Distributed Continuous Quality Assurance
Leveraging User Resources to Improve Software Quality Around-the-World, Around-the-Clock

Atif M. Memon, Adam Porter, Cemal Yilmaz, and Adithya Nagarajan
Computer Science Department
University of Maryland, College Park
{atif,aporter,cyilmaz,sadithya}@cs.umd.edu

Douglas C. Schmidt and Bala Natarajan
Electrical Engineering
& Computer Science Department
Vanderbilt University
{schmidt, bala}@dre.vanderbilt.edu

Abstract

Quality assurance (QA) tasks, such as testing, profiling, and performance evaluation, have historically been done in-house on developer-generated workloads and regression suites. Since this approach is inadequate for many systems, tools and processes are being developed to improve software quality by increasing user participation in the QA process. A limitation of these approaches is that they focus on isolated mechanisms, but not on the coordination and control policies and tools needed to make the global QA process efficient, effective, and scalable. To address these issues, we have initiated the Skoll project, which is 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.

This paper provides three contributions to the study of distributed continuous QA. First, it illustrates the structure and functionality of a generic around-the-world, around-the-clock QA process and describes several sophisticated tools that support this process. Second, it describes several scenarios implemented using these tools and process. Finally, it presents the results of a feasibility study applying these scenarios on two widely-used, large-scale open-source middleware toolkits.

The results of this study indicate that the Skoll process and its toolsuite can effectively manage and control distributed, continuous QA processes. In a matter of hours and days we identified problems that had taken the ACE and TAO developers substantially longer to find and several of which had previously not been found. Moreover, automatic analysis of QA task results often provided developers information that quickly led them to the root cause of the problems.

1. INTRODUCTION

Emerging trends and challenges. Software testing and profiling plays a key role in software quality assurance (QA). These tasks have often been performed in-house by developers, on developer platforms, using developer-generated input workloads. One benefit of in-house QA is that programs can be analyzed at a fine level of detail since QA teams have extensive knowledge of, and unrestricted access to, the software. The shortcomings of in-house QA efforts, however, are well-known and severe, including (1) increased QA cost and schedule and (2) misleading results when the test-cases and input workload differs from actual test-cases and workloads or when the developer systems and execution environments differ from fielded systems.

In-house QA processes are particularly ineffective for performance-intensive software, such as that found in (1) high-performance computing systems (*e.g.*, those that support scientific visualization, distributed database servers, and financial transaction processing), (2) distributed real-time and embedded systems that monitor and control real-world artifacts (*e.g.*, avionics mission- and flight-control software, supervisory control and data acquisition (SCADA) systems, and automotive braking systems), and (3) the operating systems, middleware, and language processing tools that support high-performance computing systems and distributed real-time and embedded systems. Software for these types of performance-intensive systems is increasingly subject to the following trends:

- **Demand for user-specific customization.** Since performance-intensive software pushes the limits of technology, it must be optimized for particular run-time contexts and application requirements. General-purpose, one-size-fits-all software solutions often have unacceptable performance.
- **Severe cost and time-to-market pressures.** Global competition and market deregulation are shrinking budgets for the development and QA of software in-house, particularly the operating system and middleware infrastructure. Moreover, performance-intensive users are often unable or less willing to pay for specialized proprietary infrastructure software. The net effect is that fewer resources are available to devote to infrastructure software development and QA activities.
- **Distributed and evolution-oriented development processes.** Today's global IT economy and n-tier architectures often involve developers distributed across geographical locations, time zones, and even business organizations. The goal of distributed development is to reduce cycle time by having developers work simultaneously, with minimal direct inter-developer coordination. Such development processes can increase churn rates in the software base, which in turn increases the need to detect, diagnose, and fix faulty changes quickly. The same situation occurs in evolution-oriented processes, where many small increments are routinely added to the base system.

As these trends accelerate, they present many challenges to de-

velopers of performance-intensive systems. A particularly vexing trend is the explosion of the software configuration space. To support customizations demanded by users, performance-intensive software must run on many hardware and OS platforms and typically have many options to configure the system at compile- and/or run-time. For example, performance-intensive middleware, such as web servers (*e.g.*, Apache), object request brokers (*e.g.*, TAO), and databases (*e.g.*, Oracle) have dozen or hundreds of options. While this flexibility promotes customization, it creates many potential system configurations, each of which may need extensive QA to validate.

When increasing configuration space is coupled with shrinking software development resources, it becomes infeasible to handle all QA in-house. For instance, developers may not have access to all the hardware, OS, and compiler platforms on which their software will run. Due to time-to-market driven environments, therefore, developers must often release their software in configurations that have not been subjected to extensive QA. Moreover, the combination of an enormous configuration space and severe development constraints mean that developers must make design and optimization decisions without precise knowledge of their consequences in fielded systems.

Solution approach: distributed continuous QA. The trends and associated challenges discussed above have yielded an environment in which the software systems tested and profiled by in-house developers and QA teams often differ substantially from the systems run by users. To address these challenges, we have begun a long-term, multi-site collaborative research project called Skoll.[1] This paper describes

- Skoll's *distributed continuous QA process* that leverages the extensive computing resources of worldwide user communities in order to improve software qualities and provide to greater insight into the behavior and performance of fielded systems.
- Skoll's tools and services, including its *model-driven intelligent steering agent* that controls and automates the QA process across large configuration spaces on a wide range of platforms.
- The results of a feasibility study that *empirically evaluates* Skoll's process and tools on two large, widely used open-source middleware toolkits, ACE [18, 19] and TAO [20].

Paper organization. The remainder of this paper is organized as follows: Section 2 examines related work and compares it with the approaches used in Skoll; Section 3 summarizes the structure and functionality of Skoll, focusing on its novel capabilities for controlling the distributed QA process; Section 4 outlines the key characteristics of the ACE and TAO middleware; Section 5 describes the feasibility study we conducted on ACE and TAO to evaluate Skoll empirically; and Section 6 presents concluding remarks and future work.

2. RELATED WORK

QA tasks have traditionally been performed in-house. For the reasons described in Section 1, however, in-house QA is increasingly being augmented with in-the-field techniques [13, 8, 15, 10]. Examples range from manual and reactive techniques (such as distributing software with prepackaged installation tests and encouraging end-users to report errors when they run into problems) to automated and proactive techniques (such as online crash reporting and auto-build scoreboard systems used in many open-source projects). This section describes existing processes and tools that support in-the-field QA and contrasts them with the distributed continuous QA approach used in Skoll.

