Skoll: Distributed Continuous Quality Assurance

A. Memon, A. Porter, C. Yilmaz, and A. Nagarajan D. Schmidt and B. Natarajan
University of Maryland, College Park *Vanderbilt University*

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, 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 several 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 QA scenarios built using these tools and process. Finally, it presents a feasibility study applying these scenarios to a 1MLOC+ software package called ACE+TAO. While much work remains to be done, the study suggests that the Skoll process and tools effectively manage and control distributed, continuous QA processes. Using Skoll we rapidly identified problems that had taken the ACE+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 quality assurance (QA) tasks are typically 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, input workload, software version and platform at the developer's site differ from those in the field. These problems are magnified in performance-intensive software, such as that found in high-performance computing systems, distributed real-time and embedded systems and the accompanying systems software (e.g., operating systems, middleware, and language processing tools). This is because this software 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. 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. In particular, customers are often unwilling to pay for customized software. The result is that limited resources are available for the development and QA of highly customizable performance-intensive software.
- **Distributed and evolution-oriented development processes.** Today's development processes are distributed across geographical locations, time zones, and business organizations. This is done to reduce cycle time by having developers work simultaneously, with minimal direct inter-developer coordination. But it can also increase software churn rates, which in turn increases the need to detect, diagnose, and fix faulty changes quickly. The same is true for evolution-oriented processes, where many small increments are routinely added to the base system.

These three trends present new challenges to developers. One major new challenge 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, 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 deserves extensive QA.

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. In this environment developers are forced to release software with configurations that have not been subjected to extensive QA. Moreover, the combination of an enormous configuration space and tight development constraints mean that developers must make design and optimization decisions without precise knowledge of the consequences in fielded systems.

Solution approach: distributed continuous QA. To address these challenges, we are conducting a collaborative research project called Skoll. Skoll is a general, continuous, feedback-driven process, supported by automated tools to carry out around-the-world, around-the-clock QA. Our approach divides QA processes into multiple subtasks that are intelligently distributed to client machines around the world, executed by them, and their results returned to central collection sites where they are fused together to complete the overall QA process.

Creating Skoll presented challenges for which we have created the following novel solutions, many of which are demonstrated in the feasibility studies of Section 4:

• **Modeling the QA subtask configuration space:** We formally model aspects of the QA subtasks and underlying software that will be varied under control of the distributed process. This includes not only process and software configuration parameters, but also constraints among them. To do this, we developed a general representation with *configuration options*, *option settings*, and *inter-option constraints*. We also developed the notion of *temporary inter-option constraints* to help us reduce configuration space size artificially in certain situations.

• **Exploring the configuration space:** The configuration space of a QA process for a performance-intensive infrastructure system can be quite large. Even with a large pool of user-supplied resources, brute-force approaches may be infeasible or simply undesirable. Consequently, we developed techniques to explore/search the configuration space. We developed a general search strategy based on *uniform sampling* of the configuration space and supplemented it with customized *adaptation strategies* to allow goal-driven process adaptation.

• **Feedback:** As subtasks are scheduled and executed in parallel at several sites, feedback (subtask results) is collected. QA processes can analyze this feedback and modify their behavior based on it. We have developed techniques for automatically characterizing such feedback, visualizing it, and adapting the QA process.

• **Resource availability:** Since QA subtasks are assigned to remote machines, volunteered by end users, we cannot know when resources will be available. Moreover, some volunteers may wish to maintain some control of how their resources will be used; for example limiting which version of a system can undergo QA on their resources. In such cases, it is impossible to pre-compute QA subtask schedules. Therefore, we have developed scheduling techniques that adapt based on a variety of factors including resource availability.

• **Process execution framework:** We developed a new process, called the Skoll process, that provides a flexible framework to integrate the above mentioned QA techniques and tools.

Paper organization. In the rest of the paper, we present in more detail, the Skoll process and infrastructure, QA processes built using Skoll, a feasibility study applying Skoll to the QA of two large software projects, and since Skoll is a new and evolving project, discuss directions for future work.

2. Related work

In this section we briefly discuss some current efforts in the area of distributed QA assurance and describe their major limitations on which Skoll tries to improve.

Online crash reporting systems, such as the Netscape Quality Feedback Agent and Microsoft XP Error Reporting, gather system state whenever a system crashes. This simplifies user participation in QA by automating problem reporting. Several other with distributed in-the-field techniques [7, 2, 6] have been developed as well. Each of these approaches however has a very limited scope, performing only a very small fraction of typical QA activities.

