Towards a Distributed Continuous Certification Process

Adam Porter
Computer Science Department
University of maryland
College Park, MD 20742
aporter@cs.umd.edu

ABSTRACT

Software scale and complexity are growing by every measure: more hardware and software, more communication links, more interdependency, more lines of code, more storage and data, etc. [18]. At the same time business trends are increasingly squeezing development resources. In particular, development processes are straining under severe cost and time-to-market pressures. Global competition and market deregulation are shrinking profit margins and thus limiting budgets for the development and QA of software.

In response to these trends, developers have begun to change the way they build and validate software systems by (among other things) moving towards more flexible product designs allowing dynamic reconfiguration.

This approach promises to improve cost, quality, and development-time, but creates other problems, especially when used in the context of safety-critical systems.

To realize this promise, however, effective certification becomes more important than ever since as static controls are removed or reduced, it becomes even more vital that (1) problems be caught as quickly as possible and (2) systems not be allowed to drift so far from their intended functional and performance requirements that rework costs overwhelm the hoped-for efficiencies.

This article will present and discuss some of our recent efforts to address these problems.

Keywords

Testing, certification, multiconfiguration systems

1. EMERGING TRENDS AND RESEARCH CHALLENGES

Software scale and complexity are growing by every measure: more hardware and software, more communication links, more interdependency, more lines of code, more storage and data, etc. [18]. At the same time business trends are increasingly squeezing development resources. In particular, development processes are straining under severe cost and time-to-market pressures. Global competition and market deregulation are shrinking profit margins and thus limiting budgets for the development and QA of software.

In response to these trends, developers have begun to change the way they build and validate software systems by moving towards more "agile" processes [26, 10] characterized by (1) decentralized development teams, (2) greater reliance on component assembly and deployment than green field code writing, (3) evolution-oriented development featuring incremental development and frequent software updates, and (4) flexible product designs supporting extensive compile time customization and runtime adaptation. These processes try to improve cost, quality, and cycle-time by limiting explicit coordination, by parallelizing or eliminating certain development activities, and by allowing systems to defer many decisions until field deployment time.

To realize these efficiencies, however, quality assurance (QA) becomes more important than ever since as process controls are removed or reduced, it becomes even more vital that a system's functional and performance characteristics be carefully exposed and explored throughout the lifecycle. In practice, these QA processes are increasingly guided by a combination of the following:

Test-driven development (TDD). In TDD-oriented projects tests become first class artifacts, treated at the same level of importance as implementations themselves. Tests are developed before (or in parallel with) implementations and are integrated into automated test harnesses (such as JUnit, CPPunit, or NUnit) where they can be invoked automatically as part of the standard build process. This automated QA process ensures that tests can be run each time the application is built, catching errors more quickly and supporting continuous evolution by minimizing unintended consequences to incremental changes [23].

Continuous build, integration, and test (CBIT). CBIT [17] enhances test-driven development by automating the system integration process each time changes are checked into the source repository by performing a complete system build, running all unit tests, measuring code coverage, enforcing coding conventions, evaluating code for consistency and running system tests. This technology helps to ensure that code written by independent groups works together and helps define stable system versions on which new functionality can be safely added. Some popular CBIT systems include CruiseControl, Apache Gump, Mozilla Tinderbox, and Dart.

While various aspects and combinations of TDD and CBIT have been used sucessfully in industry [15], our experience applying these QA techniques to highly configurable large-scale systems has identified several seri-

1-4244-0910-1/07/$20.00 ©2007 IEEE

ous limitations. In particular, we find that in practice TDD and CBIT today use unsophisticated algorithms and are thus highly inefficient, are limited to compilation and simple functional testing, and have huge gaps in configuration coverage. In particular, conventional TDD and CBIT approaches tend to:

- Concentrate QA efforts only on the most readily-available platforms and default configurations, rather than enforce and enable test diversity.
- Incur substantial redundancy and wasted effort when scaled across multiple CBIT servers because knowledge, artifacts, effort, and results are not coordinated to maximize efficiency and effectiveness.
- Support static QA processes that focus exclusively on the current system version (*e.g.*, the head branch of the software revision control system), rather than learning over time or conducting proactive analyses to improve future system versions.