Online crash reporting systems, such as the Netscape Quality Feedback Agent [5] and Microsoft XP Error Reporting [4], are small pieces of code embedded in software to gather data about what is happening in the software whenever it crashes. These agents simplify user participation in QA improvement by automating problem reporting. Auto-build scoreboards (*e.g.*, `http://tao.doc.wustl.edu/scoreboard/`) are distributed testing tools that allow software to be built/tested at multiple internal/external sites on various platforms (*e.g.*, hardware, operating systems, and compilers). Build and/or test results are gathered via common Internet protocols, web tools, and scoreboards are produced by summarizing the results as well as providing links to detailed information.

Various types of distributed testing mechanisms have been used in open-source software projects, such as GNU GCC [3], CPAN [2], Mozilla [7], VTK (The Visualization Toolkit) [6], and ACE+TAO [21]. These projects distribute test suites that end-users can run on their platform to gain confidence in their installation. Users can return the test results to the developers. Mailing lists are commonly used to improve the coordination between the developers and users who are willing to be testers.

Mozilla and ACE+TAO provide auto-build scoreboard systems that show the results of build results on various platforms, illustrating which platforms have built successfully and which fail to build and why. Bugs are reported via the Bugzilla [1] 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 [22].

VTK uses an auto-build scoreboard system called Dart [6] to reduce the burden on the end-users. Dart supports build/test sequences that start whenever something changes in the repository. Users install a Dart client on their platform and use this client to automatically check out software from a remote repository, build it, execute the tests, and submit the results to the Dart server.

Although the existing distributed QA efforts and tools help to improve the quality and performance of software, they have significant limitations. For example, since users decide (often by default) what features they will test, some configurations are tested multiple times, whereas others are never tested at all. Moreover, these approaches do not automatically adapt to or learn from the test results obtained by other users. The result is an opaque, inefficient, and *ad hoc* QA process.

To address these shortcomings, the Skoll project is developing and empirically evaluating a process, methods, and tools for around-the-world, around-the-clock QA that (1) works with highly configurable software systems, (2) uses intelligent steering mechanisms to efficiently leverage end-users resources in a QA process that adapts based on the analysis of previous results received from other sites, and (3) minimizes user effort through the judicious use of automated tools. We discuss these capabilities in the next section.

3. THE STRUCTURE AND FUNCTIONALITY OF SKOLL

As outlined in Section 1, the Skoll project is a long-term, multi-site collaborative research effort that is developing processes, methods, and tools to enable:

- **Substantial amounts of QA to be performed at fielded**

[1] Skoll is a Scandinavian myth that explains the sunrise and sunset cycles around the world.

sites using fielded resources, *i.e.*, rather than performing QA tasks solely in-house, Skoll pushes many of them to user sites. This approach provides developers and testers more effective access to user computing resources and provides visibility into the actual usage patterns and environment in which the fielded systems run.

- **Iterative improvement of the quality of performance-intensive software.** Skoll provides a control layer over the distributed QA process using models of the configuration space, the results of previous QA tasks, and navigation strategies that combine the two. This control layer determines which QA task to run next and on which part of the configuration space to run it. As the process executes, problems may be uncovered (and fixed) in the software, the models, and the navigation strategy.
- **Reduced human effort via judicious application of automation.** Skoll also uses the models developed in the previous step to help automate the role of human release managers, who monitor the stability of software repositories manually to ensure problems are fixed rapidly. In addition, Skoll uses automated web tools to minimize user effort and resource commitments, as well as ensure the security of user computing sites.

The approach we are taking to achieve these goals is based on a *distributed continuous QA process*, in which software quality and performance is iteratively and opportunistically improved around-the-clock in multiple geographically distributed locations. The Skoll project envisions distributed continuous QA via a geographically decentralized computing pool made up of thousands of machines provided by users, developers, and companies around the world.

The resources in the Skoll computing pool are scheduled and coordinated carefully via *data-driven feedback*. This adaptation is based on analysis of QA results from earlier testing tasks carried out in other locations, *i.e.*, Skoll follows the sun around the world and adapts its QA process continuously. The remainder of this section describes the components, tools, and key interactions in the Skoll architecture.

3.1 The Skoll Client/Server Architecture

To perform the distributed continuous QA process, Skoll uses a client/server architecture. Figure 1 illustrates the roles and components in this architecture, focusing primarily on the Skoll server and its interactions with various types of users. Figure 2 then shows the components in Skoll clients.

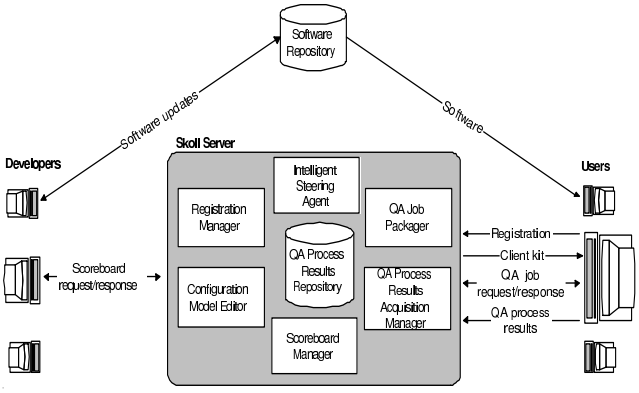

Figure 1: Components in Skoll Client/Server Architecture

User clients register with the Skoll *server registration manager* via a web-based registration form. Users characterize their client platforms (*e.g.*, the operating system, compiler, and hardware platform) from lists provided by the registration form. This information is stored in a database on the server and used by the Skoll *intelligent steering agent* (ISA). As described further in Section 3.2, the ISA automatically selects and then generates valid *job configurations*, which consists of the code artifacts, configuration parameters, build instructions, and QA tasks (e.g., regression/performance tests) associated with a software project. A job configuration also contains registration-specific information tailored for a particular client platform, along with the locations of the *CVS server* where the code artifacts actually reside.

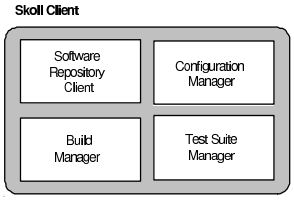

Figure 2: Skoll Client Architecture

After a registration form has been submitted and stored by the Skoll server, the *server registration manager* returns a unique ID and configuration template to the Skoll client. The template can be modified by end users who wish to restrict or specify what job configurations they will accept from the Skoll server. The Skoll client's architecture is shown in Figure 2. The Skoll client periodically requests job configurations from the server via HTTP POST requests. The server responds with a job configuration that has been customized by the server's *intelligent steering agent* in accordance with (1) the characteristics of the client platform, and (2) its knowledge of the valid configurations space and the results of previous QA tasks. The server maintains this information using the techniques described in Section 3.2.

