Techniques and Processes for Improving the Quality and Performance of Open-Source Software

Adam Porter, Cemal Yilmaz, Atif M. Memon

University of Maryland
College Park, MD

Arvind S. Krishna, Douglas C. Schmidt, Aniruddha Gokhale

Vanderbilt University
Nashville, TN

Abstract

Open-source development processes have emerged as an effective approach to reduce cycle-time and decrease design, implementation, and quality assurance costs for certain types of software, particularly systems infrastructure software, such as operating systems, compilers and language processing tools, text and drawing editors, and middleware. This paper presents two contributions to the study of open-source software processes. First, we describe key challenges of open-source software and illustrate how quality assurance (QA) processes – specifically those tailored to open-source development – help mitigate these challenges better than traditional closed-source processes do. Second, we summarize results of empirical studies that evaluate how our Skoll distributed continuous quality assurance (DCQA) techniques and processes help to resolve key challenges of developing and validating open-source software. Our results show that: (1) using models to configure and guide the DCQA process improves developer understanding of open-source software, (2) improving the diversity of platform configurations helps QA engineers find defects missed during conventional testing, and (3) centralizing control of QA activities helps to eliminate redundant work.

Keywords: Distributed Continuous Quality Assurance, Model-driven Approaches, Skoll, and Software Quality

1. Introduction

1.1. Enablers of Open-Source Success

Over the past decade, open-source development processes [O'Reilly98] have demonstrated their ability to reduce cycle-time and decrease design, implementation, and quality assurance (QA) costs for certain types of software, particularly infrastructure software, such as operating system (OS) platforms, web servers, middleware, text and language processing tools, and system/network support tools. These projects generally exhibit common properties. First, they are *general-purpose, commoditized systems infrastructure software*, whose requirements and APIs are well known. For example, the requirements and APIs for Linux, Apache, C/C++ compilers/linkers, and JBoss/CORBA middleware are well understood, so less time and effort is needed for upstream software development activities, such as requirements analysis or interface specifications. Second, they *service communities whose software needs are unmet* by, or are economically unappealing to, mass-market software providers. For example, the Linux-based Beowulf clusters and the Globus middleware for Grid computing were developed in the scientific computing community in part because their high-end computing applications run on specialized platforms that were not the focus of the traditional mass-market desktop and server suppliers. In addition, they are often *applied by relatively sophisticated user communities*, who have considerable software development skills and knowledge of development tools (such as debuggers, configuration managers, bug tracking systems, memory leak/validation tools, and performance profilers) and collaboration mechanisms (such as web/ftp-sites, mailing lists, NetMeeting, and instant messaging). When these users encounter bugs or software configuration problems they can often identify/fix the problems and submit patches.

From a software perspective, the open-source projects outlined above have generally succeeded for the reasons described below.

1.1.1. Scalable division of labor. Brooks' Law states that "adding developers to a late project makes it later" [Brooks75]. Open-source projects often exploit a "loophole" in this law, namely that software debugging and QA productivity *do* scale up as headcount increases because the more people testing the code, the more defects will be detected. All other things being equal, therefore, we would thus expect a team of 1,000 testers to find many more defects than a team of 10 testers, a phenomenon referred to in the open-source community as "to enough eyeballs, all bugs are shallow" [Raymond98]. QA activities also scale well since they do not require as much inter-personal communication as software development (particularly analysis and design activities).

To leverage the scalability of open-source QA processes, successful projects are often organized into a "core" and "periphery" structure where a relatively small number of core developers ensure the architectural integrity of the project, e.g., vetting user contributions and bug fixes, adding new features and capabilities, and tracking day-to-day progress on project goals and tasks. In contrast, the periphery consists of the hundreds or thousands of user community members who help test and debug the software released periodically by the core team. In some

open-source projects, the distinctions between core and periphery are informal and fluid, e.g., participants can play different roles at different times [Mockus02].

1.1.2. Short feedback loops between the core and the periphery is one reason for the success of well-organized, large-scale open-source development efforts, such as the Linux OS, Apache web server, and the ACE+TAO middleware. In these systems, for example, it often takes just a few minutes or hours to detect a bug at the periphery, report it to the core, and receive an official patch from a core developer [Mockus00, Osterlie03]. Moreover, the use of Internet-enabled configuration management tools, such as the GNU Concurrent Versioning System (CVS) or Subversion, allows open-source users in the periphery to resynchronize quickly with updates and fixes supplied by the core.

1.1.3. Effective leverage of user community expertise and computing resources. In today's time-to-market-driven economy, few software providers can afford long QA cycles. As a result, nearly everyone who uses a computer – particularly software application developers – is effectively a beta-tester of software that was shipped before all defects could be identified and removed. In traditional closed-source/binary-only software deployment models, these premature release cycles yield frustrated users, who have little recourse other than grousing when problems arise. Since they are often limited to little more than finding workarounds for problems they encounter, they may have little incentive to help improve closed-source products.

In contrast, open-source development leverages expertise in their communities, allowing users and developers to collaborate to improve software quality [Lakhani03]. For example, short feedback loops encourage users to help with the QA process since they are "rewarded" by rapid fixes. Moreover, since the source code is available, users at the periphery can often either fix bugs directly or can provide concise test cases that help isolate problems quickly. User efforts can therefore greatly magnify the debugging and computing resources available to an open-source project, which can improve software quality if harnessed effectively and combined with automated tools for tracking defect reports, such as Bugzilla and JIRA**.**

1.1.4. Inverted stratification of available expertise. In many organizations, testers and QA engineers are perceived to have lower status than software developers. In contrast, open-source development processes often invert this stratification so that the "testers" in the periphery are often excellent software developers who apply their debugging skills when they encounter problems with the open-source software base. The open-source model thus makes it possible to leverage the talents of these developers, who ordinarily do not work as testers in traditional software organizations.

1.1.5. Greater opportunity for analysis and validation. Open-source development techniques can help improve software quality by enabling the use of powerful analysis and validation techniques, such as whitebox testing and model checking, that require access to the source code. For example, [Zeller02] and [Michail05] have employed static analysis and testing techniques to find bugs in open-source software.

