Skoll: A Process and Infrastructure for Distributed Continuous Quality Assurance

Adam Porter, Cemal Yilmaz, Atif Memon, Douglas C. Schmidt, and Bala Natarajan

Abstract— **Software engineers increasingly emphasize agility and flexibility in their designs and development approaches. For example, they increasingly use distributed development teams, rely on component assembly and deployment rather than green field code writing, rapidly evolve the system through incremental development and frequent updating, and use flexible product designs supporting extensive end-user customization. While agility and flexibility have many benefits, they also create an enormous number of potential system configurations built from rapidly changing component implementations. Since today's quality assurance (QA) techniques do not scale to handle highly configurable systems, we are developing and validating novel software QA processes and tools that leverage the extensive computing resources of user and developer communities in a distributed, continuous manner to improve software quality significantly.**

This paper provides several contributions to the study of distributed, continuous QA (DCQA). First, it shows the structure and functionality of Skoll, which is an environment that defines a generic around-the-world, around-the-clock QA process and several sophisticated tools that support this process. Second, it describes several novel QA processes built using the Skoll environment. Third, it presents two studies using Skoll: one involving user testing of the Mozilla browser and another involving continuous build, integration, and testing of the ACE+TAO communication software package.

The results of our studies suggest that the Skoll environment can manage and control distributed continuous QA processes more effectively than conventional QA processes. For example, our DCQA processes rapidly identified problems that had taken the ACE+TAO developers much longer to find and several of which they had not found. Moreover, the automatic analysis of QA task results provided developers information that enabled them to find the root cause of quality problems quickly.

Index Terms— **Categories: D.2.9 [Management]: software quality assurance; D.2.5 [Testing and Debugging]: testing tools**

General Terms: Reliability, Performance, Measurement, Experimentation

Keywords: Distributed continuous quality assurance, performance-oriented regression testing, design of experiment theory

I. Introduction

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 are well-known and severe, however, 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 modern software systems that are increasingly subject to two trends: *distributed and evolution-oriented development processes* and *cost and time-to-market pressures*.

Today's development processes are increasingly distributed across geographical locations, time zones, and business organizations [1]. This distribution helps reduce cycle time by having developers and teams work simultaneously and virtually around the clock, with minimal direct inter-developer coordination. Distributed development can also increase software churn rates, however, 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.

Global competition and market deregulation is encouraging the use of off-the-shelf software packages. Since one-size-fits-all software solutions rarely meet user needs in a wide range of domains, these packages must often be configured and optimized for particular run-time contexts and application requirements to meet portability and performance requirements. Due to shrinking budgets for the development and QA of software in-house, however, customers are often unwilling or unable to pay much for customized software. As a result, a limited amount of resources are available for the development and QA of highly customizable and performant software.

These trends present several new challenges to developers, including the explosion of the *QA task space*. To support customizations demanded by users, software often runs on multiple hardware and OS platforms and has 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.

In addition, QA processes themselves require ever more sophisticated and flexible control mechanisms to meet the wide-ranging and often dynamic QA goals of today's complex and rapidly changing systems. For instance, QA processes might want to control input workload characteristics, vary test case selection and prioritization policies, or enable/disable specific measurement probes at different times. In earlier work [2] we developed a QA process to isolate the causes of failures in fielded systems. In this process, different instances of a system enable different sets of measurement probes, thus sharing data collection overhead across the participating instances. In addition, the choice of which measurement probes to enable in a new program instance depends on each probe's historical ability (across all previous instances) to predict system failure [3].

When increasingly larger QA task spaces are 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 QA task space and tight development constraints mean that developers must make design and optimization decisions without precise knowledge of the consequences in fielded systems.

To address the challenges described above, we have developed a collaborative research environment called *Skoll* whose ultimate goal is to support continuous, feedback-driven processes and automated tools to perform QA around-the-world, around-the-clock. Skoll QA processes are logically divided into multiple tasks that are distributed intelligently to client machines around the world and then executed by them. The results from these distributed tasks are returned to *central collection sites* where they are merged and analyzed to complete the overall QA process.

When developing and operating Skoll we encountered a number of research challenges and created novel solutions. First, to understand the QA space it is necessary to *formally model* aspects of both the QA process and the system. For example, in our feasibility studies we found it helpful to model execution platforms, static system configurations, which build tools to use, runtime optimization levels, and which subset of tests to run. To support this modeling, we developed a general representation with options that take their values from a discrete set of option settings. We also developed a tool for expressing inter-option constraints that indicate valid and invalid combinations of options and settings. In addition, we developed the notion of temporary inter-option constraints to help us restrict the configuration and control space in certain situations.

Second, because the task space of a QA process can be large, brute-force approaches may be infeasible or simply undesirable, even with a large pool of supplied resources. We therefore developed techniques to *explore/search* the QA task space. We developed a general search strategy based on uniform random sampling of the space and supplemented it with customized adaptation strategies to allow goal-driven process adaptation. One adaptation strategy called *nearest neighbor* refocuses search around a failing configuration, *e.g.*, a point in the QA task space. This strategy helps find additional failing configurations quickly and delineates the boundaries between failing and passing QA task subspaces.

Third, because QA tasks are assigned to remote machines—often volunteered by end users—it may be hard to know *a priori* when resources will be available. For instance, some volunteers may wish to control how their resources will be used, *e.g.*, limiting which version of a system can undergo QA on their resources. In such cases, it is impossible to precompute QA task schedules. We therefore developed *scheduling techniques* that adapt the mapping of QA tasks to remote machines based on a variety of factors, such as resource availability and end user preferences.

Fourth, operating a distributed continuous QA process requires the integration of many artifacts, tools, and resources, such as models of QA process' task spaces, search/navigation strategies for intelligently and adaptively allocating QA tasks to clients, and advanced mechanisms for feedback generation, including statistical analysis and visualization of QA task results. We therefore developed the *Skoll process* that provides a flexible framework to coordinate the QA techniques and tools described above. As Skoll executes, QA tasks are scheduled and executed in parallel at multiple remote sites. The results of these subtasks are collected and analyzed continually at one or more central locations. The Skoll process can use adaptation strategies to vary its behavior based on this feedback. We have also developed techniques for automatically characterizing and presenting feedback to human developers.

Finally, it is hard to evaluate this kind of research since approaches are experimental and thus risky. At the same time, the work requires a distributed setting with multiple hardware platforms, operating systems, software libraries, etc. To deal with this we have developed a *large-scale distributed evaluation testbed* consisting of a pair of dedicated clusters at University of Maryland `www.cs.umd.edu/projects/skoll` and Vanderbilt University `www.dre.vanderbilt.edu/ISISlab` and containing over 225 top-end x86 CPUs running many versions of Linux, Windows, Solaris, Mac OSX, and BSD UNIX. They also have several terabytes of disk space for long-term data storage. We are enhancing these clusters with EMUlab control software developed in an NFS-sponsored testbed at the University of Utah to facilitate experimental evaluation of networked systems.

This paper significantly extends our previous work [4] by providing new information about the Skoll system and algorithms and substantially extending our empirical evaluation of software using Skoll. The remainder of this paper is organized as follows: Section II explains the Skoll process and infrastructure, QA processes built using Skoll; Sections III and Section IV describe the design and results from feasibility studies that applied Skoll to enhance the QA processes of two substantial software projects; Section V compares our work on Skoll with related work; and Section VI presents concluding remarks and discusses directions for future work.

II. The Skoll project

To address the limitations with current QA approaches (described in Section V), the Skoll project is developing and empirically evaluating processes, methods, and support tools for distributed, continuous QA. 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. Ultimately, we envision distributed continuous QA processes involving geographically decentralized computing pools made up of thousands of machines provided by end users, developers, and companies around the world. The expected benefits of this approach include: massive parallelization of

QA processes, greatly expanded access to resources and environment not easily obtainable in-house, and (depending on the specific QA process being executed), visibility into actual fielded usage patterns. This section describes our initial steps towards realizing our vision.

A. Distributed Continuous QA processes

At a high level, distributed continuous QA processes resemble certain traditional distributed computations as shown in Figure 1. As implemented in Skoll, *tasks* are QA activities, such as testing, capturing usage patterns, and measuring system performance. They are decomposed into *subtasks*, which perform part of the overall task. Example subtasks might execute test cases on a particular platform, test a subset of system functions, monitor a subgroup of users, or measure performance under one particular workload characterization. For example, the global QA task in Section IV's feasibility study performs functional testing that "covers" the space of system configurations, *i.e.*, each individual subtask executes a set of tests in one specific system configuration.

