View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Columbia University Academic Commons

Using Runtime Testing to Detect Defects in Applications without Test Oracles

Christian Murphy, Gail Kaiser
Dept. of Computer Science
Columbia University
New York NY 10027
{cmurphy, kaiser}@cs.columbia.edu

ABSTRACT

We address the testing of complex, highly-configurable systems - particularly those without test oracles - by testing in the field using built-in oracles from functions' metamorphic properties.

1. PROBLEM STATEMENT

For large, complex software systems, it is typically impossible in terms of time and cost to reliably test all configuration options and all possible system states before releasing the product into the field. 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 these prior to release, because they didn't exist yet. Thus, we require a testing approach that allows for the testing of such systems in the context of all possible configuration options, execution environments, and system states.

This problem is compounded by the fact that, even if it were possible to test an application in all its possible configurations, environments, and states, there still remains a certain class of applications that can be called "non-testable programs" [21] because there is no reliable "test oracle" to indicate what the correct output should be for arbitrary input. Machine learning applications (among others, such as simulations and optimization algorithms) fall into this category. Formal proofs of an ML algorithm's optimal quality do not guarantee that an application implements or uses the algorithm correctly. Without an oracle, we cannot demonstrate correctness of the implementation, but we need a testing approach that can at least demonstrate the presence of defects.

2. APPROACH AND HYPOTHESES

A solution to the problem described above would need to address not only the issue of multiple possible configurations, environments, and states, but for some applications, the absence of an oracle as well.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FSE '08 Atlanta, Georgia USA
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

2.1 Proposed Approach

There are two aspects of our proposed approach. First, we suggest that continuing to execute tests in the field, after deployment, will reveal defects that are dependent on configuration and the execution environment. More importantly, by executing tests from within the software while it is running under normal operations and use, additional defects that depend on the system state (or a combination of state and environment) will also be revealed. This approach requires the creation of a new type of test that is designed to be run from within the application, as it is executing. These are tests that ensure that properties of the application hold true no matter what the application's state is, and regardless of its configuration or runtime environment.

An example can be found in Mozilla Firefox 2.0. One of the known defects is that attempting to close all other tabs from the shortcut menu of the current tab may fail with an error message on Mac OS X when there are more than 20 tabs open[1]. In this case, a test designed to run in the field would be one that calls the function to close all other tabs, then checks that no other tabs are open; this sequence should always succeed, regardless of how many tabs were open or what operating system is in use. Particular combinations of execution environment and state may not always be tested in development prior to release of the software, and one way to fully explore whether this property holds in all cases is to test it in the field, as the application is running.

The second aspect of our approach considers applications that depend heavily on configuration, environment, and state but in particular have no test oracle. Although it may be impossible to know if the output of the application is correct for arbitrary input, often these applications exhibit properties such that if the input or system state is modified in a certain way, it should be possible to predict the new output, given the original output. This approach is a variant of what is known as "metamorphic testing" [1].

For example, anomaly-based network intrusion detection systems often build up a model of "normal" behavior based on what has previously been observed; this model may be created, for instance, according to the byte distribution of incoming network payloads. When a new payload arrives, its byte distribution is then compared to that model, and anything deemed anomalous causes an alert. For a particular input, it may not be possible to know *a priori* whether it should raise an alert, since that is entirely dependent on the model. However, if while the program is running we take the

[1] http://www.mozilla.com/en-US/firefox/2.0.0.16/releasenotes/

new payload and randomly permute the order of its bytes, the result (anomalous or not) should be the same, since the model only concerns the distribution, not the order. If the result is not the same, then a defect must exist.

This approach does not require an oracle for the particular input; it only requires the software authors to specify the function's metamorphic properties. Moreover, this has all the benefits of testing in the field: the tests are conducted within the context of the runtime environment, but also within the context of the application's state, in this case the model that has been created over time. The use of such an approach in the development environment may not reveal defects if the initial test inputs are not sufficient, or if the functions rely on application state or execution environments that were not or could not have been tested prior to deployment. However, when we use this approach in the field, we will get a wide range of input values that represent actual usage, as opposed to a smaller set of test cases that are conjured up by developers in the lab.

2.2 Hypotheses

Two main hypotheses will thus be investigated. First, that executing tests within the context of an application running in the field can reveal defects that would not ordinarily otherwise be found. Second, that this approach can further be extended to applications for which there is no test oracle by using a variant of metamorphic testing at runtime. That is, the approach can reveal defects that would not be found using metamorphic testing prior to deployment.

