Preserving Distributed Systems' Critical Properties

A Model-Driven Approach

Cemal Yilmaz, Atif M. Memon, and Adam A. Porter, *University of Maryland*

Arvind S. Krishna, Douglas C. Schmidt, Aniruddha Gokhale, and Balachandran Natarajan, *Vanderbilt University*

While the connectivity and efficiency advantages of distributed systems ensure their continued expansion, our increasing dependency on distributed resources comes at a cost of reduced predictability. Much of the burden of creating the needed levels of predictability necessarily falls on the software that controls and orchestrates the distributed resources' behaviors. The need for creating predictability in distributed systems is most often specified in terms of *quality-of-service* (QoS) requirements, which help define the acceptable levels of dependability with which capabilities such as processing capacity, data throughput, or service availability reach users. For longer-term properties such as scalability, maintainability, adaptability, and system security, we can similarly use *persistent software attributes* (PSAs) to specify how and to what degree such properties must remain intact as a network expands and evolves over time.

The Skoll Distributed Continuous Quality Assurance process helps identify viable system and software configurations for meeting stringent QoS and PSA requirements by coordinating the use of distributed computing resources. The authors tested their process using the large, rapidly evolving ACE+TAO middleware suite.

In the past, a common approach to ensuring the long-term persistence of important service properties was simply to freeze all changes after a system's properties had been sufficiently validated. However, for an increasing range of important QoS-intensive systems, this kind of development process no longer suffices. Today's global information economy strongly encourages forms of development that bring together participants from across geographical locations, time zones, and business organizations. While such distributed development processes provide a powerful and economically important development model, they unfortunately also increase the churn rates in QoS-intensive software (see the "QoS and PSAs" sidebar).

0740-7459/04/$20.00 © 2004 IEEE

PRESERVING DISTRIBUTED SYSTEMS' PROPERTIES

WHY THIS MATTERS

Persistent software attributes are akin to quality-of-service guarantees: they both require that specified functional properties remain intact during operational use. The main difference between them is that QoS requirements focus on immediate operational properties, such as throughput and latency, whereas PSAs focus on the resilience of long-term software engineering properties such as scalability, security, and maintainability. PSA and QoS concepts are closely intertwined, since a failure to specify and implement PSA requirements (for example, persistent scalability) can lead directly to serious QoS failures (for example, an inability to handle peak loads due to throughput failures).

This article addresses the difficult problem of how to meet and maintain QoS and PSA requirements in large distributed systems. The authors' approach, Skoll DCQA, helps keep the PSA validation process manageable through its use of modeling techniques that intelligently guide the process of distributed and continuous validation of properties. The article includes examples of how Skoll DCQA uncovered previously unrecognized problems in a large, well-maintained, widely used framework for distributed systems.

—Terry Bollinger, Jeffrey Voas, and Maarten Boasson, guest editors

Such churn typically destabilizes QoS properties such as latency, jitter, and throughput rates as well as—perhaps even more ominously—PSA attributes such as reliability, scalability, efficiency, adaptability, maintainability, and portability. Rapid code changes can also occur due to the use of evolution-oriented processes[1] such as agile practices, where many small code change increments are routinely added to the base system. The challenge in such cases is to develop techniques that can both help coordinate remote developers and allow them to cope more efficiently with frequent software changes. These new techniques must enable distributed teams to detect, diagnose, and correct changes that could damage not only QoS constraints but also the longer-term, investment-preserving PSA properties of such systems.

In practice, budgets for development and in-house quality assurance are often too limited to perform a comprehensive exploration of the software configuration space (see the related sidebar). Instead, developers typically assess PSAs for only a few software configurations and then arbitrarily extrapolate their results to the entire configuration space. This approach is risky, however, because it can let major sources of PSA degradation escape detection until systems are fielded. Even worse, since in-house–tested settings are often selected in an ad hoc manner, the points from which such generalizations are made might not even include any actual settings in fielded systems. What we need, then, are QA techniques that can provide a scalable, effective approach to understanding and navigating large software configuration spaces.

