Validating Quality of Service for Reusable Software via Model-integrated Distributed Continuous Quality Assurance*

Arvind S. Krishna[1], Douglas C. Schmidt[1], Atif Memon[2], Adam Porter[2], and Diego Sevilla[3]

[1] Electrical Engineering & Computer Science, Vanderbilt University, TN USA, {arvindk, schmidt}@dre.vanderbilt.edu,
[2] Computer Science Department, University of Maryland College Park, MD USA, {atif, aporter}@cs.umd.edu
[3] Dept. of Computer Engineering, University of Murcia, Spain, {dsevilla@ditec.um.es}

Abstract

Quality assurance (QA) tasks, such as testing, profiling, and performance evaluation, have historically been done in-house on developer-generated workloads and regression suites. Performance-intensive systems software, such as that found in the scientific computing grid and distributed real-time and embedded (DRE) domains, increasingly run on heterogeneous combinations of OS, compiler, and hardware platforms. Such software has stringent quality of service (QoS) requirements and often provides a variety of configuration options to optimize QoS. As a result, QA performed solely in-house is inadequate since it is hard to manage software variability, i.e., ensuring software quality on all supported target platforms across all desired configuration options. This paper describes how the Skoll project is addressing these issues by developing advanced QA processes and tools that leverage the extensive computing resources of user communities in a distributed, continuous manner to improve key software quality attributes.

1 Introduction

Emerging trends and challenges. While developing quality reusable software is hard, developing it for performance-intensive systems is even harder. Examples of performance-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). Reusable software for these types of systems must not only function correctly across the multiple contexts in which it is reused and customized – it must also do so efficiently and predictably.

* This material is based upon work supported by the National Science Foundation under Grant Nos. NSF ITR CCR-0312859, CCR-0205265, CCR-0098158.

To support the customizations demanded by users, reusable performance-intensive software often must (1) run on a variety of hardware/OS/compiler platforms and (2) provide a variety of options that can be configured at compile- and/or run-time. For example, performance-intensive middleware, such as web servers (*e.g.*, Apache), object request brokers (*e.g.*, TAO), and databases (*e.g.*, Oracle) run on dozens of platforms and have dozen or hundreds of options. While this variability promotes customization, it creates many potential system configurations, each of which may need extensive quality assurance (QA) to validate. Consequently, a key challenge for developers of reusable performance-intensive software involves managing variability effectively in the face of an exploding *software configuration space*.

As software configuration spaces increase in size and software development resources decrease, it becomes infeasible to handle all QA activities in-house. For instance, developers may not have access to all the hardware, OS, and compiler platforms on which their reusable software artifacts will run. Moreover, due to time-to-market driven environments, developers may be forced to release their software in configurations that have not been subjected to sufficient QA. The combination of an enormous configuration space and severe development resource constraints therefore often force developers of reusable software to make design and optimization decisions without precise knowledge of their consequences in fielded systems.

Solution approach → Distributed continuous QA processes (DCQA). To manage this situation, we have initiated the **Skoll** (`www.cs.umd.edu/projec\ -ts/skoll`) project to develop tools and processes necessary to carry out "around-the-world, around-the-clock" QA. Our feedback-driven Skoll approach divides QA processes into multiple subtasks that are intelligently and continuously distributed to, and executed by, a grid of computing resources contributed by end-users and distributed development teams around the world. The results of these executions are returned to central collection sites where they are fused together to identify defects and guide subsequent iterations of the QA process.

Skoll QA processes are based on a client/server model. Clients distributed throughout the Skoll grid request *job configurations* (implemented as QA subtask scripts) from a Skoll server. The Skoll process is carried out as shown in Figure 1[4]. At a high level, the Skoll process is carried out as shown in Figure 1.

1. Developers create the configuration model and adaptation strategies. Developers create the generic QA subtask code that will be specialized when creating actual job configurations.

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

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

[4] A comprehensive discussion of Skoll components and infrastructure appears in [1].

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 it.
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 operators to adapt the global process.

Model-Integrated DCQA techniques Earlier work on Skoll described the structure and functionality of Skoll and presented results [1] from a feasibility study that applied Skoll tools and processes to ACE [2] and TAO [3]. The initial Skoll prototype provided a DCQA infrastructure that performed functional testing, but did not address QoS issues, nor did it minimize the cost of implementing QA subtasks. In particular, integrating new application capabilities into the Skoll infrastructure (such as benchmarks that quantified various QoS properties) required developers to write test cases manually. Likewise, extending the configuration models (*e.g.*, adding new options) required the same tedious and error-prone approach.

