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

Cemal Yilmaz, Atif M. Memon, Adam Porter
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, 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 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) creating models and setting up the DCQA process improves developer understanding of open-source software, (2) improving test diversity in terms of platform configurations helps to find defects missed during conventional testing, and (3) centralizing control of QA activities helps to eliminate redundant QA work.

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 systems, web servers, middleware, 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 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 and relatively poorly served platforms. Finally, 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). So when they encounter bugs or software configuration problems they can fix the problems and submit patches.

From a software perspective, the open-source projects outlined above have generally succeeded for the following reasons:

- **Scalable division of labor**. Open-source projects work by exploiting a "loophole" in Brooks' Law [Brooks75] that states "adding developers to a late project makes it later." In contrast, software debugging and QA productivity *does* scale up as developer headcount increases. This is because all other things being equal, having more people test the code will identify the "error-legs" much more quickly than having fewer testers. So we would expect a team of 1,000 testers to find many more bugs than a team of 10 testers, a phenomenon which is commonly referred to as "to enough eyeballs, all bugs are shallow" in the open-source community [O'Reilly98]. QA activities also scale better than software development activities (particularly analysis and design activities) since they do not require as much inter-personal communication.

  As a result successful open-source projects are often organized into a "core" and "periphery" structure. A relatively small number of core developers are responsible for ensuring the architectural integrity of the project, e.g., they vet user contributions and bug fixes, add new features and capabilities, and track 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. Naturally, these distinctions are informal and developers can play different roles at different times.

- **Short feedback loops**. One reason for the success of well-organized, large-scale open-source development efforts, such as Linux or ACE+TAO, is the short feedback loops between the core and the periphery. For example, for these systems it often takes just a few minutes or hours to detect a bug at the periphery report it to the core and re-

ceive an official patch. 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 synchronize in real-time with updates supplied by the core.

- **Effective leverage of user community expertise and computing resources**. In today's time-to-market-driven economy, fewer 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 its defects were removed. In traditional closed-source/binary-only software deployment models, these premature release cycles yield frustrated users, who have little recourse when problems arise. Since they are often limited to 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. 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 therefore greatly magnify the debugging and computing resources available to an open-source project, which can improve software quality if harnessed effectively.

- **Inverted stratification of available expertise**. In many organizations, testers 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 application developers who use their considerable debugging skills when they encounter occasional problems with the open-source software base. The open-source model thus makes it possible to leverage the talents of these gifted developers, who might not ordinarily act as testers in traditional software organizations.

- **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, see [Zeller02] and [Michail05]) who have employed analysis and testing techniques to find bugs in open-source software.

In general, traditional closed-source 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 notable successes in the systems infrastructure software domains for the reasons described in Section 1.1. Our experience working on a wide-range of open-source projects [GPERF90, TAO98, JAWS99, ACE01, CoSMIC04] for the past two decades has shown, however, that the open-source development model also creates 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 aren't unique, i.e., limit regression errors (e.g., 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 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 the end-users who receive 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.

- **Problem 2: Lack of global view of system constraints**. Large-scale open-source projects often have a high-rate of churn in contributors and infrastructure. Users may encounter problems, 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. These users, however, are rarely knowledgeable about the entire architecture of the open-source software system. As a result, they lack a global view of broader system constraints that may be affected by any given change.

- **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 run to evaluate installation success. Users can – but frequently do not – return the test results to project developers. Even when results are returned to developers, however, the testing process is often undocumented and unsystematic, e.g., developers have no record of what was tested, how it was tested, or what the results were, which loses crucial QA-related information. Moreover, many QA configurations are executed redundantly by thousands of developers, whereas others are never executed at all.

- **Problem 4: Lack of diversity in test environments**. Well-written open-source software can be ported easily to a variety of operating system (OS) and compiler platforms. In addition, since the source is available, end-users can modify and adapt their source base readily to fix bugs

[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.