We selected and merged two existing techniques to address this challenge: distributed continuous quality assurance (DCQA)[2] and the Skoll model-driven approach to coordinating distributed activities (www.cs.umd.edu/projects/skoll). We developed Skoll DCQA to help automate QoS and PSA compliance in geographically distributed systems and teams. Skoll DCQA approaches QoS and PSA compliance in much the same way as a more traditional distributed data-processing task—by assigning subtasks that can be sent iteratively, opportunistically, and continuously to partici-

QoS and PSAs

Quality-of-service and persistent-software-attributes requirements often are tightly linked; for example, a QoS need for minimum services availability might not be achievable if a broader PSA requirement for continued long-term dynamic scalability of the distributed system can't also be met. QoS-intensive systems are ones in which services must either continually meet a wide range of strict QoS limits or face serious to catastrophic consequences. QoS-intensive systems are necessarily also PSA-intensive, because large-scale distributed systems aren't sufficiently fixed over time for their QoS certifications to remain adequately stable.

QoS-intensive distributed systems also often have both real-time and embedded properties. Examples include air traffic control systems and electrical power grid control systems, for which a serious failure to meet QoS and PSA requirements could lead to significant damage and even loss of human lives. Other examples of QoS-intensive systems include high-performance scientific-computing systems, in which losing a critical system property at the wrong time could be extremely costly or could lead to the loss of one-of-a-kind data. Examples of scientific QoS-intensive systems include robotic space missions, high-energy physics experiments, and long-running computational fluid dynamics simulations.

Software Configuration Spaces

A good strategy for tackling any large problem is first to identify and analyze a smaller subset of that problem, ideally one that's still large enough to apply to a range of interesting systems. QoS-intensive systems provide just such a solid starting point in the form of dozens to hundreds of software configuration settings in the operating system, middleware, application, compiler, and runtime software of each platform in a distributed system. Focusing on configuration settings limits the problem's total complexity yet still encompasses a large and interesting set of platform configurations.

An especially useful way to visualize the full range of settings possible in a distributed system is as a multidimensional *software configuration space*. Each possible total combination of settings on a network platform becomes a single unique point within this space, with the space as a whole encompassing every possible combination of settings. Points that lie close together in this space represent combinations of settings that differ from each other only by one or two individual settings. The total size of the software configuration space depends on the number of settings available for change. If only a small number of settings are treated "in play," the resulting *subset software configuration space* might have no more than a few tens of unique points within it. However, if every setting available on a typical realistic network platform is put into play, the resulting *full software configuration space* is typically very large indeed, containing millions or more of unique points.

We can now understand QoS requirements as criteria for determining an "acceptable performance" subset of the overall software configuration space. PSA requirements further limit this QoS subspace to exclude "quick fix" solutions—that is, solutions that might meet QoS constraints in the short term but only at the cost of significantly degrading one or more longer-term PSA properties such as adaptability, portability, scalability, or maintainability.

While the vast size of the full software configuration space ensures the presence of solutions that are both adaptable and portable, it also places enormous demands on developers who must ensure that all their design decisions keep the system within the boundaries of the QoS- and PSA-compliant subspaces. In particular, our experience has shown that PSA requirements for reliability, portability, and efficiency can't be reliably assured without first performing an extensive QA analysis of how system requirements interact with operating environments.[1] This is equivalent to mapping out the impact of QoS and PSA requirements over a large subset of the full software configuration space.

pating network clients.[2] Skoll DCQA makes it easier and more efficient to automate the analysis and synthesis of customization-related artifacts such as component interfaces, implementations, glue code, configuration files, and deployment scripts.[3]

The Skoll DCQA process

The Skoll DCQA process was inspired by earlier work, including the Options Configuration Modeling language,[4] the Benchmarking Generation Modeling Language (BGML),[5] commercial systems efforts such as the Netscape Quality Feedback Agent and Microsoft's XP Error Reporting, the distributed regression test suites that open source projects use, and autobuild scoreboards such as the ACE+TAO Virtual Scoreboard (www.dre.vanderbilt.edu/scoreboard) and Dart (www.public.kitware.com/Dart).

