Leveraging Open-Source Communities To Improve the Quality & Performance of Open-Source Software

Douglas C. Schmidt

schmidt@uci.edu
Electrical & Computer Engineering Department
University of California, Irvine

Adam Porter

aporter@cs.umd.edu
Computer Science Department
University of Maryland

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 distribution middleware. This paper presents two contributions to the study of open-source software engineering. First, we describe the key challenges of open-source software, such as controlling long-term maintenance and evolution costs, ensuring acceptable levels of quality, sustaining end-user confidence and good will, and ensuring the coherency of system-wide software and usability properties. We illustrate how well-organized open-source projects make it easier to address many of these challenges compared with traditional closed-source approaches to building software. Second, we present the goals and methodology of the *Skoll* project, which focuses on developing and empirically validating novel open-source software quality assurance and optimization techniques to resolve key open-source challenges. We summarize the experimental design of a long-term case study of two widely used open-source middleware projects—ACE and TAO—that we are using in the Skoll project to devise, deploy, and evaluate techniques for improving software quality through continuous distributed testing and profiling. These techniques are designed to leverage common 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, that can improve the quality and performance of open-source software significantly.

1. Introduction

1.1. Enablers of Open-Source Success

Over the past decade, open-source development processes [O'Reilly98] have emerged as an effective approach 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, compilers and language processing tools, editors, and distribution middleware. Widely used examples of successful open-source infrastructure software include:

- UNIX/POSIX operating systems, e.g., Linux, FreeBSD, and NetBSD [BSD4.4]
- Web servers, e.g., Apache and JAWS [JAWS99]
- Distribution middleware, e.g., ACE [ACE01], TAO [TAO98], MICO [Puder99], omniORB, and JacORB [Brose99]
- Compilers, e.g., GNU C/C++ [Stallman]
- Editors, e.g., GNU emacs
- Language processing tools, e.g., Perl, GAWK, Flex, and Bison
- System/network support tools, e.g., Samba, Sendmail, and Bind.

These open-source projects are distributed under a variety of licensing schemes[1] and business models. In general, however, they all distribute their software in source code form, allow derived works to be created and freely distributed, and rely upon their user communities for many development and QA activities that have been performed internally by traditional software providers. Moreover, all these projects exhibit one or more of the following key enablers of open-source success:

- *They are general-purpose, commoditized systems infrastructure software*, whose requirements and APIs are well known. For example, the requirements and APIs for UNIX/POSIX operating systems, Web servers, C/C++ compilers/linkers, and object request broker (ORB) middleware are well understood by many software developers. Thus, there is little need to expend time and effort on costly upstream software development activities, such as requirements analysis or interface specifications. Moreover, there is an abundance of highly talented graduate students and programming aficionados who enjoy working on these types of general-purpose software infrastructure technologies.

- *They service communities with unmet software needs.* Certain technology communities, such as scientists or even the defense industry, have software needs that are unmet by (or are economically unappealing to) "mainstream" information technology (IT) software markets. These communities often lack sufficient market presence to be well supported by mass-market IT vendors, such as Microsoft, IBM, or Sun. For example, Linux-based Beowulf clusters [Beowulf97] were developed in the scientific computing community in part because their high-end computing applications run on hardware and infrastructure software configurations that are not widely supported via mass-market IT vendors.

- *They are employed by a technically sophisticated user community*. Members of certain user communities have considerable software development skills and knowledge of common development tools, such as compilers, debuggers, configuration managers, online bug tracking systems, memory leak/validation tools, and performance profilers. Likewise, they are facile with open communication mechanisms, such as web/ftp-sites and mailing lists, and advanced Internet collaboration mechanisms, such as Netmeeting or instant messaging. When members of technically sophisticated user communities encounter bugs or software configuration problems, they are not adverse to fixing the problems and submitting patches to maintainers of the software.

1.2. Open-Source Challenges

Although open-source projects have clearly succeeded in the systems infrastructure software domains outlined above, our experience working on several large-scale open-source projects for over a decade [GPERF90, TAO98, ACE01] has shown that developers of open-source software face the following key challenges:

1. *Containing long-term maintenance and evolution costs associated with quality assurance (QA).* In general, the goals of open-source software are similar to those of other types of software, i.e., limit regression errors (e.g., avoid breaking features that worked in previously 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 as a result of the following context in which it is developed:

   - *Frequent beta releases*. A cornerstone of open-source is 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.