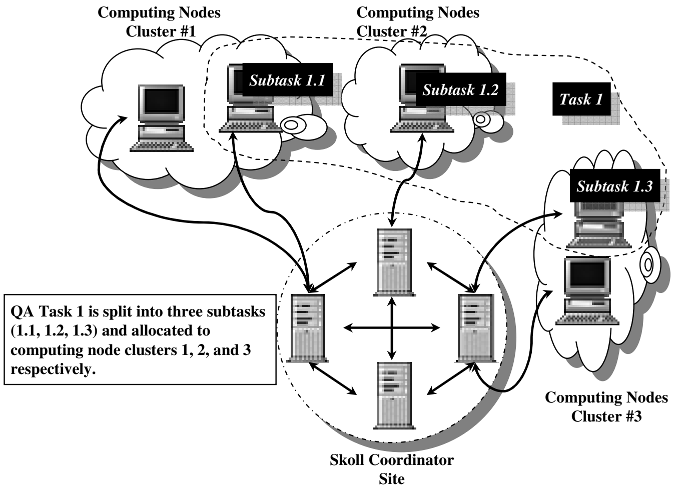

Fig. 1. Skoll Tasks/subtasks allocated to computing nodes over a network.

Skoll *allocates* subtasks to *computing nodes*, where they are executed. Computing nodes in Skoll are remote machines that elect to participate in specific distributed continuous QA processes. When these nodes decide they are available to perform QA activities they pull work from a *Skoll coordinator site*. As subtasks run, individual results are returned to *Skoll collection sites*, merged with previous results, and analyzed incrementally. Based on this analysis, control logic may dynamically steer the global computation for reasons of performance and correctness. In addition to incremental analysis, results may be analyzed manually and/or automatically after process completion to calculate the result of the entire QA task.

We envision Skoll QA processes involving geographically decentralized computing pools composed of numerous client machines provided by end users, developers, and companies around the world. This environment can perform large amounts of QA at fielded sites, giving developers unprecedented access to user resources, environments, and usage patterns.

Skoll's default behavior is to cover the configuration and control space by allocating subtasks upon request, on a random basis without replacement. The results of these subtasks are returned to collection sites and stored. They are not analyzed, however, so no effort is made to optimize or adapt the global process based on subtask results. When more dynamic behavior is desired, process designers can write *adaptation strategies*, which are programs that monitor the global process state, analyze it, and modify how Skoll makes future subtask assignments. The goal is to steer the global process in a way that improves process performance, where improvement criteria can be specified by users.

To support the distributed continuous QA processes described above, we have implemented a general set of components and services that we call the *Skoll infrastructure*. We have applied this infrastructure to prototype several distributed, continuous QA processes aimed at highly configurable software systems. We have also evaluated the Skoll infrastructure on two software projects, as described in Sections III and IV.

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

B. 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 involves numerous decisions, such as how will tasks be decomposed into subtasks, on what basis and in what order subtasks will be allocated, how will they be implemented so that they run on a wide set of client platforms, how results will 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 decisions, we have developed several components and services for use by Skoll process designers.

The first component is a formal model of a QA process' configuration and control space. The model essentially parameterizes all valid QA subtasks. This information is used in planning the global QA process, for adapting the process dynamically, and to help interprete the results.

In our model, subtasks are generic processes parameterized by configuration and control options. These options capture 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, OS version, library implementations, compiler flags, run-time optimization controls, etc. Each option must take its value from a discrete number of settings. For example, one configuration option (called `OS`) in our feasibility study in Section IV takes values from the set $\{$Win32, Linux$\}$. Skoll uses this option for a variety to tasks, *e.g.*, to select appropriate binaries and build code for the subtasks.

TABLE I
EXAMPLE OPTIONS AND CONSTRAINTS

Option	Settings	Interpretation
COMPILER	{gcc2.96, VC++6.0}	compiler
AMI	{1 = Yes, 0 = No}	Enable Feature
MINIMUM_CORBA	{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 → MINIMUM_CORBA = 0		
run(Multiple/run_test.pl) = 1 → (Compiler = VC++6.0)		

Defining a subtask involves mapping each option to one of its allowable settings. We call this mapping a *configuration*. Configurations are represented as a set { (V_1, C_1), (V_2, C_2), ..., (V_N, C_N) }, where each V_i is an 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 is not supported on OS 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 I presents some sample options and constraints from the feasibility study in Section IV (similar options appeared in the study described in Section III). The sample options refer to things like the end-user's compiler (COMPILER); whether or not to compile in certain features, such as support for asynchronous messaging invocation (AMI); 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 cannot be enabled on a minimum CORBA.

As QA subtasks are performed, their results are returned to an Intelligent Steering Agent (ISA). By default, the ISA simply stores these results. Often, however, we want to learn from incoming results, *e.g.*, when some configurations prove to be faulty, why not refocus resources on other unexplored parts of the configuration and control 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.

Since they must process subtask results, adaptation strategies must be tailored for each QA process. Consequently, adaptation strategies in Skoll are independent programs executed by the Skoll server when subtask results arrive. This decoupling of Skoll and the adaptation strategies allows us to develop, add, and remove adaptation strategies as needed. The following three general adaptation strategies are used in our feasibility studies in Sections III and IV (other strategies are discussed in Section VI).

The first ISA adaptation strategy is called *Nearest neighbor.* Suppose a test case run in a specific configuration reports a failure. Developers might want to focus on other similar configurations to see whether they pass or fail. The nearest neighbor strategy is designed to generate such configurations when the ISA is configured to choose configurations using random selection without replacement.

For example, suppose that a test on a configuration and control 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 setting: $\{1, 0, 0\}$, $\{0, 1, 0\}$ and $\{0, 0, 1\}$, *i.e.*, all distance-one neighbors. This process continues recursively.

Figure 2 depicts the nearest neighbor strategy on a configuration and control space taken from our feasibility study in Section IV-C. Nodes in this figure 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 either pass or fail for a different reason). As we show in the feasibility study IV-E, this approach quickly identifies whether similar configurations pass or fail. This information is then used by the automatic characterization service described later in Section II-C.

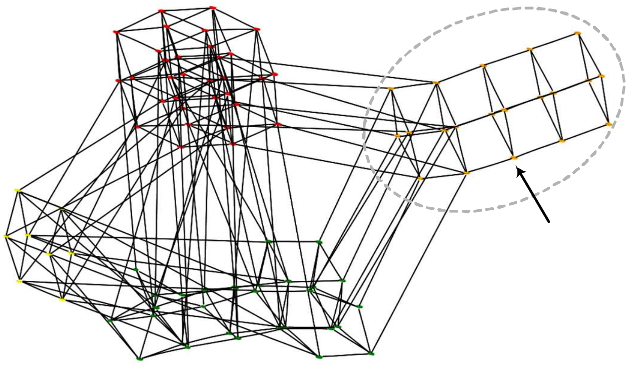

Fig. 2. Nearest Neighbor Strategy

The next ISA adaptation strategy involves the use of temporary constraints. Suppose that a software incorrectly fails to build whenever binary option AMI = 0 and binary option CORBA_MSG = 1. Suppose further that this fact can be discerned well before testing all such configurations (which comprise 25% of the entire configuration and control space). In this situation, developers would obviously want to stop testing these configurations and instead use their resources to test other parts of the configuration and control space.

To use resources more effectively, we therefore created an adaptation strategy that inserts *temporary constraints*, such as CORBA_MSG = 1 → AMI = 1 into the configuration model. This constraint 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 reenabling normal ISA execution. Once these constraints are negated they can be used to spawn new Skoll subtasks that test patches on only the previously failing configurations. We employ this strategy in our feasibility study in Section IV-C.

A third ISA adaptation strategy terminates or modifies 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.

The three ISA adaptation strategies described above are just some examples of the ones that we use in our work. As we encounter new situations, we implement new strategies. For example, we have observed that passing/failing configuration spaces are not necessarily contiguous, *i.e.*, failing subspaces may be disjoint. These situations cannot be found using the nearest neighbor strategy described above. We are therefore exploring the design of a variant of the nearest neighbor strategy that sometimes jumps across neighbors, with the goal of finding other failing subspaces that are disjoint from the subspace currently being explored.

Another useful Skoll component encapsulates the automatic characterization of subtask results. Since QA processes can unfold over long periods of time, we often want to interpret subtask results incrementally, which 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 help automate this process we have included implementations of Classification Tree Analysis (CTA) [5], [6] 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. Other approaches such as regression modeling, pattern recognition, neural networks, each with their own strengths and weaknesses, could be used instead of CTA, though they are beyond the scope of this paper.

We used CTA in our feasibility studies to determine which options and their specific settings best explained observed test case failures. Figure 3 shows a classification tree model that characterizes 3 different compilation failures and 1 success condition for the results of 89 different configurations. In this figure nodes contain predicates. When a predicate is true the right branch is followed, otherwise the left. This figure also shows 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.

Since Skoll processes are expected to generate large amounts of data, Skoll supports the organization and visualization 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. Visualizations are programmable with results presented in ways that are easy to use, readily deployed, and helpful to wide range of developers with varying needs. We also incorporated a multi-dimensional data visualizer called Treemaps `www.cs.umd.edu/hcil/treemap` to display the results of automatic characterization, which we described earlier.