Figure 1 depicts Skoll DCQA's process architecture; in this case, two of several Skoll client groups are communicating (shown as red bidirectional arrows) with a central collection site. Each collection site comprises a cluster of servers connected via high-speed links (blue bidirectional arrows). The (yellow) callouts show the servers' contents in terms of the DCQA model and configuration artifacts. We use Skoll's configuration model to intelligently guide the distribution and continuous execution of Skoll clients across a grid of computing resources. In this process, Skoll provides languages for modeling system configurations and their constraints, algorithms for scheduling and remotely executing tasks, and analysis techniques for characterizing faults.[2] The results of the distributed evaluations are returned to central Skoll servers, where they are fused together to guide subsequent iterations of the Skoll DCQA process.

Skoll DCQA's analytical cornerstone is its use of a formal model of the software configuration space in which its processes must operate. This formal model makes it possible to capture valid configurations and use them to develop specific, well-defined 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 each S_i is its value, drawn from the allowable settings. Not all configurations make sense in practice (for example, feature X might not be supported on operating system Y), so Skoll DCQA also includes inter-option constraints that limit the setting of one option based on the setting of another.

To navigate the remaining valid configurations in the space, Skoll uses an *intelligent steering agent* that assigns subtasks to clients as they become available. The ISA considers four major factors when making such assignments:

- The configuration model, which characterizes the subtasks that can be assigned legally
- The summarized results of previously completed subtasks
- A set of global process goals that define general policies for allocating subtasks, such as testing recently changed features more than unchanged ones