---

[1] See http://www.opensource.org for a discussion of alternative open-source licensing terms.

- *Platforms-independence*. Another cornerstone of open-source software is its platform-independence, which stems from its roots in the open systems—rather than proprietary systems—community. Support for platform-independence, however, can yield the Sisyphean task of keeping an open-source source software base operational despite continuous changes to the underlying operating system and compiler platforms.
- *Support for many compile-time and run-time configurations*. The availability of the software in open-source projects encourages core 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 due to the combinatorial number of code paths that must be regression tested. Moreover, since open-source projects often run on a "shoe-string" QA budget due to their minimal/non-existent licensing fees, it can be difficult to support large numbers of versions and variants simultaneously.

2. *Ensuring coherency of system-wide properties*. The decentralized nature of many open-source projects makes it hard to enforce a common "look and feel," standardize common APIs, and ensure consistent semantics when disparate components are integrated together. Since open-source projects often grow "organically" based on contributions from a range of developers and users, considerable effort must be expended to ensure architectural integrity and commonality across contributions from disparate members of the community.

1.3. Addressing Open-Source Challenges in the Skoll Project

We have begun a long-term, multi-site collaborative research project, called *Skoll*, that is leveraging key open-source assets, such as:

- Technologically sophisticated worldwide user communities
- Open access to source and
- Ubiquitous web access

to address the key challenges of open-source software development described above. The goals of the Skoll project are to develop methods and tools that can:

- *Incrementally improve the quality and performance of open-source systems*. Keeping with the spirit of open-source efforts, our aim is to opportunistically improve the software base over time, though not necessarily perfectly at each step.
- *Minimize human effort by applying automation judiciously*. We are automating the role of human "build czars," who today monitor the stability of open-source software repositories *manually* to ensure problems are fixed rapidly.
- *Minimize unnecessary end-user overhead without compromising security*. We are using automated web tools to minimize end-user effort and resource commitments, as well as ensure the security of end-user computing sites.

The approach we are taking to achieve these goals is based on the notion of *continuous testing*, which we are operationalizing in Skoll as follows:

- Regression testing and performance profiling is widely distributed and conducted in parallel on machines provided by the open-source user community during their off-peak hours and
- The resources of the user community are coordinated carefully, i.e., Skoll follows the sun around the world[2] and adapts its testing/profiling processes based on results of earlier testing and profiling in other locations.

We are pursuing these goals in the context of the ACE and TAO open-source projects, which are based at the University of California, Irvine (UCI) and Washington University, St. Louis (WUSTL). ACE [ACE01] is a widely used object-oriented framework containing a rich set of components that implement patterns for high-performance and real-time communication systems. TAO [TAO98] is an implementation of the Common Object Request Broker Architecture (CORBA) [OMG00] that uses the framework components in ACE to meet the demanding quality of service (QoS) requirements in distributed, real-time, and embedded systems.

The remainder of this paper is organized as follows: Section 2 describes the reasons why successful open-source development projects work, from both an end-user and software process perspective; Section 3 outlines the key characteristics of the ACE and TAO open-source software and explains why they are representative of other successful open-source projects; Section 4 presents an overview of the Skoll project, focusing on its experimental design and contributions to open-source software engineering; and Section 5 presents concluding remarks and summarizes the limitations of open-source software applicability.

[2] This is the origin of the term "Skoll," which is a Scandinavian myth that explains the sunrise and sunset cycles.

2. Why Open-Source Works

Before it is possible to improve the quality and performance of open-source software, it is important to understand why it works in the first place. This section summarizes why successful open-source projects work from an end-user and software process perspective. We base this discussion on our experience devising, employing, and researching open-source development processes for over a decade.

From an end-user perspective, successful open-source projects work for the following reasons:

- *Reduced software acquisition costs*. Open-source software is distributed without development or run-time license fees, though many open-source companies do charge for technical support. This pricing model is particularly attractive to application developers working in highly commoditized markets where profits are driven to marginal cost. Moreover, open-source projects typically use low-cost distribution channels, such as the Internet, so that users can access source code, examples, regression tests, and development information cheaply and rapidly.
- *Enhanced diversity and scale*. Well-written and well-configured open-source software can be ported easily to a variety of heterogeneous operating system and compiler platforms. In addition, since the source is available, end-users have the freedom to modify and adapt their source base readily to fix bugs quickly or to respond to new market opportunities with greater agility.
- *Simplified collaboration*. Open-source promotes the sharing of programs and ideas among members of technology communities that have similar needs, but who also may have diverse technology acquisition/funding strategies. This cross-fertilization can lead to new insights and breakthroughs that would not have occurred as readily without these collaborations.

