Applying Model-Driven Distributed Continuous Quality Assurance Processes to Enhance Persistent Software Attributes

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

Abstract

*Time and resource constraints often force developers of highly configurable quality of service (QoS)-intensive software systems to guarantee their system's persistent software attributes (PSAs) (*e.g.*, functional correctness, portability, efficiency, and QoS) on a few platform configurations and to extrapolate from these configurations to the entire configuration space, which allows many sources of degradation to escape detection until systems are fielded. This article illustrates how model-driven distributed continuous quality assurance (DCQA) processes can help improve the assessment and assurance of these PSAs across the large configuration spaces of QoS-intensive software systems.*

Keywords. Distributed Continuous Quality Assurance, Model-Integrated Computing, Quality of Service, Software Configurations.

1 Introduction

Emerging trends and challenges. Quality of service (QoS)-intensive software must satisfy stringent requirements for persistent software attributes (PSAs), such as assured latency/jitter/throughput values, scalability, dependability, and security. Examples of QoS-intensive software include high--performance scientific computing systems (*e.g.*, high energy physics experiments and computational fluid dynamics), distributed real-time and embedded systems (*e.g.*, control systems for the electrical power grid and commercial air traffic), and the accompanying systems software (*e.g.*, operating systems, middleware, and language processing tools). QoS-intensive software is increasingly affected by trends toward *distributed and evolution-oriented development processes* and *demand for user-specific customization*.

Today's global information technology economy and n-tier architectures often involve developers distributed across geographical locations, time zones, and business organizations. One goal of distributed development is to ensure rapid change by having developers work simultaneously, with minimal direct inter-developer coordination. Distributed development processes can increase churn rates in the software base, which can lead to rapid, unpredictable, and potentially uncontrolled changes in persistent software attributes (PSAs), including reliability, scalability, efficiency, adaptability, maintainability, and portability. The same situation occurs in evolution-oriented processes [7], where many small increments are routinely added to the base system. What is needed are quality assurance (QA) techniques that can coordinate remote developers and cope with frequent software changes by detecting, diagnosing, and helping fix faulty and/or inefficient changes as quickly and automatically as possible.

In addition, QoS-intensive software needs to be fine-tuned to specific (and often changing) user platforms/contexts to meet its performance requirements. This is commonly done by (re)adjusting a large set of (10's-100's) configuration options that control parameters such as different workloads; operating system, middleware and application feature sets; compiler flags; and/or run-time optimization settings. Moreover, the configuration options that maximize QoS for a particular set of hardware, OS and compiler platforms may not produce optimal QoS for a different platform combination.

While this "explosion" in the *software configuration space* promotes adaptability and portability, it places enormous demands on the developers who must must ensure that their decisions and modifications work across this large (and often changing) configuration space. In fact, experience has shown that the *reliability, portability, and efficiency* PSAs of QoS-intensive software cannot be assured without extensive QA on a range of requirements and operating environments [6].

In practice, however, budgets for development and in-house QA are limited. Therefore, developers often can only assess PSAs on a few software configurations and then extrapolate to the entire configuration space. This allows many sources of PSA degradation to escape detection until systems are fielded. Moreover, those few in-house tested software configurations are often selected in an *ad hoc* manner so PSAs are not evaluated systematically. What is needed are QA techniques that can cope with exploding software configuration and validation spaces.

Addressing PSA challenges with distributed continuous quality assurance and model-driven techniques. Two promising techniques for addressing the QA challenges described above include *distributed continuous quality assur-*

ance (DCQA) and *model-driven* techniques. DCQA techniques are designed to improve PSAs iteratively, opportunistically, efficiently, and continuously in multiple, geographically distributed locations [6]. In particular, DCQA techniques have been shown to help assure the *portability and reliability* PSAs of QoS-intensive software by validating functional correctness and QoS satisfaction across a wide range of hardware, OS, network, and compiler platforms. For example, the *Skoll* DCQA environment (`www.cs.umd.edu/projects/skoll`) provides a framework for executing a variety of QA tasks continuously across a grid of computers distributed around the world and analyzing the results to evaluate PSAs. Likewise, the Dart DCQA environment (`public.kitware.com/Dart`) supports a continuous build/test process that starts whenever repository check-ins occur.