Many well-known projects, such as GNU GCC, CPAN, Mozilla, VTK (The Visualization Toolkit), and ACE+TAO, distribute test suites that end-users run to evaluate installation success. Users can, but quite often don't, return the test results to the developers. One limitation of this approach is that the process is undocumented. Developers have no record of what was tested or how it was tested or what the results were. We believe that a great deal of useful information is lost this way.

Auto-build scoreboards 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). The Mozilla and ACE+TAO projects use these systems to track build results across various platforms. Bugs are reported via the Bugzilla issue tracking system, which provides inter-bug dependency recording, advanced reporting capabilities, extensive configurability, and integration with automated software configuration management systems, such as CVS. While these systems help with documenting the QA process, the decision of what to test is left to end users. Unless developers can control at least some aspects of the QA process, important gaps and inefficiencies creep in.

The VTK project uses an auto-build scoreboard system called Dart [1]. Dart supports a continuous build and test process that can start whenever repository checkins occur.

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. One major limitation of this system is that the underlying QA process is hardwired. Other QA processes or other implementations of the build and test process are not supported easily. Once started, the process doesn't change. It cannot exploit incoming results nor avoid newly discovered problems, which leads to wasted resources and lost improvement opportunities.

Although existing distributed QA approaches help to improve the quality and performance of software, they have significant limitations. Many of these systems have limited **scope** – e.g., they can be used only when software crashes. General QA support needs to be much broader. Another problem is that many of these systems fail to **document** the QA activities that have been performed. It is therefore usually impossible to determine the full extent of (or gaps in) the QA process.These systems give developers little or no **control** over the QA process. Typically users decide (often by default) what aspects of the system they will examine; so some configurations are evaluated multiple times, others not at all. Finally, 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 processes.

3. The Skoll project

To address the shortcomings of current QA approaches, the Skoll project is developing and empirically evaluating processes, methods, and support tools for distributed, continuous QA. For our research, a distributed continuous QA process is one in which software quality and performance are improved – iteratively, opportunistically, and efficiently – around-the-clock in multiple, geographically distributed locations. To support such processes, we have implemented a general set of components and services that we call the Skoll infrastructure. We used this infrastructure to prototype some distributed, continuous QA processes for highly configurable, software program families. We have also evaluated this approach on a large-scale software project.

At a high level, Skoll processes resemble certain traditional distributed computations. General tasks are decomposed into many subtasks. Subtasks are then allocated to computing nodes, where they are executed. As subtasks run, control logic may dynamically steer the global computation for reasons of performance and correctness.

In Skoll, tasks are QA activities, such as testing, capturing usage patterns, and measuring system performance. They are broken down into subtasks, which perform part of the overall task. For example, a subtask might test a subset of system functions, monitor a subgroup of users, or measure performance under one particular workload characterization. The subtasks in one feasibility study in Section 4 do functional testing for a single, specific system configuration. The global process, by executing the right set of subtasks, does functional testing that "covers" the space of system configurations.

In Skoll, computing nodes are machines volunteered by end users. These nodes request work from a server when they decide they are available. Ultimately, we envision Skoll processes involving geographically decentralized computing pools made up of thousands of machines provided by users, developers, and companies around the world. This environment will allow large amounts of QA to be performed at fielded sites using fielded resources, giving developers unprecedented access to user resources, environments, and usage patterns.

Skoll's default behavior is to allocate subtasks upon request. No effort is made to optimize or adapt the global process based on subtask results. When more dynamic behavior is desired, process designers must write programs called "adaptation strategies." These programs monitor the global process state, analyze it, and modify how Skoll makes future subtask assignments. Here the goal is to steer the global process in a way that improves process performance (where improvement criteria are application specific).

The remainder of this section describes the components, services and interactions within the Skoll infrastructure and provides a sample scenario showing how they can be used to implement Skoll processes.

3.1. The Skoll infrastructure

Skoll processes are based on a client/server model, in which clients request job configurations (QA subtask scripts) from a server that determines which subtask to allocate, bundles up all necessary scripts and artifacts, and sends them to the client. To realize such a process however involves numerous decisions, e.g.; how tasks will be decomposed into subtasks; on what basis and in what order subtasks will they be allocated, how will they be implemented so they run on a very wide set of client platforms; how will results be merged together and interpreted, if and how should the process adapt to incoming results, and how will the results of the overall process be summarized and communicated to software developers. To support these issues we have developed several components and services for use by Skoll process designers.

Configuration Space Model. A cornerstone of our approach is a formal model of a QA process' configuration space. The model captures all valid configurations for QA subtasks. This information is used in planning the global QA process, for adapting the process dynamically, and to aid in interpreting results.