In general, traditional closed-source/binary-only software development and QA processes rarely achieve the benefits outlined above as rapidly or as cost effectively as open-source processes.

1.2. Problems with Current Open-Source Processes

Open-source projects have had great success in the systems infrastructure software domains, as described in Section 1.1. Our experience working on many open-source projects [GPERF90, TAO98, JAWS99, ACE01, CoSMIC04] for the past two decades has shown, however, that the open-source development model can create significant problems in maintenance and evolution:[1]

Problem 1: Hard to maintain software quality in the face of short development cycles. The goals of open-source software development are not unique, i.e., limit regression errors to avoid breaking features or degrading performance relative to prior releases, sustain end-user confidence and good will, and minimize development and QA costs. It can be hard, however, to ensure consistent quality of open-source software due to the short feedback loops between users and core developers, which typically result in frequent "beta" releases, e.g., several times a month. Although this schedule satisfies end-users who want quick patches for bugs they found in earlier betas, it can be frustrating to other end-users who want more stable, less frequent software releases. In addition to our own experiences, [Gamma05] describes how the length of the release cycles in the Eclipse framework affected user participation and eventually the quality of the software.

Problem 2: Lack of global view of system constraints. Large-scale open-source projects often have a large number of contributors from the user community (i.e., the periphery). When these users encounter problems, they may examine the source code, propose/apply fixes locally, and then submit the results back to the core team for possible integration into the source base. Often these users in the periphery have much less knowledge of the entire architecture of an open-source software system

[1] More discussions of failed open-source projects are available at www.isr.uci.edu/research-open-source.html and www.infonomics.nl/FLOSS/report.

than the core developers. As a result, they may lack a global view of broader system constraints that can be affected by any given change, so their suggested fixed may be inappropriate.

Problem 3: Unsystematic and redundant QA activities. Many popular open-source projects (such as GNU GCC, CPAN, Mozilla, the Visualization Toolkit, and ACE+TAO) distribute regression test suites that end-users can run to evaluate the success of an installation on a user's platform. Users can – but frequently do not – return the test results to project developers. Even when results are returned to core developers, however, the testing process is often undocumented and unsystematic, e.g., core developers have no record of what configurations were tested, how they was tested, or what the results were, which loses crucial QA-related information. Moreover, many QA configurations are executed redundantly by thousands of users (e.g., on popular versions of Linux or Windows), whereas others are never executed at all (e.g., on less widely used operating systems).

Problem 4: Lack of diversity in test environments. Well-written open-source software (e.g., based on GNU auto-conf) can be ported easily to a variety of OS and compiler platforms. In addition, since the source is available, end-users can modify and adapt their source base readily to fix bugs quickly or to respond to new market opportunities with greater agility. Support for platform-independence, however, can yield the daunting task of keeping an open-source source software base operational despite continuous changes to the underlying platforms. In particular, since developers in the core may only have access to a limited number of OS/compiler configurations, they may release code that has not been tested thoroughly on all platform configurations on which users want to run the software.

Problem 5: Manually intensive execution of QA processes. The availability of source code often encourages an increase in the number of options for configuring and subsetting the software at compile- and run-time. Although this flexibility enhances the software's applicability for a broad range of use cases, it can also exacerbate QA costs due to a combinatoric increase in the QA space. Moreover, since open-source projects often run on a limited QA budget due to their minimal/non-existent licensing fees, it can be hard for core developers to validate and support large numbers of versions and variants simultaneously, particularly when regression tests and benchmarks are written and run manually.

As open-source software systems evolve, the problems outlined can compound, resulting in systems that are defective, bloated, and hard to maintain. For example, the inability to regression test a broad range of potential configurations increases the probability that certain configurations of features/options may break in new releases, thereby reducing end-user confidence/adoption and increasing subsequent development and QA costs. Without remedial action, therefore, open-source user communities may become smaller (due to frustration with software quality) until the software falls into disuse. This paper describes techniques and tools for addressing these types of problems, so it becomes easier to deliver on the promise of open-source processes.

2. Addressing Open-Source Challenges with Skoll

To address the problems with open-source software development described in Section 1.2, we have developed the Skoll *distributed continuous quality assurance* (DCQA) environment [Memon04]. Skoll provides tools for creating and validating novel software QA processes that leverage the extensive computing resources of worldwide user communities in a distributed, continuous manner to rapidly improve software quality. In particular, Skoll provides an integrated set of tools that run coordinated QA activities around-the-world, around-the-clock on a virtual computing grid provided by user machines during off-peak hours to (1) detect and resolve QA problems quickly and (2) automatically analyze and enhance key system QoS characteristics on a diverse range of system configurations and platforms.

To enhance the detection and resolution of QA problems, Skoll closes the loop from users back to developers by exploiting the inherently distributed nature of open-source user communities. In particular, users at various sites in the periphery use Skoll tools to perform a portion of the overall testing, thereby offloading the number of versions that must be maintained by core developers, while enhancing user confidence in new (beta) versions of open-source software. Each user/site participating in Skoll-guided DCQA processes conducts different instrumentations and performance benchmarks automatically to collect metrics, such as measures of memory footprint,

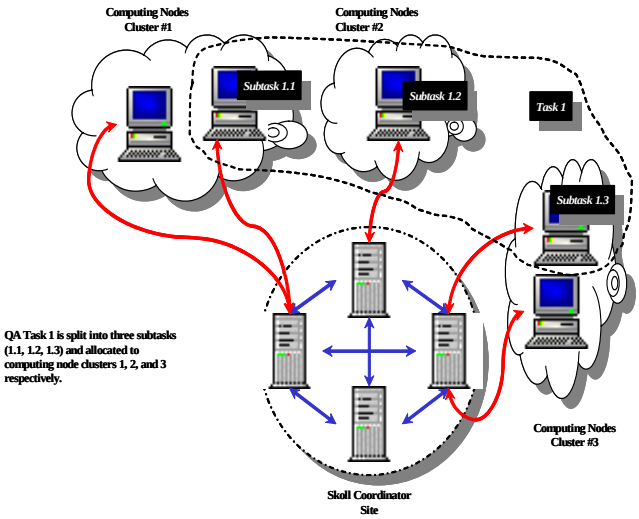

Fig. 1: Skoll Tasks/Subtasks Allocated to Computing Nodes

throughput, latency, and jitter.