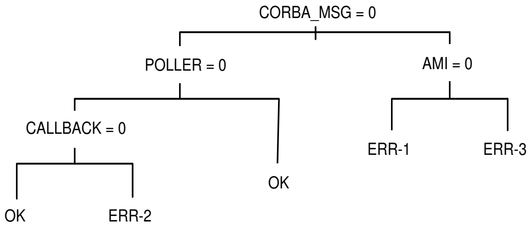

Fig. 3. Sample Classification Tree Model

C. Skoll Implementation

Skoll is designed as a client/server system. To ensure cross-platform compatibility, the Skoll system is written entirely in Perl and all communication between the Skoll server and clients is done in XML using the HTTP protocol, *e.g.*, via GET and POST methods. The remainder of this section shows Skoll's design details by tracing through a sequence of events that execute during a typical distributed, continuous QA process.

Since Skoll is designed to support a wide range of software systems and QA processes, it requires customization/-configuration before it can be used with a new software system and a QA process. Given a QA task, the first step in configuring Skoll is to create a configuration and control model that specifies how the QA task is divided into several subtasks. The interpretation and execution details of QA subtasks are application-specific and provided to the Skoll system by implementing two application-specific interfaces called `ServerSideApplicationComponent` (shown in Figure 4) and `ClientSideApplicationComponent`, respectively.[1]

A Skoll server uses the `ServerSideApplication-Component` to help the ISA interpret QA subtasks and create actual QA jobs. The Skoll server invokes the `init()` and `finalize()` methods just before starting a new DCQA process and immediately after finishing one, respectively. A Skoll client uses `ClientSideApplicationComponent` to execute the QA jobs sent by the ISA. The `init()` and `finalize()` methods of this component are invoked just before and immediately after executing a QA job, respectively.

End-users use a web-based form to register with the Skoll *server registration manager* and characterize their client platforms. This information is used by the ISA when it selects and generates job configurations to tailor generic subtask

[1] Note that, for clarity purposes, we simplified the interfaces given in this paper; in an actual implementation, these interfaces may be more complex.

```
interface ServerSideApplicationComponent {
  boolean init()
  Instructions QA_job(Configuration c)
  boolean finalize()
}

interface ClientSideApplicationComponent {
  boolean init(QAJob job)
  InstructionResult dispatch_instruction(
                          Instruction i)
  boolean finalize()
}
```

Fig. 4. Server- and Client-side Application Component Interfaces

implementation code. For example, some tailoring is for client-specific issues, such as operating system type or compiler, whereas other tailoring is 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 cannot 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 kit, consisting of cross-platform client software that provides basic services, such as contacting the Skoll server for jobs.

Once installed, the Skoll client periodically or on-demand requests QA jobs from the Skoll server. At each request, the Skoll client automatically detects information that describes its platform configuration, including its OS (*i.e.,* OS version, kernel version, vendor, etc.), compiler (*i.e.*, version, patches, etc.), and hardware specifications (*i.e.,* CPU details, number of CPUs, memory sizes, etc.). The ISA uses this information to guide the subsequent QA process, *e.g.*, to ensure that certain types of functional or performance regression tests run on the appropriate platform configuration. The Skoll client packages the platform configuration information together with the configuration template into a QA job request message (`QAJobReqMsg`) and sends it to the ISA.

The ISA responds to each incoming request with a QA job configuration (*i.e.,* QA subtask), which is customized in accordance with the characteristics of the client platform by the ISA. After a QA subtask is computed for a requesting client, the ISA consults the `ServerSideApplicationComponent` via the `QA_job()` method by passing the selected configuration as an argument. This method returns a set of instructions that assists the client in performing the assigned QA subtask. The ISA then packages these instructions and the selected configuration into a QA job (`QAJob`). A unique ID (`QAJobID`) is assigned to each QA job and stored in the Skoll database, along with the QA job information.

The Skoll client kit provides implementations for a set of generic instructions, *e.g.*, setting environment variables, downloading a software from a remote CVS repository, starting/-stopping a log, running system commands, uploading a log file, etc. Each instruction is implemented as a separate component that complies with a common interface, which ensures that Skoll's default instruction set can be expanded easily. Moreover, instruction components are loaded dynamically at runtime on demand, which allows online upgrades of Skoll client with a new set of instruction components even after deployment.

The Skoll client executes the set of instructions in the order they are received. Instructions that are not in the default set of instructions supported by the Skoll client are considered application-specific and passed to the `ClientSideApplicationComponent` component via the *dispatch_instruction()* method (Figure 4). This component is responsible for executing the instruction. Each application-specific instruction is implemented as a Perl package, conforming to a well-defined interface. The instruction interface includes method signatures for setting the environment variables to execute the instruction, executing the instruction, and logging and parsing the output of the instruction.

All client activities are stored into a log file that consists of multiple sections where each section corresponds to an instruction executed by the client (*e.g.*, "download" and "build"). After the QA subtask is completed, the client is often asked to parse the log file into an XML document, summarizing the QA subtask results.

QA job results are collected and stored in a database at the Skoll server. The Skoll database is implemented using MySQL and it contains tables to store information about clients (*e.g.*, OS, compiler, and hardware information, etc.), QA job configurations allocated (*e.g.*, QA job IDs, and the current status of the jobs, etc.), and the QA job results (*e.g.*, build results, functional test results, performance test results, etc.). After the database is populated, the ISA is notified about the incoming results. The ISA may use this information to modify future subtask allocation via adaptation strategies.

```
interface AdaptationStrategy {
  boolean init()
  Configurations adapt_to(QAJobID id)
  boolean finalize()
}
```

Fig. 5. Interface Between the ISA and Adaptation Strategies

Figure 5 shows the interface between the ISA and adaptation strategies. The `init()` and `finalize()` methods are called once DCQA processes start and finish, respectively. The ISA notifies the registered adaptation strategies via the `adapt_to()` method by passing the QA job ID as an argument. The adaptation strategies can then analyze the current state of the process and schedule configurations for future allocation.

The analysis and visualization of QA job results are application-specific. Depending on the characteristics of a particular QA task—and the preferences of developers—some analysis/visualization tools may be preferable to others. Skoll therefore provides a web-based portal to various analysis and visualization tools.

We have put together all the components discussed above to develop a Skoll process, which is described next.

D. Skoll in Action

The Skoll process performs the following steps using the components and services described in Section II-B:

Step 1. Developers create the configuration and control 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.

Step 2. A *user* requests the Skoll client kit via the registration process described earlier. The user receives the Skoll client and a configuration template. If users wish to temporarily change option settings or constrain specific options they do so by modifying the configuration template.

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

Step 4. The server queries its databases and the user-provided configuration template to determine which 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.

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

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

Step 7. Periodically—or when prompted by developers—the server updates a *virtual scoreboard* that summarizes subtask results and the current state of the overall process.

Sections III and IV present feasibility studies that demonstrate the usefulness of the Skoll process and tools described above.

III. Initial Feasibility Study of the Skoll Infrastructure and Process

To gain experience with Skoll, we developed and executed a distributed continuous QA process as an initial feasibility study. Since our goal was to gather experience, we simulated several parts of the process and ignored some issues (such as privacy and security) that would be important in an actual deployment with real end users. The application scenario is inspired by a software failure in version 1.7 of the Mozilla [7] web browser that Zeller et al. diagnosed using their Delta Debugging technique [8]. This bug occurs when a user attempts to print an HTML document containing the `select` tag (the select tag creates a drop-down list and allows users to choose one or more of its items) and results in the browser crashing (see bug report 69634 in `bugzilla.mozilla.org` for more information).

This scenario is a good test of Skoll because the failure's appearance depends on specific combinations of (1) input features (*i.e.*, an HTML document containing a `select` tag), (2) user actions (*i.e.*, printing an HTML document), and (3) execution platform (*i.e.*, version 1.7 of Mozilla, running in its default configuration.

We developed the following distributed continuous QA process to test Skoll. In practice this process would be executed by end-user machines. Clients on these machines would be divided into three groups, each of which is assigned to execute jobs coming from one of the three threads described below.

Thread-1 captures web pages for later testing by creating browser proxy components and deploy them to volunteer users. Browser proxies intercept client web page requests, retrieve the pages, and analyze them to determine whether they contains particular HTML tags (previously uncaptured) and do not contain other HTML tags (already captured and known to cause failures). Each proxy looks for a different set of HTML tags and the list of uncaptured tag sets is updated over time. If the currently requested page contains a desired tag set, users are asked to authorize sending the page to the Skoll server for further analysis. After a tag set has been found it is removed from the list of previously unseen tag sets.

Thread-2 tests the web pages captured in Thread-1. When a Thread-2 client becomes available, the ISA selects one previously captured web page, selects specific user actions to be applied to that page, and chooses the configuration under which Mozilla is to be run. The job is then sent to the client, which configures Mozilla, opens the page in it, and invokes an automated robot to carry out the selected user actions. The results (*i.e.*, failure or non-failure) are returned to the Skoll server.