In our model, subtasks are generic processes parameterized by configuration options. Configuration options cap-

Table 1. Some options and constraints.

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

ture information (1) that will be varied under process control or (2) that is needed by the software to build and execute properly. Such options are application specific, but could include workload parameters, operating system, library implementations, compiler flags, run-time optimization controls, etc. Currently, each option must take its value from a discrete number of settings.

Defining a subtask then involves mapping each option to one of its allowable settings. We call this mapping a *configuration* and represent it as a set { (V_1, C_1), (V_2, C_2), ..., (V_N, C_N) }, where each V_i is a configuration option and C_i is its value, drawn from the allowable settings of V_i.

In practice not all configurations make sense (e.g., feature X not supported on operating system Y). We therefore allow *inter-option constraints* that limit the setting of one option based on the setting of another. We represent constraints as ($P_i \rightarrow P_j$), meaning "if predicate P_i evaluates to $TRUE$, then predicate P_j must evaluate to $TRUE$." A predicate P_k can be of the form A, $\neg A$, $A \& B$, $A|B$, or simply $V_i = C_i$, where A, B are predicates, V_i is an option and C_i is one of its allowable values. A *valid configuration* is a configuration that violates no inter-option constraints.

Table 1 presents some sample options and constraints taken from the feasibility studies in Section 4. The sample options refer to things like the end user's compiler (COMPILER); whether to compile in certain features (AMI, CORBA_MSG); whether certain test cases are runnable in a given configuration (run(T)), and at what level to set a run-time optimization (ORBCollocation). One sample constraint shows that AMI support requires the presence of CORBA messaging services. The other shows that a test can only run on a platform that uses the SUN CC compiler version 5.1.

Intelligent Steering Agent. A distinguishing feature of Skoll is its use of an *intelligent steering agent* (ISA) to control the global QA process. The ISA controls the global process by deciding which valid configuration to allocate to each incoming Skoll client request. Once the valid configuration is chosen, the ISA packages the corresponding QA subtask implementation, consisting of the application code, configuration parameters, build instructions, and QA-specific code (e.g., regression/performance tests) associated with a software project. This package is called a *job configuration*.

Skoll's formal configuration model lets us cast configuration selection and implementation as a planning problem. This problem requires automated constraint solving, scheduling, and learning. Consequently, we implemented the ISA using planning technology.

Given an *initial state*, a *goal state*, a set of *operators* (specified in terms of parameterized preconditions and effects on variables), and a set of *objects*, the ISA planner (currently Blackbox [5]) returns a set of actions (or commands) with ordering constraints that achieve the goal. In Skoll, the initial state is the base subtask configuration. The base subtask configuration includes any option settings that the ISA must not modify (e.g., the client machine's OS). The goal state describes the desired configuration, consistent with the option settings specified by the end user. The operators encode all the constraints, including those resulting from previously run subtasks. The output is the job configuration.

For many planning problems, a single plan is sufficient. For Skoll, however, we need to generate many or even all acceptable plans (i.e., subtask implementations). We therefore modified the Blackbox planner so that it can iteratively generate all acceptable plans. We also added a parameter to the ISA by which each acceptable plan is generated exactly once (*random selection without replacement*) or zero or more times (*random selection with replacement*).

Going back to our sample options in Table 1, if the process designer wants to ensure that all software configurations compile cleanly, he/she would use a configuration model without the test case- or runtime-specific options and would instruct the ISA to generate plans using the random selection without replacement strategy (each valid configuration is generated exactly once). If on the other hand the task were to capture performance measures on a wide variety of user machines, then the process designer might use all available options and have the ISA use the random selection with replacement strategy (which could generate specific valid configurations more than once).

Adaptation Strategies. As QA subtasks are performed, their results are returned to the ISA. By default, the ISA ignores these results. Often, however, we want to learn from incoming results. For example, when some configurations prove to be faulty, why not refocus resources on other unexplored parts of the configuration space. When such dynamic behavior is desired, process designers develop customized *adaptation strategies*, that monitor the global process state, analyze it, and use the information to modify future subtask assignments in ways that improve process performance.

