled by the Skoll server to run on varied platforms.

4. 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. Using the results obtained, the QA engineer can identify the candidate set of configuration options. This corresponds to Phase II of main effect screening process. Using this isolated configuration space, the aforementioned process can be repeated to successively refine the configuration space.

3 Feasibility Studies

This section describes a pair of feasibility studies that empirically evaluate the DCQA approach described in Section 2.

3.1 Feasibility Study 1: Functional Correctness Testing

Our first feasibility study focuses on several scenarios in which we test ACE+TAO for different purposes across its numerous configurations. This study used three QA task scenarios applied to a specific version of ACE+TAO: (1) checking for clean compilation, (2) regression testing with default runtime options, and (3) regression testing with configurable runtime options. In addition, we enabled automatic characterization in order to give ACE+TAO developers concise descriptions of the subspace in which the test failures occurred.

We installed Skoll clients and one Skoll server across 10+ workstations distributed throughout computer science labs at the University of Maryland. All Skoll clients ran Linux 2.4.9-3 and used gcc 2.96 as their compiler. We chose a single OS and compiler to simplify our initial study and analyses. We used TAO v1.2.3 with ACE v5.2.3 as the subject software.

3.1.1 Configuring the Skoll Infrastructure

We implemented the components of the Skoll infrastructure described in Section 2.1 as follows:

Configuration model. We developed configuration models for each scenario. The Skoll system automatically translated the models into the ISA's planning language.

ISA. We configured the ISA as a stand-alone process running the Blackbox planner and using random sampling without replacement.

Adaptation strategies. We implemented three adaptation strategies – nearest neighbor, temporary constraints, and terminate/modify subtasks – as described below:

• Nearest neighbor. Suppose a test reports a configuration in which test cases are failing. Developers might want to quickly identify other similar configurations that pass or fail. The nearest neighbor strategy is designed to generate such configurations. For example, suppose that a test on a configuration space with three binary options fails in configuration 0,0,0. The nearest neighbor search strategy marks that configuration as failed and records its failure information. It then schedules for immediate testing all valid configurations that differ from the failed one in the value of exactly one option: {1,0,0}, {0,1,0} and {0,0,1}, i.e., all distance one neighbors. This process continues recursively. This approach is intended to quickly identify configurations that fail.

• Temporary constraints. Suppose that a software incorrectly fails to build whenever configuration options AMI = 0 and CORBA_MSG = 1. Developers, however, are unable to fix the bug immediately. Therefore, we may want prevent further selection of job configurations with these parameters until the problem is fixed. This adaptation strategy therefore would insert temporary constraints, such as CORBA_MSG = 1 → AMI = 1 (and others) into the configuration model.

• Terminate/modify subtasks. Suppose a test program is run at many user sites, failing continuously. At some point, continuing to run that test program on additional client machines provides little new information. Time and resources
might be better spent running some previously unexecuted test program. This adaptation strategy monitors for such situations and, depending on how it is implemented, can modify subtask characteristics or terminate the global process.

**Automatic characterization.** We developed scripts that prepare subtask results and feed them into the CTA algorithms. Figure 4 shows a sample classification tree based on some data from this study. The models covers 3 different compilation failures and 1 success condition for the results of 89 different configurations.

**Subtask execution.** We developed portable Perl scripts to be run by Skoll clients. These scripts request new QA job configurations, execute the jobs, and return the results to the server. We developed web registration forms and Skoll client software and created MySQL database schemas to manage user data and test results.

### 3.1.2 Scenario 1: Clean Compilation

Like other performance-intensive software toolkits, ACE+TAO allow many features be compiled in or out of the system. Features are often left out, for example, to reduce memory footprint in embedded systems. The QA task for Scenario 1 was to determine whether each ACE+TAO feature combination compiled without error. This is important for systems distributed in source code form, since any valid feature combination should compile. Unexpected build failures not only frustrate users, but also waste a lot of time. In fact, compiling the 1-million+ lines of code takes each client roughly 4 hours on a 933 MHz Pentium III with 400 Mbytes of RAM.

**Configuration model.** The feature interaction model for ACE+TAO is essentially undocumented. We therefore built our initial configuration model bottom-up. First, we analyzed the ACE+TAO source and interviewed several senior ACE+TAO developers. After discovering many omissions and inconsistencies in the models proposed by the developers, we settled on a subset of 10 binary-valued compile-time options that controls build time inclusion of features such as asynchronous messaging, use of software interceptors, and user-specified messaging policies.