This paper describes several previously unexamined dimensions of Skoll: *integrating model-based techniques with distributed continuous QA processes*, *improving quality of service (QoS) as opposed to functional correctness*, and *using Skoll to empirically optimize a system for specific run-time contexts*. At the heart of the Skoll work presented in this paper is **CCMPerf** [4], which is an open-source toolsuite[5] that applies generative model-based techniques [5] to measure and optimize the QoS of reusable performance-intensive software configurations. Currently, CCMPerf in concert with Skoll, focuses on evaluating QoS of implementations of the CORBA Component Model (CCM)[6], as shown in Figure 2.

Paper organization The remainder of this paper is organized as follows: Section 2 motivates and describes the design of CCMPerf, focusing on its model-based generative benchmarking capabilities; Section 3 describes a case-study that illustrates how QoS characteristics captured using CCMPerf can be captured and fed back into models to analyze system behavior at model construction time; Section 4 examines related work and compares it with the approaches used in Skoll and CCMPerf; Section 5 presents concluding remarks and future work.

2 Enhancing Skoll with a Model-based QoS Improvement Process

Reusable performance-intensive software is often used by applications with stringent quality of service (QoS) requirements, such as low latency and bounded jitter. The QoS of reusable performance-intensive software is influenced heavily

[5] CCMPerf can be downloaded from `www.dre.vanderbilt.edu/cosmic`
[6] We focus on CCM since it is standard component middleware targeted for QoS requirements for DRE systems

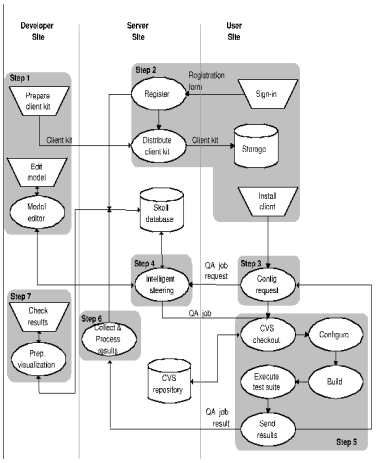

Fig. 1. Skoll QA Process View

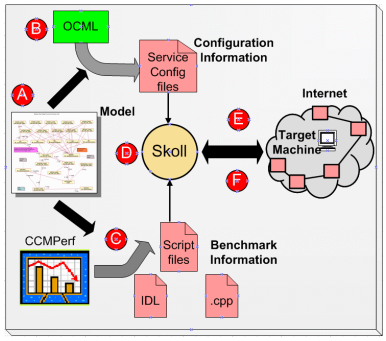

Fig. 2. Skoll QA Process View with CCM-Perf Enhancements

by factors such as (1) the configuration options set by end-users to tune the underlying hardware/software platform and (2) characteristics of the underlying platform itself. Managing these variable platform aspects effectively requires a QA process that can precisely pinpoint the consequences of mixing and matching configuration options on various platforms.

In the initial Skoll approach, creating a benchmarking experiment to measure QoS properties required QA engineers to write (1) the header files, source code, that implement the functionality, (2) the configuration and script files that tune the underlying ORB and automate running tests and output generation, and (3) project build files (*e.g.*, makefiles) required to generate the executable code. Our experience during our initial feasibility study [1] revealed how this process was tedious and error-prone. The remainder of this section describes how we have applied model-based techniques [5] to improve this situation. These improvements are embodied in CCMPerf [4], a model-based benchmarking toolsuite.

2.1 Model-based Tools for Performance Improvement

With CCMPerf, QA engineers graphically model possible interaction scenarios. For example, Figure 3 presents a model that shows an association between a facet[7] and an IDL interface. It also shows the domain-specific building blocks (such as receptacles, event sources, and event sinks) allowed in the models. Given a model, CCMPerf generates the scaffolding code needed to run the experiments. This typically includes Perl scripts that start daemon processes, spawn the component server and client, run the experiment, and display the required results.

[7] A facet is a specialized port a CCM component exposes for clients to communicate with the component.