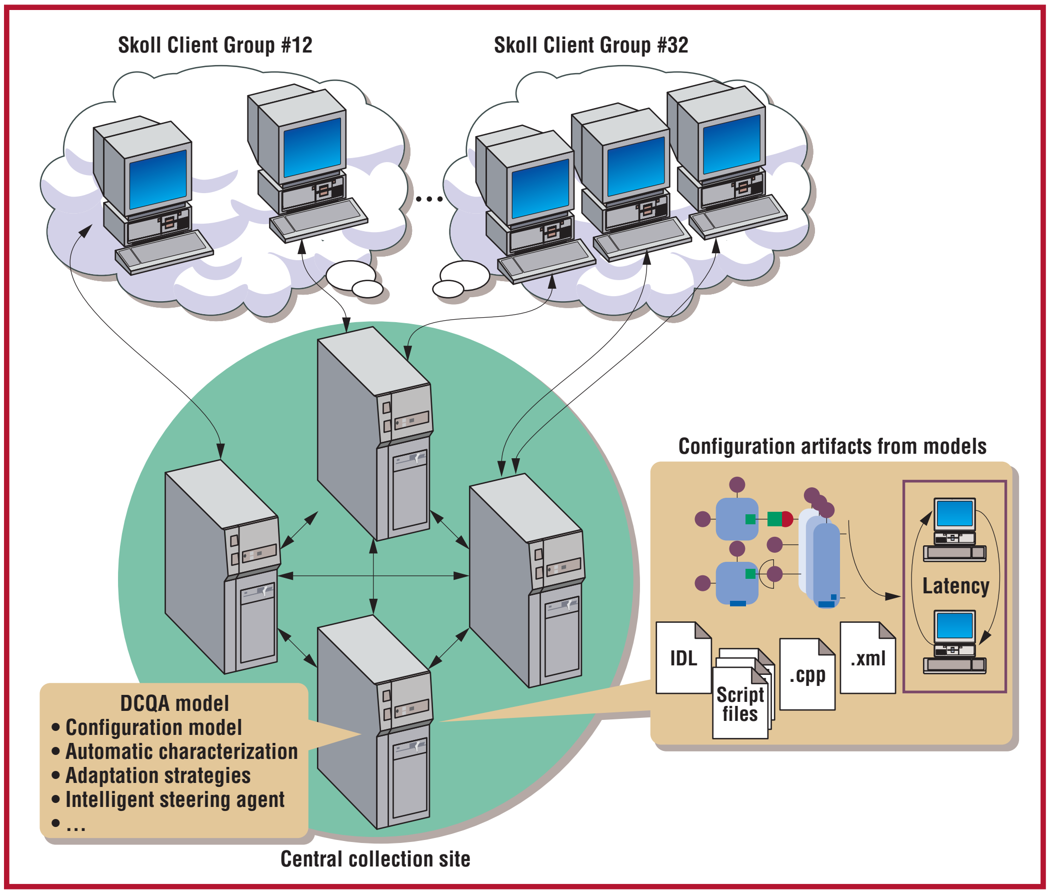


Figure 1. The Skoll DCQA process architecture.

- Clients' characteristics and participation level preferences, as described in a template provided by clients

Once the ISA has selected a configuration that meets all the factors relevant to a newly available client, it creates a *job configuration* that includes all the code artifacts, configuration parameters, build instructions, and QA-specific code (such as developer-supplied regression and performance tests) associated with that subtask. The job configuration is then sent to the Skoll client, which executes it and returns the results to the ISA for collection and analysis.

A notable feature of this process is that by looking at the results returned from many clients, the ISA can learn and adapt the overall Skoll DCQA process to make it more efficient. For example, the ISA can identify configurations that repeatedly fail to meet PSAs, thereby enabling developers to concentrate their efforts on fixing these problems. Conversely, once problem configurations have been characterized, the ISA can refocus efforts on other previously unexplored parts of the configuration space that might provide better PSA support.

To help developers implement specific Skoll DCQA processes, the Skoll DCQA environment provides a variety of model-driven tools, such as the BGML. This modeling tool lets developers model interaction scenarios visually, generate test code, reuse QA subtasks across configurations, generate scripts to control subtask distribution and execution by clients, monitor performance and functional behaviors, and evaluate software attributes including correctness, throughput, latency, and jitter.

Skoll DCQA includes a variety of analysis tools that help developers interpret and leverage its often-complex global process results. One such tool provides *classification tree analysis*,[6] which creates tree-based models that predict object class assignment based on the values of a subset of object features. CTA is used to diagnose which options and option settings are most likely causing specific PSA test failures, thus helping developers identify root causes of PSA failures quickly.

Evaluating Skoll DCQA

To evaluate the effectiveness of the Skoll DCQA process at navigating software configuration spaces, we applied it in three different contexts described next: support of the ACE+TAO code base, an avionics application of ACE+TAO, and early work on the problem of churn-induced degradation of distributed code bases.

We focused primarily on QoS and PSA requirements affecting latency, throughput, and correctness.

Applying Skoll DCQA to ACE+TAO

In this application, we used Skoll DCQA to analyze and support the popular ACE+TAO middleware suite (www.dre.vanderbilt.edu/Download.html). ACE+TAO provided a good high-end test of the Skoll DCQA process, because it includes about two million lines of continuously evolving C++ frameworks, functional regression tests, and performance benchmarks, packaged in about 4,500 files and receiving an average of over 300 code updates per week. We focused primarily on QoS and PSA requirements affecting latency, throughput, and correctness. Subtasks assigned to the clients included activities such as compiling the system, running regression tests in a particular system configuration, and evaluating system response times under different input workloads.

From this exercise (described in detail elsewhere[2]), we identified three main benefits of using Skoll DCQA:

- Detecting and pinpointing software portability and availability problems
- Identifying the subsets of options that had the most impact on specific PSAs
- Understanding the effects of system changes on PSAs, enabling developers to ensure continued support with an acceptable level of effort

Because the ACE+TAO evaluation was the most controlled of the three we conducted, we were able to use it to try out scenarios, starting with small subset configuration spaces and working toward larger ones. For simplicity and easy comparison of results, we ran all these scenarios on a single operating system (Linux 2.4.9-3) using a single compiler (gcc 2.96); since then, we've run other studies across multiple operating systems and compilers.