In Skoll, adaptation strategies are independent programs executed by the Skoll server when subtask results arrive. This loosely couples Skoll and the adaptation strategies and allows us to develop, add, and remove adaptation strategies

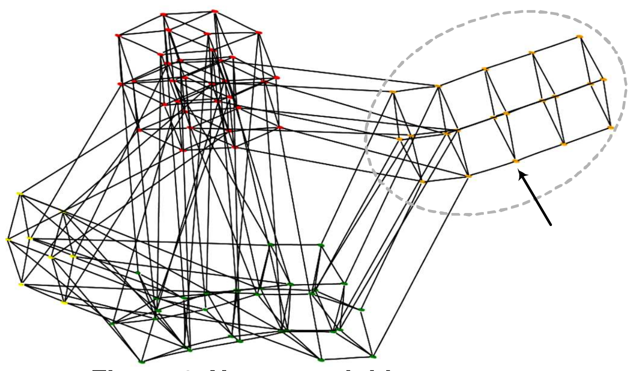
Figure 1. Nearest neighbor strategy.

at will. As they must process subtask results, adaptation strategies must be tailored for each QA process. Next we describe three general adaptation strategies used in later feasibility studies. Other strategies are discussed in Section 5.

• **Nearest neighbor:** Suppose a test reports a configuration in which test cases are failing. Developers might want to quickly identify other similar configurations that pass or fail. The nearest neighbor strategy is designed to generate such configurations. For example, suppose that a test on a configuration space with three binary options fails in configuration $\{0, 0, 0\}$. The nearest neighbor search strategy marks that configuration as failed and records its failure information. It then schedules for immediate testing all valid configurations that differ from the failed one in the value of exactly one option: $\{1, 0, 0\}$, $\{0, 1, 0\}$ and $\{0, 0, 1\}$, *i.e.*, all distance one neighbors. This process continues recursively. Figure 1 depicts the nearest neighbor strategy on a configuration space taken from our feasibility study. Nodes represent valid configurations; edges connect distance one neighbors. The dotted ellipse encircles configurations that failed for the same reason. The arrow indicates an initial failing node. Once it fails, its neighbors are tested; they fail so their neighbors are tested and so on. The process stops when nodes outside the ellipse are tested (since they will pass or fail for a different reason). As we show in the feasibility study, this approach quickly identifies whether similar configurations pass or fail. This information is then used by the automatic characterization service described later in this section.

•**Temporary constraints:** Suppose that a software incorrectly fails to build whenever configuration options AMI = 0 and CORBA_MSG = 1. Developers, however, are unable to fix the bug immediately. Therefore, we may want prevent further selection of job configurations with these parameters until the problem is fixed. This adaptation strategy therefore would insert *temporary constraints*, such as CORBA_MSG = 1 → AMI = 1 into the configuration model. This excludes configurations with the offending option settings from further exploration. Once the problem that prompted the temporary constraints has been fixed, the constraints are removed, thus allowing normal ISA execution. In fact, as we mention in our feasibility study (Section 4.2), we found these temporary constraints could be used to spawn new Skoll processes that test patches only on the previously failing configurations.

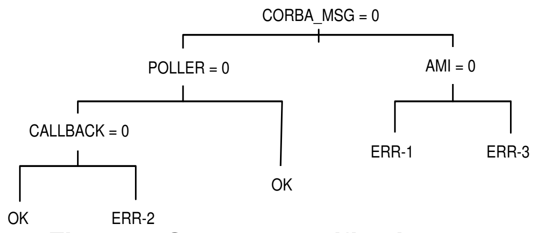

Figure 2. Sample classification tree.

•**Terminate/modify subtasks:** Suppose a test program is run at many user sites, failing continuously. At some point, continuing to run that test program provides little new information. Time and resources might be better spent running some previously unexecuted test program. This adaptation strategy monitors for such situations and, depending on how it is implemented, can modify subtask characteristics or even terminate the global process.

Automatic characterization of subtask results. Since QA processes can unfold over long periods of time, we often want to interpret subtask results incrementally. This is useful both for adapting the process and for providing developers with feedback. Given the amount and complexity of the data, this process must be automated.

To this end we have included implementations of Classification Tree Analysis (CTA) [3] in the Skoll infrastructure. CTA approaches are based on algorithms that take a set of objects, O_i, each of which is described by a set of features, F_{ij}, and a class assignment, C_i. Typically, class assignments are binary and categorical (e.g., pass or fail, yes or no), but approaches exist for multi-valued categorical, integer, and real valued class assignments. CTA's output is a tree-based model that predicts object class assignment based on the values of a subset of object features. Nodes contain predicates; when the predicate is true the right branch is followed; otherwise the left. Note the other approaches (beyond the scope of this paper) such as regression modeling, pattern recognition, neural networks, each with their own strengths and weaknesses, could be used instead of CTA.

We used CTA in our feasibility studies, for example, to determine which configuration option and their specific settings best explained observed test case failures. Figure 2 shows a classification tree model that characterizes 3 different compilation failures and 1 success condition for the results of 89 different configurations. The figure shows, among other things, that compilation fails with error message "ERR-1", whenever CORBA_MSG is disabled and AMI is enabled. It is also possible and often preferable to model different failure classes individually.