Skoll's DCQA processes are (1) *distributed*, i.e., a given QA task is divided into several subtasks that can be performed on a single user machine, (2) *opportunistic*, i.e., when a user machine becomes available, one or more subtasks are allocated to it and the results are collected and fused together at central collection sites to complete the overall QA process, and (3) *adaptive*, i.e., earlier subtask results are used to schedule and coordinate future subtask allocation. Skoll leverages important open-source project assets, such as the technological sophistication and extensive computing resources of worldwide user communities, open access to source, and ubiquitous web access, to improve the quality and performance of open-source software by automating the division of labor to make those 'thousands of eyeballs' more effective. Figure 1 illustrates how a QA task (Task 1) can be decomposed into three subtasks (subtasks 1.1, 1.2, and 1.3), which are allocated to and executed on computing node clusters. As subtasks run, Skoll's control logic may dynamically steer the global QA computations to enhance their performance and accommodate various platform constraints, such as type of OS/compiler platform, amount of disk space and CPU load available for QA tasks, etc.

At a high level, a Skoll-based DCQA process (shown in Figure 2) runs as follows:

Step 1. Developers create a *configuration and control model* (Section 3.2) and *adaptation strategies* (Section 3.3) for the open-source software. An *intelligent steering agent* (ISA) uses automated AI planning algorithms [Memon01a] to translate the model into planning operators. Developers create generic QA subtask code that will be specialized when creating *job configurations*, which consist of the code artifacts, configuration parameters, build instructions, and QA-specific code (e.g., developer-supplied regression and performance tests) associated with a software project.

Step 2. A user registers with a *Skoll server* to receive *Skoll client software*. The user receives the Skoll client software and a *configuration template*. The user can modify the configuration template to change option settings temporarily or constrain specific options.

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

Step 4. The Skoll server queries its databases and the user-provided configuration template to determine which option settings are fixed for that user and which can be set by the ISA. It then packages this information as a planning goal and queries the ISA, which generates a plan, creates the job configuration, and returns it to the client.

Step 5. The Skoll client invokes the job configuration and returns the results to the Skoll server.

Step 6. The Skoll server examines these results and invokes the designated adaptation strategies, which update the ISA operators to adapt the global process. Skoll's adaptation strategies use built-in statistical analyses that help developers quickly identify large subspaces in which QA subtasks have failed (or performed poorly).

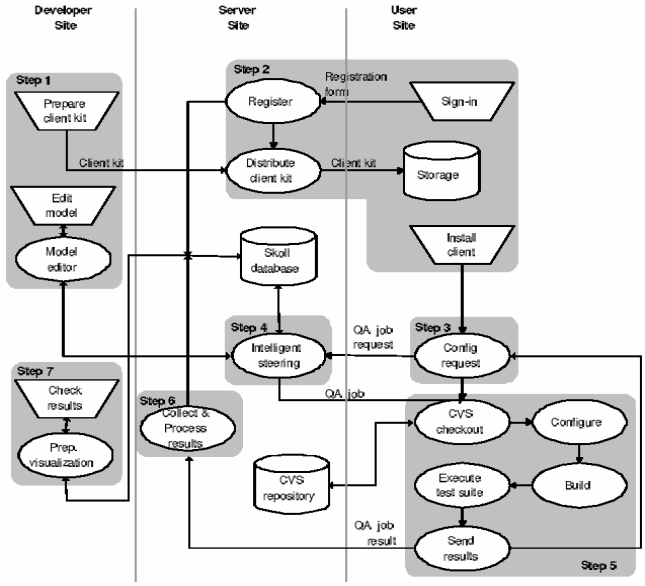

Figure 2: Skoll in Action

Step 7. Periodically, e.g., when prompted by developers or at the end of a QA session, Skoll servers prepare a visualization of their subtask results and the current state of the overall process.

Our earlier work on DCQA environments [Memon04, Yilmaz05, PSA04] has described various techniques supported by Skoll, including (1) the creation of a model of all possible software configurations, (2) division of a QA task into subtasks that can execute independently on end-user machines, (3) collection of subtask execution results and their interpretation, (4) adaptation of the overall process based on incremental QA results, (5) fault classification techniques to help pinpoint the source of errors, and (6) visualization of QA task results. This paper extends our prior work by showing how we applied Skoll to a large-scale open-source project to mitigate the problems described in Section 1.2.

3. Overview of ACE and TAO

We applied Skoll on ACE [ACE01] and TAO [TAO98], which are widely used open-source middleware platforms (www.dre.vanderbilt.edu). ACE is an object-oriented framework containing hundreds of classes that implement key patterns and frameworks for distributed real-time and embedded (DRE) systems. TAO is an im-

plementation of the Real-time CORBA specification [OMG02] that uses many frameworks and classes in ACE to meet the demanding quality of service (QoS) requirements in DRE systems. These middleware platforms allow applications to interoperate across networks without hard-coding dependencies on their location, programming language, operating system platform, communication protocols and interconnects, and hardware characteristics.