From a software process perspective, successful open-source projects work 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." The logic underlying this law is that as a general rule, software development productivity does *not* scale up as the number of developers increases. The culprit, of course, is the rapid increase in human communication and coordination costs as project size grows. Thus, a team of ~10 good developers can often produce much higher quality software with less effort and expense than a team of ~1,000 developers [Boehm80].

  In contrast, software debugging and QA productivity *does* scale up as the number of developers helping to debug the software increases. The main reason for this is that all other things being equal, having more people test the code will identify the ``error-legs" much more quickly than having just a few testers. A team of 1,000 testers will therefore usually find many more bugs than a team of 10 testers. QA activities also scale better since they do not require as much inter-personal communication compared with software development activities, particularly analysis and design activities.

  To leverage the loophole in Brooks law, therefore, most successful open-source projects are organized into a "core" and "periphery" structure, which is shown in the following figure.

  As shown in this figure, the smaller circles represent the peripheral users and the large circle in southern California represents the core developers. A relatively small number of core developers[3] 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 divisions can be informal and the same developer may play different roles at different times during the lifecycle of an open-source project.

[3] This figure shows the core developers residing in a single site, though in practice they can be distributed throughout the world.

- *Short feedback loops*. One reason for the success of well-organized open-source development efforts, such as Linux or ACE+TAO, is the short feedback loops between the core and the periphery. In successful open-source projects, for instance, it is often only a matter of minutes or hours from the point at which a bug is reported from the periphery to the point at which an official patch is supplied from the core to fix it. Moreover, the use of powerful Internet-enabled configuration management tools, such as the GNU Concurrent Versioning System (CVS) [CVS99], 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, few software providers can afford long QA cycles. As a result, nearly everyone who uses a computer—particularly software application developers—is 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 with software they purchased from vendors. Since their only option is often to find workarounds for problems they encounter, they may have little incentive to help improve closed-source products.

  In contrast, open-source development processes help to leverage expertise in their communities, thereby allowing users and developers to collaborate to improve software quality. For example, the short feedback loops mentioned in the previous bullet encourage users to help with the QA process since they are "rewarded" by rapid fixes after bugs are identified. Moreover, since the source code is open for inspection, when users at the periphery do encounter bugs they can often either fix them directly or can provide concise test cases that allow the core developers to 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*. In many organizations, testers are perceived to have less status than software developers. The open-source development model inverts this stratification in many cases 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 makes it possible to leverage the talents of these gifted developers, who would often not be satisfied with playing the role of a tester 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. Although this benefit of open-source is not yet widely employed, the next-generation of testing tools and model checkers will be more effective since they can instrument and analyze large-scale systems where open-source is used throughout many components and layers, e.g., from the network drivers, to the OS, and up to the middleware and applications.

In general, traditional closed-source software development and QA processes rarely achieve the benefits outlined above as rapidly or as cheaply as open-source processes.

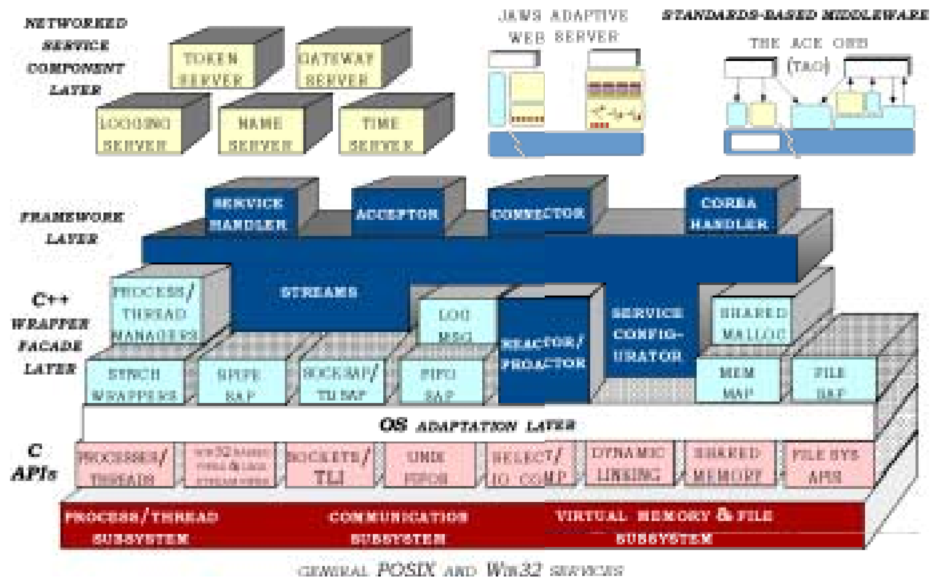
3. Overview of the ACE and TAO Open-Source Projects