Model-driven techniques help to minimize the development, validation, and evolution effort associated with DCQA activities by capturing the customizability of QoS-intensive software within higher-level models that help to automate the analysis and synthesis of various artifacts, such as component interfaces, implementations, and glue code; configuration files; and deployment scripts [3]. In particular, model-driven techniques help assure the *correctness, maintenance, and porting* PSAs of QoS-intensive software by automatically generating artifacts required for QA activities, such as regression testing and performance benchmarking. For example, the Options Configuration Modeling language (OCML) [9] allows developers to model middleware configuration options as high-level models. Likewise, the Benchmarking Generation Modeling Language (BGML) [4] allows developers to automatically generate sophisticated benchmarking experiments.

This article describes how model-driven DCQA processes and tools can work separately and together to help monitor, safeguard, enforce, and reassert desirable PSAs after changes occur in QoS-intensive software. In particular, we describe two DCQA processes implemented in the model-driven Skoll environment that help to ensure PSAs related to both correctness and performance across large configuration spaces. To validate our approach, we present results from applying our model-driven DCQA processes and tools on ACE+TAO (`www.dre.vanderbilt.edu/Download.html`), which are widely-used QoS-enabled middleware consisting of ∼2,000,000 lines of continuously evolving C++ frameworks, functional regression tests, and performance benchmarks contained in ∼4,500 files that average over 300 CVS commits per week. Our results show that (1) DCQA techniques improve the ability of developers to detect and pinpoint software portability and availability problems, (2) integrating DCQA and model-driven techniques can significantly enhance the process of identifying key subsets of options that affect PSAs, and (3) monitoring only these selected options helps developers understand effects of system changes based on these PSAs with an acceptable level of effort.

Related work. Our model-driven DCQA techniques build upon earlier efforts that address limitations with conventional in-house QA processes. As described below, these efforts gather various types of information from distributed run-time environments and usage patterns encountered *in the field, i.e.*, on user target platforms with user configuration options.

Online crash reporting systems, such as the Netscape Quality Feedback Agent and Microsoft XP Error Reporting, gather system state at a central location whenever a fielded system crashes, which simplifies user participation in QA by automating certain aspects of problem reporting. Likewise, many popular open-source projects use *distributed regression test suites* that end-users can run to evaluate installation success. Well-known examples include GNU GCC, CPAN, Mozilla, the Visualization Toolkit (VTK), and ACE+TAO.

Auto-build scoreboards are a more proactive form of distributed regression test suites that allow software to be built/tested at multiple sites on various hardware, operating system, and compiler platforms. The Mozilla Tinderbox (`www.mozilla.org/tinderbox.html`) and ACE+TAO Virtual Scoreboard (`www.dre.vanderbilt.edu/scoreboard/`) are auto-build scoreboards that track end-user build results across various platforms. Bugs are reported via problem tracking systems (such as Bugzilla or Jira), which provides inter-bug dependency recording, advanced reporting capabilities, extensive configurability, and integration with automated software configuration management systems, such as CVS or Clearcase.

Although these prior efforts have provided a good starting point, they have several limitations. For example, they have a limited focus, ignoring PSAs related to QoS and performance issues. Our approach can be customized to address a wide variety of PSAs. Moreover, while these efforts help document and automate portions of the QA process, the decision of *what* and *when* to test it is left to users. In contrast, the model-driven DCQA techniques described in this paper enable developers to control key aspects of the QA process, thereby minimizing gaps and inefficiencies. As discussed below, our approach can exploit incremental results and selectively ignore problems discovered earlier, which avoids wasting resources that could otherwise be devoted to identifying other problems.