Scenario 1: Clean compilation. This scenario 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 (a detailed explanation of the many ACE+TAO configuration options are available at www.dre.vanderbilt.edu/TAO/docs). We also identified seven inter-option constraints that take forms similar to "if option A is enabled, then option B must be disabled." This relatively small subset configuration space had 89 valid configurations.

By executing the Skoll DCQA process on this subset configuration space, we determined that 60 of the 89 valid configurations didn't even build—a result that was quite a surprise to the ACE+TAO developers. By using CTA on the results from client workstations, we were also able to identify a previously undiscovered bug. This bug centered on a particular line in the TAO source code, and it occurred in eight configurations that all had a specific pair of option settings.

Even though we intentionally kept the selected configuration space small for testing purposes, it was still large enough to contain significant problems that the maintainers of the ACE+TAO code base hadn't yet identified. Identifying these problem areas in the configuration space helped improve both QoS and PSA support by steering users toward safer regions in the configuration space and by showing ACE+TAO developers what needed to be fixed.

Scenario 2: Testing with default runtime options. This scenario assessed portability and correctness PSAs of ACE+TAO by executing regression tests on each compile-time configuration using the default runtime options (that is, the configuration that new users encounter upon installation). We used the 96 regression tests that are distributed with ACE+TAO, each containing its own test oracle that reported success or failure on exit. We expanded the configuration model to include options that captured low-level operating system and compiler information—for example, indicating the use of static versus dynamic libraries, multithreading versus singlethreading, and inlining versus non-inlining. We also added test-specific options to the configuration space because some ACE+TAO tests can only run in particular configurations, such as when multithreading is enabled.

We added one test-specific option per test, called `run(`T_i`)`, which indicates whether test T_i will run in a given compile-time configuration. We also defined constraints over these options; for example, some tests can run only on configurations that have more than Minimum CORBA features. After making these changes, the space 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. Of the 1,979 tests that did compile, 152 failed, while 1,827 passed. This process took about 52 hours of computer time on the Skoll grid that was available for the experiments.

Subsequent analysis using classification trees showed that in several cases tests failed for the same reason in the same configurations. For example, the analysis showed that test compilation always failed at a given file for the following option settings: `CORBA_MSG` = 1, `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 couldn't occur. Using our model-driven DCQA environment and process, we were able to determine whether the current version of ACE+TAO successfully completed all regression tests in its default configuration.

Scenario 3: Regression testing with configurable runtime options. This scenario assessed the functional correctness of ACE+TAO by executing the ACE+TAO regression tests over all settings of their runtime options, such as when to flush cached connections or what concurrency strategies TAO should support. This was the largest scenario in terms of configuration space. We modified the configuration model to reflect six runtime configuration options. Overall, there were 648 different combinations of CORBA runtime policies.

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

Several tests failed in this scenario, although they hadn't failed in Scenario 2 when they were run with default runtime 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. Bugs in option setting and processing code were often the sources of these problems. ACE+TAO developers were intrigued by these findings because in practice they rely heavily on user testing at installation time (Scenario 2) to verify proper installation and provide feedback on system correctness. Our feasibility study raises questions about the adequacy of that approach.

Another group of tests had particularly interesting failure patterns. Three tests failed between 2,500 and 4,400 times (out of 18,792 executions). Using CTA, we automatically discovered that the failures occurred only when `ORBCollocation = NO` was selected (that is, no other option influenced these failures). This option lets objects in the same address space communicate directly, saving marshaling, demarshaling, 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, further scrutiny revealed the problem's source was a bug in the ACE+TAO routines for marshaling and demarshaling object references. Our DCQA process thus helped us not only to systematically evaluate the functional correctness PSA across many different runtime configurations but also to leverage that information to help pinpoint the causes of specific failures.