ACE and TAO are large, widely-deployed open-source[4] middleware software toolkits that C++ programmers can reuse and extend to simplify the development of their own distributed applications. ACE [ACE01] is an object-oriented framework that implements core concurrency and distribution patterns for networked application software. It is targeted at developers of high-performance, real-time services and applications across a wide range of OS platforms. Components in the ACE toolkit support a variety of reusable network programming capabilities, such as connection establishment and service initialization; event demultiplexing and event handler dispatching; interprocess communication and shared memory management; static and dynamic configuration of distributed communication services; concurrency and synchronization; networked services (e.g., naming, logging, and event routing); and standards-based middleware (e.g., web servers and ORBs).

The ACE ORB (TAO) is an implementation of the CORBA standard [OMG00] that is constructed using the patterns [POSA2] and framework components provided by the ACE toolkit. TAO simplifies the development of distributed applications by automating and encapsulating object location, connection and memory management, parameter (de)marshaling, event and request demultiplexing, error handling and fault tolerance, object and server activation, concurrency, and security. These capabilities 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 [H&V99]. The components, layers, and relationships between the components/layers in ACE+TAO are shown in the figure below.

---

[4] The ACE source can be downloaded at http://www.cs.wustl.edu/~schmidt/ACE.html and the TAO source can be downloaded at http://www.cs.wustl.edu/~schmidt/TAO.html.



ACE and TAO are ideal study candidates for the Skoll project because they share the following key characteristics with other successful open-source projects:

- *Large and mature source code base*. The ACE+TAO source base has evolved over the past decade and now 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 OS and compiler platforms. These platforms change over time, e.g., to support new features in the C++ standard, different versions of POSIX/UNIX, as well as different versions of non-UNIX OS platforms, including all variants of Microsoft Win32 and many real-time and embedded 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 2.4 specification, Minimum CORBA, Real-time CORBA, CORBA Messaging, and many different variations of CORBA services.
- *Core development team*. ACE+TAO are maintained and enhanced by a core—yet geographically distributed—team of ~40 developers. Many of these core developers have worked on ACE+TAO for years. There is also a continual influx of new developers into the core, many of who start out as graduate students at WUSTL or UCI.
- *Comprehensive source code control and bug tracking*. The ACE and TAO source code resides in a CVS repository, which provides revision control and change tracking. The CVS repository is hosted at the Center for Distributed Object Computing (DOC) at WUSTL and is mirrored at the UCI DOC Lab. External read-only access is available to the ACE+TAO user community via the Web. Write access is granted only to present and some former DOC members, and a few trusted outsiders. Software defects are tracked using the Bugzilla bug tracking system (BTS) (http://ace.cs.wustl.edu/bugzilla), which is a Web-based tool that helps ACE+TAO developers resolve problem reports and other issues in a timely and robust manner.
- *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 an average of 300+ 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 occasional "stable" releases.* Beta releases contain bug fixes and new features that are lightly tested on the platforms the core ACE+TAO team use for their daily development work. The usual interval between beta releases is relatively frequent, e.g., around every two to three weeks. In contrast, the so-called "stable" versions of ACE+TAO are released much less frequently, e.g., once a year. The stable releases are tested extensively on all the OS and compiler platforms to which ACE+TAO have been ported. The stable ACE and TAO releases are supported commercially by Riverace (http://www.riverace.com) and OCI (http://www.theaceorb.com), respectively.