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, individual developers may only have access to a limited number of OS/compiler configurations, which may cause them to release code that has not been tested thoroughly on all platform configurations on which their software needs to run.

- **Problem 5: Manually intensive execution of QA processes**. Source availability encourages developers to increase the number of options for configuring and subsetting the software at compile-time and run-time. Although this flexibility enhances the software's applicability for a broad range of use cases, it can also greatly magnify QA costs. Moreover, since open-source projects often run on a limited QA budget due to their minimal/non-existent licensing fees, it can be hard to 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 and increasing subsequent development and QA costs. Without remedial action, therefore, the open-source user communities may become smaller (due to frustration with software quality) until the software falls out of use. Until these types of problems are addressed adequately, it will be hard to deliver on the promise of open-source processes.

1.3. Addressing Open-Source Challenges with Skoll

To address the problems with open-source software development described in Section 1.2, we have developed Skoll [Skoll04], which is a *distributed continuous quality assurance* (DCQA) environment for developing and validating novel software QA processes and tools that leverage the extensive computing resources of worldwide user communities in a distributed, continuous manner to significantly and rapidly improve software quality. In particular, Skoll provides an integrated set of technologies and 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:

- **Detect and resolve QA problems quickly**. Skoll closes the loop from users back to developers rapidly; it exploits the inherently distributed nature of open-source sites/users, each one performing a portion of the overall testing to offload the number of versions that must be maintained by the core developers, while simultaneously enhancing user confidence in new beta versions of open-source software.

- **Automatically analyze and enhance key system quality of service (QoS) characteristics on multiple platforms**. Each site/user conducts different instrumentations and performance benchmarks automatically to collect metrics. These metrics include static and dynamic measures of memory footprint, as well as performance measures, such as 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

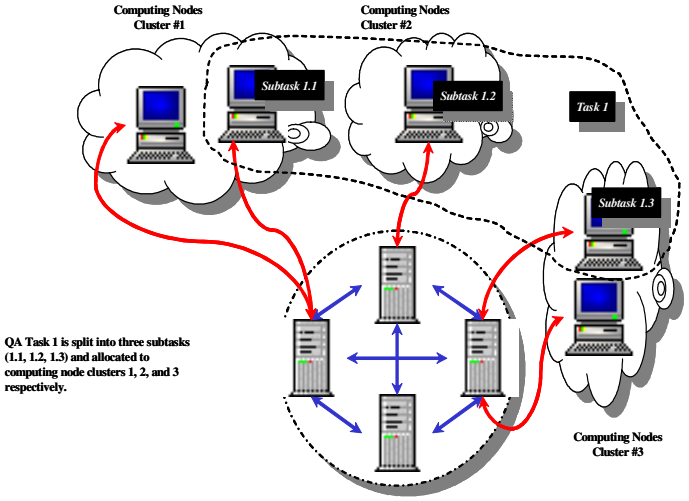

Fig. 1: Skoll Tasks/Subtasks Allocated to Computing Nodes

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 significantly 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). The subtasks are allocated to computing node clusters, where they are executed. As subtasks run, Skoll's control logic may dynamically steer the global computation for reasons of performance and correctness.

Our earlier work on Skoll [Skoll04, Skoll05, PSA04] has described various DCQA 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 be executed independently on user machines, (3) collec-

tion of subtask execution results and their interpretation, (4) adaptation of the overall process based on the results, (5) fault classification techniques to help pinpoint errors, and (6) visualization of QA task results. This paper extends our prior work by focusing on how Skoll, when applied to a large-scale open-source project, has helped to mitigate problems common to many open-source projects.

2. Overview of ACE and TAO

We have use Skoll with ACE and TAO, which are widely used open-source middleware platforms (downloadable from www.dre.vanderbilt.edu). ACE [ACE01] is an object-oriented framework containing hundreds of classes that implement key patterns and frameworks for distributed real-time and embedded (DRE) systems. TAO [TAO98] is an implementation 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 systems 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 language processing tools:

- **Large and mature source code base**. The ACE+TAO source base contains over one million source lines of C++ middleware systems source code, examples, and regression tests split into over 4,500 files as follows.