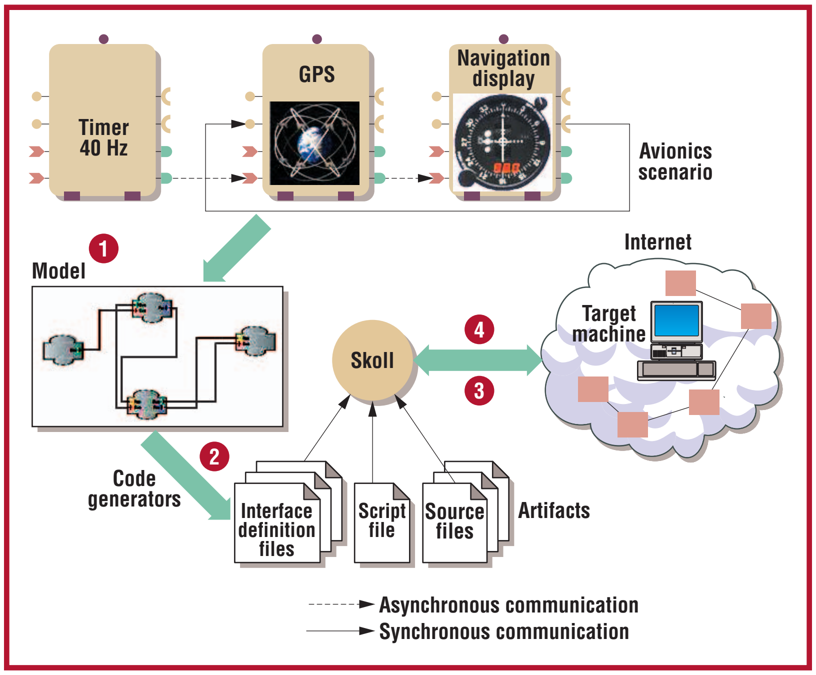


Figure 2. Using Skoll DCQA on an avionics system.

Applying Skoll DCQA to avionics

Figure 2 gives a high-level overview of how we used Skoll DCQA to analyze and support PSAs for one part of an avionics mission computing system based on Boeing's Bold Stroke software archtecture.[7] This system was developed using the same ACE+TAO middleware described in the first example. This scenario's

Finding and characterizing "dangerous" regions in configuration spaces helps ensure that important properties can be preserved as their code base expands over time.

goal was to leverage the generative capabilities of our BGML tool[5] and integrate it with Skoll DCQA. There were four major steps in applying Skoll DCQA to the avionics system.

Step 1: Define the application scenario. Using the Generic Modeling Environment model-driven tool suite (www.isis.vanderbilt.edu/Projects/gme), developers created BGML models of both the avionics software system and the PSA-specific evaluation activities that are needed to ensure acceptable operational behavior. The models detailed system configuration options and interoption constraints, and also captured PSA-specific information such as the metrics calculated in benchmarking experiments, the number and execution frequency of low-level profiling probes, and which event patterns should be filtered out or logged by the system. For example, in the avionics mission computing scenario, we used a three-component, basic single-processor scenario (known as the BasicSP scenario) that receives global-position updates from a GPS device and then displays them on a user interface in real time.

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

Generating the required source and configuration files from higher-level models removed the need to handcraft these files. Moreover, the generated code was syntactically and semantically correct, thus eliminating common sources of errors. The Skoll DCQA process just described illustrates how DCQA capabilities can address both performance-related PSAs and functional-correctness PSAs.

Applying Skoll DCQA to change-induced performance degradation

Given its successes in the first two studies described earlier, we were interested in whether we could also apply Skoll DCQA to the problem of ensuring consistent performance in an environment of rapid code change, which can lead to performance degradation over time. We call the strategy we used *main effects screening*, which is based on the following three steps.

Step 1: Generate a formal model of the most relevant effects. This formal model is based on the Skoll DCQA system configuration model and uses *screening designs*[8] to help reveal low-order effects—that is, small changes to single, paired, or tripled option settings that have seemingly disproportionate effects on performance. We call the most influential of these option settings *main effects*. This technique works when the relationships between such settings and performance remain reasonably linear.

Step 2: Execute the resulting model on the DCQA grid. Each task involves running and measuring benchmarks on a single configuration dictated by the experimental model devised in Step 1. As before, we used the model-driven BGML tool to simplify benchmark creation, execution, and analysis.