All told, today's QA approaches weren't designed to scale to the combinatorially exploding *software configuration and control spaces* found in modern dynamically adaptive systems.

1.1 Assessment of Current Technologies and Related Work

There have been a number of attempts to address the challenges and limitations of the existing QA processes described in Section 1. These efforts gather various types of information from systems in a wide variety of configurations. Below, we describe some of these efforts and discuss their pros and cons.

1. Remote data collection systems. Online crash reporting systems, such as the Netscape Quality Feedback Agent [5] and Microsoft XP Error Reporting [4], gather system state at a central location whenever a fielded system crashes. A key assumption of these techniques is that fielded systems inherently run in numerous configurations. Recent work by Liblit et al. [22] extends these approaches to capture data on both crashing and non-crashing executions, using statistical learning algorithms to identify data that predicts each outcome. Orso et al. [25] have also 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.

A limitation shared by these remote data collection systems, however, is their limited scope, *i.e.*, they perform only a small fraction of typical QA activities, ignoring for example issues associated with quality of service (QoS) performance. Moreover, they are largely *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).

2. Distributed regression test suites. Many popular open-source projects distribute regression test suites that end-users run to evaluate installation success. Well-known examples include GNU GCC [3], CPAN [1], Mozilla [30], the Visualization Toolkit (VTK) [19], and ACE+TAO [16]. Users can—but frequently do not—return the test results to project developers. Even when results are returned to developers, moreover, the testing process is often undocumented and unsystematic. For example, developers have no record of what was tested, how it was tested, or what the results were, resulting in the loss of crucial QA-related information.

3. Auto-build scoreboards and build farms. Auto-build scoreboards and build farms are a more proactive form of distributed regression test suites that allow developers to build/test their software at multiple sites on various hardware, operating system, and compiler platforms. The Mozilla Tinderbox [7] and ACE+TAO Virtual Scoreboard [2] are auto-build scoreboards that track end-user build results across various platforms. Bugs are reported via the Bugzilla issue tracking system [29], which provides inter-bug dependency recording, advanced reporting capabilities, extensive configurability, and integration with automated software configuration management systems, such as CVS [28]. While these auto-build systems help document the QA process, the decision of what to put under QA and how to do it is left to users. Unless developers can control at least key aspects of the QA process, important gaps and inefficiencies will still occur.

4. Distributed continuous quality assurance (DCQA) environments, are designed to coordinate QA efforts around-the-clock using resources in multiple, geographically distributed locations. For example, the VTK project uses a DCQA environment called Dart [6], which supports a continuous build and test process that is initiated whenever repository check-ins occur. Developers install a Dart client on their platform and use this client to automatically check out software from a remote repository, build it, execute the tests, and submit the results to the Dart server. Another similar system is BuildBot [8]. A key limitation of these DCQA systems, 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 does not change once it starts. These systems therefore cannot exploit incoming results nor avoid already discovered problems, which wastes resources and misses important improvement opportunities.

Although prior efforts on QA described above can help improve the quality and performance of software, they have significant limitations. First, existing approaches are largely *ad hoc* and have **no scientific basis** for assuring that anomaly detection, QoS evaluation, and integration testing is performed systematically and comprehensively. Second, many existing approaches are **reactive** and **have limited scope** (*e.g.*, they can be used only when software crashes or only focus on functional regression tests), whereas effective QA support needs to be much broader and more proactive (*e.g.*, seeking to resolve problems before users encounter them and trying to identify/optimize performance bottlenecks). Third, existing approaches **inadequately document** the QA activities that have been performed, which makes it hard to determine the full extent of (or gaps in) the QA process. Fourth, existing approaches **limit developer control** over the QA

process, *e.g.*, although developers may be able to decide what aspects of their software to examine, some configurations are evaluated multiple times, whereas others are not evaluated at all. Finally, existing approaches **do not intelligently adapt** by learning from QA results obtained earlier by other users. These limitations collectively yield inefficient and opaque in-the-field QA processes that are insufficient to address the emerging trends and requirements of software systems described in Section 1.