Software Toolkit	Source Files	Source Lines of Code
ACE	1,860	393,000
TAO	2,744	695,000
Total	4,604	1,088,000

- **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, and many different variations of CORBA services.
- **Core development team**. ACE+TAO are maintained by a core – yet geographically distributed – team of ~20 developers. Many of these core developers have worked on ACE+TAO for several years. Yet 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 part of the core development team.
- **Continuous evolution**, i.e., 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, 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 core 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 weeks. 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 also suffers from the problems discussed in Section 1.

3. Applying Skoll to ACE+TAO

This section describes how we have applied DCQA processes using Skoll environment to address the problems with open-source processes discussed in Section 1.2. To make the discussion concrete, we focus on the application of 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 any large scale open-source project.

3.1. 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 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 all possible configurations in a timely manner. As a result some parts of the system 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 have 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

- **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
- **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 options and 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. [Skoll04].

3.2. Ensuring a Consistent Global View of System Configuration Constraints

A fundamental strength of open-source software is its distribution in source-code form. An open-source software user is free to download, build, and execute the software on any platform that has the right combination of compiler/build tools and run-time support. This flexibility, however, creates a very large number of potential platform configurations (e.g., compiler settings, OS version, versions of installed libraries) in which an open-source software can be executed. Complexity is also introduced by the large number of compile- and run-time options/settings typically seen in open-source systems infrastructure software. Our experience with open-source software has shown that a large number of errors are encountered in the field by users who employ a new, unexplored configuration. 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 magnified by turnover in core developers.

For a successful DCQA process, it is essential to model valid configurations explicitly. Standard documentation does a reasonable job of conveying the options and their settings. It rarely, however, captures inter-option constraints. For example, in our case study with ACE+TAO [Skoll04], 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 such an open-source system 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 is very large, 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 -> 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. It can indicate the range of platforms over which the QA process can run correctly. It can also cover test cases which often 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. This also confuses developers who have to remember which of the several hundred test cases to pay attention to and which can be safely ignored.

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.

Table 1 presents some sample options and constraints for ACE+TAO. These include 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 runtime optimization (ORBCollocation). One sample constraint shows that AMI support requires the presence of CORBA messaging services. The other shows that a certain test only runs on platforms with the SUN CC compiler version 5.1.

We learned several lessons building the ACE+TAO configuration model. 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. The use of Skoll enabled us to quickly identify many coding errors (some previously undiscovered) that prevented the software from compiling in certain configurations. Moreover, 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 binary code.

3.3. 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; each developer typically decides (often by default) what aspects of the system he/she will 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. Each developer, however, is responsible for deciding which configurations and tests he/she wants to run on his/her platform. Our experience [skoll04, skoll05] shows that (1) configurations proven to be faulty are tested over and over again and (2) some configurations are evaluated multiple times, others not at all, which leads to wasted resources and lost opportunities and lets redundancies and important gaps 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 advanced 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 than 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*, which consists of the code artifacts, configuration parameters, build instructions, and QA-specific code (*e.g.*, developer-supplied regression/performance tests) associated with a software project. The job configuration is then sent to the requesting Skoll client, which executes the job configuration and returns the results to the ISA. The default behavior of the ISA is to allocate each configuration exactly once (i.e., *random selection without replacement*) or

zero or more times (i.e., *random selection with replacement*); it ignores the 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 *adaptation strategies* -- pluggable ISA components that monitor the global process state, analyze it, and use the information to modify future subtask assignments to improve overall DCQA process performance.

In one of our case studies with ACE+TAO [Skoll04], we observed that ACE+TAO failed to build whenever configuration options AMI = 0 and CORBA MSG = 1 were selected. Developers, however, were unable to fix the bug immediately. Therefore, we developed an adaptation strategy, which inserts *temporary constraints*, such as CORBA MSG = 1 -> AMI = 1 into the configuration model, excluding further exploration of the offending option settings until the problem is fixed. After fixing it, the constraints are removed which allows normal ISA execution. Temporary constraints could be used to spawn new Skoll processes that test patches only on the previously failing configurations.

