The In Vivo Approach to Testing Software Applications

Christian Murphy, Gail Kaiser, Matt Chu
Department of Computer Science, Columbia University, New York NY 10027
{cmurphy, kaiser, mwc2110}@cs.columbia.edu

ABSTRACT

Software products released into the field typically have some number of residual bugs 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, assumptions made during the creation of test cases, or the infeasibility of testing the sheer number of possible configurations for a complex system. Testing approaches such as perpetual testing or continuous testing seek to continue to test these applications even after deployment, in hopes of finding any remaining flaws. In this paper, we present our initial work towards a testing methodology we call *in vivo testing*, in which unit tests are continuously executed inside a running application in the deployment environment. These tests execute within the current state of the program (rather than by creating a clean slate) without affecting or altering that state. Our approach can reveal defects both in the applications of interest and in the unit tests themselves. It can also be used for detecting concurrency or robustness issues that may not have appeared in a testing lab. Here we describe the approach and the testing framework called Invite that we have developed for Java applications. We also enumerate the classes of bugs our approach can discover, and provide the results of a case study on a publicly-available application, as well as the results of experiments to measure the added overhead.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Verification

Keywords

Software Testing, Perpetual Testing, Continuous Testing

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 ("bugs") in an application is severely hampered by numerous factors. 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. A recent report [48] indicates that 40% of companies consider insufficient pre-release 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 assumptions by the authors. And, of course, despite progress in measuring test coverage and formal verification, it is only possible to detect the presence of bugs, not their absence.

One proposed way of addressing this problem has been to continue testing the application in the field, even after it has been deployed. The theory of this "perpetual testing" [44] approach is that, over time, defects will reveal themselves given that multiple instances of the same application may be run globally with different configurations, under different patterns of usage, and in different system states.

In this paper, we present a testing methodology we call *in vivo testing*, in which unit tests are continuously executed inside a running application in the deployment environment. In this new approach, tests execute within the current state of the program without affecting or altering that state. This approach can reveal defects both in the applications under test and in the unit tests themselves. It can also be used for detecting concurrency or robustness issues that may not have appeared in a testing lab (the "in vitro" environment).

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. In vivo testing can be used to detect bugs hidden by assumptions of a clean state in the unit 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. Our approach goes beyond application monitoring in that it actively tests the application, using the same unit tests from the development stage, with minimal modification to the application and unit test code.

Our main contribution is an approach to executing unit tests within the environment of a running system, and doing so without altering that system's state.

2. THE IN VIVO TESTING APPROACH

In vivo testing is a methodology by which unit tests are executed in the deployment environment, in the context of the running application, as opposed to a controlled or blank-slate environment. Tests are run continuously as the application runs, at appropriate points in the program execution. Crucial to the approach is the notion that the test must not alter the state of the application. In a live system in the deployment environment, it is clearly undesirable to have a test application altering the system in such a way that it affects the users of the system, causing them to see the results of the test code rather than of their own actions. This is done by executing the test in a separate process, which has been created as an exact copy of the original.

2.1 Conditions

In order for in vivo testing to be useful in practice for a given unit test and a corresponding piece of software to be tested, three conditions must be met. First, the unit test must pass in the development environment, even though there are unknown defects in the software under test (if the unit test fails before deployment, then obviously in vivo testing is not necessary). Second, under certain potentially-unanticipated circumstances the running application should give erroneous results or behavior in the deployment environment, *i.e.* have a bug. Lastly, for some process state or condition of use, the unit test must subsequently fail. If these conditions are met, it is possible for in vivo testing to detect that there is a bug. The bug may be one in the application code, or in the unit test code, or both.

2.2 Categories and Motivating Examples

Although the necessary conditions described above may appear to be rather restrictive, many such bugs exist in production applications. To examine the feasibility of our testing approach, we investigated the documented defects of some popular, open-source applications to see which of them could have been discovered using in vivo testing. The first application we considered, OSCache version 2.3 [6], is an open-source multi-level caching solution designed for use with JSP pages and Servlet-generated web content. In addition, we looked at different versions of Apache Tomcat [1], a Java Servlet container.