3. MODEL

This section provides more detail about our approach. Here we identify the types of tests that we propose, and a framework that is used to execute these tests from within the running application.

3.1 In Vivo Tests

We first identify a new type of tests, called *in vivo tests*, that are designed to be executed in the context of the running application. In object-oriented programming languages (Java, C#, *etc.*), for example, these tests would reside in the same class as (or a subclass of) the class whose methods they are testing, so that objects can be tested "from within", using their current accumulated state, as opposed to testing from a clean or constructed state, as is typical in unit testing [12]. In vivo tests go beyond program invariants or assertions [5], and focus more on sequences of actions for which the correct response or final state is expected to hold, regardless of the state of the application or the environment.

It is important to note that in vivo tests are not intended to replace unit or integration tests, but rather to enhance them by making it possible to check for correct behavior within the context of an application running in the field, which may be in a previously untested or unanticipated state. In fact, in vivo tests could be used in the development environment as well, and the creation of these tests may aid in the creation of further unit tests.

3.2 Metamorphic In Vivo Tests

Next, to address the absence of a test oracle, we will create tests called *metamorphic in vivo tests* that are to be executed in the running application, using the arguments to selected functions as they are called. The arguments are modified according to the specification of the metamorphic properties, and the output of the function with the original input is compared to that of the modified input; if the results are not as expected, then a defect has been exposed. This will allow us to not only execute tests in the field, within the context of the running application, but also to test those applications without a test oracle, by using the metamorphic tests themselves as built-in "pseudo-oracles" [7].

3.3 In Vivo Testing Framework

A testing framework that supports these tests has two primary requirements: execute the tests from within the context of the running application; and do so without affecting the user's application state, so that the user does not see the results of the test code rather than of his own actions.

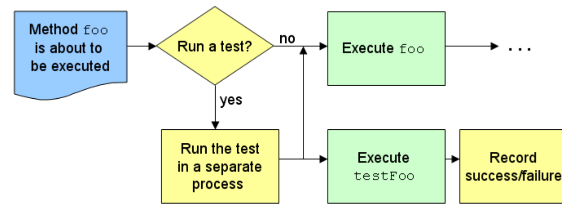

Figure 1: Testing Framework Model

In our model of the testing framework, tests are logically attached to the functions that they are designed to test. Thus, prior to a function's execution, the framework invokes the corresponding test with some probability. In order not to have the user see the effects of the test, the testing framework will execute the in vivo test in a separate process, so that any changes to the state are not reflected in the original process. This also means that the two processes execute in parallel: the in vivo test does not preempt the execution of the application code, which can continue as normal. Figure 1 demonstrates the model we will use for conducting in vivo tests. To ensure that the in vivo test does not make any changes to the file system, external databases, network traffic, *etc.*, we have begun to investigate integration with DejaView [14], an application which creates a virtual execution environment that isolates the process running the test.

The testing framework is also configurable, for instance to allow a developer to specify the probability with which in vivo tests will be run, on a per-method basis. To address some of the performance concerns with conducting tests from within a running application, the framework configuration can also specify the maximum number of concurrent test processes that are allowed to run simultaneously. The framework can also take advantage of multi-processor/core architectures by assigning the test process to a separate processor/core, so as to further reduce overhead. All configuration can be done by the software vendor or a system administrator prior to deployment.

4. FEASIBILITY

In [16] we describe a Java implementation of our in vivo testing framework, called Invite, which uses AspectJ to instrument the code at compile time. Here we summarize additional experiments and results evaluating the feasibility of using the in vivo testing approach, and also explore the performance impact of our approach.

4.1 Detecting Software Defects

We have investigated OSCache 2.1.1 [18], which contained three known defects that we speculated could be detected with the in vivo testing approach. These defects included one that only appeared when the application was in a certain state, one that depended on the system configuration, and another that only appeared intermittently; these are the exact types of defects that in vivo testing is designed to detect.

Unfortunately the unit tests that are distributed with that version of OSCache do not cover the methods in which those defects are found, so to show that "traditional testing" would not detect the defects, we created our own unit tests that would reasonably exercise those parts of the application. Those tests passed (*i.e.* did not reveal the defect) in the development environment during traditional unit testing, primarily because we had created the tests assuming a clean state which we could control (which, we feel, is a reasonable and common assumption [12]).

We then developed in vivo tests, using those unit tests as a starting point; it took less than one hour to complete this task. Next we instrumented the corresponding classes in OSCache with the Invite framework, and created a test application that randomly added and removed elements of random size from the cache, and randomly flushed the cache. All three defects were revealed by Invite in less than two hours. A similar experiment was also conducted with Apache JCS version 1.3, and we are now implementing a Java version of a framework to support metamorphic in vivo testing.