ACE and TAO were ideal study candidates for the Skoll DCQA environment since they share the following characteristics – as well as problems – with other large-scale open-source projects, such as Linux, Apache, and the GNU compiler and language processing tools:

- **Large and mature source code base**. The ACE+TAO source base contains over two million source lines of C++ middleware systems source code, examples, and regression tests split into over 9,800 files as follows:

Software Toolkit	Source Files	Source Lines of Code
ACE	2,016	448,974
TAO	7,824	1,755,789
Total	9,840	2,204,763

- **Heterogeneous platform support**. ACE+TAO runs on dozens of platforms, including most POSIX/UNIX variants, all versions of Microsoft Win32, many real-time and embedded operating systems, MVS OpenEdition, and Cray. These platforms change over time, e.g., to support new features in the C++ standard and newer versions of the operating systems.
- **Highly configurable**, e.g., numerous interdependent options supporting a wide variety of program families [Parnas79] and standards. Common examples of different options include multi-threaded vs. single-threaded configurations, debugging vs. release versions, inlined vs. non-inlined optimized versions, and complete ACE vs. ACE subsets. Examples of different program families and standards include the baseline CORBA 3.0 specification, Minimum CORBA, Real-time CORBA, CORBA Messaging, the Lightweight CORBA Component Model, and many different variations of CORBA services.
- **Core development team**. ACE+TAO are maintained by a core – yet geographically distributed – team of ~20 developers. While many of these core developers have worked on ACE+TAO for several years, there is also a continual influx of developers into and out of the core.
- **Comprehensive source code control and bug tracking**. The ACE and TAO source code resides in a CVS repository, hosted at the Institute for Software Integrated Systems (ISIS) at Vanderbilt University. External read-only access is available to the ACE+TAO user community via the Web. Write access is granted only to certain group members and trusted external contributors. Software defects are tracked using Bugzilla, which is a Web-based tool that helps ACE+TAO developers resolve problem reports and other issues.
- **Large and active user community**. Over the past decade ACE+TAO have been used by more than 20,000 application developers, who work for thousands of companies in dozens of countries around the world. Since ACE and TAO are open-source systems, changes are often made and submitted by users in the periphery who are not (initially) part of the core development team.
- **Continuous evolution**, i.e., ACE+TAO have a dynamically changing and growing code base that has hundreds of CVS repository commits per week. Although the interfaces of the core ACE+TAO libraries are relatively stable, their implementations are enhanced continually to improve correctness, time/space overhead, user convenience, portability, safety, and another desired aspects. The ACE+TAO software distributions also contain many examples, standalone applications, and tests for functionality and performance. These artifacts change more frequently than the main ACE+TAO libraries, and are often not as thoroughly tested on all the supported platforms.
- **Frequent beta releases and periodic "stable" releases.** Beta releases contain bug fixes and new features that are lightly tested on the platforms that the core ACE+TAO team uses for their daily development work. The usual interval between beta releases averages around every two to three months. In contrast, the "stable" versions of ACE+TAO are released less frequently, e.g., once a year, and are tested extensively on all the OS and compiler platforms to which ACE+TAO have been ported. The stable releases are supported commercially by over half a dozen companies worldwide.

Despite the broad use of ACE+TAO, this open-source effort suffers from the problems discussed in Section 1.2.

4. Applying Skoll to ACE+TAO

This section describes how we have applied DCQA processes using Skoll to address the problems with open-source processes discussed in Section 1.2. To make the discussion concrete, we focus applying Skoll to improve the quality and performance of ACE+TAO by developing a scalable QA process based on continuous testing and profiling. Although we describe these problems in the

context of ACE+TAO, they are issues for many other large-scale open-source projects.

4.1. Ensuring Software Quality in the Context of Short Development Cycles

One reason why ACE+TAO are widely used in both commercial and research projects is that they are customizable to many different runtime contexts, i.e., they have hundreds of features/options that can be enabled/disabled for application-specific use cases. In the context of short "beta-driven" development cycles, however, the core ACE+TAO developers cannot test all possible configurations because there are simply not enough people, OS/compiler, platforms, CPU cycles, or disk space in house to run the hundreds of ACE+TAO regression tests over the combinatorial number of configurations in a timely manner. As a result, some parts of the middleware are released untested, which greatly increases the probability that certain configurations of features/options may break in new releases, thereby reducing end-user confidence and increasing subsequent development and QA costs.

To mitigate this problem we designed a Skoll DCQA process that runs automated regression tests continuously across a grid of external computing resources. These tests include ~100 ACE tests and ~250 TAO tests that serve several purposes, including (1) *user acceptance and assurance*, which involves building and testing the ACE and/or TAO libraries on a wide variety of different platforms to validate the integrity of the builds and any assumptions made about the operating platform and (2) *smoke testing*, where build/test scripts run a varying subset of the ACE+TAO regression test suite whenever developers commit their changes to the CVS repository. The Skoll DCQA process is currently running on 60+ workstations and servers at over a dozen sites around the world (see www.dre.vanderbilt.edu/scoreboard for a summary). This parallelization of the DCQA process allows much more work to be done in a shorter time frame.

Each full build and test ranges can take anywhere from three hours on quad-CPU Linux machines to 8 or more hours on less powerful machines. Given the size of the configuration space and the shortness of the development cycle, we try to improve efficiency by adapting Skoll's DCQA process to incoming test results. One adaptation strategy is called *nearest neighbor search*, where when a configuration fails all configurations that differ from it in the setting of exactly one option are scheduled for testing with the highest priority to quickly identify sets of similar configurations that both pass and fail. This information is then fed to classification tree algorithms [Porter91] that (1) build models of the specific option settings that may be causing the failures and (2) summarize the large volumes of data into feedback that developers can use to help focus their debugging efforts. For more information on this work see Memon et al. [Memon04].

4.2. Ensuring a Consistent Global View of System Configuration Constraints

A fundamental strength of open-source software is its distribution in source-code form that users are free to download, build, and execute on any platform that has the right combination of compiler/build tools and runtime support. This flexibility, however, creates an enormous number of potential platform configurations (e.g., compiler and OS settings, versions of the OS and installed libraries, etc.) in which open-source software can execute. Complexity is also introduced by the large number of compile- and run-time options/settings typically seen in open-source software itself. Our experience with ACE+TAO has shown that many errors are encountered in the field by users who employ new, unexplored configurations. For example, the Bugzilla databases (deuce.doc.wustl.edu/bugzilla/index.cgi) for ACE+ TAO show many cases of bug reports that are in fact misconfigurations by users. These problems are exacerbated by the turnover in core developers.