We identified five different categories of defects that in vivo testing could potentially detect. The categories are listed in Table 1. There may be other categories of bugs that could be found with in vivo testing, but these are the ones identified so far.

The first category of defects likely to be found by in vivo testing are those in which the corresponding unit test assumes a clean slate, but the code does not work correctly otherwise. By clean slate, we mean a state in which all objects have been created anew and are modified only by the unit test or methods it calls, such that the unit test has complete control of the system. Generally unit tests are written in such a way that the objects being tested are created and modified to obtain a desirable state prior to testing [5]. In these cases, the code may pass unit tests coincidentally, but not work properly once executed in the field, revealing bugs in both the test code and the code itself. State-based testing [49] or static analysis [20] could be used to look for defects in this category, though these may not be as useful as in vivo testing when the system state depends heavily on external systems or user input sequences, which may not be able to be anticipated prior to deployment, or with respect to bugs caused by values determined at runtime, such as pointer variables or array indices.

Table 1: Categories of defects that can be detected with in vivo testing

1	Unit tests make incorrect assumptions about the state of objects in the application
2	Possible field configurations were not tested in the lab
3	A legal user action puts the system in an unexpected state
4	An unanticipated user action breaks the system
5	Those that only appear intermittently

One of the OSCache bugs[1] we discovered notes that, under certain configurations, the method to remove an entry from the cache is unable to delete a disk-cached file if the cache is at full capacity. In this case, the corresponding unit test for testing cache removal may simply add something to the cache, remove it, and then check that it is no longer there. A unit test that assumes an empty or new cache would pass, but when the cache is full, the test would fail, revealing a bug that may not have been caught in the development environment.

The second category of defects concerns those that come about from field configurations that were not tested in the lab. These, too, may reveal a bug in the code or in the unit test. Java server applications may require testing on multiple platforms with multiple JDK versions and multiple revisions of the application code; this is not always feasible for testing in a single test lab, particularly given the frequency with which companies must release their applications to be competitive in the marketplace. Additionally, a new JDK may be released after the software is deployed. System administrators of such applications may have numerous runtime configuration options, and not all combinations may have been tested before release. We note that a testing approach using a system like Skoll [24] [34] to run tests at the production site before deployment of the software could potentially find some defects in this category, but others will only reveal themselves once the application has been running for a while, and would not be detected prior to the application's deployment and widespread use.

Another OSCache bug[2] falls in this category. In this bug, setting the cache capacity programmatically does not override the initial capacity specified in a properties file when the value set programmatically is smaller. A unit test for the method to set the cache capacity may assume a fixed value in the properties file and only execute tests in which it sets the cache capacity to something larger; this test would pass. However, if a system administrator sets the capacity to a large number in the properties file, the unit test would fail when it tries to set the cache capacity to a smaller value, revealing the bug. This also happens to reveal a flaw in a poorly-designed unit test, as well, which is a side benefit of

[1] http://jira.opensymphony.com/browse/CACHE-236
[2] http://jira.opensymphony.com/browse/CACHE-158

the in vivo testing approach.

The third type of defect targeted by in vivo testing are ones that stem from a (legal) user action that puts the system in an unexpected state. This could happen when objects in the same process are shared between users, and one user's activities modify an object such that it does not work correctly for other users.

Concurrency bugs are a very common type of defect in this category. We noticed one of the concurrency bugs[3] in Apache Tomcat, in which a particular method used in the creation of a session is not threadsafe. If the thread that invalidates expired sessions happens to execute at the same time as a session is being created, it is possible that an exception would occur because one of the objects being used in the session creation could be set to null. A unit test that is simply testing the creation of sessions is not likely to detect this bug because at that time there may not be any other sessions to invalidate (this is also a case of the first type of defect targeted by in vivo testing, in which the unit test assumes a blank slate). However, in the deployment environment, this unit test may fail if the session invalidation thread is cleaning up other sessions at the same time. Note that here, we are in a sense testing the unit test as well as the application code, which is a differentiating feature of the in vivo testing approach.