2 Applying Model-driven DCQA Techniques to Address PSA Challenges

To maintain and evaluate PSAs across large configuration spaces, we have developed the *model-driven Skoll* environment to support DCQA. We use these processes to evaluate

PSAs (such as latency, throughput, and correctness) "around-the-world, around-the-clock." To do this, Skoll's modeling tools divide the overall QA process into multiple subtasks, *e.g.*, running regression tests in a particular system configuration, evaluating system response time under a different input workloads, or measuring usage errors for a system with several alternative GUI designs. As illustrated in Figure 1, these

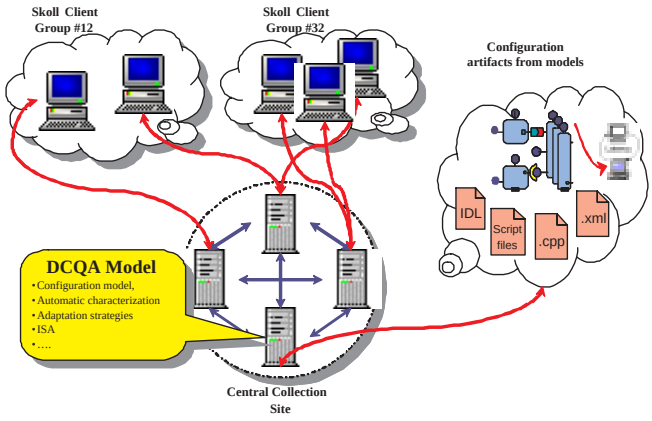

Figure 1: **The Skoll Model-driven DCQA Architecture**

tasks are then intelligently and continuously distributed to – and executed by – Skoll clients across a grid of computing resources contributed by end-users and distributed development teams. The results of these evaluations are returned to a server at a central collection site, where they are fused together to guide subsequent iterations of Skoll DCQA processes.

This section describes some of the components and services provided by the model-driven Skoll environment so that developers can implement and analyze DCQA processes. In particular, Skoll [6] provides languages for modeling system configurations and their constraints, algorithms for scheduling and remotely executing tasks, and analysis techniques for characterizing faults. Finally, we show how developers use Skoll to implement and execute large-scale, PSA-specific DCQA processes.

2.1 Managing PSA Challenges with Model-driven Skoll

The cornerstone of Skoll is its formal model of a DCQA process's configuration space, which captures the valid configurations for QA subtasks. A configuration in Skoll is represented formally as a set $\{ (O_1, S_1), (O_2, S_2), \ldots, (O_N, S_N) \}$, where each O_i is a configuration option and S_i is its value, drawn from the allowable settings of O_i. Since in practice not all configurations make sense (*e.g.*, feature X may not be supported on operating system Y), we define *inter-option constraints* that limit the setting of one option based on the setting of another. We represent constraints as $(P_i \rightarrow P_j)$, meaning "if predicate P_i evaluates to $TRUE$, then predicate P_j must evaluate to $TRUE$." A predicate P_k can be of the form A, $\neg A$, $A\&B$, $A|B$, or simply $O_i = S_j$, where A, B are predicates, O_i is an option and S_j is one of its allowable settings. A *valid configuration* is a configuration that violates no inter-option constraints. Skoll uses this configuration space model to plan global QA processes, adapt processes dynamically, and aid in analyzing and interpreting results.

Since the configuration spaces can be quite large, Skoll has an *Intelligent Steering Agent* (ISA) which controls DCQA processes by deciding which valid configuration to allocate to each incoming Skoll client request. When a client becomes available, the ISA decides which subtask to assign it. To do this, the ISA can consider many factors, including (1) *the configuration model*, *e.g.*, which characterizes the subtasks that can legally be assigned, (2) *the results of previous subtasks*, *e.g.*, which captures what tasks have already been done and whether the results were successful, (3) *global process goals*, *e.g.*, test popular configurations more than rarely used ones or test recently changed features more than heavily than unchanged features, and (4) *client characteristics and preferences*. The configuration must be compatible with physical realities such as the OS running on the remote machine. Also, client preferences, which are declared in a Skoll *client template* must be respected. For example, suppose a product runs in normal or superuser mode. A security conscious user might only want configurations in which the mode is normal.