Thread-3 applies Zeller's Delta Debugging [8], [9] algorithm (in parallel across multiple nodes) to minimize test cases that failed in Thread-2. Starting with such a failed test case, Delta Debugging works by removing a portion of test case and then retesting the remainder to determine whether the removed subset affects the failure's manifestation. This process continues until removing any more of the test causes the test to no longer fail. After this minimization is completed, the set of HTML tags remaining in the test case is added to the sets of HTML tags known to cause a failure. This action prevents Thread-1 from searching for further pages containing this set of tags.

The overall goal of this process is to gather a wide variety of web pages efficiently, where each page contains a diverse combination of HTML tags. We then test Mozilla in numerous run-time configurations by applying a wide set of user actions to the pages. When test fail, we use Zeller's Delta Debugging to help identify which subset of the test case (*e.g.*, HTML tags, user actions and runtime configurations) caused Mozilla to crash. This information is then fedback to the process to prevent retesting of conditions known to cause failure.

We implemented the process described above using Skoll, simulating several steps, such as the user interaction in Thread-1 that issues web page requests and the crashing of the Mozilla browser. We first developed a configuration and control model for this process, a subset of which is shown in Table II. This model captures four types of options: (1) the HTML tags that may be present in a web page, (2) user actions that may be executed on the page, (3) Mozilla run-time configuration

TABLE II
EXAMPLE CONFIGURATION MODEL

Option	Type	Settings
select	HTML tag	{1 = Yes, X = Don't Care}
table	HTML tag	{1 = Yes, X = Don't Care}
. . .		
print	User action	{1 = Yes, 0 = No}
bookmark	User action	{1 = Yes, 0 = No}
. . .		
safe-mode	Run-time option	{1 = Yes, 0 = No}
sync	Run-time option	{1 = Yes, 0 = No}
. . .		
Thread-ID	Client characteristic	{1, 2 or 3}

options, and (4) client characteristics. There were a total of 26 HTML tag options, 6 run-time configuration options, 3 user action options, and 1 client characteristic option, which induce an enormous configuration and control space over which we want to test. The Skoll system translated this model automatically into the ISA's planning language.

Next, we wrote the necessary QA subtask code that implements QA tasks, such as (1) preparing and deploying browser configuration scripts and browser proxy components that intercept web page requests and then analyze the requested web page to see whether it contains certain sets of HTML tags, (2) deploying and executing a test case, where executing a test case requires viewing one web page within a specific configuration of the Mozilla browser and then invoking a specific set of user actions on it, and (3) executing steps of the Delta Debugging algorithm to minimize the input to previously failed test cases.

To execute the QA process, we instructed the ISA to navigate the configuration and control space using random sampling without replacement, *i.e.*, each valid configuration was scheduled exactly once by having the ISA randomly select (1) HTML tags to locate in a webpage, (2) runtime configuration options for Mozilla, and (3) a set of user actions and the order in which they should execute. This configuration is then placed on both the Thread-1 and Thread-2 job list (only the HTML tags are important for Thread-1).

For each job request from a Thread-1 client, the ISA randomly selects a job from the Thread-1 job list. Next, a browser proxy component is configured to search for the HTML tags indicated by the configuration. The browser proxy is then deployed on the client machine. To facilitate the demonstration, we configured the proxy to generate and return canned web pages containing the required tags.

Upon receiving a job request from a Thread-2 client, the ISA randomly selects a job from the Thread-2 job list for which the corresponding web page has already been captured by a Thread-1 client. The ISA then builds a job package directing the client to (1) download the Mozilla software from a remote repository, (2) configure it, (3) open the HTML document in the browser, (4) execute the sequence of user actions using a GUI test automation tool called GUITAR [10], and then (5) send the results (*i.e.*, Mozilla crashed or did not crash as determined by our simulated oracle) to the Skoll server. Each step was realized as an individual instruction in the QA jobs sent by the Skoll server. We implemented only the application specific instructions for step (3) and (4). For the rest, we used Skoll's default set of instructions. When a test case fails in Thread-2, it is subdivided into two pieces as dictated by the Delta Debugging algorithm and both pieces are added to the Thread-3 subtask queue.

For each Thread-3 job request, a job (if available) is pulled from the Thread-3 job queue and tested using the same infrastructure as in Thread-2. If the test fails (as determined by our simulated oracle), the Delta Debugging algorithm is invoked. This recursive process reduces the input and retests looking to create the smallest it can that still fails. The algorithm stops when any further reduction of test case causes the failure to disappear. After the algorithm terminates, we manually analyze the results to identify the failure inducing tag sets. We then used adaptation strategies to create temporary constraints that prevent the creation of new QA subtasks involving these tags.

We spent ∼30 person hours directly implementing this scenario using the Skoll infrastructure. Two-thirds of this time (∼20 person hours) was sent fixing bugs in Skoll uncovered during our exploration. To run the process, we installed ten Skoll clients and one Skoll server across workstations distributed throughout computer science labs at the University of Maryland. All Skoll clients ran on Linux 2.4.9-3 platform. We used Mozilla v1.7 as our subject software. We then executed various QA processes for over 100 hours as described above.

As a result of conducting the initial feasibility study on Mozilla we gained valuable experience with our Skoll process and infrastructure. For example, we demonstrated that Skoll's configuration and control model was sufficient to define a test space consisting of subspaces for input cases (*i.e.*, HTML tag options), user actions (*i.e.*, user action options), traditional software configuration options (*i.e.*, Mozilla's run-time configuration options), and client characteristics (*assignment of clients to thread pools*). One deficiency of the model was that it did not naturally support ordering among different options. Specifically, we wanted to generate an ordering to the user actions (*e.g., first bookmark, then print*). Although this can be done using constraints, doing so is quite cumbersome. We therefore chose to model only the presence or absence of each action in the current test case and then randomized the order of their application.

Our feasibility study also demonstrated the generality of the Skoll system and distributed continuous QA concepts by developing and executing a simple yet interesting process at a reasonable level of effort. We were able to integrate an existing test automation tool (*i.e.*, a GUI test automation tool [10]), an analysis technique (*i.e.*, Delta Debugging algorithm), and an adaptation strategy (*i.e.*, temporary constraints) within the Skoll infrastructure. During this activity we encountered serious difficulties implementing this process because our initial Skoll implementation hardwired many aspects of the DCQA process workflow. This experience led us to create the APIs described in Section II-C, which greatly simplify adding application-specific QA instructions to a DCQA process.

Finally, we also uncovered some practical limitations. For example, we found we needed a debugging mode for Skoll since there was no easy way to see what actions would happen during a process without actually executing them, using resources, and updating Skoll's internal databases. We therefore added a feature to Skoll that echoes the instructions that should be executed, but does not actually run them (this behavior is similar to what happens when the `make` program is run with the "-n" flag). We also identified a need for a systematic method of terminating outstanding jobs or classes of jobs. For instance, while running Delta Debugging we often found solutions on one branch of the recursively defined algorithm. At this point there was no reason to continue running jobs from other branches, so we had to extend Skoll to shut them down safely.

IV. A Multi-platform Feasibility Study of Skoll

Based on the success of our initial Mozilla feasibility study described in Section III, we decided to explore the use of Skoll on a larger project: ACE+TAO, where ACE [11] implements core concurrency and distribution services and TAO [12] is a CORBA object request broker (ORB) built using ACE. We conjectured that the Skoll prototype would be superior to the *ad hoc* QA processes used by ACE+TAO developers because it (1) automatically manages and coordinates QA processes, (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 the results of our second feasibility study that addressed these conjectures.

A. Motivation and Design of the ACE+TAO Study

We chose ACE+TAO since they embody many of the challenging characteristics of modern software systems. For example, they have a 2 MLOC+ 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 ∼40 developers whose code base changes dynamically and averages over 400+ CVS repository commits per week.

The ACE+TAO developers currently run the regression tests continuously on 100+ largely uncoordinated workstations and servers at a dozen sites around the world. The results of their testing appear at `www.dre.vanderbilt.edu/scoreboard`. The interval between build/test runs ranges from 3 hours on modern quad-CPU machines to 12-18 hours on less powerful machines. The platforms vary from different versions of UNIX (*e.g.*, Solaris, AIX, HP, and Linux) to Windows (Windows XP, Windows 2000, Windows CE) to Mac OS, as well as to real-time operating systems, such as VxWorks and LynxOS.

Another motivation for choosing ACE+TAO is that their 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. The number of possible configurations is thus far beyond the resources of the core ACE+TAO development team. These characteristics of ACE+TAO are similar to much other complex software-intensive systems.