For a successful DCQA process, it is essential to model valid configurations explicitly. Although documentation generally does a good job of conveying the options and their settings, it rarely captures inter-option constraints. For example, in our case study with ACE+TAO [Memon04], we found that many core ACE+TAO developers did not understand the configuration option constraints for their very complex system, i.e., they provided us with both erroneous and missing constraints.

A useful way to visualize the full range of settings possible in open-source software is as a *multidimensional software configuration space*. Each possible combination of settings becomes a single unique point within this space, with the space as a whole encompassing every possible combination of 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 of the software configuration space might have no more than a few tens of unique points. If every setting available on a typical realistic open-source software is put into play, however, the resulting full software configuration space can be enormous, containing millions or more unique points.

Skoll uses a formal model of the software's configuration space to maintain a consistent global view of system constraints. This model captures all valid configurations, which are mappings represented as a set $\{(V_1, C_1), (V_2, C_2), \ldots, (V_N, C_N)\}$, where each V_i is a configuration option and C_i is its value, drawn from the allowable settings of V_i. As noted, not all configurations make sense (e.g., feature X not supported on operating system Y). We

therefore allow 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 (V_i=C_i), where A, B are predicates, V_i is an option and C_i is one of its allowable values. A valid configuration is a configuration that violates no inter-option constraints.

The Skoll configuration model can cover more than just software options, e.g., it can indicate the range of platforms over which the QA process can run correctly. It can also cover test cases that run correctly only in certain configurations. Since this information is typically not written down anywhere, users often run regression tests anyway, which confuses them into believing that the resulting test failure indicates an unknown installation problem. These test failures also confuse developers who must remember which of the several hundred test cases to pay attention to and which can be ignored safely.

Table 1 presents some sample options and constraints for ACE+TAO, including the end platform compiler (`COMPILER`), whether to compile in certain features (`AMI`, `CORBA_MSG`), whether certain test cases are runnable in a given configuration (run(T)), and at what level to set a run-time optimization (`ORBCollocation`). One sample constraint shows that asynchronous method invocation (`AMI`) support requires the presence of CORBA messaging services (`CORBA_MSG`). The other shows that a certain test only runs on platforms with the SUN CC compiler version 5.1.

Option	Settings	Interpretation
COMPILER	{ gcc2.96, SUNCC5_1 }	compiler
AMI	{ 1 = Yes, 0 = No }	Enable Feature
CORBA_MSG	{ 1 = Yes, 0 = No }	Enable Feature
run(T)	{ 1 = True, 0 = False }	Test T runnable
ORBCollocation	{ global, per-orb, NO }	runtime control
Constraints		
AMI = 1 → CORBA_MSG = 1		
run(Multiple/run_test.pl) = 1 → (Compiler = SUNCC5_1)		

Table 1. Some Options and Constraints.

We learned several lessons building the ACE+TAO configuration model. For example, we learned that the configuration model for ACE+TAO was undocumented, so we had to build our initial model bottom-up. We also found that different core developers had conflicting views on what the constraints really were and whether certain constraints were current or had been superseded by recent changes, which taught us that building configuration models is an iterative process.

By using Skoll in place of manual QA processes, we quickly identified previously undiscovered coding errors that prevented the software from compiling in certain configurations. For example, the ACE+TAO build failed at line 137 in the `RT_ORBInitializer.cpp` source file (20 configurations) whenever CORBA MSG = 0 due to a missing #include statement that was conditionally included (via a #define block) only when `CORBA_MSG` = 1. The ACE+TAO build also failed at line 38 in the source file `Asynch_Reply_Dispatcher.h` (8 configurations) whenever `CALLBACK` = 0 and `POLLER` = 1. Since this configuration should be legal, this was determined to be a previously undiscovered bug.

In several cases, tests failed for the same reason on the same configurations. For example, test compilation failed at line 596 of the source file `ami_testC.h` for 7 tests, each when `CORBA_MSG` = 1 and `POLLER` = 0 and `CALLBACK` = 0, which was a previously undiscovered bug. It turned out that certain code within TAO implementing CORBA Messaging incorrectly assumed that at least one of the `POLLER` or `CALLBACK` options would always be set to 1. ACE+TAO developers also noticed that the failure manifested itself no matter what the setting of the `AMI` option, which was a second previously undiscovered problem because these tests should not have been executed when `AMI` = 0. Consequently, there was a missing testing constraint, which we then included in the test constraint set.

In all the cases described above, we found that access to the source code of ACE+TAO enabled us to create and validate our configuration models more quickly and effectively than if we only had access to the binaries.

4.3. Ensuring Coherency and Reducing Redundancy in QA Activities

While conventional QA approaches employed in open-source projects help improve the quality and performance of software, they have significant limitations, e.g., there is little, if any, control over the QA tasks being executed. What to test is left to developers, i.e., each developer typically decides (often by default) what aspects of the system to examine. For example, ACE+TAO developers continuously test their software using a battery of automated tests whose results are published at www.dre.vanderbilt.edu/scoreboard. Developers are responsible, however, for deciding which configurations and tests to run on their platforms. Our experience [Memon04, Yilmaz05] shows that (1) configurations proven to be faulty are tested repeatedly and (2) some configurations are evaluated multiple times, whereas others are never evaluated, which leads to wasted resources and lost opportunities and lets redundancies and important gaps in QA coverage creep in.

To ensure greater coherency and less redundancy in QA activities, Skoll provides an *Intelligent Steering Agent* (ISA) to control the global QA process. The ISA uses AI planning algorithms [Memon01a] to decide which valid

configuration to allocate to each incoming Skoll client request. When a client becomes available, the ISA decides which subtask to assign it by considering various 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., testing popular configurations more than rarely used ones or testing recently changed features more heavily than unchanged features, and (4) *client characteristics and preferences*., e.g., the configuration must be compatible with physical realities, such as the OS running on the remote machine.