CCMPerf is built on top of the Generic Modeling Environment (GME) [6], which provides a meta-programmable framework for creating domain-specific modeling languages and generative tools. GME is programmed via *meta-models* and *model interpreters*. The meta-models define modeling languages called paradigms that specify allowed modeling elements, their properties, and their relationships. Model interpreters associated with a paradigm can also be built to traverse the paradigm's modeling elements, performing analysis and generating code. CCMPerf was developed by creating the following two paradigms:

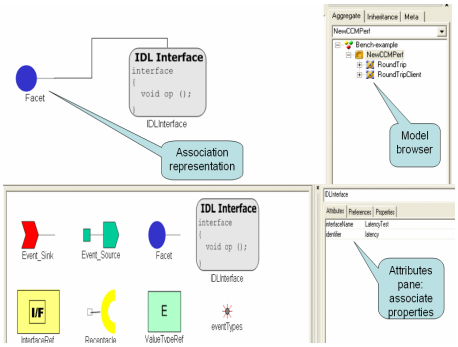

Fig. 3. CCMPerf modeling paradigm

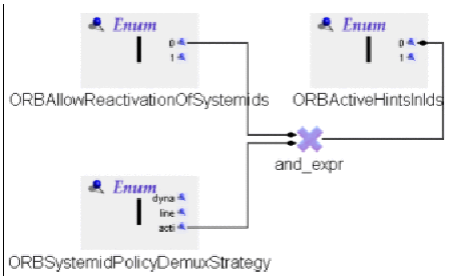

Fig. 4. OCML Modeling Paradigm

The Options Configuration Modeling Language (OCML) [7] paradigm models the various configuration parameters provided by ACE, TAO, and CIAO. Figure 4 shows ORB configuration options modeled with OCML. Inter-option dependencies are captured as model constraints. For instance, Figure 4 shows that when the demultiplexing option is set to *active demultiplexing* and the reactivation of system ids is disabled, the use of active hints is disallowed. Specific OCML models are then used to automatically generate large portions of Skoll's QA subtask code.

The Benchmark Generation Modeling Language (BGML) [4] paradigm models (1) how clients and servers interact with each other and (2) represents metrics that can be applied to specific configuration options and platforms. Figure 3 illustrates the BGML modeling paradigm.

We also developed one model interpreter for each of CCMPerf's two paradigms. The OCML model interpreter generates configuration files to configure the underlying middleware. For ACE, TAO, and CIAO these configuration files conform to a specified format that can be parsed by the middleware. The BGML model interpreter parses the model and synthesizes the code required to benchmark the modeled configuration.

2.2 Integrating CCMPerf into the Skoll QA Process

Figure 2 presents an overview of how we have integrated CCMPerf with the existing Skoll infrastructure.
A. A QA engineer defines a test configuration using CCMPerf models. The necessary experimentation details are captured in the models, *e.g.*, the ORB configuration options used, the IDL interface exchanged between the client and the server, and the benchmark metric performed by the experiment.
B & C. The QA engineer then uses CCMPerf to interpret the model. The OCML paradigm interpreter parses the modeled ORB configuration options and generates the required configuration files to configure the underlying ORB. The CCMPerf paradigm interpreter then generates the required benchmarking code, *i.e.*, IDL files, the required header and source files, and necessary script files to run the experiment. Steps A, B, and C are integrated with Step 1 of the Skoll process described in Section 1.
D. When users register with the Skoll infrastructure they obtain the Skoll client software and configuration template. This step happens in concert with Step 2, 3, and 4 of the Skoll process.
E & F. The client executes the experiment and returns the result to the Skoll server, which updates its internal database. When prompted by developers, Skoll displays execution results using an on demand scoreboard. This scoreboard displays graphs and charts for QoS metrics, *e.g.*, performance graphs, latency measures and foot-print metrics. Steps E and F correspond to steps 5, 6, and 7 of the Skoll process.

3 Feedback-driven, Model-integrated Skoll: A Case Study

Study motivation and design. Measuring QoS for a highly configurable system such as CIAO involves capturing and analyzing system performance in terms of throughput, latency, and jitter across many different system configurations running on a wide range of hardware, OS, and compiler platforms. We treat this problem as a large-scale scientific experiment, *i.e.*, we rely on design of experiments theory to determine which configurations to examine, how many observations to capture, and the techniques needed to analyze and interpret the resulting data. We use the CCMPerf modeling tools presented in Section 2 to model configuration parameters and generate benchmarking code that measures and analyzes the QoS characteristics. Using the collected data, we derive two categories of information: (1) *platform-specific*, whose behavior differs on particular platforms and (2) *platform-independent*, whose behavior is common across a range of platforms. This information can then be fed-back into the models to specify QoS characteristics at model construction time.

To make this discussion concrete, we present a simple example of our approach (in production systems these experiments would be much larger and more complex). This experiment measures only one aspect of QoS: *round-trip throughput calculated at the client side as the number of events processed/sec.* We then use the OCML paradigm described in Section 2.1 to model the software

configuration options that set the request processing discipline within the ORB. All other options are simply set to their default values.