We also identified 7 inter-option constraints. One example constraint is \([AM\bar{I} = 1 \rightarrow MIN\_COR\bar{B}A = 0]\). This means that the asynchronous method invocation (AMI) feature is not supported by the minimal CORBA implementation. In total, this configuration space has 89 valid configurations.

**Scenario execution.** Because the configuration space was small, we configured the ISA to use uniform random sampling without replacement since we considered one observation per valid configuration to be sufficient. Of the 89 valid configurations only 29 compiled without errors. For the 60 configurations that did not build, automatic characterization helped to clarify the conditions in which they failed.

**Results and observations.** Beyond identifying failures, in several cases, automatic characterization provided concise, statistically significant descriptions of the failing configuration subspace. Below we describe one example failure, present the automatically generated characterization, and discuss the action taken by ACE+TAO developers.

The ACE+TAO build failed line 38 in AsynchReply_Dispatcher.h (8 configurations) whenever \(CALLBACK = 0\) and \(POL\bar{L}ER = 1\). Since this configuration should be legal, this was determined to be a previously undiscovered bug. Until the bug could be fixed, we temporarily added a new constraint \(POL\bar{L}ER = 1 \rightarrow CALLBACK = 1\).

**Lessons learned.** We learned several things from this scenario. We found that even ACE+TAO developers do not completely understand the configuration model for their very complex system. Model building is therefore an iterative process. Using Skoll we quickly identified coding errors (some previously undiscovered) that prevented the software from compiling in certain configurations. We learned that the Temporary constraints and Process termination adaptation strategies performed well, directing the global process towards useful activities, rather than wasting effort on configurations that would surely fail without providing any new information. ACE+TAO developers also told us that the automatic characterization services were very useful to them in that the characterizations greatly narrowed down the number of issues they had to examine in tracking down the root cause of the failure.

Before moving on to the next scenario we fixed those errors we could. We worked around the more complex ones by leaving the appropriate temporary constraints in Scenario 2’s configuration model.

### 3.1.3 Scenario 2: Testing with Default Runtime Options

The QA task for the second scenario was to determine whether each configuration would run the ACE+TAO regression tests
without error with the system’s default runtime options. This activity is important for systems that distribute tests to run at installation time. This is intended to give the user confidence that he or she has correctly installed the system. To perform this task, each client must compile ACE+TAO, compile the tests, and execute the tests. On our machines this took around 8 hours: about 4 hours to compile ACE+TAO, about 3.5 hours to compile all tests, and 30 minutes to execute them.

Configuration model. In this scenario we used 96 ACE+TAO tests, each containing its own oracle and reporting success or failure on exit. These tests are often intended to run in limited situations, so we extended the configuration space, adding test-specific options. We also added some options capturing low-level system information, indicating the use of static or dynamic libraries, whether multithreading support is enabled, etc.

The new test-specific options contain one option per test. They indicate whether that test is runnable in the configuration represented by the compile time options. For convenience, we named these options \( \text{run}(T_i) \). We also defined constraints over these options. For example, some tests should run only on configurations with more the Minimum CORBA features. So for all such tests, \( T_i \), we added a constraint \( \text{run}(T_i) = 1 \rightarrow \text{MIN\_CORBA} = 0 \). This prevents us from running tests that are bound to fail.

Scenario execution. After making these changes, the space had 14 compile time options with 12 constraints and 96 test-specific options with an additional 120 constraints. We again configured the ISA for uniform random sampling without replacement.

Results and observations. In this section we detail some of the results and observations from this scenario. Overall, we compiled 2,077 individual tests. Of these 98 did not compile, 1,979 did. Of these, 152 failed, while 1,827 passed. This process took \( \sim 52 \) hours of computer time. As in Scenario 1, we now describe some example failures we uncovered, the automatically-generated failure characterizations, and the action taken by ACE+TAO developers.

In several cases tests failed for the same reason in the same configurations. For example, test compilation failed at line 596 of ami_testC.h for 7 tests, each when \( \text{CORBA\_MSG} = 1 \) and \( \text{POLLE\_R} = 0 \) and \( \text{CALLBACK} = 0 \). This was a previously undiscovered bug. It turned out that certain files within TAO responsible for implementing CORBA Messaging incorrectly assumed that at least one of the \( \text{POLLE\_R} \) or \( \text{CALLBACK} \) options would always be set to 1. ACE+TAO developers also noticed that the failure manifested itself no matter what the setting of the AMI was. This was a second previously undiscovered problem because these tests should not have been runnable when \( \text{AMI} = 0 \). Consequently, there was a missing testing constraint, which we then included in the test constraint set.