Our study applied several QA scenarios to ACE+TAO, testing it 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. We also 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, which allowed us to qualitatively compare Skoll's performance to that of ACE+TAO's *ad hoc* process. The tasks involved in these scenarios are typically done by developers and involve fairly heavyweight activities, such as downloading a large code base from CVS, compiling the system, and running resource intensive test cases. This process is therefore conducted mostly on resources volunteered by project developers and by companies that use the software in their products.

We installed Linux and Windows Skoll clients and one Skoll server across 25 (11 Linux and 14 Windows) workstations distributed throughout computer science labs at the University of Maryland. All Linux Skoll clients ran on Linux 2.4.9-3 stations and used gcc v2.96 as their compiler; the Windows clients ran on Windows XP stations with Microsoft's Visual C++ v6.0 compiler. On both platforms, we used TAO v1.2.3 with ACE v5.2.3 as the subject software.

B. Configuring the Skoll Infrastructure

We implemented all the components of the Skoll infrastructure described in Section II-B. We then developed different QA task models for each scenario. We configured the ISA and instructed it to navigate the QA task space using random sampling without replacement.

We used several adaptation strategies provided by Skoll. Specifically, we integrated the nearest adaptation strategies neighbor, temporary constraints, and terminate/modify subtasks described in Section II-B into these distributed continuous QA processes. We used temporary constraints and terminate/modify subtasks adaptation in each scenario, but used nearest neighbor only when the QA task space was considered large. In practice, process designers determine the criteria for deciding when a QA task space is large or small.

We developed scripts that prepare task results and feed them into the CTA algorithms for automatic fault characterization. We also wrote scripts that used the classification tree models as input to visualizations.

The QA tasks for these studies must run on both the Windows and Linux operating systems. We therefore implemented client side QA tasks as portable Perl scripts. 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 kit. 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.

C. Study 1: Clean Compilation

Study design. ACE+TAO features can be compiled in or out of the system, *e.g.*, features are often omitted reduce memory footprint in embedded systems. The QA task for this study was to determine whether each ACE+TAO feature combination compiled without error, which is important for open-source software since any valid feature combination should compile. Unexpected build failures not only frustrate users, but also waste time. For example, compiling the 2 MLOC+ took us roughly 4 hours on a 933 MHz Pentium III with 400 Mbytes of RAM, running Linux.

Our first step was to build a QA task model. The feature interaction model for ACE+TAO was undocumented, so we built the QA task model bottom-up. First, we analyzed the source and interviewed several senior ACE+TAO developers. We selected 18 options; one of these options was the OS; the remaining 17 were binary-valued compile-time options that control build time inclusion of various CORBA features.

We also identified 35 inter-option constraints. For example, one constraint is (AMI = 1 → MINIMUM_CORBA = 0), which means that asynchronous method invocation (AMI) is not supported by the minimal CORBA implementation. This QA task space has over 164,000 valid configurations. Since no constraints were related to the OS option, the space was divided equally by OS, *i.e.*, 82,000 valid configurations per OS. Since the QA task space was large, we used the nearest neighbor adaptation strategy to navigate this model. We also configured the ISA to use random sampling replacement without replacement since one observation per valid configuration was sufficient.

After testing ~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. Automatic characterization showed that the problem stemmed from 7 options providing fine-grained control over CORBA messaging policies. This code had been modified and moved to another library, but developers (and users) had forgotten to check whether 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 QA task 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 → B=1) and (B=1 → C=1))—removing them simplified the QA model considerably. As a result, the QA task model contained 11 options (one being OS) and 7 constraints, yielding only 178 valid configurations. Of course, we investigated only a small subset of ACE+TAO's total QA task space; the actual space is much larger.

We then continued the study using the new QA task model and removing the nearest neighbor adaptation strategy (since now we could easily build all valid configurations). Of the 178 valid configurations only 58 compiled without errors. For the 120 (178-58) remaining configurations that did not build, automatic characterization helped clarify the conditions under which they failed.

Analysis of results. After conducting the study we analyzed the data and drew some preliminary conclusions. Beyond identifying failures, in several cases, automatic characterization provided concise, statistically significant descriptions of the subspaces in which 120 configurations failed. Below we describe the cause of 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` (64 configurations - 32 per OS) whenever AMI = 1 and CORBA_MSG = 0. ACE+TAO developers determined that the constraint AMI = 1 → CORBA_MSG = 1 was missing from the model. We therefore refined the model (for later studies) by adding this constraint.

The ACE+TAO build also failed line 38 (line 37 for Windows[2]) in `Asynch_Reply_Dispatcher.h` (16 configurations) whenever CALLBACK = 0 and POLLER = 1. Since this configuration should be legal, we had identified a previously undiscovered bug. Until the bug could be fixed, we temporarily added a new constraint POLLER = 1 → CALLBACK = 1, which we also used in later studies.

The ACE+TAO build failed at line 137 in `RT_ORBInitializer.cpp` (40 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. Again, the error was reported on line 665 in file `RT_Policy_i.cpp` when the system was compiled under Windows; we attribute this difference to the compiler and not an ACE+TAO platform-specific problem.

This study did not find any actual platform-specific compilation problems since the faults characterized as platform-specific were actually due to differences in how compilers generated error messages, and reported error locations. Before moving on to the next study we fixed those errors that we could and worked around those we could not fix by leaving the appropriate temporary constraints in the second study's QA task model.

Lessons learned. We learned several important lessons from Study 1. For example, we found that even ACE+TAO developers did not completely understand the QA task model for their complex software. In fact, they provided us with both erroneous and missing model constraints.

We also discovered that model building is 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

[2] We noted that the compilers (gcc and MSVC++) reported different line numbers for the same error, requiring manual examination and matching of error messages.

useful activities, rather than wasting effort on configurations that would surely fail without providing any new information.

ACE+TAO developers 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, *i.e.*, the new Skoll process tested the patched software only for those configuration that had failed previously.

D. Study 2: Testing with Default Runtime Options

Study design. 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 users confidence that they installed the system correctly. To perform this task, users compile ACE+TAO, compile the tests, and execute the tests.

On our Linux machines it took around 4 hours to compile ACE+TAO, about 3.5 hours to compile all tests, and 30 minutes to execute them, for a total of around 8 hours. On our Windows machines, it took 19 minutes to compile ACE+TAO, about 22 minutes to compile all tests, and 37 minutes to execute them, for a total of around 1.5 hours. These speed differences occurred because the Windows experiments ran on faster machines with more memory.

We first created the QA task 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 QA task 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 was necessary since clients were running on Windows and Linux machines, each with its own low-level policies.

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 → MINIMUM_CORBA = 0, which prevented us from running tests that are bound to fail. By default, we assumed that all test were runnable unless otherwise constrained.

After making these changes, the space had 15 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 did not use the nearest neighbor adaptation strategy since we only tested the 58 configurations that built in Study 1. In Study 2, automatic characterization is done separately for each test and error message combination, but is based only on the settings of the compile time-options.

Analysis of results. Overall, we compiled 4,154 individual tests. Of these 196 did not compile, 3,958 did. Of these, 304 failed, while 3,654 passed. This process took ∼52 hours of computer time. We now describe some interesting failures we uncovered, the automatically-generated failure characterizations, and the action taken by ACE+TAO developers.

Three tests failed in all configurations regardless of the OS. Even though the underlying problem that led to the failures was not configuration-specific, the overall Skoll automation process helped uncover it. The failures were caused by memory corruption due to command-line processing. Whenever the test script used a particular command-line option, namely `ORBSkipServiceConfigOpen`, the tests failed. The usage of the above mentioned option is not mandatory for the scripts, but Skoll used it during the model-building and stepwise refinement of command line options, identifying this previously undiscovered problem.

Three tests failed only when the option (OS = Windows) was enabled. These tests failed because the ACE server failed to start on Windows platforms; this failure is caused by incorrect coding (Linux vs. Windows) of server-invocation scripts in the tests. Improving test diversity (*i.e.*, increasing the number of platforms on which tests run) helped pinpoint this problem.

Two tests failed in 17 configurations when the options (OS=Windows and AMI_POLLER = 0) were enabled. These tests failed because clients did not get correct (or any) response from the server. Although these tests should actually have failed on Linux, it tolerates some amount of invalid memory scribbling without killing the process, thereby allowing the test to pass, even though it should have failed. The failure is revealed only on Windows because it is more rigorous in its memory management. We were able to detect this previously unrevealed problem by increasing the number of platforms and thus enlarging the test diversity.

Two tests failed in 3 configurations when the options (OS = Linux and AMI = 1 and AMI_POLLER and DIOP = 0 and INTERCEPTORS = 1) were enabled. The same 2 tests failed in 6 configurations when the options (OS = Linux and AMI = 1 and AMI_POLLER = 0 and DIOP = 1) were enabled. The same 2 tests failed 29 configurations when the option (OS = Windows) was enabled. According to the ACE+TAO developers, this problem occurs sporadically due to a quirk in the way the `TP_Reactor` (the default event demultiplexer in TAO) handles active handles in an FD_SET.[3] The `TP_reactor` was therefore not picking up the sockets. This error still occurs but not all the time, which suggests that testing each configuration exactly once may be inadequate to detect rarely occurring, nondeterministic faults.