One of the most novel aspects of the ACE+TAO open-source projects is their use of highly automated regression testing. The ACE and TAO software distributions contain both functional and performance tests. Many of the functional tests are used during regression testing. This regression test suite includes over 80 ACE tests and 240 TAO tests that serve several purposes:

1. *User acceptance and assurance*. After users build the ACE and/or TAO libraries on their machines, they are encouraged to run the appropriate tests to verify the integrity of the builds and any assumptions made about the operating platform.
2. *Continuous testing*. Build/test scripts run the ACE+TAO regression test suite run automatically several times a day to assure the integrity of the source code base.
3. *Unit testing*. Developers can run all or part of the test suite at any time, e.g., before committing their changes to the CVS repository.

The team of ~40 core developers of ACE and TAO run the regression tests on the 60+ workstations and servers at their respective sites in Irvine, California and St. Louis, Missouri. The interval between build/test runs ranges from three hours on quad-CPU Linux machines to nightly on less powerful machines.

One reason for the broad adoption of ACE+TAO in commercial projects is the high level of software quality resulting from its open-source development and automated regression-testing processes described above. It is important to recognize, however, that the core ACE+TAO developers cannot test all possible platform and OS combinations because there simply are not enough people, OS/compiler, platforms, CPU cycles, or disk space to run the hundreds of ACE+TAO regression tests in a timely manner. Moreover, ACE+TAO are designed for ease of subsetting and several hundred orthogonal features/options can be enabled/disabled for application-specific use-cases. Thus, there are a combinatorial number of possible configurations that could be tested at any given point, which is far beyond the resources of the core ACE+TAO open-source development team to handle in isolation.

The inability to regression test a broad range of potential configurations is problematic since it 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. Although we describe this problem in the context of ACE+TAO, it's actually an issue for any open-source project that exhibits the characteristics we describe above, such as Linux, the GNU compiler and language processing tools, Perl, and Apache. To address this problem effectively, the open-source community needs an integrated set of technologies and tools that can leverage the inherent assets of open-source software and the extensive resources of open-source user communities in order to:

- *Detect and resolve QA problems quickly*. To close the loop from users back to developers rapidly, it is necessary to exploit 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 should conduct different instrumentations and performance benchmarks automatically to collect metrics, such as static and dynamic memory footprint metrics, as well as throughput, latency, and jitter performance metrics.
- *Understand and enhance key system usability characteristics*, such as common usage errors.

The following section describes how we are addressing these issues in the Skoll project, which is helping to improve the quality and performance of ACE+TAO and similar open-source efforts by developing a scalable QA process based on continuous testing and profiling.

4. Overview of the Skoll Project

This section describes the Skoll project, focusing on its experimental design and contributions to open-source software engineering.

4.1. Prototype Experiments

Our goals for the Skoll project are ambitious. To maximize the chances of achieving these goals—as well as to minimize the cost of obtaining them—we are conducting a series of increasingly realistic and complex experiments in several stages. Our initial experiments occur in a highly controlled setting, i.e., using the existing 60+ workstations and servers distributed throughout the DOC labs at WUSTL and UCI. Based on this experience, we are refining our hypotheses, experimental designs, and analysis methods, and repeating experiments when necessary. Only after our results are encouraging will we

1. Scale up to less-controlled experiments involving a broad segment of the ACE+TAO open-source user community worldwide and
2. Fully implement the Skoll tools and testing environment.

4.2. Initial Hypothesis

For our initial experiment, we hypothesized that a systematic QA process based on distributed continuous testing and profiling would prove markedly superior to the *ad hoc* QA processes used by the open-source ACE+TAO project described in Section 3. We believe in the superiority of distributed continuous testing and profiling because it will:

1. Detect software bugs that would not have otherwise been found
2. Detect bugs more quickly (on the average) than current processes and
3. Pinpoint performance bottlenecks on a range of platforms more quickly than current processes.

Below, we use the phrase "QA process performance" to indicate the ability of a particular development process to satisfy the criteria enumerated above.