Test \( \text{RTCORBA/Client\_Protocol/run\_test.pl} \) failed 25 out of 29 times. In this case, automatic characterization failed to uncover a statistically significant model. This suggests that the problem is related to configuration options, but rather to some more pervasive problem (either a bug in the test code itself or some configuration-wide software problem). This problem was found to be due to a race condition in the shared memory Internet object protocol (SHMIOP) implementation.

Lessons learned. We learned several things from Scenario 2. We easily extended and refined the initial configuration model to create more complex QA processes. We again were able to carry out a sophisticated QA process across networked user sites on a continuous basis. In this case, we exhaustively explored the configuration space in less than a day and quickly flagged numerous real problems with ACE+TAO. Some of these problems had not been found with ACE+TAO’s \( \text{ad hoc} \) QA processes.

We also learned several things about automatic problem characterization. In particular, it is important to remember that generated models may be unreliable. We use notions of statistical significance to help indicate weak models, but more investigation is necessary. Also, the tree models we use may not be reliable when failures are non-deterministic and the ISA has been configured to generate only a single observation per valid configuration. In the presence of potentially non-deterministic failures, therefore, it may desirable to configure the ISA for random selection with replacement (rather than without replacement as we used in this scenario).

3.1.4 Scenario 3: Regression Testing with Configurable Runtime Options

The QA task for Scenario 3 was to determine whether each configuration would run the ACE+TAO regression tests without error over all settings of the system’s runtime options. This is important for building confidence in the system’s correctness. To do this users compile ACE+TAO, compile the tests, set the appropriate runtime options, and execute the tests. For us, each task takes about 8 hours.

Configuration model. To examine ACE+TAO’s behavior under differing runtime conditions, we modified the configuration model to reflect 6 runtime configuration options. These options set up to 648 different combinations of CORBA runtime policies: when to flush cached connections, what concurrency strategies the ORB should support, etc. (See Table 1). Since these runtime options are independent, we did not add any new constraints.

After making these changes, the compile-time option space had 14 options and 12 constraints, there were 96 test-specific options with an additional 120 constraints, and there were 6 runtime options with no new constraints.
Scenario execution. The configuration space for this scenario had 18,792 valid configurations. At roughly 30 minutes per test suite, the entire test process (not including compilation) involved around 9,400 hours of computer time. Given the large number of configurations, we used the nearest-neighbor adaptation strategy.

Results and observations. One observation is that several tests failed in this scenario even though they had not failed in Scenario 2 (when running tests with default runtime options). Some even failed on every single configuration (including the default configuration tested earlier), despite not failing in Scenario 1! In the latter case, the problems were often caused by bugs in option setting and processing code. In the former case, the problems were often in feature-specific code. ACE+TAO developers were intrigued by these findings because they rely heavily on testing by users at installation time, not just to verify proper installation, but to provide feedback on system correctness.

Another group of tests had particularly interesting failure patterns. Three of these tests failed between 2,500 and 4,400 times. In each case automatic characterization showed that the failures occurred when ORBCollocation = NO. No other option influenced failure manifestation.

TAO’s ORBCollocation option controls the conditions under which the ORB should treat objects as being collocated. The NO setting indicates that objects should never be treated as being collocated. When objects are not co-located they call each other’s methods by sending messages across the network. When they are collocated, they can communicate directly, saving networking overhead. The fact that these tests worked when objects communicated directly, but failed when they talked over the network clearly suggested a problem related to message passing. In fact, the source of the problem was a bug in their routines for marshaling/unmarshalling object references.

Lessons learned. We learned several things from Scenario 3. First, we confirmed that our general approach could scale well to larger configuration spaces. We also reconﬁrmed one of key conjectures: that data from the distributed QA process can be analyzed and automatically characterized to provide useful information to developers. We also saw how the Skoll process gives better coverage of the configuration space than does that used by ACE+TAO (and, by inference, many other projects).

Finally, we note that in the time it takes to compile the system we could run an entire test suite under several different runtime conﬁgurations. This suggests an interesting future work item adding cost metrics to the ISA’s planning operators.