In several cases, multiple 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 options (CORBA_MSG = 1 and POLLER = 0 and CALLBACK = 0) were enabled. This bug was previously undiscovered and stemmed from the fact that certain files in TAO implementing CORBA Messaging incorrectly assumed that at least one of the

[3] See doc.wustl.edu/bugzilla/show_bug.cgi?id=982 for more details.

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, which also had not been discovered previously 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 28 of 58 configurations with an error message indicating response timeout. No statistically significant model could be found, which suggests that (1) the error report might be covering multiple underlying failures, (2) the failure(s) manifests themselves intermittently, or (3) some other factor not related to configuration options is causing the problem. This particular problem appears intermittently and is related to inconsistent timer behavior on certain OS/hardware platform combinations.

Lessons learned. We learned several lessons from Study 2. For example, we found it was relatively easy to extend and refine the initial QA task model to create more complex QA processes. We were again able to conduct a sophisticated QA process across remote user sites on a continuous basis. For example, we exhaustively explored the QA task 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. In fact, the model-building and automation process led to the discovery of the improper handling of command-line options.

We also learned several things about automatic problem characterization. In particular, Skoll's generated models can be unreliable. We use notions of statistical significance to help indicate weak models, but more investigation is necessary. The tree models we use may also 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.

E. Study 3: Testing with Configurable Options

Study design. The QA task for the third study was to determine whether each configuration would run the ACE+TAO regression tests without errors for all settings of the system's runtime options, which is important for building confidence in the system's correctness. This task involves compiling ACE+TAO, compiling the tests, setting the appropriate runtime options, and executing the tests. Doing this for one configuration took from 4 to 8 hours on our machines.

First, we develop the QA task model. To examine ACE+TAO's behavior under differing runtime conditions, we modified the QA task model to reflect 6 multi-valued (non-binary) runtime configuration options. These options set up to 648 different combinations of CORBA runtime policies, *e.g.*, when to flush cached connections, what concurrency strategies the ORB should support, etc. Since these runtime options are independent, we added no new constraints.

After making these changes, the compile-time option space had 15 options and 13 constraints. There were 96 test-specific options with an additional 120 constraints and 6 runtime options with no new constraints.

Analysis of results. The QA task space for this study had 37,584 valid configurations. At roughly 30 minutes per test suite, the entire process involved around 18,800 hours of computer time. Given the large number of configurations, we used the nearest neighbor adaptation strategy. The total number of test executions was 3,608,064. Of these, 689,603 test failed, with 458 unique error messages. We analyze these executions and failures below.

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 former case, the problems were often in feature-specific code, whereas in the latter case the problems were often caused by bugs in option setting and processing code. ACE+TAO developers were intrigued by these findings because they rely heavily on testing of the default configuration by users at installation time, not just to verify proper installation, but to provide feedback on system correctness.

Eight tests failed when option (ORBCollocation = NO) was enabled in 12,441 configurations. These failures stemmed from a bug in TAO where object references were created properly but not activated properly. The ACE+TAO developers have fixed this very serious problem.

One test `RTCORBA_Policy_Combinations_run_test` failed when option (OS = Windows) was enabled in 18,585 configurations. This bug was due to a race condition in the SHMIOP code in TAO and has also now been fixed.

A group of three tests had particularly interesting failure patterns. These tests failed between 2,500 and 4,400 times. In each case automatic characterization showed that the failures occurred when option `ORBCollocation` = NO was enabled. 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 collocated in a single process and thus should communicate directly via method calls instead of sending messages through the OS protocol stack. The `NO` option setting means that objects should not be colocated. The fact that these tests worked when objects communicated directly, but failed when sending messaging through the protocol stack clearly suggested a problem related to message passing. In fact, the source of the problem was a bug in TAO's (de)marshaling of object references.

Three tests failed in 6 configurations when options (OS=Linux and AMI_POLLER = 0 and INTERCEPTORS = 0 and NAMED_RT_MUTEXES = 1) were enabled. The same 3 tests failed in 10 configurations when options (OS=Linux and AMI_POLLER = 0 and INTERCEPTORS = 1) were enabled. This failure was a side-effect of the order in which the test cases ran and had nothing to do with the specific test cases themselves or the options (except OS=Linux), *i.e.*, this problem was specific only to the Linux platform.

These test failures occurred when Linux ran out of the shared memory segments available to the OS. We discovered that TAO leaked these segments on Unix-based platforms. If enough tests were run on a particular Linux machine, the machine ran out of the shared memory segments, causing all subsequent tests to fail. If these particular tests had been run earlier, they would not have failed. In effect, we inadvertently conducted a load test on some machines.

Lessons learned. We learned several things from Study 3. First, we confirmed that our general approach could scale to larger QA task 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 provided better coverage of the QA task space than 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. Much work will therefore be done where a large subspace is failing, (*e.g.*, as described above in roughly 5,000 out of a total 20,000 configurations, `ORBCollocation` = NO and the test failed). In this case we could have stopped the search much earlier and still correctly characterized the failing subspace. In future work, we will explore criteria for stopping the search process.

F. Discussion of the Studies Results

The three studies presented in this section confirmed or reinforced multiple lessons learned about the characteristics of Skoll, distributed continuous QA, and the specific subject applications. First, our conjectures about Skoll were supported, *i.e.*, the overall approach worked well. In particular, Skoll was superior to the *ad hoc* QA processes used by ACE+TAO developers.

ACE+TAO developers were also happy with the results and will use Skoll more aggressively in the future. They report that the Skoll-based process is significantly more effectively than their current QA process. It detected problems quickly, several of which they were not aware of. They also benefited from Skoll's automatic fault characterization, which helped them narrow down the set of possible failure causes quickly, avoiding multiple rounds of fruitlessly guessing the causes of specific failures.

Our use of the QA task model helped us to extend the studies quickly to a completely different platform (*i.e.*, Windows vs. Linux vs. Solaris) with little work and code modification. The fundamental change required for this extension was the addition of a new variable `OS`. We also added some options to capture low-level system information, indicating the use of static or dynamic libraries, whether multithreading support is enabled, etc. This step was necessary since clients were running on Windows and Linux machines and each OS has its own low-level policies.

Constraints associated with the option variables outlined above helped control platform-specific test cases; these test cases were already available for ACE+TAO. Since many of these tests are often intended to run in limited situations, we extended the QA task space by adding test-specific options. Much of the Skoll code is portable (*e.g.*, the control scripts are implemented in Perl), we could reuse it across platforms. We are confident that future extensions can also be added easily.

We learned that full automation of all Skoll processes will require adaptation of several low-level tools. For example, automatic characterization of errors requires that all platform-specific tools (*e.g.*, compliers) report the errors in a similar format. Study 2 showed us that some tools (such as compilers) report the same error differently across platforms. In the future, we will need to wrap the error messages generated by these tools so that they look similar to our automatic characterization algorithms. We envision that some manual work will be needed to write these wrappers each time a new error is encountered; subsequent encounters should be handled automatically.

We discovered that there is significant effort associated with putting each application under Skoll control. These costs were not necessarily caused by Skoll, however, *e.g.*, understanding the ACE+TAO QA task space required significant interaction with ACE+TAO developers, who did not completely understand their own system. We also needed to discover and eliminate several errors in ACE+TAO's command-line option processing before the Skoll scripts could be used. We expect to find and fix more of these errors as new distributed continuous QA processes are developed.

We learned (from Study 3) that the *order* in which sub-tasks are executed may also have an impact on their results. This result uncovered a deeper issue that we need to handle carefully in the future: the context in which a test case executes has an impact on its outcome. We will need to improve the Skoll task execution policies to handle context more effectively and robustly. Each task should execute in a pre-determined clean context; each task should also restore the system environment so that subsequent tasks remain unaffected.

V. Related work

Our research is closely related to other efforts in the area of remote analysis and measurement of software systems. The particular application to which we apply our research is related to software engineering techniques used to create, manage and validate configurable systems.

A. Remote Analysis and Measurement of Software Systems

Several prior attempts have been made to feedback fielded behavioral information to designers. As described below, these approaches gather various types of runtime and environmental information from programs deployed *in the field*, *i.e.*, on user platforms with user configurations.

Distributed regression test suites. Many popular projects distribute regression test suites that end-users run to evaluate installation success. Well-known examples include GNU GCC [13], CPAN [14], and Mozilla [7]. Users can—but frequently do not—return the test results to project staff. Even when results are returned, however, the testing process is often undocumented and unsystematic. Developers therefore have no record of what was tested, how it was tested, or what the results were, resulting in the loss of crucial information.