1.2 Summary of Prior Work

In prior work [13, 24, 34], we developed a prototype DCQA environment called Skoll [27] that improves upon earlier system approaches described in Section 1.1. In particular, Skoll provides an *Intelligent Steering Agent* (ISA) that guides the QA process across large configuration spaces by decomposing QA analyses (such as anomaly detection, QoS evaluation, and integration testing QA processes) into multiple tasks and then distributing/executing these tasks continuously across a grid of computing resources contributed by end-users and distributed developers around the world. The results of these executions are returned to a central collection site [2] that merges and analyzes the results to complete the original QA analysis (*e.g.,* identifying defects, performance bottlenecks, and other quality problems) and (2) guide subsequent iterations of the QA process.

The Skoll process and infrastructure. Skoll is a process and a tool-supported infrastructure for defining and executing DCQA processes. Skoll's analytical cornerstone is a model of the QA task space that captures all QA task configurations in which QA tasks might run. A task configuration is a set of name and value tuples, which serve as parameters to generic QA task definitions. In actual systems, not all parameter combinations are legal, so Skoll also supports inter-parameter constraints that limit the setting of one parameter based on the settings of others.

Given a configuration space and generic QA tasks that operate on points in this space, DCQA processes are defined by (1) creating programs that systematically "visit" points in the configuration space, where visiting involves a remote client executing a QA task in the configuration defined by that point and returning the results to a Skoll server, (2) defining analysis techniques that merge and analyze incremental QA task results, and (3) creating decision rules to dynamically and intelligently steer the visitor programs based on the incoming results.

At execution time, Skoll's DCQA processes run on volunteered clients using software that calls into a Skoll server when they are available to perform QA tasks. When contacted, Skoll uses intelligent planning technology to assign the current best QA task to that client, where "best" is defined by the navigation and adaptation strategies discussed above, the state of the DCQA process, and characteristics of the client machine offering service. After a QA task is selected, Skoll creates the code artifacts, assembly parameters, build instructions, and QA-specific code associated with it. This data is then sent to the client, which executes it and returns the results for collection and analysis. As results return, decision rules are triggered to effect any desired process steering.

We have developed several novel **DCQA Processes** using the Skoll infrastructure, some of which we describe below.

DCQA fault characterization creates models describing the configuration options and settings that best predict failure. These models help developers quickly narrow down the causes of specific failures by field testing many different system configurations and feeding the results to a predictive model building process. Since there are many QA subtasks—each taking hours to complete—we developed search-based strategies to improve early fault characterization. For example, one strategy zooms in on parts of the subtask configuration space in which tests fail, allowing quick characterization of specific problems. Conversely, after a problematic configuration subspace has been characterized, we steer further efforts away from this subspace and towards other parts of the subtasks configuration space for which QA information is still needed.

We applied this DCQA fault characterization process to one version of ACE+TAO for which QA information was already available, using several hundred clients across 120 CPUs in our evaluation testbed. The largest subtask configuration models had ∼115K possible configurations. Our results confirmed Skoll's effectiveness and strongly suggest that our DCQA process worked better than the conventional QA approach used by ACE+TAO developers. For example, although ACE+TAO developers could not execute large-scale QA processes due to the combinatoric number of configurations, we used Skoll to execute all tests on all compilable configurations in 8 days. We quickly identified problems that they had not found or had taken much longer to find. Finally, the automatic fault characterization enabled them to find the root cause of quality problems quickly. See [24] for more details.

One limitation with the Skoll QA process applied to ACE+TAO is that it must ultimately test the entire configuration space, which does not scale gracefully. We therefore decided instead to systematically sample the configuration space, testing only the selected configurations and conducting fault characterization on the resulting data. Our sampling approach is based on two types of computing mathematical objects called *covering* arrays. The first type (basic covering arrays) induces a subtask configuration sample in which all t-way interactions between options are observed at least once. The second type (variable-strength covering arrays) is similar, but allows us to vary t across different subsets of the subtask configuration space. Our evaluation showed that these sampling approaches were nearly as accurate as those based on exhaustive data, but were much less expensive, providing from 50% to 99% reduction in the number of configurations tested. See [12, 31] for more details.