After a valid configuration has been chosen, the ISA packages the corresponding QA subtask implementation into a *job configuration* (defined in step 1 in Section 2). The job configuration is then sent to the requesting Skoll client, which executes the job configuration and returns the results to the ISA. The ISA's two default behaviors are (1) to allocate each configuration exactly once (i.e., *random selection without replacement*) or (2) to allocate them zero or more times (i.e., *random selection with replacement*). In both cases, the ISA ignores subtask results. Often, however, we want to learn from incoming results, e.g., when some configurations prove to be faulty, resources should be refocused on other unexplored parts of the configuration space. When such dynamic behavior is desired, process designers develop pluggable ISA components called *adaptation strategies* that monitor the global process state, analyze it, and use the information to modify future subtask assignments to improve overall performance of the DCQA process.

In the ACE+TAO case study mentioned previously in Section 4.2, we observed that ACE+TAO failed to build whenever configuration options `AMI` = 0 and `CORBA_MSG` = 1 were selected. Developers were unable to fix the bug immediately, however, so we developed an adaptation strategy that inserted *temporary constraints*, such as `CORBA_MSG` = 1 → `AMI` = 1 into the configuration model, excluding further exploration of the offending option settings until the problem was fixed. After fixing it, the constraints were removed to restore normal ISA execution. Temporary constraints can also be used to spawn new Skoll processes that test patches only on the previously failing configurations.

Skoll's configuration model therefore makes it possible for the ISA and adaptation strategies to steer the QA process in ways that ensure coherency and reduce redundancy in QA activities. Skoll also allows more sophisticated algorithms and techniques to be applied to improve software quality, e.g., its configuration model essentially defines a combinatorial object against which a wide variety of statistical tools can be applied. In another case study [Yilmaz05], we leveraged this feature to develop a DCQA process called *main effects screening* for monitoring performance degradations in evolving systems.

Main effects screening is a technique that can be used to detect performance degradation rapidly across a large configuration space as a software system changes. This technique relies on experimental design theory to efficiently determine a small subset of the configuration options that substantially effect performance. To implement this technique we compute a highly-efficient formal experimental design called *screening designs* based on the configuration model and conduct the resulting experiment over the Skoll grid. The outcome is a small set of "important options," i.e., those that have the main effects on key system quality factors. From this point on, whenever the system changes, we systematically benchmark all combinations of the important options (while randomizing the rest) to get a reliable estimate of the performance across the entire configuration space. We then monitor the performance estimates to detect performance degradations. Since important options can change over time, we can recalibrate the important options by restarting the process.

We evaluated the main effects screening process via several industrial strength feasibility studies on ACE+TAO software systems. In these studies we created a configuration model containing 14 binary runtime options (a total of 16,384 valid configurations). Our application scenario evaluated the system performance across this configuration space using a benchmarking regression test.

We created a screening design that contained only 32 configurations to identify the important options reliably. We ran the benchmarking test on each of these configurations and found that only 2 (out of 14) options affected the performance of the system significantly. We then monitored the system performance using 4 configurations (all possible combinations of the 2 important options) during a period of time as the system evolved.

Our results showed that (1) screening designs can correctly identify important options, (2) these options can be used to quickly produce reliable estimates of the performance across the entire configuration space at a fraction of the cost of exhaustive testing, (3) the alternative approach of *ad hoc* or random sampling can give highly unreliable results, (4) the main effects screening process can detect performance degradations in evolving software systems, and (5) monitoring all combinations of important options while defaulting or randomizing all the others can provide more precise performance estimates than *ad hoc* or random approaches [Yilmaz05].

4.4. Supporting Diversity in Test Environments

When configuration space explosion is coupled with frequent software updates and increasing platform heterogeneity, ensuring the quality of open-source software can be hard since individual developers may only have access to a limited number of software and hardware platforms. Moreover, frequent code changes may cause development teams to release code that has not been tested thoroughly on all platform and configuration combinations. For example, in one of our case studies with ACE+TAO, we built a configuration model by interviewing the core ACE+TAO developers. After testing several hundred configurations, we found that every configuration failed to compile. We discovered that the problem stemmed from options providing fine-grained control over CORBA messaging policies that had been modified and moved to another library and developers (and users) failed to establish if these options still worked. Based on this feedback the ACE+TAO developers chose to control these policies at link-time rather than compile-time.

To support greater diversity in testing, one QA task we implemented in Skoll is to systematically sample configuration spaces [Yilmaz04]. The approach is based on calculating a mathematical object called a *covering array* with certain coverage properties over the configuration space. A t-way covering array (where *t* is called the strength of the array) is a minimal set of configurations in which all t-way combinations of option settings appear at least once. For a given configuration model and a level of coverage criterion (i.e., a value for *t*), the ISA computes a covering array and allocates only the selected configurations to the requesting clients. The goal is to efficiently improve developer's confidence that options interact with each other as expected.

We conducted a set of feasibility studies on ACE+TAO where we evaluated covering arrays as a sampling strategy in complex configuration spaces [Yilmaz04]. In these studies, our configuration model consisted of 10 compile-time and 6 runtime options. Each compile-time option was binary-valued, while the runtime options had differing numbers of settings: four options with three levels, one option with four levels, and one option with two levels. This configuration space has 18,792 valid configurations. We tested each configuration using 96 regression tests, each of which was designed to emit an error message for failure cases. We captured these error messages, indexed them, and used them in our analysis.

To evaluate the use of covering arrays, we created five different t-way covering arrays for this configuration space. We allowed t to range between 2 and 6. We reran the regression tests on each of these t-way suites and used classification trees to automatically characterize the test results. We then compared the fault characterizations obtained from t-way suites to the ones obtained from exhaustive testing.

Covering Array Strength (t)	# of configurations	% of reduction
2	116	99.4
3	348	98.2
4	1229-1236	93.5-93.4
5	3369-3372	82.1-82.0
6	9433-9453	49.8-49.7

Table 2. Percentage Reduction in Number of Configurations Tested (Compared to Exhaustive Testing)

Table 2 gives the covering array size N for each value of t and the average percentage of reductions in the number of configurations to be tested compared to exhaustive testing. When $t \leq 3$ all five arrays were the same size N. For these we were able to construct covering arrays with the smallest mathematically possible number of rows. When $t \geq 4$, the problem of building a small N is harder, so we obtained a range of sizes.