Auto-build scoreboards. Auto-build scoreboards monitor multiple sites that build/test a software system on various hardware, operating system, and compiler platforms. The Mozilla Tinderbox [15] and ACE+TAO Virtual Scoreboard [16] are examples of auto-build scoreboards that track end-user build results across various volunteered platforms. Bugs are reported via the Bugzilla issue tracking system [17], which provides inter-bug dependency recording and integration with automated software configuration management systems, such as CVS or Subversion. While these systems help to document multiple build processes, deciding what to put under system control and how to do it is left to users. Unless developers can control at least some aspects of the build and process, however, important gaps and inefficiencies will still occur.

Remote data collection systems. Online crash reporting systems, such as the Netscape Quality Feedback Agent [18] and Microsoft XP Error Reporting [19], gather system state at a central location when fielded systems crash, simplifying user participation by automating some parts of problem reporting. Orso et al. [20] developed the GAMMA system to collect partial runtime information from multiple fielded instances of a software system. GAMMA allows users to conduct a variety of different analyses, but is limited to tasks for which capturing low-level profiling information is appropriate. One limitation of these approaches is their limited scope, *i.e.*, they capture only a small fraction of interesting behavioral information. Moreover, they are *reactive* (*i.e.*, the reports are only generated *after* systems crash), rather than *proactive* (*i.e.*, attempting to detect, identify, and remedy problems *before* users encounter them).

Remote data analysis techniques. The emergence of remote data collection systems has spurred research into better remote analysis techniques. Podgursky et al. [21]–[25] present techniques for clustering program executions. Their goal is to support automated fault detection and failure classification. Bowring et al. [26] classify program executions using a technique based on Markov models. Brun and Ernst [27] use two machine learning approaches to identify types of invariants that are likely to be good fault indicators. Liblit et al. [28], [29] remotely capture data on both crashing and non-crashing executions, using statistical learning algorithms to identify data that predicts each outcome. Elbaum and Diep [30] investigate ways to efficiently collect field data and use them for improving the representativeness of test suites. Michail and Xie's [31] Stabilizer system correlates users' partial event histories with failures they report. The models are then linked back into running systems allowing them to predict reoccurences of the failure. Users are also offered a chance to terminate the current operation when imminent failure is predicted.

Earlier remote data analysis techniques share several limitations. Most of them consider only a few specific features of program executions, such as program branches, method entry counts or variable values. They do not support broader types of quality assurance techniques. Moreover, many such techniques require heavyweight data collection, which creates considerable overhead in terms of code bloat, data transmission and analysis costs and, in most cases, execution time.

Distributed continuous quality assurance (QA) environments. Distributed continuous QA environments are designed to support the design, implementation, and execution of remote data analysis techniques such as the ones described in the previous paragraph. For example, Dart [32] and CruiseControl [33] are continuous integration servers that initiate build and test processes whenever repository check-ins occur. Users install clients that automatically check out software from a remote repository, builds it, executes the tests, and submits the results to the Dart server. A major limitation of Dart and CruiseControl, however, is that the underlying QA process is hard-wired, *i.e.*, other QA processes or other implementations of the build and test process are not easily supported and the process cannot be steered. As a result, these QA processes cannot exploit incoming results nor avoid already discovered problems, which leads to wasted resources and lost improvement opportunities.

Although these efforts described above can provide some insight into fielded behavior, they have significant limitations. For example, they are largely *ad hoc* and often have **no scientific basis** for assuring that information is gathered systematically and comprehensively. Moreover, many existing approaches are **reactive** and **have limited scope** (*e.g.*, they can be used only when software crashes or only focus only on a single, narrow task), whereas effective measurement and analysis support needs to be much broader and more proactive (*e.g.*, seeking to collect and analyze important information continuously, before problems occur).

Existing approaches also often **inadequately document** the activities that have been performed, which makes it hard to determine the full extent of (or gaps in) the measurement and analysis process. These approaches also **limit developer control** over the measurement and analysis process (*e.g.*, although developers may be able to decide what aspects of their software to examine, some usage contexts are evaluated multiple times, whereas others are not evaluated at all). Finally, most existing approaches **do not intelligently adapt** by learning from measurement results obtained earlier by other users. These limitations collectively yield inefficient and opaque in-the-field measurement and analysis processes that are insufficient to support today's software designers.

Our work with Skoll is intended to improve this situation. For example, we have used Skoll to support distributed continuous performance assessment [34]. In that effort we developed a new adaptation strategy based on using *Design of Experiments* (DOE) theory to identify a small set of observations (an experimental design selecting configurations to test) that allows Skoll to determine which combinations of options and settings significantly affect performance. This information allowed us to then quickly estimate whether future changes to the system degraded performance.

B. Software Engineering for Configurable Systems

This paper focuses on how we applied our distributed, continuous QA Skoll approach to test the functionality of highly configurable software systems. Our work is therefore related to the following research activities that have created, managed, and validated configurable software systems. Note, however, that Skoll is not limited to only configurable software systems.

Software development approaches that emphasize portability, customizability, large-scale reuse or incremental development (*e.g.*, product line architectures, feature- and middleware-oriented development, or agile development) often rely on identifying and leveraging the commonalities and variabilities of their target application domain [35]. Several researchers have therefore created techniques to model the configuration spaces and interdependencies of such systems. Most processes for developing product-line architectures, for example, incorporate visual models [36]–[38] of the system's variation points. More recent work has focused specifically on system variability and supporting various types of reasoning and analysis over the models [39], [40]. With appropriate translators, most of these models could easily be translated into Skoll's format.

Some work has also been done to develop techniques for testing highly configurable systems. One example is the use of covering arrays to reduce the number of input combinations needed to test a program [41]–[46]. Mandl [46] first used orthogonal arrays, a special type of covering array in which all t-sets occur *exactly* once, to test enumerated types in ADA compiler software. This idea was extended by Brownlie *et al.* [41] who developed the orthogonal array testing system (OATS). They provided empirical results to suggest that the use of orthogonal arrays is effective in fault detection and provides good code coverage. Dalal *et al.* [44] argue that the testing of all pairwise interactions in a software system finds a large percentage of the existing faults. In further work, Burr *et al.* [42], Dunietz *et al.* [45], and Kuhn *et al.* [47] provide more empirical results to show that this type of test coverage is effective.

The studies described above focus on finding unknown faults in already tested systems and equate covering arrays with code coverage metrics [43], [45]. Yilmaz et al. [48], [49] apply covering arrays to test configurable systems. They show that covering arrays were effective not only in detecting failures, but also in characterizing the specific failure inducing options.

VI. Concluding Remarks and Future Work

This paper described the results of our initial efforts designing, executing, and evaluating distributed continuous quality assurance (QA) processes. We first presented Skoll, which is an environment for implementing feedback-driven distributed continuous QA processes that leverage distributed computing resources to improve software quality. We then implemented several such processes using Skoll and evaluated their effectiveness in two feasibility studies that applied Skoll to Mozilla and ACE+TAO, which are several large-scale open-source software systems containing millions of lines of code.

Using Skoll, we iteratively modeled complex QA task spaces, developed novel large-scale distributed continuous QA processes, and executed them on multiple clients. As a result, we found bugs, some of which had not been identified previously. Moreover, the ACE+TAO developers reported that Skoll's automatic failure characterization greatly simplified identifying the root causes of certain failures.

Our work on the Skoll environment is part of an ongoing research project. In addition to providing insight into Skoll's current benefits and limitations, therefore, the results of our feasibility studies are guiding our future work, as summarized below.

Our initial feasibility studies were limited to a small number of machines at the University of Maryland. We are extending and generalizing this work in two dimensions. First, we recently built a large-scale, heterogeneous computing cluster with hundreds of CPUs to support our research, as described in Section I. We have rerun the experiments described in this article on this cluster (the results were the same) and will expand our use of it in the future.

Second, we are replicating our feasibility studies on a dozen test sites and hundreds of machines provided by ACE+TAO developers and user groups in two continents (`www.dre.vanderbilt.edu/scoreboard` lists sites that are contributing machines). As the scope of our work increases we will investigate security and privacy issues more thoroughly. For now, ACE+TAO participants are accustomed to downloading, compiling and testing ACE+TAO, so no special security and privacy precautions were necessary. We are developing security and privacy policies based on existing volunteer computing systems, such as Microsoft's Watson system and the Berkeley Open Infrastructure for Network Computing (which supports projects such as seti@home).

We are applying Skoll to a broader range of application domains, including running prototyping experiments for enterprise distributed systems and large-scale shipboard computing environments, that have many configuration parameters and options, some of which must be evaluated dynamically as well as statically. We are also enriching Skoll's QA task models to support hierarchical models, not just the flat option spaces supported currently. We are incorporating priorities in the model so that different parts of the configuration space can be explored with different frequencies and are incorporating real-valued option settings into the models.

We are enhancing Skoll's Intelligent Steering Agent (ISA) to allow planning based on cost models and probabilistic information. For example, if historical data suggests that users with certain platforms send requests at certain rates, it can take this information into account when allocating job configurations. We are also exploring the use of higher level ISA planners that simultaneously plan for multiple QA processes (not just one at a time as the ISA does now).