DCQA performance modeling. In addition to functional testing we also have developed initial DCQA processes targeting performance measurement. DCQA performance modeling helps developers determine which small subset of system configurations must be bench-

marked to accurately estimate performance across all configurations. Since benchmarking all system configurations is infeasible in large-scale systems, developers currently limit their analyses to a few configurations, after which they (unreliably) extrapolate to the entire configuration space, which allows performance degradations to escape to the field.

To address the limitations with existing performance estimation, we developed a DCQA process called "reliable effects screening" that we implemented using Skoll. This process executes statistically-designed experiments across a QA grid to identify a small subset of the most important performance-related configuration options (currently limited to binary options only). Whenever software changes occur thereafter, developers can quickly estimate system performance across the entire configuration space by exhaustively exploring all combinations of (the small number of) important options, while randomizing the rest.

We evaluated the reliable effects screening process on a 14 option subset of ACE+TAO and associated software. The results indicated that (1) this process cheaply and correctly identifies the subset of options that are most important to system performance, (2) monitoring only these selected options can detect performance degradation quickly with an acceptable level of effort, and (3) alternative strategies with equivalent effort yield less reliable results. See [32, 33] for more details. An interesting aspect of this approach is that by computing the key effects before changes occur, we cut down total benchmarking time from 2 days to 5 minutes, which is fast enough to make this part of the source code check-in process. By working proactively, therefore, we can give developers that illusion of very fast response without significantly compromising the quality of the analysis.

While the results described above have been promising, they are only a first step to realizing the promise of DCQA technologies. We must therefore overcome the following weaknesses and areas for improvement that motivate the research efforts in this proposal:

- **Scalability**. Our work to date on Skoll has been designed for, and evaluated on, problems of relatively modest scale on a single software subject. For example, our configuration models of ACE+TAO assumed a fixed set of options, option settings were limited to nominal categories (*i.e.*, a small number of discrete settings), and experimentation was limited to 20 or fewer configuration options. Our broader goal is to run DCQA processes across large virtual computing grids provided by software development companies in the form of commodity computing clusters or volunteered resources connected via intranets or by world-wide user communities via the Internet. To make significant progress on this effort, therefore, we need more powerful algorithms, theories, and tools; a larger and more diverse application set; a more heterogeneous evaluation grid; and must engage appropriate development organizations and user communities to broaden our application scenarios.

- **Automation and ease of use**. Many key artifacts (such as configuration models, build scripts, test scaffolding, and algorithm definition) used in Skoll must be implemented manually today and redeveloped for every new application or system change. Moreover, developers often do not completely understand the configuration model for their large-scale, complex systems, so their models often have erroneous and missing constraints. Explicit and automated support for model building and validation is therefore required, particularly as we explore ultra-large-scale configuration and control spaces [18].

- **Range of applicability**. So far, we have applied Skoll to only a very restricted set of problems and we have assumed that QA tasks must be very lightweight to run on end-user machines. These previous restrictions prevented us from tackling interesting use cases, such as QA processes for component-based systems with multiple versions of each component, cost and time optimized QA processes run on reconfigurable hardware and software platforms, and QA tasks involving multiple client machines. We therefore need to focus on QA processes used by developers and run on developer-provided machines (which may of course be distributed around the world in large, decentralized organizations).

To address these limitations we are working with researchers from the Institute for Software Integrated Systems (ISIS) at Vanderbilt University (VU) to develop, validate, and deploy next generation of QA environments for complex and adaptive software systems.

1.3 Project Research Goals and Technical Focus Areas

To address the challenges discussed above—and overcome the limitations with existing QA processes described in Sections 1.1 and 1.2—we are conducting a research effort called *QUality Assurance as a Service Infrastructure* (QUASI) aimed at developing and validating the next generation of QA technologies that support around-the-world, around-the-clock QA on a virtual computing grid provided largely by (possibly distributed) development teams. Our research goals in the proposed QUASI project focus on:

- **Developing new technologies and support infrastructure**, including languages for modeling key characteristics of software configurations and QA process control, control algorithms for scheduling and remotely executing QA tasks, and analysis techniques that accurately and scalably characterize software faults and performance problems.