Once 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.*, developer-supplied regression/performance tests) associated with a software project. The job configuration is then sent to the requesting Skoll client, which executes the job configuration and returns the results to the ISA so it can learn from the results and adapt the process. For example, if some configurations fail to maintain certain PSAs, developers may want to pinpoint the source of the problems or refocus on other unexplored parts of the configuration space. To do this Skoll process designers can 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.

Skoll applies various model-driven tools that raise the level of abstraction and reduce the accidental complexity of dealing with Skoll's internal formats. For example, Skoll employs the Benchmarking Generation Modeling Language (BGML) [4] that allows developers to (1) visually model interaction scenarios between configuration options and system components using domain-specific building blocks, (2) automate genera-

tion of common parts of test code and reuse QA subtask code across configurations, (3) generate control scripts to distribute and execute the experiments across the Skoll grid to monitor performance and functional behavior in a wide range of execution contexts, and (4) enable evaluation of multiple software attributes such as correctness, throughput, latency, jitter, and other criteria.

Since DCQA processes are complex we often need help to interpret and leverage process results. Therefore a wide variety of analysis tools can be plugged into Skoll. One such tool we added to Skoll implements Classification Tree Analysis (CTA) [2]. CTA's output is a tree-based model that predicts object class assignment based on the values of a subset of object features. As we show in Section 3.1, we used CTA to diagnose which specific options and option settings were most likely causing specific PSA test failures. This helped developers quickly identify the root causes of some failures.

2.2 An Example Model-driven DCQA Process

Figure 2 presents a high-level overview of how the BGML tool described above has been employed with the Skoll client/server infrastructure to support model-driven DCQA processes. Below we present an example model-driven DCQA

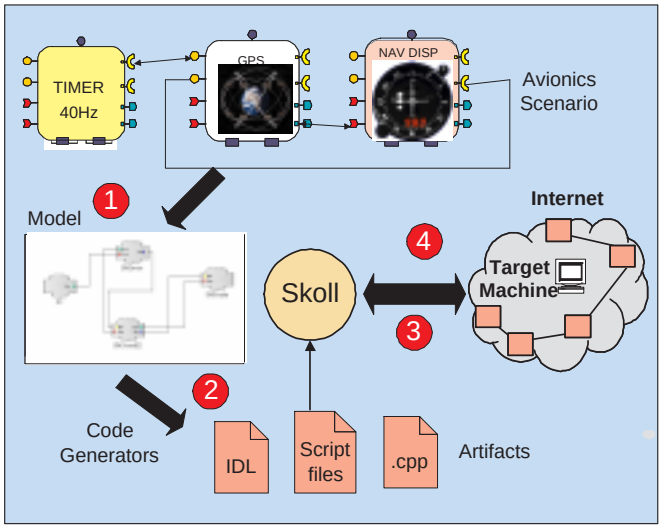

Figure 2: **Using Model-Driven Skoll**

process that illustrates the use and interactions of the Skoll components and BMGL tool. This example focuses on an avionics mission computing system [8] system developed using the ACE+TAO middleware described in Section 1.

Step 1: Define the application scenario. A developer uses BGML to model the software system and PSA-specific evaluation activities. Specifically, the models are visually composed via the Generic Modeling Environment (GME) model-driven toolsuite (`www.isis.vanderbilt.edu/Projects/gme/`). The resulting models detail the system's configuration options and inter-option constraints and capture PSA-specific information, such as the metrics calculated in a benchmarking experiment, the number and execution frequency of low-level profiling probes, or event patterns to monitor for or filter out of system logging server. For example, in the mission computing scenario, we use a three component *BasicSP* scenario that receives global position updates from a GPS device and displays them at a GUI display in real-time.