The *CVS client* component is responsible for downloading software from the CVS repository. The information required to perform this task, such as the version and module name (CVS terminology) of the software, are sent in the job configuration. The *client configuration manager* component prepares the software by creating and/or customizing the appropriate header files. The instructions from the server provide mapping information between each parameter and the header file in which the parameter must be defined. The *client build manager* component builds the software by using the compiler specified at the registration time. The *client test suite manager component* is responsible for locating and executing the tests in the test suite.

For each job configuration, the Skoll client records all of its activities into a log file accessible from the Skoll server. Each log file consists of multiple sections, where each section corresponds to an operation performed by the client, such as CVS check out, build, and execute QA tasks. As the builds and tests complete, the client log files are sent to the Skoll server, which uses the *server QA process results acquisition manager* shown in Figure 1 to parse the log files and store them into a database.

Since the Skoll architecture is designed to support a user community with heterogeneous software infrastructures, developers must be able to examine the results without concern for platform compatibility and local software installation. We therefore employ web-based scoreboards that use XML to display the build and test results for job configurations. The *server scoreboard manager* provides a web-based scoreboard retrieval form to developers through

which they can browse a scoreboard for a particular job configuration. This component is responsible for retrieving the scoreboard data and creating the scoreboard GUI. Results are presented in a way that is easy to use, readily deployed, and helpful to wide range of developers with varying needs. For example, Figure 5 illustrates how Skoll uses a multi-dimensional, hierarchical data visualizer called Treemaps `http://www.cs.umd.edu/hcil/treemap` to display which configurations are passing or failing their build and test phases.

3.2 The Intelligent Steering Agent

Portions of the Skoll architecture described above are similar to those used by other distributed QA systems described in Section 2. A distinguishing feature of Skoll, however, is its use of an *intelligent steering agent* (ISA) to control the process. The ISA controls the process by deciding which configurations, in which order, to give to each incoming Skoll client request.

As QA tasks are carried out, their results are returned to the Skoll server and made available to the ISA. The ISA can therefore learn from past results, using that knowledge when generating new configurations. To accomplish this, the ISA performs automated constraint solving, scheduling, and learning. Consequently, we chose to implement the ISA using AI planning technology [23, 24]. Such technology has been used successfully in other QA efforts [11].

The configuration model The most basic element of ISA approach is a formal model of the system's configuration space. Each software system controlled by Skoll has a set of configurable options, each with a small, discrete number of settings. Each option value must be set before the system executes. Creating a job configuration, therefore, involves mapping each option to one of its allowable settings. This mapping is called the configuration and we represent it as a set $\{ (V_1, C_1), (V_2, C_2), \ldots, (V_N, C_N) \}$, where each V_i is a variable representing an option in the configuration and C_i is its value, drawn from a set of constants associated with V_i.

Since the configuration options may take many values, the *configuration space* can be quite large. Not all possible configurations are valid, however. We define which configurations are valid by imposing *inter-option constraints* on values of options. We represent the constraints using rules of the form $(P_i \rightarrow P_j)$ to mean that "if predicate P_i evaluates to $TRUE$, then predicate P_j must evaluate to $TRUE$" as well. A predicate P_k can be of the form $\neg A$, $A\&B$, $A|B$, or simply (V_i, C_i), where A, B are predicates, V_i is a option and C_i is its value.

Planning internals. Given Skoll's formal configuration model, we can cast the configuration generation problem as a planning problem. Given an *initial state*, a *goal state*, a set of *operators*, and a set of *objects*, the ISA planner returns a set of actions (or commands) with ordering constraints to achieve the goal. In Skoll, the initial state is the default job configuration of the software. The goal state is a description of the desired configuration, partly specified by the end user. The operators encode all the constraints, including knowledge of past test executions. The resulting plan is the configuration (*i.e.*, the mapping of options to their settings).

We use the standard Planning Domain Definition Language[2] (PDDL) to represent planning operators. Operators are specified in terms of parameterized preconditions and effects on variables, allowing intuitive expression of constraints. The ISA uses an efficient plan generator called the Interference Progression Planner (IPP) [9]. IPP uses extremely fast planning algorithms by converting the representation of a planning problem into a propositional encoding. Plans are then found by means of a search through a leveled graph, where *even levels* $(0, 2, \ldots, i)$ represent all the (grounded) propositions that might be true at stage i of the plan, and *odd levels* $(1, 3, \ldots i+1)$ represent actions that might be performed at time $i + 1$. The planners in the IPP family have shown increases in planning speeds of several orders of magnitude on a wide range of problems compared to earlier planning systems that rely on the full propositional representation and a graph search requiring unification of unbound variables.

We modified the Skoll planner so that, it can iteratively generate all acceptable plans, unlike typical planning systems that usually generate a single plan for a given set of constraints. Since the ISA generates multiple plans, we also added "navigation strategies," which are algorithms that allow us to schedule or prioritze among a set of multiple acceptable plans. This capability is useful in Skoll to cover a set of configurations, yet only proceed one step at a time in response to Skoll client requests. These algorithms also allow Skoll to add new information derived from previous QA task results to the planning process.

Planner output. The Skoll client requests a job configuration from a Skoll server. The Skoll server then queries its databases and (if provided by the user) a configuration template to determine which configuration option settings are fixed for that user and which must be set by the ISA. This information is packaged as a planning goal and sent to the ISA to be solved. Using this goal, the ISA planner generates a plan that is processed by the Skoll server, which ultimately returns all instructions necessary for running the QA task on the user's platform. These instructions are called the *job configuration.*

3.3 Skoll in Action

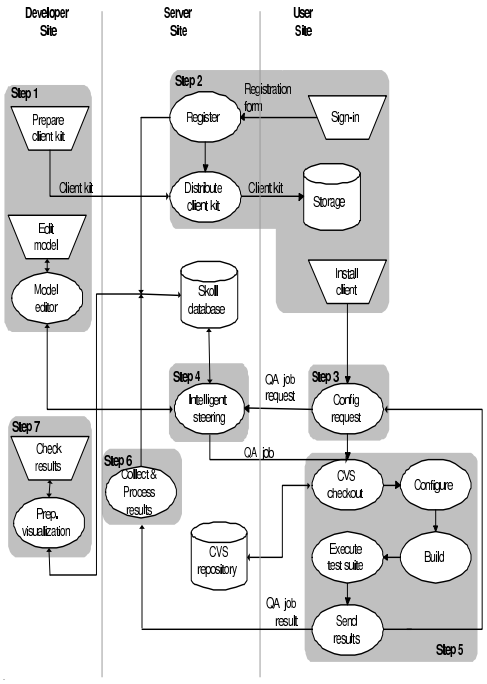

Figure 3: Process View

At a high level, the Skoll process is carried out as shown in Figure 3 and described below:

1. Developers create the configuration model and navigation strategies. The ISA configuration model editor then automatically translates the model into planning operators and stores

[2]Entire documentation available at `//www.cs.yale.edu/pub/mcdermott/software/pddl.tar.gz`

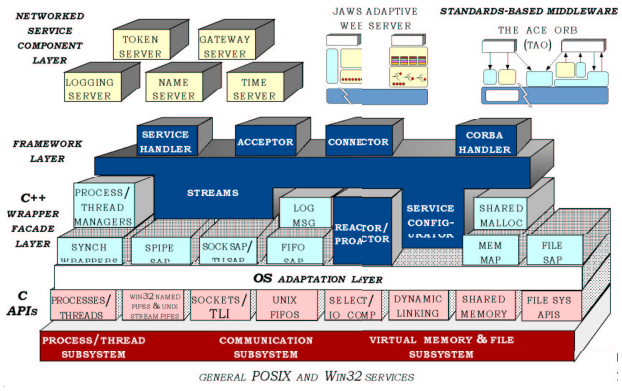

Figure 4: ACE+TAO infrastructure

them in an ISA database. Developers also create the client kit.

2. A *user* submits a request to download the software via the registration process described earlier. The user then receives the Skoll client software and a configuration template. If users wish to temporarily change configuration settings or constrain specific options they do so by modifying the configuration template.
3. The Skoll client periodically requests a job configuration from a Skoll server.
4. In response to a client request, the Skoll server queries its databases and the user-provided configuration template to determine which configuration option settings are fixed for that user and which must be set by the ISA. It then packages this information as a planning goal and queries the ISA. The ISA generates a plan and returns it to the Skoll server. Finally, the Skoll server creates the job configuration and returns it to the Skoll client.
5. The Skoll client invokes the job configuration and returns the results to the Skoll server.
6. The Skoll server examines these results and updates the ISA operators to reflect them.
7. Periodically and when prompted by developers the Skoll server prepares a *virtual scoreboard*, which depicts all known test failures and their details. It also performs statistical analysis of the failing options and prepares visualizations that help developers quickly identify large subspaces in which tests have failed.

4. ACE+TAO OVERVIEW

We are performing our initial studies on the ACE and TAO (ACE+TAO) open-source Project. ACE+TAO are large, widely-deployed open-source[3] middleware software toolkits that can be reused and extended to simplify the development of performance-intensive distributed software applications. ACE [16] is an object-oriented framework that implements core concurrency and distribution patterns for networked application software. The ACE ORB (TAO) is an implementation of the CORBA standard [12] that is constructed using the patterns [17] and framework components provided by the ACE toolkit. The components, layers, and relationships between the components/layers in ACE+TAO are shown in the Figure 4. ACE and TAO are ideal study candidates for the Skoll project because they share the following key characteristics common to performance-intensive infrastructure software.

[3]The ACE+TAO source can be downloaded at `deuce.doc.wustl.edu/Download.html`.

Table 1: Evolution of ACE and TAO source base

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

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.

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. ACE+TAO has a large number of interdependent options supporting a wide variety of program families [14] 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.

Distributed development. ACE+TAO are maintained and enhanced by a core yet geographically distributed team of $\sim$140 developers. Many of these core developers have worked on ACE+TAO for over five years. There is also a continual influx of new developers into the core.

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 and VU DOC Labs, as well as several other sites in Europe. Software defects are tracked using the Bugzilla bug tracking system (BTS) (`deuce.doc.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. For any particular release of ACE+TAO, Bugzilla indicates which bugs have been found and the time and effort required to detect and resolve them.

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. ACE+TAO have a dynamically changing and growing code base that has averages over 400+ 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.

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 must be tested extensively on all the OS and compiler platforms to which ACE+TAO have been ported.

Dependence of 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. Currently, the ACE+TAO developers run the regression tests continuously on 100+ workstations and servers at a dozen sites around the world (see `tao.doc.wustl.edu/scoreboard` for a list of sites). The interval between build/test runs ranges from 3 hours on quad-CPU Linux machines to 12-18 hours on less powerful machines.

It is important to recognize 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.

5. FEASIBILITY STUDY

The goals of the Skoll project described in Section 1 are ambitious. To achieve these goals, we are conducting a multi-step feasibility study based on the ACE+TAO open-source middleware toolkits described in Section 4. The initial steps of the study were conducted in a controlled setting, *i.e.*, using 10+ workstations and servers distributed throughout computer science labs at the University of Maryland (UMD). Our conjecture is that a Skoll-supported process will be superior to ACE+TAO's *ad hoc* QA processes outlined in Section 4 since it will (1) automatically manage and coordinate the QA process, (2) detect problems more quickly on the average, and (3) automatically characterize test results, giving developers more information as to potential causes of a given problem. This section describes a multi-phase feasibility study that implemented, executed, and analysed these study questions using three QA task scenarios applied to specific version of ACE+TAO.

5.1 ACE+TAO QA Process Scenarios

As discussed in Section 3, Skoll is designed to support a wide variety of QA tasks. Our initial study, however, focused on several basic scenarios, all involving software testing. The goal of these scenarios was to test ACE+TAO for different purposes across its numerous configurations. In addition, we wanted to give ACE+TAO developers useful feedback, specifically about the parts of the configuration space over which test cases failed. This information was provided in the form of concise descriptions of the subspace in which the failures occurred.

To carry out a QA process, the Skoll server must be able to tell Skoll clients what QA task to run and what configuration to run it on. For this study we executed three QA scenarios: (1) checking for clean compilation, (2) regression testing with default runtime options, and (3) regression testing with configurable runtime options. To perform these scenarios, we implemented the following components and integrated them into the Skoll architecture described in Section 3.1:

- We wrote textual representations of the configuration model. The Skoll server's configuration model editor automatically translated these into the ISA's planning language. We also hand-coded navigation strategies into the ISA for each testing scenario.
- We developed *client kits*, which are portable Perl scripts to be run by the Skoll client. These scripts make HTTP calls on a Skoll server to (1) request new QA job configurations, (2) receive, parse, and execute the QA jobs, and (3) make the results available to the server. QA jobs are transmitted in XML format and contain a test suite, the configuration option setting determined by the server, CVS instructions to download the source code, makefiles to build the software, and instructions for executing the test suite.
- We implemented the Skoll server as a servlet running in the Tomcat application server.
- We developed web forms to let users register and characterize their default testing platforms using HTTP calls.
- To manage user registration data and testing results, we developed schemas for a MySQL database.
- We developed scripts that created test scoreboards from the test results databases. The scoreboard shows all known test failures and their details. These scripts also prepared this data for Treemap visualization.

