Software Testing Techniques and Experimental Research drawn from Inferences

Sagar Sharma, Rahul Magow, Siddharth Kathpal, Aman Jatain
ITM University Gurgaon 122017 India

ABSTRACT
Software products released into the field typically have some number of residual defects that either were not detected or could not have been detected during testing. This may be the result of flaws in the test cases themselves, incorrect assumptions made during the creation of test cases, or the infeasibility of testing the sheer number of possible configurations for a complex system; these defects may also be due to application states that were not considered during lab testing, or corrupted states that could arise due to a security violation. One approach to this problem is to continue to test these applications even after deployment, in hopes of finding any remaining flaws. In this paper, we present a testing methodology we call in vivo testing, in which tests are continuously executed in the deployment environment applications called Invite. We also provide the results of case studies that demonstrate Invite’s effectiveness and efficiency.

1. INTRODUCTION
Thorough testing of a software product is unquestionably a crucial part of the development process, but the ability to faithfully detect all defects in an application is severely hampered by numerous factors. A recent report [30] indicates that 40% of IT companies consider insufficient prerelease testing to be a major cause of later production problems, and the problem only worsens as changes are rolled out into production without being thoroughly tested. Furthermore, it is possible that the test code itself may have flaws in it, too, perhaps because of oversights or incorrect assumptions made by the authors. A key issue is that, for large, complex software systems, it is typically impossible in terms of time and cost to reliably test all configuration options before releasing the product into the field. For instance, Microsoft Internet Explorer has over 19 trillion possible combinations of configuration settings [9]. Even given infinite time and resources to test an application and all its configurations, once a product is released, the other software packages on which it depends (libraries, virtual machines, etc.) may also be updated; therefore, it would be impossible to test with these dependencies prior to the application’s release, because they did not exist yet. A last emerging issue is the fact that, as multiprocessor and multi-core systems become more and more prevalent, multi-threaded applications that had only been tested on single-processor/core machines are more likely to start to reveal concurrency bugs [21].

One proposed way of addressing this problem has been to continue testing the application in the field, after it has been deployed. The theory of this “perpetual testing” [26] approach is that, over time, defects will reveal themselves given that multiple instances of the same application may be run globally with different configurations, in different environments, under different patterns of usage, and in different system states.

2. QUALITY ASSURANCE USING VIVO TECHNIQUE
In this paper, we present a testing methodology we call in vivo testing, in which tests are continuously executed in the deployment environment. We also introduce a new type of test called in vivo tests, which are designed to run from within the executing application and be used with this approach. These tests improve on traditional unit or integration tests by foregoing the assumption of a clean state created by a test harness, and focusing on aspects of the program that should hold true regardless of what state the system is in. These tests execute within the current state of the program without affecting or altering that state, as potentially visible to users. The approach can be used for detecting concurrency, security, or robustness issues, as well as defects that may not have appeared in a testing lab (the “in vitro” environment).

Our three main contributions are an approach (in vivo testing) to executing tests within the deployment environment, without altering that...
system’s state; a new style of tests (in vivo tests) that exercise parts of the application as the system is running, no matter what its current state; and a prototype implementation of the testing framework, called Invite, developed in Java. In [6], we briefly sketched an earlier version of the Invite framework, focused on distributed execution of the tests; in this work, we present for the first time the complete system in full, including a more detailed description of in vivo tests, case studies in which the approach reveals defects in real-world applications, and evidence that the performance overhead of the approach is reasonable and yet a single application instance can still execute millions of tests per day.

The foundation of the in vivo testing approach is the fact that many (if not all) software products are released into deployment environments with latent defects still residing in them, as well as our claim that these defects may reveal themselves when the application executes in states that were unanticipated and/or untested in the development environment. The in vivo testing approach can be used to detect defects hidden by assumptions of a clean state in the tests, errors that occur in field configurations not tested before deployment, and problems caused by unexpected user actions that put the system in an unanticipated state; these flaws may also be due to corrupted states that could arise due to a security violation. Our approach goes beyond passive application monitoring [24] in that it actively tests the application as it runs in the field.

3. AUTOMATED SOFTWARE TESTING

Software quality assurance is in dire need of substantial progress. Software testing is resource-hungry, time-consuming, labor-intensive, and prone to human omission and error. Despite massive investments in quality assurance, serious code defects are routinely discovered after software has been released [16], and fixing them at so late a stage carries substantial cost [13]. Thorough testing of large, complex software involves great effort, and the software industry still employs relatively primitive testing techniques.

We need a “disruptive technology” to substantially improve software quality. Various studies have found the average bug density in production-ready software to have stayed relatively constant over time, while average code volume of software has increased along an exponential curve [13], with the net effect that the number of bugs per product is increasing. It is therefore necessary to quickly find a way of reducing bug density by at least an order of magnitude. A promising direction is to reduce reliance on human labor through automated testing techniques, and recent proposals [4, 5, 2, 1] have made promising progress along these lines. Alas, they are still not ready to handle real-sized software (1 million lines of code or more), mainly due to high CPU and memory requirements. We believe cloud computing can come to the rescue.