Step 2: Create benchmarks using the model-driven BGML tool. In the *BasicSP* scenario, the `GPS` component serves as the source for multiple components requiring position updates at a regular interval. This component's concurrency mechanism should therefore be tuned to serve multiple requests simultaneously in parallel. Moreover, the requirements that the desired data request and the display frequencies are fixed at 40 Hz is captured in within the models. The BGML model interpreter processes these models to generate the lower-level XML based configuration files, the required benchmarking code (*e.g.*, IDL files, required header and source files), and necessary script files to for executing the DCQA process. This step reduces accidental complexities associated with tedious and error-prone handcrafting of source code for a potentially large set of configurations. The configuration file is input to the ISA, which schedules the subtasks to execute as clients become available.

Step 3: Register and download clients. Remote users register with the Skoll infrastructure and obtain the Skoll client software and configuration template that was generated by the BGML model interpreter. Clients can run periodically at user-specified times, continuously, or on-demand.

Step 4: Execute DCQA Process. As each client request arrives, the ISA examines its internal rule base and Skoll databases and selects a valid configuration, packages the job configuration, and sends it to the client. The client executes it and returns the results to the Skoll server, which updates its databases and executes any adaptation strategies triggered by the new results.

3 Evaluating Correctness and Performance of QoS-Intensive Software

Ensuring the (re)usability of QoS-intensive software requires that it be adaptable and portable, *i.e.*, it should be configurable to run efficiently, robustly, and predictably on a wide range of hardware, OS, network, and compiler platforms that provide fine-grained knobs to tune QoS behavior. It is therefore important that functional correctness be verified and QoS properties

be validated for the configured operating contexts and environments. This section describes several feasibility studies that how Skoll can be used to assess and assure PSAs associated with functional correctness and run-time performance.

3.1 Evaluating Functional Correctness Across Large Configuration Spaces

Our first study examine three scenarios in which we test different PSAs of ACE+TAO across its numerous configurations, exploring the following hypotheses:

1. Our DCQA process helps to strengthen system-wide PSAs, such as portability and correctness.
2. The DCQA process can be easily used to quickly identify problems with software portability and compile-/run-time customizations.

We implemented the DCQA processes using the model-driven Skoll environment and then installed Skoll clients and one Skoll server across 10+ workstations distributed across a network. All clients ran Linux 2.4.9-3 and used gcc 2.96 as their compiler (we used a single OS and compiler to simplify the initial study and analysis, but have since run other studies across multiple operating systems and compilers). We then applied functional correctness QA task scenarios to ACE v5.2.3 and TAO v1.2.3 to check for clean compilation and perform regression testing with both default and configurable run-time options.

3.1.1 Scenario 1: Clean Compilation

Scenario 1 assessed whether each ACE+TAO feature combination compiled without error. We selected 10 binary-valued compile-time options that control build time inclusion of features, such as asynchronous messaging, use of software interceptors, and user-specified messaging policies[1]. We also identified 7 inter-option constraints, *e.g.*, ($A = 1 \longrightarrow B = 0$), which means that if option A is enabled then option B has to be disabled. The configuration space chosen has a total of 89 valid configurations.

By executing the process we determined that 60 of the 89 valid configurations did not even build – which was a quite surprise to the ACE+TAO developers! Using CTA analysis on the results we, for example, automatically characterized a previously undiscovered bug. This bug centered on a particular line within the TAO source code and occurred in exactly 8 configurations each of which shared a particular pair of options settings. Using our model-driven DCQA environment and process, we therefore successfully assessed the error-free compilation attribute of ACE+TAO, which is a necessary (though not a sufficient) condition to validate the functional correctness PSA.

[1] A detailed explanation of the many ACE+TAO configuration options are available from `www.dre.vanderbilt/TAO/docs/`

3.1.2 Scenario 2: Testing with Default Run-Time Options

Scenario 2 assessed the portability and correctness PSAs of ACE+TAO by executing regression tests on each compile-time configuration using the default run-time options (*i.e.*, the configuration new users encounter upon installation). We used the 96 regression tests that are distributed with ACE+TAO, each containing its own oracle and reporting success or failure on exit. We expanded the configuration model to include options that captured low-level OS and compiler information, *e.g.*, indicating the use of static vs. dynamic libraries, multithreading vs. singlethreading, and inlining vs. non-inlining. Also, some ACE+TAO tests can only run in particular configurations (such as when the multithreading is selected), so we also adding test-specific options to the configuration space.