We learned several things from our covering array studies. First, we rapidly identified problems that had taken the developers substantially longer to find or which had previously not been found. Second, we observed that diverse testing also allows sophisticated analysis techniques to be applied to the resulting data to reason about the root causes of failures, e.g., the classification tree analysis techniques described in Section 3.1 will perform poorly if the input data is skewed towards specific configurations. Third, using covering arrays in complex configuration spaces resulted in fault characterization models that are nearly as accurate as the ones obtained from exhaustive testing, but are much cheaper (provides 50-99% reductions in the number of configurations to be tested) [Yilmaz04].

4.5. Automating the Execution of QA Processes

To evaluate key QoS characteristics of performance intensive software, QA engineers today often handcraft individual QA tasks. For example, for a simple QA task, initial versions of Skoll required the artifacts such as (1) the configuration settings and options for ACE+TAO that need to be evaluated, (2) the evaluation/benchmarking code used to evaluate the configuration settings and provide feedback, (3) interface definitions that represent the contract between the client and server, and (3) support code, e.g., script files, build files to build and execute the experiments. Our earlier work [Memon04] revealed how manually implementing these steps is tedious and error-prone since each step may be repeated many times for every QA experiment.

To redress this shortcoming, we applied *model-driven generative programming techniques* [Czarnecki:00] to

automate the generation of scaffolding code from higher level models, which helps ensure that the generated code is both syntactically correct and semantically valid, thus shielding QA engineers from tedious and error-prone low-level source code generation. This technique also enables QA engineers to compose the experiments via model artifacts rather than source code, thereby raising the level of abstraction. We have integrated the following two modeling capabilities into Skoll:

- The *Options Configuration Modeling Language* (OCML) [RTAS05] modeling tool, which enables users to select a set of middleware-specific configuration options required to support the application needs and
- The *Benchmark Generation Modeling Language* (BGML) [RTAS05], which is a model-driven benchmarking tool that allows component middleware QA engineers to visually model interaction scenarios between configuration options and system components using domain-specific building blocks, i.e., capture software variability in higher-level models rather than in lower-level source code.

OCML model interpreters based on the requesting client characteristics (*e.g.*, OS, compiler, and hardware) and the configuration model generate platform-specific configuration information, which serves as the basis for generating the job configuration. As described in Section 4.3, a job configuration is used to run a particular QA subtask at the client site. The BGML model interpreters generate benchmarking code generation and reuse QA task code across configurations.

In earlier work [Yilmaz05, ICSR8], we showed how BGML can be used to auto-generate ~90% of the code required to set up a benchmarking experiment. This generated code was also interfaced with our main effects screening tools. Our work on model-driven automation of the QA process revealed that having access to the source code enabled the QA engineers to reuse instrumentation and evaluation code for different configuration combinations. Similarly, QA engineers can easily tailor the generated code for different IDL interfaces by adding/modifying the generated IDL from the model.

5. Related Work

There are several efforts addressing the QA challenges of open-source software systems by gathering 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. This section describes the most visible of these efforts.

Online crash reporting systems 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. For example, Netscape Quality Feedback Agent and Microsoft XP Error Reporting aid users in reporting errors by automatically generating error reports during crashes to enable users to report failures with ease, thus aiding QA teams to improve software quality. These approaches, however, have a very limited scope, i.e., they perform only a small fraction of typical QA activities and ignore issues associated with QoS and performance. Moreover, they are *reactive* (i.e., the reports are only generated when systems crash), rather than *proactive* (e.g., attempting to detect, identify, and remedy problems before users encounter them). In contrast, Skoll's ISA uses several techniques (e.g., nearest neighbor) that study previous results and predict configurations that are likely to create problems and explores them before they are encountered in the field.

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, OS, and compiler platforms. The Mozilla Tinderbox and ACE+TAO Virtual Scoreboard are auto-build scoreboards that track end-user build results across various platforms. Bugs are reported via defect tracking systems (such as Bugzilla and JIRA), which provide inter-bug dependency recording, advanced reporting capabilities, extensive configurability, and integration with automated software configuration management systems (such as CVS and Subversion). While these systems help document the QA process, the decision of *what* to test is left to end users. Unless the core developers can control at least some aspects of the QA process, important gaps and inefficiencies will still occur. In contrast, Skoll is essentially driven by core developers in multiple ways. For example, adaptation strategies in the ISA are coded by the core developers. This has a direct impact on which configurations should be tested and when.

6. Concluding Remarks

Open-source has proven to be an effective development process for many software application domains [Raymond01]. The overall quality of open-source software, however, can be compromised by recurring problems, such as frequent releases of patches, lack of global view of system constraints, unsystematic and inadequate QA, and large configuration spaces,. This paper described how the Skoll project provides technologies and tools to help minimize the effects of these problems by leveraging key strengths of open-source development processes, such as open access to source code, ubiquitous web access, and their scalability to large user communities, where technologically sophisticated application programmers and end-users in the field can assist with QA activities, documentation, mentoring, and technical support traditionally performed in-house. Throughout this paper we describe how intelligent leverage of the expertise and extensive computing resources of user communities is

essential to overcome common problems that can impede the success of large-scale open-source software projects.

7. Acknowledgements

We thank the anonymous reviewers for their helpful comments. This material is based on work supported by the US National Science Foundation under NSF grants ITR CCR-0312859, CCR-0205265, CCF-0447864, and CCR-0098158, ONR grant N00014-05-1-0421, as well as funding from BBN Technologies, Lockheed Martin, Raytheon, and Siemens.

Bibliography

[ACE01] Schmidt D., Huston S., *C++ Network Programming: Resolving Complexity with ACE and Patterns*, Addison-Wesley, Reading, MA, 2001.