To use the Skoll process we defined several configuration models. The specific of these models are described in greater detail in Section 5.5.4. We also defined two general navigation strategies used by the ISA: *random walk* and *nearest neighbor search*. In the random walk strategy, the next configuration to investigate is chosen randomly without replacement *i.e.*, after a configuration is tested it is removed from further consideration. The basic goal of random walk is to ensure coverage of the valid configurations. If we need to investigate individual configurations several times, then random selection with replacement may be more desirable.

The nearest neighbor search starts by using the random walk strategy. When a QA task fails, however, it switches to a strategy designed to identify quickly whether similar configurations pass or fail. This information is used to characterize those configuration options whose values may influence whether or not the failure manifests itself. For example, suppose that a test on a configuration with three binary options fails in configurations $0, 0, 0$. If we then test $1, 0, 0$ and find that it does not fail, we assume that option 1's being 0 influences the failure's manifestation. Of course, if the underlying failure has nothing to do with the options, this assumption will prove false.

The nearest neighbor search strategy continues as follows: when configuration c fails we mark it as failed and record its failure information (*e.g.*, the first error message). We then schedule for testing all configurations that differ from c in the value of exactly one option, *i.e.*, test c's neighbors. Now continue the process recursively on all scheduled tests. If at any time there are no more scheduled tests, return to the random walk strategy on the remaining, untested configurations.

As the QA process runs, it generates one characterization for each observed failure. We conjecture that these characterizations will tend to overlap when configuration options are actually influencing failures, but be disjoint otherwise. We also conjecture that having these characterization will improve the developers' ability to assess and identify failure causes.

5.2 Operational Model and Test Execution Environment

Our feasibility study used TAO v1.2.3 with ACE v5.2.3 as the subject software. We used this version since it was considered to

be highly stable by the ACE+TAO developers and a susbstantial amount of information on previously-discovered bugs is recorded in the ACE+TAO bug tracking system. We downloaded Skoll clients and one Skoll server across 10+ workstations distributed throughout computer science labs at UMD. All Skoll clients ran Linux 2.4.9-3 and used gcc 2.96 as their compiler. We chose a single OS and compiler to simplify our initial study and analyses. Note, however, that the current Skoll implementation handles OS and compilers as just another set of configurable options, whose values can be used to select the machine-specific build instructions that are sent to Skoll clients.

As we identified problems with the ACE+TAO middleware during the study, we timestamped them and recorded pertinent information. This data allowed us to qualitatively compare Skoll's performance to that of ACE+TAO's *ad hoc* process. As discussed in Section 4, ACE+TAO's use of Bugzilla and CVS provide a wealth of historical data against which to compare the process performance improvements provided by Skoll.

5.3 Study 1: Compiling Different Features into ACE+TAO:

As with many other performance-intensive software toolkits, ACE+TAO allow a wide range of features that can be chosen during the initial compilation. Hence the QA task for our first study was to determine whether each of ∼80,000 possible ACE+TAO feature combinations that can be picked during compilation can be built without errors. This activity is important for systems like ACE+TAO that are distributed in source code form since the code base should either build fine or fail gracefully for any feature combination. Unexpected build failures can frustrate users. It is also important since building the million+ lines of ACE+TAO code takes time, *e.g.*, we averaged roughly 4 hours per complete build on a 933 MHz Pentium III with 400 Mbytes of RAM.

5.3.1 Configuration model

The feature configuration model for ACE+TAO has not been extensively documented. We therefore built our initial model bottom-up. First, we analyzed the ACE+TAO source and interviewed several senior ACE+TAO developers. From this we settled on a subset of 17 binary-valued compile-time options that control various features in ACE+TAO, including support for (1) asynchronous messaging, (2) software interceptors, and (3) user-specified messaging policies. These options are used to include/exclude specific CORBA features at build-time, *e.g.*, embedded applications typically disable many features like messaging policies to minimize memory footprint.

We also identified 35 inter-option constraints. One example constraint is ($AMI = 1 => MIN\_CORBA = 0$). This constraint is needed because the asynchronous method invocation (AMI) feature is not supported by the minimal CORBA implementation. In total, this option space and constraint set yields over 82,000 valid configurations.

We began Study 1 using the nearest neighbor navigation strategy described in Section 5.2. After testing ∼500 configurations, we realized that every one failed to compile. ACE+TAO developers determined that the problem was caused by 7 options designed to provide fine-grained control over CORBA messaging policies. Somewhere in the development process, CORBA messaging code was modified and moved to another library and developers (and users) failed to establish whether these options still worked.

Based on feedback from the Skoll project, ACE+TAO developers chose to fix the CORBA messaging problems by making these policies available at link-time, rather than during compilation. We therefore refined our configuration model by removing the 7 options and their corresponding constraints. Since these options appeared in many constraints—and because the remaining constraints are tightly coupled (*e.g.*, $A = 1 => B = 1$ and $B = 1 => C = 1$, etc.)—removing them simplified the configuration model considerably. As a result, the configuration model contained 10 options and 7 constraints, yielding only 89 valid configurations, which was much more manageable than 82,000!

5.3.2 Study Execution

We then continued the study using the new model and by switching to the random walk navigation strategy (nearest neighbor searching was not necessary since we could easily build all legal configurations). Of the 89 valid configurations only 29 compiled without errors. For the 60 configurations that did not build, we applied the nearest neighbor search strategy on the error reports to automatically characterize build failures by the configurations in which they failed.

5.3.3 Results and Observations

In addition to identifying failures in many configurations, in several cases the nearest neighbor characterizations[4] provided insight into the causes of failures. For example, the ACE+TAO build failed at line 630 in `userorbconf.h` (32 configurations) whenever $AMI = 1$ and $CORBA\_MSG = 0$. After investigating this, ACE+TAO developers determined that $AMI = 1 => CORBA\_MSG = 1$ was a missing constraint. Therefore, we refined the model by adding this constraint.