The new test-specific options contain one option per test, ($run(T_i)$), which indicates if that test T_i is runnable in a given compile-time configuration. We also defined constraints over these options, *e.g.*, some tests should run only on configurations that have more than Minimum CORBA features. After making these changes, the space now had 14 compile time options with 12 constraints and an additional 120 test-specific constraints.

After resolving the constraints, we compiled 2,077 individual tests, of which 98 did not compile and 1,979 did. Of these 1,979 tests, 152 failed, while 1,827 passed. This process took $\sim$52 hours of computer time on the Skoll grid available for the experiments.

In several cases, tests failed for the same reason in the same configurations. For example, CTA analysis showed that test compilation failed at a given file for the following option settings ($CORBA\_MSG = 1$ and $POLLER = 0$ and $CALLBACK = 0$). This compilation error stemmed from a previously undiscovered bug that occurred because certain TAO files assumed these settings were invalid and thus could not occur. Using our model-driven DCQA environment and process, we were therefore able to determine whether the current version of ACE+TAO successfully completes all regression tests in its default configuration.

3.1.3 Scenario 3: Regression Testing with Configurable Run-Time Options

The goal of scenario 3 involved assessing the portability of ACE+TAO via execution-based test cases and run-time options by executing the ACE+TAO regression tests over all settings of their run-time options (such as when to flush cached connections or what concurrency strategies the ORB should support. See Table 1 for a summary of option settings). We

Name	Possible Settings
ORBCollocation	global, per-orb, NO
ORBConnectionPurgingStrategy	lru, lfu, fifo, null
ORBFlushingStrategy	leader_follower, reactive, blocking
ORBConcurrency	reactive, thread-per-connection
ORBClientConnectionHandler	MT, ST, RW
ORBConnectStrategy	Blocked, Reactive, LF

Table 1: **Six ACE+TAO Run-Time Options and their Settings.**

modified the configuration model to reflect 6 run-time configuration options. Overall, there were 648 different combinations of CORBA run-time policies

After making these changes, the compile-time option space had 14 options and 12 constraints, there were 120 test-specific constraints, and 6 run-time options with no new constraints. Thus, the configuration space for this scenario grew to 18,792 valid configurations (648 run-time x 29 compile-time configurations). At roughly 30 minutes per testsuite, the entire test process involved around 9,400 hours of computer time on the Skoll grid.

Several tests failed in this scenario, even though they had not failed in scenario 2 when they were run with default run-time options. These problems were often located in feature-specific code. Interestingly, some tests failed on every single configuration (including the default configuration tested earlier), despite succeeding in Scenario 2! These problems were often caused by bugs in option setting and processing code. ACE+TAO developers were intrigued by these findings because in practice they rely heavily on testing by users at installation time (scenario 2) to verify proper installation and provide feedback on system correctness. Our feasibility study raises some questions about the adequacy of that approach.

Another group of tests had particularly interesting failure patterns. Three of these tests failed between 2,500 and 4,400 times (out of 18,792 executions). We discovered that the failures occurred only when `ORBCollocation` = NO was selected (*i.e.*, no other option influenced these failures). This option allows objects within the same address space to communicate directly, saving (de)marshaling and protocol processing overhead. The fact that these tests worked when objects communicated directly – but failed when they talked over the network – suggested a problem related to message passing. In fact, the source of the problem was a bug in the ACE+TAO routines for (de)marshaling object references. Our DCQA process thus helped us to not only systematically evaluate the functional correctness PSA across many different runtime configurations, but also leveraged that information to help pinpoint the causes of specific failures.