4.2 Performance Testing

We have conducted tests to determine the additional overhead introduced by the Invite framework. For our performance testing, we used the Ashes Fast Fourier Transform benchmark on Java 1.6.0 on a Linux Ubuntu 2.7.1 system with a quad-core 2.4GHz CPU and 1 GB of memory. We configured the framework so that it would only run one test at a time, and also so that the main process would run on one core and the test would run on another.

Our results indicate that incurring an overhead of just 5% (which we believe is typically less than what is noticeable to a human user) still achieves almost 700,000 tests per day, and an overhead of 10% achieves over four million tests per day. Although more investigation is still needed, this experiment demonstrates that it is possible to gain the benefits of in vivo testing with limited performance overhead.

In [4], we also investigated a distributed approach to in vivo testing and demonstrated that distributing the testing load would reduce performance overhead to an expected degree, but maintain the same global number of tests.

5. METHODOLOGY

Now that the feasibility of our approach has been demonstrated, we are in the process of proving our two hypotheses through a number of experiments. To demonstrate that in vivo testing can reveal defects that would not ordinarily otherwise be found, we will identify and then test a non-trivial application (we are considering some in the domains of scientific calculation and computer supported cooperative work) for which we have access to its unit tests, and demonstrate that these tests would not detect some (known) defects through traditional unit testing. We will then create a suite of new in vivo tests and then use them during the program's execution in a simulated live environment to show that the defects can, however, be revealed with our approach. Moreover, to show that in vivo testing advances the state of the art, we will compare it to a system such as DIDUCE [10], which detects and then performs runtime checks of program invariants. We will demonstrate that in some cases there are certain defects that cannot be revealed using only invariants, and that in other cases in vivo tests are more efficient and reveal the defects more quickly.

Next, to prove that runtime metamorphic testing is a feasible way of revealing defects in applications for which there is no test oracle, we will select one or more machine learning applications (we already have candidates from the fields of intrusion detection systems and computational biology), identify their metamorphic properties, and instrument them with the framework. We will then conduct runtime metamorphic testing on these applications as they run under normal operation in the field, and we expect that we will reveal new defects that were not previously known; we will also show that these defects would not have been detected by using metamorphic testing prior to deployment. And, as above, we will demonstrate that program invariants alone are not expressive enough to reveal these defects.

6. RELATED WORK

While the idea of "self-checking software" is by no means new [22], our work is principally inspired by the notion of "perpetual testing" [19], which suggests that analysis and testing of software should not only be a core part of the development phase, but also continue into the deployment phase and throughout the entire lifetime of the application. The in vivo testing approach is a type of perpetual testing in which the tests are executed from within the context of the running application and do not alter the state of that application from the users' perspective.

The Skoll project [13] takes a similar approach of extending testing into the deployment environment by the execution of tests at distributed installation sites, and then gathering the results back at a central server; to date this has mostly focused on acceptance testing of compilation and installation on different target platforms. Other approaches to testing software in the field include the monitoring, analysis, and profiling of deployed software, as surveyed in [8]. One of these, the GAMMA system [17], uses software tomography for dividing monitoring tasks and reassembling gathered information; this information can then be used for onsite modification of the code (for instance, by distributing a patch) to fix defects. Liblit's work on Cooperative Bug Isolation [15] enables large numbers of software instances in the field to perform analysis on themselves with low performance impact, and then report their findings to a central server, where statistical debugging is then used to help developers isolate and fix defects. All of these strategies could make use of in vivo testing as part of their implementation.

As the in vivo tests themselves can be considered extended program invariants, these are also similar to algebraic specifications [6]. Others have looked at the automatic detection of these specifications [11] and of detecting invariants in general (*e.g.* DIDUCE [10], Daikon [9], *etc.*), and then checking them at runtime [20]. We are considering the automatic generation of in vivo tests and of metamorphic properties, though this is outside the scope of our current work.

Applying metamorphic testing to situations in which there

is no test oracle has been studied in great detail by Chen *et al.*, *e.g.* [3]. Our work builds on theirs by applying metamorphic testing to the runtime environment, instead of using it to create new test cases prior to deployment. Additionally, whereas their work has primarily focused on functions with simple numerical input domains [2], we are working with inputs that consist of larger, alphanumeric data sets, as a result of the types of applications we are investigating.