4.3. Operational Model and Instrumentation

Since we are conducting our experiments on specific releases of ACE+TAO, we have a substantial amount of information in the DOC CVS repository and Bugzilla BTS. For any particular release of ACE+TAO, this information indicates which bugs have occurred and the time and effort required to detect and resolve them. There may be some latent bugs in the ACE+TAO release for which we have no relevant data, though we will naturally be delighted to find and fix them! For the purposes of our experiments, however, we just compare the performance of the Skoll-enabled QA process with the existing *ad hoc* QA process based on previously known bugs. During this experiment we test ACE+TAO in a distributed fashion according to a schedule set by the experimental design described in Section 4.5. As bugs are found, we record them and note the time at which they are discovered. We then compare the performance of these QA metrics against the historical information collected by the Bugzilla BTS during previous releases of ACE+TAO.

4.4. Test Execution Model

ACE+TAO are designed using a family of patterns [Gam95, POSA2] that make it possible to select or subset their components into various *configurations*. Each configuration consists of particular settings of conditional compilation options that control which features or components are combined to customize ACE+TAO for a particular use-case. A key step in the distributed testing process is therefore determining which configuration to exercise next. We call this step the *test execution model* and are experimenting with the following four models:

1. *Ad hoc:* This is the approach used in the existing ACE+TAO QA process. We start with the most common configurations (e.g., build the complete libraries on Win2K, Linux, or Solaris) used by the ACE+TAO community and let the users and developers test the configurations that suit their particular needs.

2. *Random:* With this approach the next configuration to run is selected randomly without replacement, i.e., after a configuration is tested it is removed from further consideration.

3. *Classification-based:* Here we treat the choice of configuration as a type of search problem by recording the results of previous runs and determining whether we can identify options that are common in failing configurations. We use statistical classification methods [Porter91] to identify these common failure configurations.

4. *Modeled, Classification-based:* This model is similar to the previous approach, but incorporates external information that ACE+TAO developers have about the code. For example, if a certain feature is implemented across multiple configurations with the same source code, this information can be used to reduce the space of configurations that must be considered when doing classification.

4.5. Initial Experimental Design

Our initial experiments manipulate the following two *independent variables*:

1. The subject software (i.e., TAO or ACE) and
2. The test execution model (e.g., one of random, classification-based, or modeled, classification-based).

On each experimental run we compute the following four *dependent variables*:

1. The number of test cases run
2. The number and identity of any faults detection
3. The calendar time at which they were detected and
4. The number of testing sessions over which those faults remained undiscovered.

Using this information, we can quantify the performance of the various QA processes we're investigating, with respect to the criteria outlined in Section 4.2.

4.6. Data and Analysis of Results

> ***Note to reviewers****: we are currently conducting the initial experiment described above. Thus far, our approach has found bugs that were not found with the current ad hoc ACE+TAO QA process. Consequently, we are proceeding with the experiment and plan to include these results in the final version of this paper.*

4.7. Long-range Experiment Structure

As we work on our prototype experiments, we are also designing a longer-range set of experiments using ACE and TAO as our subject programs. These experiments will enlist the ACE+TAO user community as study participants to perform the following activities:

1. Use a web-based registration form through which study participants can characterize their testing platforms, i.e., their operating system and compiler/linker characteristics.
2. Use a mailing list to communicate with study participants. ACE+TAO users already communicate with the core developers via several heavily trafficked mailing lists (ace-users@cs.wustl.edu and tao-users@cs.wustl.edu) and the news://comp.soft-sys.ace USENET newsgroup.
3. Distribute a portable Perl script to automatically download ACE+TAO from the main CVS repository where it resides at WUSTL.
4. At periodic intervals designated by the study participants, every participant will be sent a configuration file that is customized for their platform. To provide coverage of the entire space of possible configuration options, this configuration file will be generated randomly, based on rules that ensure its semantic validity.
5. Participants will compile ACE+TAO using this configuration, run all the tests, and send any problems back to the core developers or link it into a "virtual scoreboard" (such as http://ringil.ece.uci.edu/scoreboard/) that can be examined later via a Web browser. Each core developer responsible for different components of ACE+TAO will have their own view into the problem results or virtual scoreboard so that they know what bugs to fix.
6. Steps 1 through 5 will be repeated continuously, running two types of tests:
    a. Tests to evaluate the syntactic and semantic correctness of ACE+TAO with respect to the generated configuration. Syntactic checks will be performed at compile-time. More advanced semantic checks will be performed by running regression tests. Two different approaches are reserved for the run-time problems
        i. Maximize diversity to find the greatest range of problems and
        ii. Focus the available resources to pinpoint the problem when problems are found.
    b. Tests that collect metrics on throughput, latency, scalability, predictability, and static/dynamic footprint to pinpoint when and why ACE+TAO's performance decreases on particular platforms.