Visualization. To help developers organize and visualize large amounts of process results, we employ web-based scoreboards that use XML to display job configuration results. The *server scoreboard manager* provides a web-based query form allowing developers to browse Skoll databases for the results of particular job configurations.

Subtask Execution. This section presents several services aimed at implementing QA subtasks that will run on client machines.

End users register with the Skoll *server registration manager* via a web-based registration form, characterizing their client platforms. This information is used by the ISA when it selects and generates *job configurations* to tailor generic subtask implementation code. For example, some tailoring is for client-specific issues such as operating system type or compiler; some for task-specific issues such as identifying the location of the project's *CVS server*.

After a registration form has been submitted, the *server registration manager* returns a unique ID and configuration template to the end user. The configuration template contains any user-specific information that may not be modified by the ISA when generating job configurations. The template can be modified by end users who wish to restrict which job configurations they will accept from the Skoll server.

The end user also receives a Skoll client which periodically or on-demand requests job configurations from the server. The server responds with a job configuration that has been customized by the ISA using the techniques described in Section 3.1.

For each job configuration, the Skoll client logs its activities. When QA subtasks complete, the log files are sent to the Skoll server, where they are stored and processed by adaptation strategies.

3.2. Skoll in action

At a high level, the Skoll process runs as follows:

1. Developers create the configuration model and adaptation strategies. The ISA automatically translates the model into planning operators. Developers create the generic QA subtask code that will be specialized when creating actual job configurations.

2. A *user* requests Skoll client software via the registration process described earlier. The user receives the Skoll client 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 client periodically (or on-demand) requests a job configuration from a server.

4. The 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, creates the job configuration and returns it to the client.

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

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

7. Periodically and when prompted by developers the server prepares a *virtual scoreboard*, which summarizes subtask results and the current state of the overall process.

4. Feasibility study

The Skoll project's goals are ambitious. To help achieve them, we conducted a large feasibility study using the ACE+TAO projects. ACE + TAO are large middleware projects for performance-intensive distributed software applications. ACE [8] implements core concurrency and distribution services. TAO is a CORBA ORB built on top of ACE [9].

We chose these projects for several reasons. First, they share the key characteristics common to performance-intensive infrastructure software. They have a 1MLOC+ source code base and substantial test code. ACE+TAO run on dozens of OS and compiler platforms and are highly configurable, with hundreds of options supporting a wide variety of program families. ACE+TAO are maintained by a geographically distributed core team of ∼140 developers. Their code base is dynamically changing and growing with 400+ CVS repository commits per week on the avg. Currently, the ACE+TAO developers run the regression tests continuously on 100+ workstations and servers at a dozen sites around the world. The interval between build/test runs ranges from 3 hours on quad-CPU Linux machines to 12-18 hours on less powerful machines.

The second reason is that, like many performance intensive infrastructure systems, 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, since ACE+TAO are designed for ease of subsetting, several hundred orthogonal features/options can be enabled/disabled for application-specific use-cases. Thus, number of possible configurations is far beyond the resources of the core ACE+TAO development team.

We conjecture the Skoll infrastructure is easy to use and can implement a variety of QA processes. We also conjecture that our prototype Skoll QA process is superior to ACE+TAO's *ad hoc* QA processes as it (1) automatically manages and coordinates the QA process, (2) detects problems more quickly on the average, and (3) automatically characterizes subtask results, directing developers to potential causes of a given problem. This section describes a feasibility study that addresses these conjectures. We focus on several scenarios, testing ACE+TAO for different purposes across its numerous configurations. We used three QA task scenarios applied to a specific version of ACE+TAO: (1) checking for clean compilation, (2) testing with default runtime options, and (3) testing with configurable runtime options. In addition, we enabled automatic characterization to give ACE+TAO developers concise descriptions of failing subspaces. As we identified problems with the ACE+TAO, we time-stamped them and recorded pertinent information. This allowed us to qualitatively compare Skoll's performance to that of ACE+TAO's *ad hoc* process.

We installed Skoll clients and one Skoll server across 10+ workstations distributed throughout computer science labs at the University of Maryland. All Skoll clients ran Linux 2.4.9-3 and used gcc 2.96 as their compiler. We used TAO v1.2.3 with ACE v5.2.3 as the subject software.

4.1. Setting up the Skoll infrastructure

We implemented all the components of the Skoll infrastructure described in Section 3.1.

Configuration model: We developed configuration models for each scenario. The Skoll system automatically translated the models into the ISA's planning language.