For this study, we modeled a leader/followers [8] request processing approach, where a pool of threads take turns demultiplexing, dispatching, and processing requests via a thread-pool reactor (`TP_Reactor`) [8]. In this scheme the following two configuration parameters can be varied to tune the QoS characteristics:

- The **thread-pool size** determines the number of threads in the ORB's `TP_Reactor` that will demultiplex, dispatch, and process requests and
- The **Scheduling policy** determines how the threads take turns in processing requests. We use two scheduling policies for the experiment: (1) *FIFO scheduling*, where the thread that enters the queue first processes the request first, and (2) *LIFO scheduling*, where the thread entering last processes the request first.

Execution testbed. We chose the following four testbeds with varying hardware, OS, and compiler configurations. Table 1 summarizes the key features of these four testbed platforms.

	DOC	**ACE**	**Lindy**	**Tango**
CPU Type	AMD Athlon	AMD Athlon	Intel Xeon	Intel Xeon
Speed (GHz)	2	2	2.4	1.9
Cache (KB)	512	512	1024	2048
Compiler (gcc)	3.2.2	3.3	3.3.2	3.3.2
OS (Linux)	Red Hat 9	Red Hat 8	Fedora Core I	Debian

Table 1. Testbed Summary

Study execution. To identify the influence of scheduling policy and thread pool size on round-trip throughput, we conducted the experimental task that used two components communicating with each other. The Skoll system next distributed the experimental tasks to clients running on the four platforms. Each task involved 250,000 iterations. Skoll continued distributing the tasks until the entire experimental design was completed. For example, on the machine called DOC (See Table 1), different clients and servers executed every combination of FIFO and LIFO policies with the number of request processing threads set to 2 and 4. Each combination can be categorized as a tuple (t_1, t_2), where t_1 denotes number of threads used on the client and t_2 number of threads used on the server.

Study analysis. The Skoll infrastructure provides the framework for automatically collecting data that can then be analyzed to glean platform-specific and platform-independent information. The experimental results shown in Figure 5 were obtained by using Skoll to run experiments on each platform described in Table 1. Our analysis of the results in Figure 5 yielded the following observations:

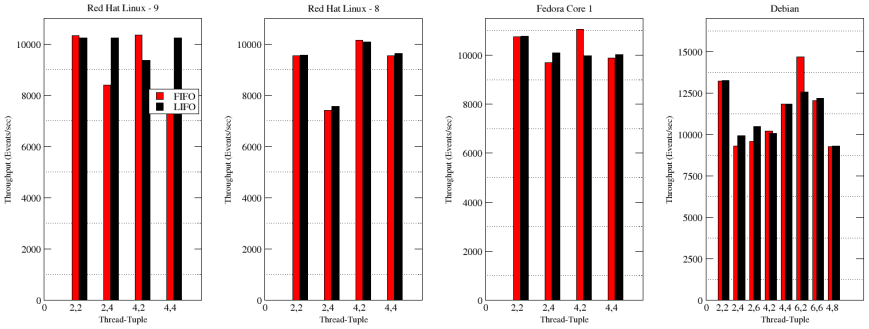

Fig. 5. Results Summary for Study Execution

Machine	Tuple	FIFO	LIFO
		Throughput (events/sec)	
DOC	(4,2)	10,470	9354
ACE	(4,2)	10,152	10,087
Lindy	(4,2)	11,057	9982
Tango	(4,2)	10,196	10,067
	(6,2)	14,696	12,556

Table 2. Platform-specific Information

• **Observation 1.** On average, LIFO scheduling yielded higher throughput than FIFO scheduling strategy. Our analysis suggests that this occurs because the LIFO strategy increases cache thread affinity, leading to the cache lines not being invalidated after every request. This observation holds for all the platforms we conducted the experiment, though it is obviously influenced by the underlying hardware, OS and compiler platforms. Assuming our testbed platforms as a complete configuration space, this information is *platform-independent*, *i.e.*, the LIFO strategy yields higher throughput when the leader/followers model for request processing is chosen. Such information helps developers understand the first-order effects of different configuration options.

• **Observation 2.** While observation 1 holds as the general case, we detected finer effects, as shown in Table 2. Also, for specific test cases, FIFO produces higher throughput than LIFO. In particular, whenever the number of server threads is low (*i.e.*, 2) FIFO performs as well or better than LIFO. Moreover, the degree of improvement increases as the number of client threads increases. This *platform-specific* information holds only for certain configurations in our configuration space, which consists of the hardware, OS, compiler and software configuration options.

Lessons learned. Although this feasibility study was purposely simplified, it indicates how the model-integrated Skoll framework enables more powerful ex-

periments that help identify performance bottlenecks and provide general guidelines to developers and users of software systems. The platform specific and independent information help in developing re-usable configurations that maximize QoS for a given operational context. The formal designed experiment illustrated how our Skoll framework can be used to codify these re-usable configuration solutions. These configurations when validated across a range of hardware, OS and compiler platforms represents a *Configuration & Customization (C&C)* [9] pattern.