Based on the results of the initial study outlined above, we plan to conduct more advanced studies that will use the results and feedback from temporally earlier experiments to focus subsequent tests (and subsequent versions) that are distributed around the world. For example, results of tests run in India can be used to guide the configurations tested in Europe six hours later. We also plan to automate error detection (e.g., via CVS rollback capabilities) and provide tools that will recommend a course of action to human "build czars" if errors cannot be detected automatically.

5. Concluding Remarks

Although software quality has historically lagged behind hardware quality, the maturation of R&D efforts on patterns [POSA2, Gam95], frameworks [John97], middleware [Schantz01, ACE01, TAO98], and open systems [OMG00] over the past decade have enabled the use of open-source technologies in an increasing number of application domains, such as:

- Distributed real-time and embedded applications, such as avionics mission computing, hot rolling mills, backbone routers, and high-speed network switches and
- Pervasive computing applications, such as wireless and wireline phones, PDAs, and other consumer electronic devices.

As the open-source community continues to improve the quality and ubiquity of its technologies, we believe its presence will increase in many—though by no means all—application domains. In particular, open-source may not be effective or desirable in the following domains:

- *Highly vertical domains*. Certain domains, such as distributed electronic medical imaging or enterprise resource planning, require highly specialized technical skills and are not standardized. As a result, it is hard to establish an open-source developer community since graduate students and programming aficionados rarely have sufficient domain-knowledge or interest to work on these topics.
- *Highly competitive domains*. In some domains, such as online trading systems, the financial incentives are so high that developers in these domains are not motivated to make their work freely available for open-source use by end-users or competitors.
- *Highly secure domains.* In other domains, such command and control systems for ballistic missile defense, security issues are paramount and there are national security/export laws that restrict the dissemination of certain types of software.
- *High-confidence domains*. In safety-critical domains, such as nuclear power plan process control or civil aviation flight-control, there are strict certification procedures that must be followed to validate that software is sufficiently robust for mission-critical operations. The development and QA processes for most open-source projects today lack sufficient rigor to pass these certification tests.

While we believe that it may ultimately be possible to apply techniques from the Skoll project to address concerns with using open-source in high-confidence domains, the business and security issues may preclude open-source from being adopted in the other three types of domains outlined above. Fortunately, open-source has proven to be an effective development process in many other software application domains [Raymond01]. The Skoll project described in this paper 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 is its 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 effective leverage of the expertise and extensive computing resources of user communities is essential to the long-term success of open-source projects.

Bibliography

[O'Reilly98] O'Reilly, T., *The Open-Source Revolution*, O'Reilly, 1998.

[BSD4.4] McKusick, M., Bostic, K., Karels, M., Quarterman J., *The Design and Implementation of the 4.4 BSD Operating System*, Addison Wesley, 1996.

[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*.

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

[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.

[Puder99] Puder, A.., Romer K., *MICO: An Open Source CORBA Implementation*, Morgan Kaufmann, 2000.

[Brose99] Brose G., *Java Programming with CORBA*, Wiley and Sons, 2001.

[Stallman] Stallman, R.M., "Using and Porting GNU CC," Free Software Foundation, 2001.

[Beowulf97] Ridge, D, Becker, D., Merkey P., Sterling T., "Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs," Proceedings of the IEEE Aerospace Conference, 1997.

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

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

[Boehm80] Boehm, B., *Software Engineering Economics*, Prentice Hall, 1981.

[CVS99] Hack, S., "CVS: Concurrent Versioning System," University of Illinois at Urbana-Champaign, Technical Report 99-011, http://www.itg.uiuc.edu/publications/techreports/99-011/.

[OMG00] Object Management Group, "The Common Object Request Broker: Architecture and Specification Revision 2.4, OMG Technical Document formal/00-11-07", October 2000.

[POSA2] Schmidt D., Stal M., Rohnert H., Buschmann F., *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, Wiley and Sons, 2000.

[H&V99] Henning M., Vinoski S., *Advanced CORBA Programming with C++*, Addison-Wesley Longman, 1999.

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

[Gam95] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

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

[John97] Johnson R., "Frameworks = Patterns + Components", *Communications of the ACM*, Volume 40, Number 10, October 1997.

[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.

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