The ACE+TAO build also failed line 38 in `Asynch_Reply_Dispatcher.h` (8 configurations) whenever $CALLBACK = 0$ and $POLLER = 1$. ACE+TAO developers told us that such configurations should be legal. This was therefore a previously undiscovered bug. Until the bug could be fixed, we temporarily added a new constraint $POLLER = 1 => CALLBACK = 1$.

Likewise, the ACE+TAO build failed at line 137 in `RT_ORBInitializer.cpp` (20 configurations) whenever $CORBA\_MSG = 0$. ACE+TAO developers determined that the problem was due to a `#include` statement, missing because it was conditionally included (via a `#define` block) only when $CORBA\_MSG = 1$.

5.3.4 Lessons Learned

We learned several specific things from this study. First, we identified obtrusive, erroneous and missing model constraints. In some cases the proper fix was to refine the model. In others, ACE+TAO developers chose to change their software. We quickly identified coding errors that prevented the software from compiling in certain configurations. Errors that could be fixed easily were corrected before proceeding to the next study. To work around more complex errors, we added temporary constraints to our configuration model.

We also learned that as fixes to these problems are proposed, we can easily test them by spawning a new Skoll process that uses the failing configurations as new constraints, thereby forcing the ISA to retest them with the updated software. Finally, we learned that because of the automatic characterization of the nearest neighbor strategy, several of these errors were easier to find than might otherwise have been the case.

5.4 Study 2: Regression Testing with Default Runtime Options

The QA task for the second study was to determine whether each configuration would run the ACE+TAO regression tests without er-

[4]Note: when one option's value constrains another, we omit the constrained value from our description.

ror with the system's default runtime options. This activity is important for systems like ACE+TAO since the regression tests are packaged with the system and run when users install the system. To perform this task, users compile ACE+TAO, compile the tests, and execute the tests. For us, this process would have taken around 8 hours: about 4 hours to compile ACE+TAO, about 3.5 hours to compile all tests, and 30 minutes to execute them. We did not need to compile ACE+TAO for this study since we had saved the binaries from Study 1.

5.4.1 Configuration model

In this study we used 96 ACE+TAO regression tests. Since many of these tests are intended to run only in certain situations, we extended the existing configuration space (which contains compile-time options) and also created new test-specific options. To the compile-time options we added options that capture low-level system information, such as whether the system is compiled with static or dynamic libraries, whether multithreading support is enabled or disabled, etc.

The new test-specific options contain one option per test. These options indicate whether that test is runnable in the configuration represented by the compile time options. For convenience, we named these options $run(T_i)$. We also defined constraints over these options. For example, some tests should run only on configurations that implement the Minimum Corba specification. So for all such tests, T_i, we added a constraint $run(T) = 1 => MIN\_CORBA = 0$. These constraints prevented us from running tests that are bound to fail, thereby wasting resources and making problem localization hard. By default, we assume that all test are runnable unless constrained to be otherwise.

5.4.2 Study Execution

After making these changes, the compile-time option space had 14 options and 12 constraints and there were 96 test-specific options with an additional 120 constraints. We then modified the navigation strategies such that navigation is done over the compile-time constraints only, while still computing values for the test-specific options.

As with Study 1, we used the random walk navigation strategy to explore the configuration space. Here we only tested the 29 configurations that built in Section 5.3. In practice, we might have fixed some or all of the failing builds before continuing. To better understand the test failures we used the nearest neighbor search strategy on the error reports to automatically characterize failures by the configurations in which they failed.

5.4.3 Results and Observations

In this section we detail some of the results and lessons learned from this study. Overall, we compiled 2,077 individual tests. Of these 98 ended in a compilation failure, leaving 1,979 tests to run. Of these 152 failed, while 1,827 passed. This process took ∼52 hours of computer time.

In several cases tests failed for the same reasons over the same configurations. For example, test compilation failed at line 596 of `ami_testC.h` for 7 tests, each in the same 14 configurations when ($CORBA\_MSG = 1$ and $POLLER = 0$ and $CALLBACK = 0$). ACE+TAO developers determined that this was a previously undiscovered bug. It turned out that certain files within TAO responsible for implementing CORBA Messaging assumed that at least one of the $POLLER$ or $CALLBACK$ options would always be set to 1, which is an unwarranted assumption. ACE+TAO developers also noticed that the failure manifested itself when AMI was set to 1 or 0. This is actually a second problem because these tests should not have been runnable when $AMI = 0$. Consequently, there was a missing testing constraint, which we then included in the test constraint set.

Test `RTCORBA/Client_Protocol/run_test.pl` failed 25 out of 29 times. In this case, since the test failed in nearly all configurations we paid particular attention, not only to the characterization of failing configurations, but also to that of the passing configurations. We did not find any clear patterns in the passing configurations, however. This led us to believe that the problem was not related to configuration, but rather a more pervasive problem (either a bug in the test itself or configuration-wide software problem). In fact, ACE+TAO developers had seen this problem before and confirmed that it was due to a race their shared memory Internet object protocol (SHMIOP) implementation.

Another test, `Persistent_IOR/run_test.pl`, failed in exactly 1 configuration. This problem had not been previously seen by the ACE+TAO developers and, actually, even we had a hard time recreating it. In any event, we saw this as a success in that our automated approach identified the failure.

The test `MT_Timeout/run_test.pl` failed in 14 configurations with an error message indicating that the response time of certain requests exceeded allowable limits. Unlike in previous cases where the nearest neighbor strategy effectively returned one broad characterization covering many failing configurations, this time it returned many descriptions each of which covered few failing configurations. In other words, the failing configurations had nothing obvious in common with each other. This suggested to us that the error report might be covering multiple underlying failures, that the failure(s) manifests themselves intermittently, or that some other factor, not related to configuration options, is causing the problem. ACE+TAO developers informed us that they have seen this particular problem intermittently, and they believe it is related to inconsistent timer behavior on certain OS/hardware platform combinations.

5.4.4 Lessons Learned

We learned several things during Study 2. We were able to extend and refine the configuration model to allow new process functionality. These changes were easily handled by the ISA planner. We again were able to carry out a sophisticated QA process across networked user sites on a continuous basis. In this case, we exhaustively explored the configuration space in a few days and quickly flagged numerous real problems with ACE+TAO. Some of these problems had not been found with ACE+TAO's *ad hoc* QA processes.

We also learned several things about problem characterization. In particular, it may be possible (at least heuristically) to identify situations in which problems are likely or unlikely to be related to configuration options. That is, we may be able to determine when the characterization has a *strong signal* (problem likely configuration-related) or has a *weak signal* (probably not likely configuration-related).