ISA: We configured the ISA as a stand-alone process running the Blackbox planner and using random sampling without replacement.

Adaptation strategies: We implemented the nearest neighbor, temporary constraints, and terminate/modify subtasks adaptation strategies. We used temporary constraints and terminate/modify subtasks adaptation in each scenario, but used nearest neighbor only when the valid subtask configuration space was considered large. In practice, process designers determine the criteria for deciding when a configuration space is large or small.

Automatic Characterization: We developed scripts that prepare subtask results and feed them into the CTA algorithms. We also wrote scripts that used the classification tree models as input to visualizations.

Subtask execution: We developed portable Perl scripts to be run by Skoll clients. These scripts request new QA job configurations, receive, parse, and execute the jobs, and return results to the server. We also developed web registration forms and Skoll client software. Skoll clients are initialized with the registration information, but this information is rechecked on the client machine before sending a job request. We developed MySQL database schemas to manage user data and test results.

4.2. Study 1: Clean compilation

ACE+TAO allow many features to be compiled in or out of the system. Features are often left out, for example, to reduce memory footprint in embedded systems. The QA task for this study was to determine whether each ACE+TAO feature combination compiled without error. This is important for systems distributed in source code form, since any valid feature combination should compile. Unexpected build failures not only frustrate users, but also waste a lot of time. In fact, compiling the 1MLOC+ took us roughly 4 hours on a 933 MHz Pentium III with 400 Mbytes of RAM.

Configuration model. The feature interaction model for ACE+TAO is undocumented, so we built our initial configuration model bottom-up. First, we analyzed the source and interviewed several senior ACE+TAO developers. We selected a subset of 17 binary-valued compile-time options that controls build time inclusion of various CORBA features. We also identified 35 inter-option constraints. One constraint is (AMI = 1 $\rightarrow$ MIN_CORBA = 0). This means that asynchronous method invocation (AMI) is not supported by the minimal CORBA implementation. This configuration space has over 82,000 valid configurations.

Study execution. Because the configuration was large, we used the nearest neighbor adaptation strategy. We also configured the ISA to use random sampling replacement without replacement since we felt that one observation per valid configuration was sufficient.

After testing $\sim$500 configurations, the terminate/modify adaptation strategy signaled that every configuration had failed to compile. We terminated the process and discussed the results with ACE + TAO developers. The problem lay in 7 options providing fine-grained control over CORBA messaging policies. It turned out that the code had been modified and moved to another library and developers (and users) failed to establish if these options still worked.

Based on this feedback ACE+TAO developers chose to control these policies at link-time, not at compile time. We therefore refined our configuration model by removing the options and corresponding constraints. Since these options appeared in many constraints – and because the remaining constraints are tightly coupled (*e.g.*, were of the form (A=1 $\rightarrow$ B=1) and (B=1 $\rightarrow$ C=1)) – removing them simplified the configuration model considerably. As a result, the configuration model contained 10 options and 7 constraints, yielding only 89 valid configurations. Of course, this was just a small subset of the total, so the actual configuration is much larger than 89.

We then continued the study using the new configuration model and removing the nearest neighbor adaptation strategy (since now we could easily build all valid configurations). Of the 89 valid configurations only 29 compiled without errors. For the 60 configurations that did not build, automatic characterization helped to clarify the conditions in which they failed.

Results and observations. Beyond identifying failures, in several cases, automatic characterization provided concise, statistically significant descriptions of the failing configuration subspace. Below we describe the failure, present the automatically generated characterization, and discuss the action taken by ACE+TAO developers.

The ACE+TAO build failed at line 630 in `userorbconf.h` (32 configurations) whenever AMI = 1 and CORBA_MSG = 0. ACE+TAO developers determined that the constraint AMI = 1 $\rightarrow$ CORBA_MSG = 1 was missing from the model. 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. Since this configuration should be legal, this was determined to be a previously undiscovered bug. Until the bug could be fixed, we temporarily added a new constraint POLLER = 1 $\rightarrow$ CALLBACK = 1.

The ACE+TAO build failed at line 137 in `RT_ORBInitializer.cpp` (20 configurations) whenever CORBA_MSG = 0. The problem was due to a `#include` statement, missing because it was conditionally included (via a `#define` block) only when CORBA_MSG = 1.

Lessons learned. We found that even ACE+TAO developers do not completely understand the configuration model for their very complex system. In fact, they provided us with both erroneous and missing model constraints. Model building is therefore an iterative process. Using Skoll we quickly identified coding errors (some previously undiscovered) that prevented the software from compiling in certain configurations. We learned that the temporary constraints and terminate/modify subtasks adaptation strategies performed well, directing the global process towards useful activities, rather than wasting effort on configurations that would surely fail without providing any new information.