The fourth type of defect that can be found by using in vivo testing are ones in which an unanticipated (but legal) user action causes the system to stop running (crash) or simply stop responding (hang). This may generally seem more like "monitoring" than "testing", but can still be addressed by our approach. Unlike the third category, in which the application continues to respond to users and appears to run normally, these are defects that cause the system to stop responding or to repeatedly give error messages.

For instance, one of the Apache Tomcat bugs[4] we studied is one in which there is a resource leak in the database connection pool. A single unit test to create, use, and release connections from the pool may not detect the leak if it is not executed enough times, particularly if the application is restarted between tests. However, in the field this error may arise if the test is executed repeatedly, and finally the test would fail when it could not obtain a connection. Because this does not result in a runtime error (the application just hangs while waiting for a free connection), a system monitor that is checking for uncaught exceptions would not detect this situation. On the other hand, a unit test that is run in vivo would eventually reveal this bug.

The fifth and final type of defect is one that only appears occasionally. These defects may be discovered by a continuous testing approach [46] during only the development phase, but the fact that our approach continuously tests the application even after deployment increases the chance of finding such a bug.

One such defect appears in OSCache[5], whereby flushing the cache, adding an item, and attempting to retrieve the item can occasionally result in an error, particularly if two calls to flush the cache happen within the same millisecond. A unit test that tries this sequence of actions may simply never encounter the error by chance during testing in the development environment, but an application fitted with the in vivo framework would catch it when it eventually occurs.

It is conceivable that some of the bugs documented here could have been discovered prior to release of the application given more time, better unit tests, and a little luck. But these examples demonstrate that a testing methodology that continues to execute unit tests on an application in the field greatly improves the chances of the errors being detected. More importantly, certain bugs will in practice only manifest themselves in the field (because of limited time and resources in the testing lab), and these are the ones for which in vivo testing is most useful.

2.3 In Vivo Testing Fundamentals

To apply the in vivo testing approach, the application vendor must first perform some preparation steps (described in Section 4.1), including the instrumentation of the portions of the application that are to be tested in the production environment. After these preparation steps have been performed and the application has been configured to take advantage of in vivo testing, it is deployed in usual fashion: the application user does nothing special and would not even know that in vivo testing is being performed. In vivo testing then works as follows: when an instrumented part of the application is to be executed, with some probability its corresponding unit test is then executed in a separate "sandbox" that allows the test to run without altering the state of the original application process. The application then continues its normal operation as the unit test runs to completion in a separate process, and the results of the test are logged. Note that the unit tests are only invoked as a result of the execution of the code they are testing, so that commonly used code is tested more often.

Although the in vivo testing approach is a general testing approach suitable to most types of applications, it is most appropriate for server-side network applications like web servers, SMTP servers, *etc.* These types of applications are long-lived and thus have frequent method calls and (presumably) little idle time, which increases the chances of unit tests being executed, and thus the effectiveness of in vivo testing.

3. RELATED WORK

Our work is principally inspired by the notion of "perpetual testing" [41] [44] [45] [54], 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. Perpetual testing advocates that analysis and testing should be on-going activities that improve quality through several generations of the product, in the development environment (the lab, or "in vitro") as well as the deployment environment (the field, or "in vivo"). The in vivo testing approach is a type of perpetual testing in which the same unit tests can be used in both environments with only minor modifications, and the tests do not alter the state of the application under test.

In vivo testing is also a form of "residual testing" [42]. This type of testing is motivated by the fact that software products are typically released with less than 100% coverage, so testers assume that any potential defects in the untested code (the residue) occur so rarely so as not to bear consideration. Much of the research in this area to date has focused

[3]http://issues.apache.org/bugzilla/show_bug.cgi?id=42803
[4]http://issues.apache.org/bugzilla/show_bug.cgi?id=42856
[5]http://jira.opensymphony.com/browse/CACHE-175

on measuring the coverage provided by this approach by looking at untested residue [38] [42] or by comparing the coverage to specifications [37]. However, this work does not consider the actual execution of unit tests in the deployment environment, as we describe here. Those approaches describe measurements of the residue, whereas we are attempting to discover the residual bugs by conducting tests. Our approach does not currently address coverage, but could be extended to do so, *e.g.* emphasizing testing of the residue but not restricting the testing to only the residue, since bugs could reside in already-tested code.