A final lesson learned is that the current Skoll process cannot identify intermittent failures that are related to configuration options. This limitation is partly because Skoll is running each test only once. To identify these failures, Skoll will need to run individual tests multiple times, not just once. Otherwise, Skoll cannot distinguish intermittent configuration-related failures from intermittent failures not related to configuration options.

5.5 Study 3: Regression Testing with Configurable Runtime Options

The QA task for the third study was to determine whether each configuration would run the ACE+TAO regression tests without er-

Name	Possible Settings
ORBCollocation	global, per-orb, NO
ORBConnectionPurgingStrategy	lru, lfu, fifo, null
ORBFlushingStrategy	leader_follower, reactive, blocking
ORBConcurrency	reactive, thread-per-connection
ORBClientConnectionHandler	MT, ST, RW
ORBConnectStrategy	Blocked, Reactive, LF

Table 2: Six ACE+TAO Runtime Options and Their Settings.

ror over all settings of the system's runtime options. This activity is important for building confidence in the system's correctness. To perform this task, users compile ACE+TAO, compile the tests, set the appropriate runtime options, and execute the tests. For us, each task would have taken about 8 hours. For this study, however, we did not need to compile ACE+TAO or the tests since we had saved the binaries from our previous two studies.

5.5.1 Configuration Model

To examine ACE+TAO's behavior under differing runtime conditions, we modified the configuration to reflect the runtime configuration options. We decided to examine a subset of 6 such options. These options set upto 648 different values of CORBA runtime policies, such as how to flush cached connections and what concurrency strategies the ORB should support, as shown in Table 5.5.1. Since all of these runtime options are intended to be independent we did not need to add any new constraints.

We also discussed adding test constraints for tests that are not intended to run with certain option settings. We learned from the developers that such constraints probably exist, but that no one knew them all, knew them for sure, or had documented them. We therefore expected that initially many tests would fail, for reasons other than software faults.

After making these changes, the compile-time option space had 14 options and 12 constraints, there were 96 test-specific options with an additional 120 constraints, and there were 6 runtime options with no new constraints. We also modified the navigation strategies so that navigation is done over both the compile-time and the runtime constraints.

5.5.2 Study Execution

After the various steps outlined above, 18,792 valid configurations remained. At roughly 30 minutes per test suite, the entire process involved around 9,400 hours of computer time. Given the large number of configurations, we used the nearest neighbor navigation strategy from the outset, *i.e.*, in this study we used it to (1) select the configuration to be tested next and (1) automatically characterize failures by the configurations in which they failed.

Since the study has generated gigabytes of data, we only analyzed portions of it. As a result, we know that a number of tests have failed. ACE+TAO developers have traced several of these failures to previously identified bugs.

5.5.3 Results and Observations

In this section we detail some of the results and observations from this study. One interesting observation we made is that several tests failed in this study even though they had not failed in Study 2 (when running tests with default runtime options). Some even failed on every single configuration, despite not failing previously. In the latter case, the problems were typically caused by bugs in option setting and processing code, where as in the former case, the problems were often in feature-specific code. This finding was quite interesting to ACE+TAO developers because they rely heavily on testing by users at installation time, not just to verify proper installation, but to provide feedback on system correctness. It may also help explain why we did not observe many faults in Study 2.

Another group of tests had particularly interesting failure patterns. Three of these tests failed between 2,500 and 4,400 times. In each case the nearest neighbor characterizations singled out `ORBCollocation` = NO as only influential option. In fact, it turned out that this setting was in effect 3,475 of 3,490 times when Test `Big_Twoways/run_test.pl` failed, 4,391 of 4,412 times when `Param_Test/run_test.pl` failed, and 2,488 of 2,489 times when `MT_BiDir/run_test.pl` failed.

TAO's `ORBCollocation` option controls the conditions under which the ORB should treat objects as being co-located. The `NO` setting indicates that objects should never be treated as being collocated. When objects are not collocated they call each other's methods by sending messages across the network. When they are collocated, they can communicate directly, saving networking overhead.

The fact that these tests worked when objects communicated directly, but failed when they talked over the network clearly suggested a problem related to some aspect of message passing, *e.g.*, data marshaling, transmission, data unmarshaling, etc. In tracking down the underlying fault, ACE+TAO developers used this information to minimize the number of issues they had to consider. They discovered that the source of the problem was, in fact, a bug in their routines for marshaling/unmarshaling object references.

5.5.4 Lessons Learned

We learned several things as a result of conducting Study 3. First, we confirmed that our general approach could scale well to larger configuration spaces. We also reconfirmed one of key conjectures: that data from the distributed QA process can be analyzed to provide useful information to developers. At the same time, it is clear that we need better ways to summarize and visualize the testing data.

We also saw how the Skoll process gives better coverage of the configuration space than does that used by ACE+TAO (and, by inference, many other projects). We also note that our current implementation of the nearest neighbor navigation strategy continues to explore configurations until it finds no more failing configurations. In situations where a large subspace is failing, it might be useful at some point to make a statistical inference about the dimensions of the subspace and to stop the search. Of course, the inference could be incorrect, but time might be saved and used to explore other untested configurations.

Finally, we saw that in the time it took to compile the system we could run an entire test suite under several different configurations. This suggests an interesting future work item - adding cost metrics to the ISA's planning operators.

6. CONCLUDING REMARKS AND FUTURE WORK

This paper presents an overview of the Skoll project, which is designed to help resolve limitations with existing in-house and in-the-field QA processes. The primary focus of Skoll is "around-the-world, around-the-clock QA." Skoll is based on feedback-driven processes that leverage the extensive computing resources of worldwide user communities to significantly and rapidly improve software quality by intelligently steering the application of QA tasks in distributed and continuous manner.

To demonstrate the benefits of Skoll, this paper evaluated its impact via experiments on ACE [18, 19] and TAO [20]. ACE+TAO are production quality performance-intensive middleware consisting of well over one million lines of C++ code and regression tests contained in ∼4,500 files. Hundreds of developed around the world have worked on ACE+TAO for more than a decade, providing us with an ideal test-bed for our distributed continuous QA tools and processes.

The results presented in Section 5 provided valuable insight into the benefits and limitations of the current Skoll processes. Skoll can iteratively model complex configuration spaces and use this information to perform complex testing processes. As a result of those processes, we identified a number of test failures corresponding to real bugs, some of which had not been found. We also observed that developers benefitted greatly from our automatic problem characterization when localizing the root causes of certain test failures. These results enabled the core ACE+TAO developers to improve their code base. For example, managing CORBA messaging policies was improved considerably and a number of subtle coding errors were also identified and fixed.