We are also integrating Skoll with model-based test-case generation techniques, *e.g.*, our work with GUITAR [50]. We envision that this model will supplement the QA task space. While traversing the QA task space, Skoll's navigation/-adaptation strategies may use the test-case generation techniques to obtain new test cases *on demand*.

Currently individual QA tasks must be executed on a single computing node. This restriction prevents us from answering certain kinds of questions, such as what is the average response time for requests sent from users in one geographical region to servers in another region. We are therefore investigating how peer-to-peer and overlay network technologies can help to broaden the QA tasks Skoll can handle.

VII. Acknowledgments

This material is based on work supported by the US National Science Foundation under NSF grants ITR CCR-0312859, CCF-0447864, CCR-0205265, and CCR-0098158, as well as funding from BBN, Cisco, DARPA, Lockheed Martin Advanced Technology Lab and Advanced Technology Center, ONR, Qualcomm, Raytheon, and Siemens. We thank the anonymous reviewers for their comments that helped improve this paper.

References

[1] R. Sangwan, M. Bass, N. Mullick, D. J. Paulish, and J. Kazmeier, *Global Software Development Handbook*. Auerbach Series on Applied Software Engineering, September 2006.

[2] M. Haran, A. Karr, A. Orso, A. Porter, and A. Sanil, “Applying classification techniques to remotely-collected program execution data,” in *Proceedings of the International Symposium on the Foundations of Software Engineering*, Lisbon, Portugal, Sept. 2005.

[3] M. Haran, A. F. Karr, M. Last, A. Orso, A. Porter, A. Sanil, and S. Fouche, “Classifying software failures to support remote measurement and analysis of software systems.”

[4] Atif Memon and Adam Porter and Cemal Yilmaz and Adithya Nagarajan and Douglas C. Schmidt and Bala Natarajan, “Skoll: Distributed continuous quality assurance,” in *Proceedings of the 26th International Conference on Software Engineering*, May 2004.

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

[6] R. W. Selby and A. A. Porter, “Learning from examples: Generation and evaluation of decision trees for software resource analysis,” *IEEE Trans. Software Engr.*, vol. 14, no. 12, pp. 1743–1757, December 1988.

[7] The Mozilla Organization, “Mozilla,” www.mozilla.org/, 1998.

[8] A. Zeller and R. Hildebrandt, “Simplifying and isolating failure-inducing input,” *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 183–200, 2002.

[9] A. Zeller, “Yesterday, my program worked. today, it does not. why?” in *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*. Springer-Verlag, 1999, pp. 253–267.

[10] X11-GUITest-0.20, search.cpan.org/~ctrondlp/X11-GUITest-0.20.

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

[12] D. C. Schmidt, D. L. Levine, and S. Mungee, “The Design and Performance of Real-Time Object Request Brokers,” *Computer Communications*, vol. 21, no. 4, pp. 294–324, Apr. 1998.

[13] GNU. Gnu gcc. [Online]. Available: http://gcc.gnu.org

[14] CPAN. Comprehensive perl archive network (cpan). [Online]. Available: http://www.cpan.org

[15] Mozilla. Tinderbox. [Online]. Available: http://www.mozzila.org

[16] “Doc group virtual scoreboard,” www.dre.vanderbilt.edu/scoreboard/.

[17] The Mozilla Organization. (1998) bugs. [Online]. Available: http://www.mozilla.org/bugs/

[18] Netscape. Netscape quality feedback system. [Online]. Available: http://www.netscape.com

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

[20] A. Orso, D. Liang, M. J. Harrold, and R. Lipton, “Gamma system: Continuous evolution of software after deployment,” in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002)*, Rome, Italy, july 2002, pp. 65–69.

[21] W. Dickinson, D. Leon, and A. Podgurski, “Pursuing failure: the distribution of program failures in a profile space,” in *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of Software Engineering*, September 2001, pp. 246–255.

[22] W. Dickinson, D. Leon, and A. Podgursky, “Finding failures by cluster analysis of execution profiles,” in *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001)*, May 2001, pp. 339–348.

[23] P. Francis, D. Leon, M. Minch, and A. Podgurski, “Tree-based methods for classifying software failures,” in *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE’04)*, November 2004, pp. 451–462.

[24] D. Leon, A. Podgurski, and L. J. White, “Multivariate visualization in observation-based testing,” in *Proceedings of the 22nd international conference on Software engineering (ICSE 2000)*, May 2000, pp. 116–125.

[25] A. Podgurski, D. Leon, P. Francis, W. Masri, M. M. Sun, and B. Wang, “Automated support for classifying software failure reports,” in *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, May 2003, pp. 465–474.

[26] J. F. Bowring, J. M. Rehg, and M. J. Harrold, “Active learning for automatic classification of software behavior,” in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2004)*, July 2004, pp. 195–205.

[27] Y. Brun and M. D. Ernst, “Finding latent code errors via machine learning over program executions,” in *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, May 2004, pp. 480–490.

[28] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, “Bug isolation via remote program sampling,” in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI 2003)*, June 2003, pp. 141–154.

[29] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, “Scalable statistical bug isolation,” in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI 2005)*, June 2005.

[30] S. Elbaum and M. Diep, “Profiling Deployed Software: Assessing Strategies and Testing Opportunities,” *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 312–327, 2005.

[31] A. Michail and T. Xie, “Helping users avoid bugs in gui applications,” in *ICSE ’05: Proceedings of the 27th international conference on Software engineering*. New York, NY, USA: ACM Press, 2005, pp. 107–116.

[32] Kitware. public.kitware.com. [Online]. Available: http://public.kitware.com

[33] CruiseControl, “Continuous Integration Toolkit,” http://cruisecontrol.sourceforge.net.

[34] C. Yilmaz, A. Krishna, A. Memon, A. Porter, D. Schmidt, A. Gokhale, and B. Natarajan, “Main effects screening: A distributed continuous quality assurance process for monitoring performance degradation in evolving software systems,” in *ICSE ’05: Proceedings of the 27th International Conference on Software Engineering*. IEEE Computer Society, 2005.

[35] J. van Gurp, J. Bosch, and M. Svahnberg, “On the notion of variability in software product lines,” in *Proceedings 2nd Working IEEE / IFIP Conference on Software Architecture (WICSA)*, 2001, pp. 45–54.

[36] D. M. Weiss and C. T. R. Lai, “Software product-line engineering: A family-based software development process,” 1999.

[37] H. Gomaa, “Modeling software product lines with uml,” in *Proceedings of the Second International Workshop on Software Product Lines: Economics, Architectures, and Implications*, May 2001, pp. 27–31.

[38] M. Griss, J. Favaro, and M. d’Alessandro, “Integrating feature modeling with the rseb,” in *Proceedings of International Conference on Software Reuse*, June 1998, pp. 36–45.

[39] S. Buhne, G. Halmans, and K. Pohl, “Modeling dependencies between variation points in use case diagrams,” in *Proceedings of 9th Intl. Workshop on Requirements Engineering - Foundations for Software Quality*, June 2003, pp. 59–70.

[40] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, “Feature-oriented domain analysis (foda) feasibility study,” Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-90-TR-21, November 1990.

[41] R. Brownlie, J. Prowse, and M. S. Phadke, “Robust testing of AT&T PMX/StarMAIL using OATS,” AT&T Technical Journal, vol. 71, no. 3, pp. 41–7, 1992.

[42] K. Burr and W. Young, “Combinatorial test techniques: Table-based automation, test generation and code coverage,” in *Proc. of the Intl. Conf. on Software Testing Analysis & Review*, 1998.

[43] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, “The AETG system: an approach to testing based on combinatorial design,” *IEEE Transactions on Software Engineering*, vol. 23, no. 7, pp. 437–44, 1997.

[44] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, "Model-based testing in practice," in *Proc. of the Intl. Conf. on Software Engineering, (ICSE)*, 1999, pp. 285–294.

[45] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. M. ws, and A. Iannino, "Applying design of experiments to software testing," in *Proceedings of the International Conference on Software Engineering, (ICSE)*, 1997, pp. 205–215.

[46] R. Mandl, "Orthogonal Latin squares: an application of experiment design to compiler testing," *Communications of the ACM*, vol. 28, no. 10, pp. 1054–1058, 1985.

[47] D. Kuhn and M. Reilly, "An investigation of the applicability of design of experiments to software testing," in *Proceedings of the 27th Annual NASA Goddard/IEEE Software Engi neering Workshop*, 2002, pp. 91–95.

[48] C. Yilmaz, M. B. Cohen, and A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces." in *ISSTA*, 2004, pp. 45–54.

[49] M. C. C. Yilmaz and A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," *IEEE Transactions on Software Engineering*, vol. 31, no. 1, pp. 20–34, 2006.

[50] A. M. Memon and Q. Xie, "Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 884–896, Oct. 2005.