Step 3: Collect and analyze the data to identify main effects. By first centralizing and then iteratively redistributing the lessons learned provided by a wide range of participating developers, it becomes possible over time to form an improved consensus on the main effects' settings and their thresholds. We've found that periodically monitoring just these main effects gives an inexpensive and rapid way to detect performance degradations across the configuration space.

The results from our Skoll DCQA studies have been encouraging. They've demonstrated the approach's value at finding and characterizing "dangerous" regions in configuration spaces, which in turn helps developers ensure that important properties can be preserved as their code base expands over time. We aren't the only group that has addressed this need via DCQA processes,[1,2,9,10] but much more remains to be done. One important challenge is to increase the overall level of automation in DCQA processes. Specifying DCQA processes, for example, is currently a more labor-intensive process than we'd like. Another problem is applying DCQA processes to PSA properties such as usability and maintainability that aren't easily measured by automated means, and so again require labor-intensive human intervention. Other unresolved challenges and risks include how best to structure DCQA processes, what types of QA tasks can be distributed effectively, and how the costs and benefits of DCQA processes compare to conventional in-house QA processes.

To address these issues, we're working with other researchers in the Remote Analysis and Measurement of Software Systems community (http://measure.cc.gt.atl.ga.us/ramss) to develop tools, services, and algorithms needed to prototype and test DCQA processes. Scaling is an interesting issue; in general, we expect Skoll DCQA's quality to improve as the number of participating clients and users increases. This positive scaling effect is reminiscent of the scaling phenomenon that Terry Bollinger postulates (*Software Cooperatives: Infrastructure in the Internet Era*, www.terrybollinger.com/swcoops/swcoops) is behind the recent rapid economic growth in the use of open source development.

Despite such challenges, the overall Skoll DCQA approach of using iterative, model-coordinated, highly distributed evaluations of software properties holds considerable promise for better understanding, characterizing, and improving distributed systems. We look forward to helping this field evolve over the next few years and evaluating the degree to which such methods end up supporting and validating important properties that must be made persistent over time to be fully usable.

Acknowledgments

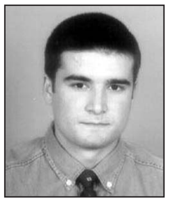
We thank Terry Bollinger for extensive comments that greatly improved this article's form and content. This material is based on work supported by the US National Science Foundation under NSF grants ITR CCR-0312859, CCR- 0205265, and CCR-0098158, as well as funding from BBN Technologies, Lockheed Martin, Raytheon, and Siemens.

References

1. A. Orso et al., "Gamma System: Continuous Evolution of Software After Deployment," *Proc. Int'l Symp. Software Testing and Analysis*, ACM Press, 2002, pp. 65–69.
2. A. Memon et al., "Skoll: Distributed Continuous Quality Assurance," *Proc. 26th IEEE/ACM Int'l Conf. Software Eng.*, IEEE CS Press, 2004, pp. 459–468.
3. G. Karsai et al., "Model-Integrated Development of Embedded Software," *Proc. IEEE*, vol. 91, no. 1, 2003, pp. 145–164.
4. E. Turkaye, A. Gokhale, and B. Natarajan, "Addressing the Middleware Configuration Challenges using Model-based Techniques," *Proc. 42nd Ann. Southeast Conf.*, ACM Press, 2004.
5. A.S. Krishna et al., "CCMPerf: A Benchmarking Tool for CORBA Component Model Implementations," *Proc. 10th Real-Time Technology and Application Symp.* (RTAS '04), IEEE Press, 2004, pp. 140–147.
6. L. Breiman et al., *Classification and Regression Trees*, Wadsworth, 1984.
7. D.C. Sharp and W.C. Roll, "Model-Based Integration of Reusable Component-Based Avionics System," *Proc. Workshop Model-Driven Embedded Systems in RTAS 2003*, IEEE Press, 2003.
8. C.F.J. Wu and M. Hamada, *Experiments: Planning, Analysis, and Parameter Design Optimization*, John Wiley & Sons, 2000.
9. J. Bowring, A. Orso, and M.J. Harrold, "Monitoring Deployed Software Using Software Tomography," *Proc. 2002 ACM SIGPLAN/SIGSOFT Workshop Program Analysis for Software Tools and Eng.*, ACM Press, 2002, pp. 2–9.
10. B. Liblit, A. Aiken, and A.X. Zheng, "Bug Isolation via Remote Program Sampling," *Proc. ACM SIGPLAN 2003 Conf. Programming Languages Design and Implementation* (PLDI 03), ACM Press, 2003, pp. 141–154.