Thus Skoll's configuration model 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. Another advantage is that it allows more sophisticated algorithms and techniques to be applied to improve software quality. For example, our configuration model essentially defines a combinatorial object against which a wide variety of statistical tools can be applied. In another case study [Skoll05], 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 for rapidly detecting performance degradation across a large configuration space as a result of system changes. We cast this as an experimental design question in which the problem is to determine a small subset of the configuration options that substantially effect performance. To do this 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". 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. Since the important options can change over time, depending on how fast the system changes, we recalibrate the important options by restarting the process. We evaluated the main effects screening process via several industrial strength feasibility studies on ACE+TAO. Our results indicate that main effects screening can reliably and accurately detect key sources of performance degradation in large-scale systems with significantly less effort than conventional techniques [Skoll05].

3.4. 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 idea here is to efficiently improve developer's confidence that options interact with each other as expected. We evaluated this approach via several feasibility studies on ACE+TAO. In these studies, we rapidly identified problems that had taken the developers substantially longer to find or which had previously not been found [Yilmaz04].

Another advantage of diverse testing is that it allows the application of sophisticated techniques 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. Overall, we found that increasing test diversity (1) allowed fault characterization models to quickly pinpoint the root causes of several failures, leading to much quicker turn-around time for bug fixes and (2) using covering arrays in complex configuration spaces resulted in fault characterization models, which 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).

3.5. 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 [Skoll04] 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 provide the following two capabilities to Skoll from the modeling level:

- 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 3.3, 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 [Skoll05, 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 the main-effects screening. Our work also revealed that having access to the source code, i.e., benchmarking and CORBA IDL files enables the QA experimenter to reuse instrumentation and evaluation code for different configuration combinations. Similarly, the QA experimenter can easily tailor the generated code for different IDL interfaces by adding/modifying the generated IDL from the model.

4. Related Work

Several research efforts attempt to address the challenges of open-source software systems. These efforts gather various types of information from distributed run-time environments and usage patterns encountered in the field, i.e., on user target platforms with user configuration options.

Online crash reporting systems, such as the Netscape Quality Feedback Agent and Microsoft XP Error Reporting, gather system state at a central location whenever a fielded system crashes, which simplifies user participation in QA by automating certain aspects of problem reporting. Each of these approaches, however, has 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). To aid users in reporting errors, even Microsoft tools such as Visual Studio, generate automatic error reports during crashes to enable users to report failures with ease thus aiding QA teams to improve software quality.

Auto-build scoreboards, which are a more proactive form of distributed regression test suites mentioned in Section 1.2 that allow software to be built/tested at multiple sites on various hardware, operating system, 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 the Bugzilla issue tracking system, which provides inter-bug dependency recording, advanced reporting capabilities, extensive configurability, and integration with automated software configuration management systems, such as CVS. While these systems help with documenting the QA process, the decision of what to test is left to end users. Unless developers can control at least some aspects of the QA process, important gaps and inefficiencies will still occur.

5. Concluding Remarks

Open-source has proven to be an effective development process in many software application domains [Raymond01]. The Skoll project is leveraging key aspects of open-source development, such as its worldwide user communities, open access to source code, and ubiquitous web access, to further improve the quality and performance of open-source software. A particularly important strength of open-source development processes are their scalability to large user communities, where technologically sophisticated application programmers and end-users can assist with many QA activities, documentation, mentoring, and technical support. Throughout this paper we have described how

intelligent leverage of the expertise and extensive computing resources of user communities is essential to overcome common problems that can impede the long-term success of large-scale open-source projects.

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.

[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.

[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, To appear in 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.

[O'Reilly98] *The Open-Source Revolution*, O'Reilly, 1998.

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

[Skoll04] 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.

[Skoll05] 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.