[Brooks75] Brooks, F., *The Mythical Man-Month*, Addison-Wesley, 1975.

[Raymond98] E. Raymond, "The Cathedral and the Bazaar," *First Monday,* volume 3, number 3 (March), http://www.firstmonday.org/issues/issue3_3/raymond/, 1998.

[Mockus02] A. Mockus, R.T. Fielding, and J. Herbsleb, "Two Case Studies of Open Source Software Development: Apache and Mozilla," ACM Trans. Software Engineering and Methodology, 11(3), 309-346, 2002.

[Mockus00] Mockus, A., Fielding, R.T., and Herbsleb, J. (2000). "A Case Study of Open Source Software Development: The Apache Server," Proceedings of the 22nd International Conference on Software Engineering. IEEE Computer Society, 263-272. A quantitative study of Apache development.

[Osterlie03] Thomas Østerlie and Knut H. Rolland, "Unveiling Distributed Organizing in Open Source Development: Practices of Using, Aligning, and Wedging," in Proceedings of the Workshop on Open Source Software Movements and Communities, 19-22 September 2003, Amsterdam

[Lakhani03] Lakhani, K.R., E. von Hippel. 2003. "How Open Source Software Works: 'Free" User-to-User Assistance," Research Policy. 32 923-943.

[CoSMIC05] Krishnakumar Balasubramanian, Arvind S. Krishna, Emre Turkay, Jaiganesh Balasubramanian, Jeff Parsons, Aniruddha Gokhale, and Douglas C. Schmidt, "Applying Model-Driven Development to Distributed Real-time and Embedded Avionics Systems", *International Journal of Embedded Systems* special issue on Design and Verification of Real-Time Embedded Software, Kluwer, April 2005.

[Gamma05] Erich Gamma, "Agile, open source, distributed, and on-time: inside the eclipse development process", Keynote Address, International Conference on Software Engineering (ICSE) 2005, St Louis, MO USA.

[GPERF90] Schmidt, D., "GPERF: A Perfect Hash Function Generator," Proceedings of the 2nd USENIX C++ Conference, San Francisco, April 1990.

[JAWS99] Hu, J., Pyarali I., and Schmidt D., "The Object-Oriented Design and Performance of JAWS: A High-performance Web Server Optimized for High-speed Networks," *Parallel and Distributed Computing Practices Journal*.

[**ICSR8**] Arvind S. Krishna, Douglas C. Schmidt, Adam Porter, Atif Memon and Diego Sevilla-Ruiz "Improving the Quality of Performance-intensive Software via Model-integrated Distributed Continuous Quality Assurance", "Proceedings of the 8th International Conference on Software Reuse, Madrid, Spain, July 2004.

[**Memon01a**] Atif M. Memon, Martha E. Pollack and Mary Lou Soffa, "Hierarchical GUI Test Case Generation Using Automated Planning," IEEE Transactions on Software Engineering. vol. 27, no. 2, pp. 144-155, Feb. 2001.

[Michail05] Amir Michail and Tao Xie, Helping Users Avoid Bugs in GUI Applications, Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), St. Louis, Missouri, USA, May 2005.

[OMG02] Object Management Group, "Real-time CORBA, OMG Technical Document formal/02-08-02", August 2002.

[Parnas79] Parnas, D., "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering*, March 1979.

[Porter91] Adam Porter, R. Selby, "Empirically Guided Software Development Using Metric-Based Classification Trees," *IEEE Software*, March 1990.

[**PSA04**] Arvind S. Krishna, Cemal Yilmaz, Atif Memon, Adam Porter, Douglas C. Schmidt, Aniruddha Gokhale, and Balachandran Natarajan, "Preserving Distributed Systems Critical Properties: a Model-Driven Approach," the *IEEE Software* special issue on the Persistent Software Attributes, Nov/Dec 2004.

[Raymond01] Raymond E., *The Cathedral and the Bazaar: Musings on Linux and Open-source by an Accidental Revolutionary*, O'Reilly, 2001.

[RTAS05] Arvind S. Krishna, Emre Turkay, Aniruddha Gokhale, and Douglas C. Schmidt, "Model-Driven Techniques for Evaluating the QoS of Middleware Configurations for DRE Systems," Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium, San Francisco, CA, March 2005.

[Schantz01] Schantz R. and Schmidt D., "Middleware for Distributed Systems: Evolving the Common Struc-

ture for Network-centric Applications," *Encyclopedia of Software Engineering*, Wiley & Sons, 2001.

[Memon04] Atif Memon, Adam Porter, Cemal Yilmaz, Adithya Nagarajan, Douglas C. Schmidt and Bala Natarajan, "Skoll: Distributed Continuous Quality Assurance", Proceedings of the 26th IEEE/ACM International Conference on Software Engineering, IEEE/ACM, Edinburgh, Scotland, May 2004.

[Yilmaz05] Cemal Yilmaz, Arvind Krishna, Atif Memon, Adam Porter, Douglas C. Schmidt, Aniruddha Gokhale, and Bala Natarajan, "Main Effects Screening: A Distributed Continuous Quality Assurance Process for Monitoring Performance Degradation in Evolving Software Systems," proceedings of the 27th International Conference on Software Engineering, St. Louis, MO, May 15-21, 2005.

[TAO98] Schmidt D., Levine D., Mungee S. "The Design and Performance of the TAO Real-Time Object Request Broker", *Computer Communications Special Issue on Building Quality of Service into Distributed Systems,* 21(4), 1998.

[Czarnecki:00] Krzysztof Czarnecki and Ulrich Eisenecker, "Generative Programming: Methods, Tools, and Applications", Addison-Wesley, Boston 2000.

[Yilmaz04] Cemal Yilmaz, Myra Cohen and Adam Porter, "Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces", Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Boston, Massachusetts, 2004.

[**Zeller02**] Andreas Zeller, "Isolating Cause-Effect Chains from Computer Programs," Proceedings of the ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE-10), Charleston, South Carolina, November 2000.