4 Related Work

This section compares our work on model-driven performance evaluation techniques in Skoll and CCMPerf with other related research efforts that use empirical data and mathematical models to identify performance bottlenecks. For example, the ATLAS [10] numerical algebra library uses an empirical optimization engine to decide the values of optimization parameters by generating different program versions that are run on various hardware/OS platforms. The output from these runs are used to select parameter values that provide the best performance. Mathematical models are also used to estimate optimization parameters based on the underlying architecture, though empirical data is not fed into the models to refine it.

Like ATLAS, CCMPerf uses an optimization engine to configure/customize middleware parameters in accordance to available OS platform characteristics (such as the type of threading, synchronization, and demultiplexing mechanisms) and characteristics of the underlying hardware (such as the type of CPU, amount of main memory, and size of cache). CCMPerf enhances ATLAS, however, by feeding back platform-specific information into the models to identifying performance bottlenecks at model construction time. This information can be used to select optimal configurations ahead of time that maximize QoS behavior.

Other research initiatives to validate if software components meet QoS, include automatic validation techniques [11] for J2EE components using Aspects. In this approach, agents at run-time, conduct validation tests such as, single-client response time, functional operation and data storage/ retrieval tests. Our approach on Skoll, differs from run-time validation as all our testing is done offline. Using our approach, no cost is incurred at deployment time for the component. Further, extensive testing and QoS behavior analysis is done on a range of hardware, OS and compiler platforms to identify performance bottlenecks by modeling the operational context and synthesizing scaffolding code. The results are then used to select optimal configuration at deployment time rather than incur overhead of run-time monitoring.

5 Concluding Remarks

Reusable software for performance-intensive systems increasingly has a multitude of configuration options and runs on a wide variety of hardware, com-

piler, network, OS, and middleware platforms. Our work on Skoll addresses two key dimensions of applying distributed continuous QA processes to reusable performance-intensive software. The Skoll framework described in [1] address software functional correctness issues, *e.g.*, ensuring software compiles and runs on various hardware, OS, and compiler platforms. The CCMPerf tools described in this paper address software QoS issues, *e.g.*, modeling and benchmarking interaction scenarios on various platforms by mixing and matching configuration options. These model-based QA techniques enhance Skoll by allowing developers to model configuration/interaction aspects and associate metrics to benchmark the interaction. These techniques also minimize the cost of testing and profiling new configurations by moving the complexity of writing error-prone code from QA engineers into model interpreters, thereby increasing productivity and quality. Model-based tools such as CCMPerf simplify the work of QA engineers by allowing them to focus on domain specific details rather than write source code.

References

1. A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. C. Schmidt, and B. Natarajan, "Skoll: Distributed Continuous Quality Assurance," in *Proceedings of the 26th IEEE/ACM International Conference on Software Engineering*, (Edinburgh, Scotland), IEEE/ACM, May 2004.
2. D. C. Schmidt and S. D. Huston, *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*. Boston: Addison-Wesley, 2002.
3. D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.
4. A. S. Krishna, J. Balasubramanian, A. Gokhale, D. C. Schmidt, D. Sevilla, and G. Thaker, "Empirically Evaluating CORBA Component Model Implementations," in *Proceedings of the OOPSLA 2003 Workshop on Middleware Benchmarking*, (Anaheim, CA), ACM, Oct. 2003.
5. J. Sztipanovits and G. Karsai, "Model-Integrated Computing," *IEEE Computer*, vol. 30, pp. 110–112, Apr. 1997.
6. A. Ledeczi, A. Bakay, M. Maroti, P. Volgysei, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing Domain-Specific Design Environments," *IEEE Computer*, Nov. 2001.
7. A. Gokhale, "Component Synthesis using Model Integrated Computing." `www.dre.vanderbilt.edu/cosmic`, 2003.
8. D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. New York: Wiley & Sons, 2000.
9. S. Berczuk and B. Appleton, *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Addison-Wesley, 2003.
10. Kamen Yotov and Xiaoming Li and Gan Ren et.al, "A Comparison of Empirical and Model-driven Optimization," in *Proceedings of ACM SIGPLAN conference on Programming Language Design and Implementation*, June 2003.
11. J. Grundy and G. Ding, "Automatic Validation of Deployed J2EE Components Using Aspects," in *17^{t}h International Conference on Automated Software Engineering, Linz Austria*, IEEE, Sept. 2002.