Our future work is focusing on refining our hypotheses, study designs, analysis methods, and tools – repeating and enhancing experiments as necessary. In particular, we will run experiments that demonstrate empirically how Skoll can reduce the time needed to find/fix bugs in the ACE+TAO code base. As we gain more experience, we will extend the Skoll tools and extend the studies to larger-scale experiments. For example, we are currently replicating our feasibility study using the dozen test sites and hundreds of machines provided by the core ACE+TAO developers in two contintents (see `tao.doc.wustl.edu/scoreboard`). We ultimately plan to involve a broad segment of the ACE+TAO open-source user community in over fifty countries worldwide to establish a large-scale distributed continuous QA test-bed.

The sophisticated QA process provided by Skoll has motivated the ACE+TAO developers to undertake several bold new initiatives. For example, they are starting to refactor ACE to shrink its memory footprint and enhance its run-time performance. To faciliate this effort, we are working closely with the ACE+TAO developers to generalize Skoll's processes to cover a broader range of QA activities, including various performance measures. In particular, Skoll will be used to measure the footprint and performance at every check-in across different configurations while simultaneously ensuring correctness via automated and intelligent regression testing. We conjecture that the timely feedback provided by Skoll will significantly enhance the quality and productivity of this effort.

As we continue to automate key steps in the Skoll process, we are enhancing the design of the Intelligent Steering Agent (ISA) described in Section 3.2 as follows: (1) we are enriching the ISA's planner to include a cost model for each QA task so that it can make sophisticated decisions, *e.g.*, if the ISA knows that user X has already compiled the system (and compilation is expensive) it may let the user test several different *run-time* configurations, rather than those that require recompilation," (2) we are integrating the ISA planner with system input models to enable test case generation, (3) we are linking the ISA with an issue tracking database, such as Bugzilla, so that when problems are resolved the ISA will be notified, automatically remove the temporary constraints, and generate new job configurations that evaluate whether the problems are actually resolved, (4) we are continuing to investigate ways to visualize our highly multivariate QA data, and (5) we are investigated how and when to allow end users to submit their own tests for inclusion in Skoll-controlled test suites.

7. REFERENCES

[1] Bugzilla: Bug tracking system. http://www.bugzilla.org.

[2] Comprehensive perl archive network (cpan). http://www.cpan.org.

[3] Gnu gcc. http://gcc.gnu.org.

[4] Microsoft xp error reporting. http://support.microsoft.com/?kbid=310414.

[5] Netscape quality feedback system. http://www.netscape.com.

[6] public.kitware.com. http://public.kitware.com.

[7] Tinderbox. http://www.mozzila.org.

[8] J. Bowring, A. Orso, and M. J. Harrold. Monitoring deployed software using software tomography. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 2–9. ACM Press, 2002.

[9] J. Koehler, B. Nebel, J. Hoffman, and Y. Dimopoulos. Extending planning graphs to an ADL subset. *Lecture Notes in Computer Science*, 1348:273, 1997.

[10] B. Liblit, A. Aiken, and A. X. Zheng. Distributed program sampling. In *Proceedings of PLDI'03*, San Diego, California, June 2003.

[11] A. M. Memon, M. E. Pollack, and M. L. Soffa. Hierarchical gui test case generation using automated planning. *IEEE Transactions on Software Engineering*, 27(2):144–155, February 2001.

[12] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 3.0 edition, June 2002.

[13] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. Gamma system: continuous evolution of software after deployment. In *Proceedings of the international symposium on Software testing and analysis*, pages 65–69. ACM Press, 2002.

[14] D. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, March 1979.

[15] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *Proceedings of the 21st international conference on Software engineering*, pages 277–284. IEEE Computer Society Press, 1999.

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

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

[18] D. C. Schmidt and S. D. Huston. *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*. Addison-Wesley, Boston, 2002.

[19] D. C. Schmidt and S. D. Huston. *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Addison-Wesley, Reading, Massachusetts, 2002.

[20] D. C. Schmidt, D. L. Levine, and S. Mungee. The Design and Performance of Real-Time Object Request Brokers. *Computer Communications*, 21(4):294–324, Apr. 1998.

[21] D. C. Schmidt and A. Porter. Leveraging Open-Source Communities to Improve the Quality and Performance of Open-Source Software. In *First Workshop on Open-Source Software Engineering*, 23^{rd} *International Conference on Software Engineering*, May 2001.

[22] SourceGear Corporation. CVS.

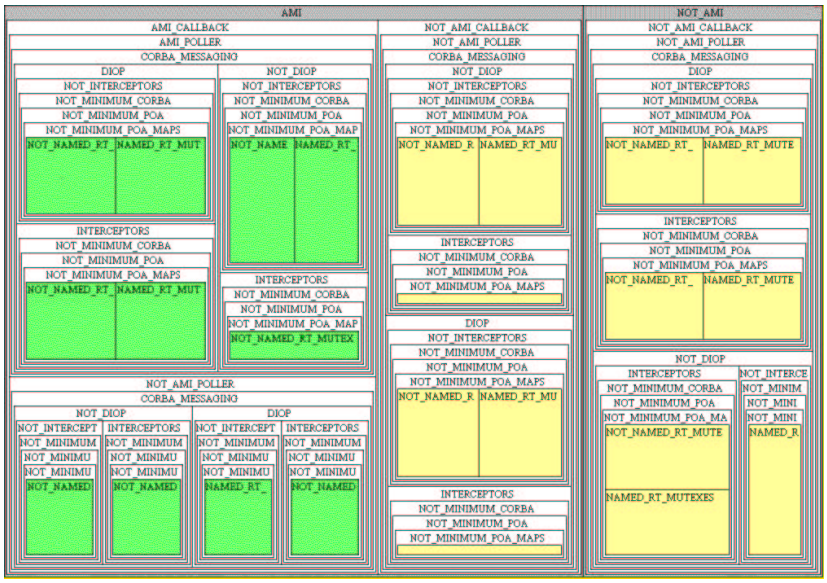

Figure 5: Treemaps Visualization: failing/passing configurations organized by failure characterization. Failing configurations are shown here with lighter color.

`www.sourcegear.com/CVS`, 1999.

[23] D. S. Weld. An introduction to least commitment planning. *AI Magazine*, 15(4):27–61, 1994.

[24] D. S. Weld. Recent advances in ai planning. *AI Magazine*, 20(1):55–64, 1999.