- **Applying and validating individual technologies, algorithms, and infrastructure components** in a controlled environment using the *QUASI evaluation grid*, which is our dedicated cluster of ∼235 diverse and geographically-distributed top-end CPUs recently funded by ONR.

- **Demonstrating and evaluating our approach at scale** by using our results to enhance the QA

process of multiple large-scale projects and companies. In particular, our project partners include major companies and popular open-source projects.

Individually, these efforts provide the technical foundation for producing the advanced software methods and tools required for next-generation software systems. Combined together, they provide a powerful framework for cost-effective, time-bounded assurance of these systems via model-based distributed continuous QA technologies that define, deploy, and manage proactive and adaptive QA processes (*e.g.*, monitoring, testing, and performance assessment) across networked testbeds to greatly improve (or tradeoff) the cost, quality, and time needed to build confidence that these systems are meet their functional and performance requirements.

To support our QA research goals we are creating, validating, and disseminating novel technologies in the focus areas described below:

1. Design and evaluation of scalable DCQA applications. To date only a handful of research efforts [25, 11, 22, 9, 14, 21] have studied DCQA processes. It is not yet clear, therefore, how best to structure these processes, what types of QA tasks can be distributed effectively, or how the costs/benefits of DCQA processes compare to conventional QA processes. To address these issues, we propose to create, prototype, and evaluate several types of DCQA processes for the large-scale software systems developed by our partners at major companies and popular open-source projects. Since these DCQA processes must scale to large configuration spaces, applying brute-force QA processes to detect, identify, and remedy faults and QoS performance bottlenecks is infeasible, even with a large grid of computing resources. To handle this scale and complexity, we will develop techniques to explore and control the QA configuration space **intelligently**. For example, we are exploring the use of customizable strategies to create goal-driven DCQA process adaptation based on a variety of factors, such as task importance, cost, and resource availability.

2. Model-driven QA configuration specification, analysis, and synthesis. Today's software systems are characterized by myriad infrastructure and application variabilities that are typically configured and optimized manually [20]. Unfortunately, this *ad hoc* customization has no scientific basis for assuring that the resulting configuration delivers the required functionality or QoS performance. To handle this variability rigorously, we are **formally modeling** aspects of the QA subtasks and underlying software that will be varied under the control of the QUASI DCQA process. Modeling includes not only software configuration and process control parameters, but also their constraints, module interconnections, and interdependencies; QoS characteristics; and estimated workloads. High-level, coarse-grained models are used initially, which are then gradually refined into finer-grained models and/or implementations as the QA process proceeds. Our approach is predicated on the assumption that the configuration models can be analyzed, and the analysis results will help to optimize QA processes applied to the software and enable automated generation and execution of many QA steps.

3. Implementation and at-scale validation of the QUASI next-generation CBIT server infrastructure. Our DCQA processes work by decomposing QA processes into multiple tasks, distributing and executing them on a large-scale computing grid, continuously merging and analyzing the results, and adapting the process based on these results until the desired analysis is finished. To ensure that our research has a firm experimental basis, we are developing and validating a prototype of the *QUASI infrastructure*, which consists of reusable tools and services needed to implement, execute, and evaluate our research plan outlined above.

Acknowledgment

This material is based on work supported by the US National Science Foundation under NSF grants ITR CCR0312859, CCF0447864, CCR0205265, and CCR-0098158.

2. REFERENCES

[1] Comprehensive Perl Archive Network (CPAN). www.cpan.org.

[2] Doc group virtual scoreboard. www.dre.vanderbilt.edu/scoreboard/.

[3] GNU GCC. gcc.gnu.org.

[4] Microsoft XP Error Reporting. support.microsoft.com/?kbid=310414.

[5] Netscape Quality Feedback Agent. www.mozilla.org/quality/qfa.html.

[6] public.kitware.com. public.kitware.com.

[7] Tinderbox. www.mozilla.org/tinderbox.html.

[8] Buildbot. http://buildbot.sourceforge.net, 2006.

[9] 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.

[10] B. Boehm and R. Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley Professional, August 2003.

[11] J. Bowring, A. Orso, and M. J. Harrold. Monitoring deployed software using software tomography. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools And Engineering*, pages 2–9. ACM Press, 2002.