3.2 Identifying Performance Degradation Across Large Configuration Spaces

As QoS-intensive software evolves, developers often run benchmark tests to check for unintended side effects on performance. As with testing, benchmarking highly configurable QoS-intensive software systems is difficult due to their enormous configuration spaces. This problem is compounded for evolving systems in which the the number of configurations that can be routinely examined before the system changes again is severely limited. As a result, developers only have a limited view of their system's performance PSAs, so problems not readily seen in the few tested configurations can (and do) escape detection until such systems are fielded.

Another challenge is that developers often have to handcraft individual QA tasks (such as regression test cases and benchmarking experiments) to evaluate key performance-related PSAs, *e.g.*, by writing such code as interface definitions, component implementations, client test applications and scaffolding and startup code. Of course, manually writing this code is error-prone since each step may be repeated many times for every QA experiment during each (re)validation phase.

To address these problems, we used model-driven Skoll to develop and implement a new DCQA process we call *main effects screening*. Main effects screening tries to *rapidly* detect degradations in performance PSAs across a large configuration space whenever the system changes. At a high level, main effects screening involves the following steps:

1. Compute a formal *experimental design* based on the system's configuration model. Our approach uses a class of experimental designs called *screening designs* [10], which are highly economical and can reveal significant *low order effects* (such as individual option settings and option pairs/triples) that significantly affect performance. We call these most influential option settings "main effects." The tradeoff is that these designs (and the main effects screening process itself) are inappropriate for systems with many significant higher order effects.
2. Execute that experimental design across the DCQA DCQA grid. Each task involves running and measuring benchmarks on a single configuration dictated by the experimental design devised in step 1. We used the model-driven BGML tool to simplify benchmark creation, execution, and analysis.
3. Collect and analyze the data to identify the main effects. The significance level that demarcates influential options can be set by developers.

Now we shift our QA process back to in-house resources. Whenever the software changes we evaluate all combinations

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 2: **ACE+TAO Options and their Settings**

of the main effects (while defaulting or randomizing all other options). We can do this quickly in-house because the set of main effects options is much smaller than the total configuration space. Our hope is that the performances of the main effects set mirrors those of the entire configuration space. If true, we can get nearly the same information as from exhaustive testing at a fraction of the cost.

Since the main effects can change over time, the process can be restarted periodically to recalibrate the main effects options. Recalibration frequency will depend on how and how fast the system changes.

We now show results from a two-phase feasibility study that explored the following hypotheses:

1. Main effects screening quickly identifies a small subset of options whose effect on performance is significant, allowing the rapid identification and monitoring of the software's performance attributes.
2. Evaluating all combinations of the main effects set produces performance data that is (1) representative of the system's performance across the entire configuration space and (2) more representative of the overall performance than that produced by observing a similarly-sized random sample of configurations.

Below, we describe the four steps we followed to evaluate the main effects DCQA process.

Step 1: Define the application scenario. As a result of changes to the ACE+TAO message queuing strategy, ACE+TAO developers want to monitor (1) the latency for each request and (2) total message throughput (events/second) between the ACE+TAO client and server. For this version of ACE+TAO, the developers identified 14 run-time options they felt affected latency and throughput. Each option is binary as shown in Table 2 and the entire configuration space is $2^{14} = 16,384$.

Step 2: Create benchmarks using the model-driven BGML tool. ACE+TAO developers used BGML (Section 2.1) to compose benchmarking experiments, which involved graphically modeling the desired number of clients and servers, workload characteristics, and performance metrics to be calculated. The graphical model is then interpreted to produce a large portion of the benchmarking code (over 90%).

Step 3: Apply the main effects screening process. This step creates a *resolution IV* screening design, which computes effects involving either one or two options, while assuming that no significant higher order effects exist (in the interests of space we have not included the statistical details of computing and interpreting screening designs, which are described in Wu [10]). The final screening design examines only 32 configurations of the 16,384 total configurations. Note that benchmarking the entire configuration space takes over 48 CPU hours, while benchmarking the screening design takes less than 6 minutes.