About the Authors

Arvind S. Krishna is a PhD student in the Electrical Engineering and Computer Science Department at Vanderbilt University and a member of the Institute for Software Integrated Systems. He received his MA in management and his MS in computer science from University of California, Irvine. His research interests include patterns, real-time Java technologies for Real-Time Corba, model-integrated QA techniques, and tools for partial evaluation and specialization of middleware. He is a student member of the IEEE and ACM. Contact him at the Inst. for Software Integrated Systems, 2015 Terrace Pl., Nashville, TN 37203; arvindk@dre.vanderbilt.edu.



Cemal Yilmaz is a PhD student in computer science at the University of Maryland, College Park. His research interests include distributed, continuous QA, software testing, performance evaluation, and application of design of experiment theory to software QA. Contact him at the Dept. of Computer Science, Univ. of Maryland, College Park, MD 20742; cyilmaz@cs.umd.edu.

Atif M. Memon is an assistant professor in the Department of Computer Science at the University of Maryland. His research focuses on program testing, software engineering, artificial intelligence, plan generation, reverse engineering, and program structures. He received his PhD in computer science from the University of Pittsburgh and is a member of the IEEE Computer Society and ACM. Contact him at the Dept of Computer Science, 4115 A.V. Williams Bldg., Univ. of Maryland, College Park, MD 20742; atif@cs.umd.edu.

Adam A. Porter is an associate professor in the Department of Computer Science and the Institute for Advanced Computer Studies at the University of Maryland. His research interests include empirical methods for identifying and eliminating bottlenecks in industrial development processes, experimental evaluation of fundamental software engineering hypotheses, and development of tools that demonstrably improve the software development process. He received his PhD in information and computer science from the University of California, Irvine, and is a member of the IEEE Computer Society and ACM. Contact him at the Dept. of Computer Science, Univ. of Maryland, College Park, MD 20742; aporter@cs.umd.edu.

Douglas C. Schmidt is a professor in the Electrical Engineering and Computer Science Department at Vanderbilt University and a senior research scientist at the Institute for Software Integrated Systems. His research interests include patterns, optimization techniques, and empirical analyses of software frameworks and domain-specific modeling environments that facilitate the development of distributed real-time and embedded middleware and applications running over high-speed networks and embedded system interconnects. He received his PhD in information and computer science at the University of California, Irvine. Contact him at the Inst. for Software Integrated Systems, 2015 Terrace Pl., Nashville, TN 37203; schmidt@dre.vanderbilt.edu.

Aniruddha Gokhale is an assistant professor in the Electrical Engineering and Computer Science Department at Vanderbilt University and a senior research scientist at the Institute for Software Integrated Systems. His research focuses on real-time component middleware optimizations, distributed systems and networks, model-driven software synthesis applied to component middleware-based distributed systems, and distributed resource management. He received his PhD in computer science from Washington University. Contact him at the Inst. for Software Integrated Systems, 2015 Terrace Pl., Nashville, TN 37203; a.gokhale@vanderbilt.edu.

Balachandran Natarajan is a senior staff engineer at the Institute for Software Integrated Systems and a PhD student in electrical engineering and computer science at Vanderbilt University. His research focuses on applying patterns, optimization principles, and frameworks to build high-performance, dependable, and real-time distributed systems. He received his MS in computer science from Washington University. Contact him at the Inst. for Software Integrated Systems, 2015 Terrace Pl., Nashville, TN 37203; bala@dre.vanderbilt.edu.