[12] M. C. C. Yilmaz and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, 31(1):20–34, 2006.

[13] Cemal Yilmaz and Adam Porter and Douglas C. Schmidt. Distributed Continuous Quality Assurance: The Skoll Project. In *Workshop on Remote Analysis and Measurement of Software Systems (RAMSS)*, Portland, Oregon, May 2003. IEEE/ACM.

[14] Cemal Yilmaz and Adam Porter and Douglas C. Schmidt. Distributed Continuous Quality Assurance: The Skoll Project. In *Workshop on Remote Analysis and Measurement of Software Systems (RAMSS)*, Portland, Oregon, May 2003. IEEE/ACM.

[15] M. A. Cusumano and R. W. Selby. How microsoft builds software. *Commun. ACM*, 40(6):53–61, 1997.

[16] DOC Group. ACE and TAO. deuce.doc.wustl.edu/Download.html/, 2004.

[17] M. Fowler. Continuous integration. www.martinfowler.com/articles/continuousIntegration.html, 2006.

[18] S. E. Institute. Ultra-Large-Scale Systems: Software Challenge of the Future. Technical report, Pittsburgh, PA, USA, Jun 2006.

[19] Kitware. The Visualization Toolkit. public.kitware.com/VTK/, 2004.

[20] A. Krishna, A. Gokhale, D. C. Schmidt, J. Hatcliff, and V. Ranganath. Context-Specific Middleware Specialization Techniques for Optimizing Software Product-line Architectures. In *Proceedings of EuroSys 2006*, Leuven, Belgium, Apr. 2006. ACM.

[21] A. S. Krishna, N. Wang, B. Natarajan, A. Gokhale, D. C. Schmidt, and G. Thaker. CCMPerf: A Benchmarking Tool for CORBA Component Model Implementations. In *Proceedings of the 10th Real-time Technology and Application Symposium (RTAS '04)*, Toronto, CA, May 2004. IEEE.

[22] B. Liblit, A. Aiken, and A. X. Zheng. Distributed program sampling. In *Proceedings of ACM Programming Languages Design and Implementation (PLDI) '03*, San Diego, California, June 2003.

[23] E. M. Maximilien and L. Williams. Assessing test-driven development at ibm. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 564–569, Washington, DC, USA, 2003. IEEE Computer Society.

[24] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. C. Schmidt, and B. Natarajan. Skoll: Distributed Continuous Quality Assurance. In *Proceedings of the 26th IEEE/ACM International Conference on Software Engineering*, Edinburgh, Scotland, May 2004. IEEE/ACM.

[25] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. Gamma system: Continuous evolution of software after deployment. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 65–69. ACM Press, 2002.

[26] Pekka Abrahamsson and Juhani Warsta and Mikko T. Siponen and Jussi Ronkainen. New Directions on Agile Methods: A Comparative Analysis. In *International Conference on Software Engineering (ICSE)*, Portland, Oregon, May 2003. IEEE/ACM.

[27] A. Porter, D. C. Schmidt, A. Memon, C. Yilmaz, B. Natarajan, and D. Hinton. Skoll – a distributed continuous quality assurance environment. www.cs.umd.edu/projects/skoll.

[28] SourceGear Corporation. CVS. `www.sourcegear.com/CVS`, 1999.

[29] The Mozilla Organization. bugs. `www.mozilla.org/bugs/`, 1998.

[30] The Mozilla Organization. Mozilla. www.mozilla.org/, 1998.

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

[32] 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.

[33] C. Yilmaz, A. Krishna, A. Memon, A. Porter, D. Schmidt, A. Gokhale, and B. Natarajan. Reliable effects screening: A distributed continuous quality assurance process for monitoring performance degradation in evolving software systems. *IEEE Transactions on Software Engineering*, 2006. (In review.).

[34] C. Yilmaz, A. Krishna, A. Memon, A. Porter, D. C. Schmidt, A. Gokhale, and B. Natarajan. Main Effects Screening: A Distributed Continuous Quality Assurance Process for Monitoring Performance Degradation in Evolving Software Systems. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, St. Louis, MO, May 2005.