Step 4: Compare to exhaustive and random testing. For comparison purposes, we collected performance data for the entire configuration space. We also conducted random samples from this data to do further comparisons.

After performing these 4 steps and analyzing the results, we found that only options o2 and o10 had a significant effect on latency and throughout across the entire configuration space. These results surprised ACE+TAO developers since they thought that all 14 run-time options would contribute substantially to latency and throughput. Our analysis of the screening design data give the same results. We were therefore able to get accurate data at $\frac{1}{512\,th}$ the cost.

In the second phase of the process we 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.

We then measured latency and throughput on the benchmark test applications.

Our results showed that the performance distributions obtained from the main effects set were similar to the ones obtained from the exhaustive runs at a fraction of the cost. In contrast, randomly sampled configurations (*i.e.*, 4 chosen at random) produced very different data. It would therefore be an unreliable indicator of performance degradation following system changes. 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.

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

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

4 Concluding Remarks

This article motivated the need for – and design of – model-driven distributed continuous quality assurance (DCQA) processes and showed several examples of how these processes can be used by developers of QoS-intensive software to help assess and assure persistent software attributes (PSAs). We rapidly implemented two such DCQA processes using model-driven tools, executed them in the Skoll environment, and demonstrated their effectiveness via several feasibility studies involving the widely used ACE+TAO middleware.

These studies, presented in Section 3 showed how DCQA processes helped ensure key PSAs, such as functional correctness, maintainability, portability, and efficiency by: (1) *monitoring* (*e.g.*, the clean compilation scenarios described in Section 3.1.1), (2) *safeguarding* (*e.g.*, via feedback to developers on failing configurations and its isolation as described in Section 3.1.3), (3) *summarizing* (*e.g.*, via identification of main effects configurations that highly influence QoS as described in Section 3.2), and (4) *reasserting* (*e.g.*, via rerunning tests to validate main effects options) these attributes on a range of platforms. They also showed how Skoll-based DCQA processes reduced the level of effort – both in time and resources – required to assure the PSAs mentioned above in rapidly changing environments characterized by a multitude of configuration options and diverse OS/compiler platforms.

Despite our initial progress, much work remains to be done. For example, currently the overhead of specifying DCQA processes is more than we would like. We also do not yet support QA tasks that require human evaluation, such as evaluating usability or code maintainability. Moreover, our experiments thus far show that Skoll works best if we have a large number of client machines to run the experiments, so recruiting users to donate their computing resources – and assuring the security of these resources – is becoming increasingly important.

It's also important to note that the work presented here is only an initial foray into a broader R&D agenda on DCQA processes for *Remote Analysis and Measurement of Software Systems* (RAMSS) (see `measure.cc.gt.atl.ga.us/ramss`). To date, only a handful of research efforts [7, 1, 5, 6] have studied such processes systematically, so there are many unresolved challenges and risks, such as how best to structure DCQA processes, what types of QA tasks can be distributed effectively, and how the costs/benefits of DCQA processes compare to conventional in-house QA processes. To address these issues, we are working with other researchers in the RAMSS community to develop tools, services, and algorithms needed to create, prototype, and evaluate various types of DCQA processes focused on functional testing, QoS evaluation, and usage profiling of highly configurable software program families.

References

[1] J. Bowring, A. Orso, and M. J. Harrold. Monitoring deployed software using software tomography. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools And Engineering*, pages 2–9. ACM Press, 2002.

[2] L. Breiman, J. Freidman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth, Monterey, CA, 1984.

[3] 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.

[4] 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.

[5] B. Liblit, A. Aiken, and A. X. Zheng. Distributed program sampling. In *Proceedings of ACM Programming Languages Design and Implementation (PLDI) '03*, San Diego, California, June 2003.

[6] 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.

[7] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. Gamma system: Continuous evolution of software after deployment. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 65–69. ACM Press, 2002.

[8] 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.

[9] E. Turkay, 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.

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