Last, although there has been much work that applies machine learning techniques to software engineering in general and software testing in particular, there has thus far been very little published work in the reverse sense: applying software testing techniques to ML applications that have no reliable test oracle. Most so-called "testing frameworks" and reusable data sets in this domain are focused on comparing the quality of the results, *i.e.* how well the algorithm learns, and not evaluating the "correctness" of the implementations.

7. EXPECTED CONTRIBUTIONS

The contributions of this thesis are anticipated to include:

1. A new type of test called *in vivo tests*, which are state-based tests that ensure that properties of the application hold true no matter the application's state. These tests are designed to be executed from within the application, *i.e.* the application launches its own tests, within its current context.

2. A variation of these tests called *metamorphic in vivo tests*. These are similar to in vivo tests in that they execute within the application's current state, but they also test an application's metamorphic properties and thus are model-based as well. Unlike in vivo tests, these do not require an oracle upon their creation; rather, the properties act as built-in test oracles.

3. An *in vivo testing framework*, which supports the execution of the different types of tests from within the context of the running application, either in the development environment or the deployment environment. The testing framework will ensure that the execution of the tests does not affect the state of the original application process.

8. REFERENCES

[1] T. Y. Chen, S. C. Cheung, and S. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, 1998.

[2] T. Y. Chen, F.-C. Kuo, T. H. Tse, and Z. Q. Zhou. Metamorphic testing and beyond. In *Proc. of the International Workshop on Software Technology and Engineering Practice (STEP)*, pages 94–100, 2004.

[3] T. Y. Chen, T. H. Tse, and Z. Q. Zhou. Fault-based testing without the need of oracles. *Information and Software Technology*, 44(15):923–931, 2002.

[4] M. Chu, C. Murphy, and G. Kaiser. Distributed in vivo testing of software applications. In *Proc. of the First International Conference on Software Testing, Verification and Validation*, April 2008.

[5] L. A. Clarke and D. S. Rosenblum. A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Software Engineering Notes*, 31(3):25–37, May 2006.

[6] W. J. Cody Jr. and W. Waite. *Software Manual for the Elementary Functions*. Prentice Hall, 1980.

[7] M. D. Davis and E. J. Weyuker. Pseudo-oracles for non-testable programs. In *Proc. of the ACM '81 Conference*, pages 254–257, 1981.

[8] S. Elbaum and M. Hardojo. An empirical study of profiling strategies for released software and their impact on testing activities. In *Proc. of ISSTA 2004*, pages 65–75, 2004.

[9] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely programming invariants to support program evolution. In *Proc. of the 21st International Conference on Software Engineering (ICSE)*, pages 213–224, 1999.

[10] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proc. of the 24th International Conference on Software Engineering (ICSE)*, pages 291–301, 2002.

[11] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *Proc. of the 17th European Conference on Object-Oriented Programming ECOOP*, 2003.

[12] JUnit Cookbook. http://junit.sourceforge.net/doc/cookbook/cookbook.htm.

[13] A. Krishna et al. A distributed continuous quality assurance process to manage variability in performance-intensive software. In *19th ACM OOPSLA Workshop on Component and Middleware Performance*, 2004.

[14] O. Laadan, R. A. Baratto, D. B. Phung, S. Potter, and J. Nieh. Dejaview: a personal virtual computer recorder. In *Proceedings of twenty-first ACM SIGOPS symposium on operating systems principles (SOSP)*, pages 279–292, 2007.

[15] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Public deployment of cooperative bug isolation. In *Proceedings of the Second International Workshop on Remote Analysis and Measurement of Software Systems*, pages 57–62, May 2004.

[16] C. Murphy, G. Kaiser, and M. Chu. The in vivo approach to testing software applications. Technical Report CUCS-007-08, Department of Computer Science, Columbia University, 2008.

[17] A. Orso, T. Apiwattanapong, and M.J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proc. of the 9th European Software Engineering Conf.*, pages 128–137, 2003.

[18] OSCache. http://www.opensymphony.com/oscache.

[19] L. Osterweil. Perpetually testing software. In *The Ninth International Software Quality Week*, May 1996.

[20] S. Sankar. Run-time consistency checking of algebraic specifications. In *Proceedings of the 1991 international symposium on software testing, analysis, and verification*, pages 123–129, 1991.

[21] E. J. Weyuker. On testing non-testable programs. *Computer Journal*, 25(4):465–470, November 1982.

[22] S. S. Yau and R.C. Cheung. Design of self-checking software. In *Proc. of the International Conference on Reliable Software*, pages 450–455, 1975.