ACE+TAO developers also told us that automatic characterization was useful to them because it greatly narrowed down the issues they had to examine in tracking down the root cause of the failure. We also learned that as fixes to problems were proposed, we could easily test them by spawning a new Skoll process based on the previously inserted temporary constraints. That is, the new Skoll process tested the patched software only for those configuration that had failed previously.

Before moving on to the next study we fixed those errors we could. We worked around the most complex ones by leaving the appropriate temporary constraints in the second study's configuration model.

4.3. Study 2: 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 error with the system's default runtime options. This activity is important for systems that distribute tests to run at installation time because it is intended to give the user confidence that he or she has correctly installed the system. To perform this task, users compile ACE+TAO, compile the tests, and execute the tests. On our machines this took around 8 hours: about 4 hours to compile ACE+TAO, about 3.5 hours to compile all tests, and 30 minutes to execute them.

Configuration model. In this study we used 96 ACE+TAO tests, each containing its own test oracle and reporting success or failure on exit. These tests are often intended to run in limited situations, so we extended the configuration space, adding test-specific options. We also added some options capturing low-level system information, indicating the use of static or dynamic libraries, whether multithreading support is enabled, etc. This last step isn't strictly necessary since all clients were running Linux machines with the system software. We did it just to gain experience in modifying the configuration model.

The new test-specific options contain one option per test. They 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 with more than the Minimum CORBA features. So for all such tests, T_i, we added a constraint $run(T_i)$ = 1 $\rightarrow$ MIN_CORBA = 0. This prevents us from running tests that are bound to fail. By default, we assume that all test are runnable unless constrained to be otherwise.

Study execution. After making these changes, the space had 14 compile time options with 13 constraints and 96 test-specific options with an additional 120 constraints. We again configured the ISA for random sampling without replacement. We do not use the Nearest Neighbor adaptation strategy since we only tested the 29 configurations that built in Study 1. In this study, automatic characterization is done separately for each test and error message combination, but is based only on the settings of the compile time-options.

Results and observations. Overall, we compiled 2,077 individual tests. Of these 98 did not compile, 1,979 did. Of these, 152 failed, while 1,827 passed. This process took $\sim$52 hours of computer time. As in the first study we now describe some of the failures we uncovered, the automatically-generated failure characterizations, and the action taken by ACE+TAO developers.

In several cases, tests failed for the same reason on the same configurations. For example, test compilation failed at line 596 of `ami_testC.h` for 7 tests, each when

(CORBA_MSG = 1 and POLLER = 0 and CALLBACK = 0). This was a previously undiscovered bug. It turned out that certain files within TAO implementing CORBA Messaging incorrectly assumed that at least one of the POLLER or CALLBACK options would always be set to 1. ACE+TAO developers also noticed that the failure manifested itself no matter what the setting of the AMI was. This was a second previously undiscovered 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.

The test `MT_Timeout/run_test.pl` failed in 14 of 29 configurations with an error message indicating response timeout. No statistically significant model could be found. This suggests 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. It appears that particular problem appears intermittently and is related to inconsistent timer behavior on certain OS/hardware platform combinations.

Lessons learned. We easily extended and refined the initial configuration model to create more complex QA processes. 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 less than a day 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 automatic problem characterization. In particular, the generated models can be unreliable. We use notions of statistical significance to help indicate weak models, but more investigation is necessary. Also, the tree models we use may not be reliable when failures are non-deterministic and the ISA has been configured to generate only a single observation per valid configuration. In the presence of potentially non-deterministic failures, therefore, it may desirable to configure the ISA for random selection with replacement.

4.4. Study 3: Testing with configurable options

The QA task for the third study was to determine whether each configuration would run the ACE+TAO regression tests without error over all settings of the system's runtime options. This is important for building confidence in the system's correctness. To do this 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.

Configuration model. To examine ACE+TAO's behavior under differing runtime conditions, we modified the configuration model to reflect 6 multi-valued (non-binary) runtime configuration options. These options set up to 648 different combinations of CORBA runtime policies: when to flush cached connections, what concurrency strategies the ORB should support, etc. Since these runtime options are independent, we did not add any new constraints.

After making these changes, the compile-time option space had 14 options and 13 constraints, there were 96 test-specific options with an additional 120 constraints, and there were 6 runtime options with no new constraints.

Study execution. The configuration space for this study had 18,792 valid configurations. 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 adaptation strategy.