The Promise of Automated Testing as a Service (TaaS) Software testing essentially consists of exercising as many paths through a program as possible and checking that certain properties hold along those paths (no crashes, no buffer overflows, etc.) TaaS combines two ideas: (1) offering software testing as a competitive, easily accessible Web service, and (2) doing fully automated testing in the cloud, to harness vast, elastic resources toward making automated testing practical for real software. A software-testing Web service allows users to upload the software of interest, instruct the service what type of testing to perform, click a button, and then obtain a report with the results within minutes or hours. This report is a list of bugs found, or the level of coverage obtained by tests with successful outcomes. Such a service can have a basic interface, where an end user uploads, e.g., the latest Windows service pack and then chooses from a menu of possible test types (e.g., comprehensive testing, security testing). A service can also have an expert interface, to be used by software developers to provide sophisticated definitions of what “a bug” may be, thus teaching the testing service what kinds of correctness violations to look for. For professional uses, TaaS can integrate directly with the development process and test the code as it is written.

4. CONTEMPORARY ASPECTS OF SOFTWARE TESTING

Software testing, as a practice, has been able to successfully evolve over time and provide efficient and constant support for improvements in software quality. On the other hand, testing is still notorious for its massive resource consumption within software projects. To this date, much of the research efforts on software testing have been focusing on designing new techniques, as well as investigating their effectiveness in real development contexts. However, during the entire history of software development, testing methods and techniques have struggled to keep up with the ever faster evolution and trends in software development paradigms. We cannot expect any favorable change in this state of affairs, unless a conscious effort is made in anticipating the trends, learning the stakeholder mindsets, and pinpointing the problem areas. It is our belief that such an effort could help in efficiently allocating the testing resources toward a specific context or in
proactively deciding the testing research agenda in general.

To our knowledge, there exist no detailed investigations with such a perspective. The research presented in this paper is a small step in this direction, in that it specially focuses on the stakeholder perspectives on some contemporary aspects related to testing. The specific research question we address in this paper is: Is it possible to identify and list main discrepancies between current and preferred testing practices that could be considered as obstacles for software testing practitioners?

By qualitatively and quantitatively analyzing the results of a recent questionnaire survey on practices and preferences in industrial software development, with respect to the above research question, we have identified a number of areas and practices where the preferred practice significantly differs from what is perceived as the actual current practice. We believe that these areas and practices assist in pointing out directions for future research within software testing.

5. SHEALTH TEST
Software testing is a critical component of the software engineering process. Despite studies showing that testing accounts for more than half the software development cost [11], software bugs persist. A NIST study [33] indicates that software bugs cost the U.S. economy an estimated $60 billion annually. The emergence of multicore architectures and the concomitant push toward concurrent multithreaded software compounds the complexity of the testing process by significantly increasing a program’s possible execution paths. New techniques are needed to make software testing tractable and effective for real-world software development.

One promising approach is on-line testing [18, 26, 28], where tests are run on deployed software systems. By running tests across a wider range of scenarios, these approaches can test a much larger and more diverse set of executions compared to traditional lab-based testing. Safe online testing must be transparent to the system’s execution (i.e., not alter program state and correctness). Even if a test leads to corrupted state, the transparency property prevents the application from observing it. Currently, providing functional transparency for online tests could incur significant overheads [5, 34]; since systems rely on application memory snap shooting via process forking. Our results (Section III (A)) confirm that using fork () for online testing can reduce an apache web server’s maximum throughput by as much as 40%. In our view, software customers are not likely to tolerate significant performance artifacts for activities traditionally done during development. Thus, widespread adoption of online testing depends upon mechanisms that provide both functional and performance transparency.

6. EFFECTIVENESS OF SOFTWARE TESTING FOR SAFETY
Evaluation of effectiveness of software testing has been considered in [1-5] and many other research papers. Selection of the appropriate testing approach for each specific software allows increasing software quality. It is especially important for safety-critical software when failures can have catastrophic consequences and affect human life. Effective testing may be expensive but, for safety-critical software, it is compensated by the higher reliability and safety level of software. One important software testing technique is combinatorial testing; it generates test sets that are relatively small while the input domain is astronomically large [6]. Well known examples of combinatorial testing include pair-wise, three-wise, and n-wise testing [7]. Many researchers have explored various combinatorial algorithms for test generation [8] as well as the efficiency of pair-wise testing and other approaches, yet more research still needs to be conducted. This is particularly significant for safety-critical systems because it can help predict the quality of testing, which in turn guarantees reliability of such systems.

This paper provides the results of experimental evaluation of effectiveness of pair-wise testing and is structured as follows. Section 2 presents a brief review of related works. We are especially focused on results in experimental evaluation of the combinatorial testing methods. In Section 3 we consider methods and tools which we use for testing effectiveness evaluation. Two existing software tools are used to generate test cases according to pair-wise criterion. We developed our own software tool for experimental numerical assessment of effectiveness. This tool generates faults of various specific types and applies the previously derived test sets to determine if the testing method detected those faults. Section 4 offers the experimental results and compares them with results by other researchers. We investigate fault detection for five various types of faults in 20 different logical expressions which contain from 5 to 14 logical variables each. Three different pair-wise test sets, generated by two different tools, are applied for each logical expression and each generated fault. Finally, Section 5 addresses general conclusions and directions for future work.
7. CONCLUSIONS

We have presented in vivo testing, a novel testing approach that supports the execution of tests in the deployment environment, without affecting that application’s state. We have also presented in vivo tests, which execute within a running application and test properties of the application that must hold regardless of the state the process is in. Last, we have described a Java implementation of our framework, called Invite. Through our initial findings, we have presented some real-world examples of defects that could be detected, and have demonstrated that our approach and the current implementation add limited overhead in terms of system performance and code modification. Testing in the deployment environment has been identified as a future challenge for the software testing community [5], and we expect that vivo testing will provide a foundation for future work in this field.

8. REFERENCES