### Table 1: Six ACE+TAO Runtime Options and Their Settings.

<table>
<thead>
<tr>
<th>Name</th>
<th>Possible Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORBCollocation</td>
<td>global, per-orb, NO, lru, llf, fifo, null</td>
</tr>
<tr>
<td>ORBConnectionPurgingStrategy</td>
<td>reactive, blocking, thread-per-connection</td>
</tr>
<tr>
<td>ORBFlushingStrategy</td>
<td>MT, ST, RW Blocked, Reactive, LF</td>
</tr>
<tr>
<td>ORBConcurrency</td>
<td></td>
</tr>
<tr>
<td>ORBClientConnectionHandler</td>
<td></td>
</tr>
<tr>
<td>ORBConnectStrategy</td>
<td></td>
</tr>
</tbody>
</table>

### 3.2 Feasibility Study 2: Performance Evaluation

We now describe a feasibility study that assesses the implementation cost and the effectiveness of the main effects screening process on a large QoS-intensive software system by exploring the following 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.

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.

**Step 1: Subject application.** We used ACE v5.4 + TAO v1.4 [4] + CIAO v0.4 for this study. CIAO is a QoS-enabled implementation of CCM that simplifies the development of DRE applications by enabling developers to declaratively provision QoS policies end-to-end when assembling a system.

**Step 2: Application scenario.** Due to recent changes made to the message queuing 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 runtime options they felt affected latency and throughput. Each option is binary as shown in Table 2 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.
Table 2: The Options and Their Settings

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

Table 3: Range of Performance Metrics Covered by Screening and Random Design

<table>
<thead>
<tr>
<th>Metric</th>
<th>Screening</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>latency</td>
<td>77%</td>
<td>46%</td>
</tr>
<tr>
<td>latency variance</td>
<td>64%</td>
<td>30%</td>
</tr>
<tr>
<td>throughput</td>
<td>75%</td>
<td>55%</td>
</tr>
</tbody>
</table>

Step 3: BGML tool. ACE+TAO+CIAO QA engineers used the BGML tool as described below to generate the screening experiments to quantify the behavior of latency and throughput:

1. Using the BGML modeling paradigm QA engineers composed the experiment.
2. In the experiment modeled, QA engineers associate the QoS characteristic (in this case roundtrip latency and throughput) that will be captured in the experiment.
3. Using the experiment modeled, BGML interpreters generate the benchmarking code required to set-up, run and tear-down the experiment. The files generated include, component implementation files (.h, .cpp), IDL files (.idl), Component IDL files (.cidl) and Benchmarking code (.cpp) files. The generated file is executed and QoS characteristics are measured.

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 decided to use a resolution IV screening design, which means that, among other things, that no main effects are aliased with any other main effects or with any two-factor interactions. The final screening design examines 14 factors in $2^{14} = 16,384$ runs, which is a $2^9$ fraction of the exhaustive design.

Step 5: Generating variation for the entire configuration space. For comparison purposed, we also obtained the performance variation for the entire configuration space, i.e., 16,384 configurations. It took 48 hours of computer time to run all the benchmarking experiments.

Results. Across the entire configuration space, 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 runtime options would contribute substantially to latency. The same result appears for latency variation and for throughput. Therefore, only options o2 and o10 show a significant effect on performance.

Looking instead at just the 32 data points from the “screening” design we see that we would 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 (in this case 4) configurations for the binary options 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 that distributions obtained from the screening experiments are very similar to the ones obtained from the exhaustive runs and the medians, not shown, were nearly identical. In contrast, the distributions for random configurations (4 chosen at random) were very different.

Table 3 shows the percentage 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 performance and covered more of the performance range than the random design did.

Lessons learned. From this study, we observed that over the course of several hours, we used BGML to generate scaffold-
ing 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 runtime 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 QoS 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 [11] 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 [8] 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 [6] to adapt adapt QoS measures. Our approach in BGML and Skoll uses screening experiments to determine first order configuration parameters 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 [2] 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 runtime 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 runtime. 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 [9] 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 QoS-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 article described a model-based distributed continuous quality assurance (DCQA) process called main effects screening. This process leverages formally-designed screening experiments to isolate the most significant individual options in the large configuration spaces found in QoS-intensive software. We rapidly implemented this process using BGML, executed it on the Skoll DCQA infrastructure, and demonstrated its effectiveness via two feasibility studies involving ACE+TAO middleware. These studies showed how a model-based DCQA process can 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).

The results of the work presented in this article have also motivated research in several new directions. 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. In addition, 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 runtime 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 [7].

References