Results and observations. One observation 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 (including the default configuration tested earlier), despite not failing in Study 2! In the latter case, the problems were often caused by bugs in option setting and processing code. In the former case, the problems were often in feature-specific code. ACE+TAO developers were intrigued by these findings because they rely heavily on testing by users at installation time, not just to verify proper installation, but to provide feedback on system correctness.

Another group of tests had particularly interesting failure patterns. Three of these tests failed between 2,500 and 4,400 times. In each case automatic characterization showed that the failures occurred when `ORBCollocation` = NO. No other option influenced failure manifestation. In fact, it turned out that this setting was in effect over 99% of the time when Tests `Big_Twoways/run_test.pl`, `Param_Test/run_test.pl`, or `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 means that objects are never co-located. When objects are not co-located they talk to each other via the network. When they are co-located, they can communicate directly. The fact that these tests worked when objects communicated directly, but failed when they talked over the network clearly suggested a problem related to message passing. In fact, the source of the problem was a bug in their routines for marshalling/unmarshalling object references.

Lessons learned. We learned several things from Study 3. First, we confirmed that our general approach could scale well to larger configuration spaces. We also reconfirmed one of our key conjectures: that data from the distributed QA process can be analyzed and automatically characterized to provide useful information to developers. We also saw how the Skoll process gives better coverage of the configuration space than does the process used by ACE+TAO (and, by inference, many other projects).

We also note that our Nearest Neighbor adaptation strategy explores configurations until it finds no more failing configurations. In cases where a large subspace is failing a lot of work will be done (e.g., as described above in roughly 5,000 out of a total 20,000 configurations, `ORBCollocation` = NO and the test failed). Looking back at the data it's clear that we could have made stopped the search much earlier and still correctly identified the problem. We intend to explore this issue in the future.

5. Summary

This paper presents the Skoll project, a process and infrastructure for implementing feedback-driven processes that leverage worldwide user community resources to improve software quality. It also presents an initial feasibility study, applying Skoll to ACE+TAO – two large scale systems containing over 1MLOC. The study provides valuable insight into Skoll's current benefits and limitations.

Using Skoll, we iteratively modeled complex subtask configuration spaces, developed large scale QA processes, and executed them on multiple clients. As a result, we found real bugs in ACE+TAO, some of which had not been identified previously. ACE+TAO developers also reported that Skoll's automatic failure characterization greatly simplified tracking down the root causes of certain failures.

Skoll is a research project, so the feasibility study also helps direct future work. Our future work focuses on more experimentation, refining the infrastructure and extending functionality.

• We will continue to work with ACE+TAO project and are replicating our study using the dozen test sites and hundreds of machines provided by the core ACE+TAO developers in two continents. We ultimately plan to involve a broad segment of the ACE+TAO worldwide user community to establish a large-scale distributed continuous QA test-bed. Skoll is being used to help refactor ACE to shrink its memory footprint and increase its performance. We have developed new QA subtasks to measure ACE's footprint and performance at every check-in across different configurations while ensuring correctness via testing.

• We are enriching Skoll configuration models. We will add hierarchical models, not just the flat option spaces currently supported. We are incorporating priorities in the model so that different parts of the configuration space can be explored with different frequencies. We are also looking at how to incorporate real-valued option settings into configuration model.

• We are enhancing the ISA to allow planning based on cost models and probabilistic information, *e.g.*, if historical data suggests that users with certain platforms send requests at certain rates, it can take this information in account when allocating job configurations. We are also exploring higher level planners to simultaneously plan for multiple QA processes (not just one at a time as the ISA does now).

• We are investigating new adaptation strategies applying Design of Experiment (DOE) theory to subtask selection. The objective of DOE is to select a minimal set of input values (experimental design) that still allows one to determine which combinations of dependent variables (options and settings) significantly affect the independent variables (e.g., test failures, performance degradation). For example, suppose, we have 10 options with 4 possible settings each. Exhaustive testing of this space requires 4^{10} (> 1M) configurations to be tested whereas some DOE approaches that restrict themselves to *2*-way effects require only 25 configurations [4]. These approaches will be modified to compute schedules online as the QA process executes.

Acknowledgements

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

References

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

[2] 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.

[3] L. Breiman, J. Freidman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth, Monterey, CA, 1984.

[4] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn. Constructing test suites for interaction testing. In *Proceedings of the 25th international conference on Software engineering*, pages 38–48, 2003.

[5] H. Kautz and B. Selman. Unifying SAT-based and graph-based planning. In J. Minker, editor, *Workshop on Logic-Based Artificial Intelligence, Washington, DC, June 14–16, 1999*, College Park, Maryland, 1999. Computer Science Department, University of Maryland.

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

[7] 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.

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

[9] 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.