Also related to perpetual testing is "continuous testing", which refers to round-the-clock execution of tests, though typically in the development environment [46] [47]. However, the Skoll project [24] [34] has extended this into the deployment environment by carefully managed facilitation of the execution of tests at distributed installation sites, and then gathering the results back at a central server. The principal idea is that there are simply too many possible configurations and options to test in the development environment, so tests can be run on-site to ensure proper quality assurance. Whereas the Skoll work to date has mostly focused on acceptance testing of compilation and installation on different target platforms, in vivo testing is different in that it seeks to execute unit tests within the application while it is running under normal operation. Rather than check to see whether the installation and build procedure completed successfully, as in Skoll, in vivo testing seeks to execute unit tests as the application runs in its deployment environment. Additionally, although the in vivo approach does not currently address performance testing, as Skoll does, our approach could be enhanced to maintain records of resource utilization of the individual units tested, for instance to help detect bottlenecks where optimization may be warranted, or in cases where *a priori* assumptions about resource utilization turn out to be off base in the field for a particular installation.

While the notion of "self-checking software" is by no means new [53], much of the recent work in executing tests in the field has focused on COTS component-based software. This stems from the fact that users of these components often do not have the components' source code and cannot be certain about their quality. Approaches to solving this problem include using retrospectors [29] to record testing and execution history and make the information available to a software tester, and "just-in-time testing" [28] to check component compatibility with client software. Work in "built-in-testing" [51] has included investigation of how to make components testable [12] [13] [14] [33], and frameworks for executing the tests [17] [32] [35], including those in embedded systems [43] and Java programs [18], or through the use of aspect-oriented programming [31].

In light of all these important contributions, in vivo testing differentiates itself by providing the ability to test any arbitrary part of the system (not just COTS components) and by utilizing existing unit test code, rather than requiring extensive modification to the original source to provide special functional and testing interfaces [8] [50] or enforcing a rearchitecture of the application to allow for the use of testers and controllers/handlers [10] [36] [50]. The advantage of the in vivo testing approach over these others is that we are providing a framework for perpetual testing of an existing application with minimal modification, as opposed to prescribing a methodology for developing an application so that it may be tested after its deployment.

Other approaches to testing software in the field include the monitoring, analysis, and profiling of deployed software, as surveyed in [19]. One of these, the GAMMA system [39] [40], 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 [27] 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 bugs. Clause [16] has looked at methods of recording, reproducing, and minimizing failures to enable and support in-house debugging, and Baah [9] uses machine learning approaches to detect anomalies in deployed software. All of these strategies could make use of in vivo testing as part of their implementation.

4. THE IN VIVO TESTING FRAMEWORK

The in vivo testing framework, which we call Invite (IN VIvo TEsting framework), has been implemented in Java and has been designed to reuse existing test code as much as possible, while not imposing restrictions on the design of the software application. This section describes the steps that must be followed to prepare an application for in vivo testing, and how the tests are actually executed in the deployment environment.

4.1 Preparation

Here we describe the steps that a software vendor would need to take to use the Invite framework. We start by assuming that unit tests have already been written in the JUnit [4] style, though this may not always be the case. In situations where unit tests have not been written at all, then they should be created according to the following guidelines.

Step 1. Create test code. In order to use Invite, the software vendor must ensure that the unit test methods reside in the same class as the code they are testing. Also, the unit test for a method "foo" should be a public method called "testFoo", which returns a boolean (to indicate whether or not the test passed, so that Invite can log the result and possibly take some appropriate action) and takes no arguments. Additionally, rather than create new objects to test in the test methods, those methods should use existing objects (*i.e.* the one in which the method resides, or other objects directly accessible through it), since the goal of in vivo testing is that, when the test is run in the field, it is using the object that has been modified over the course of the application's execution.

Certainly there are unit tests that may not seem appropriate for in vivo testing. Take, for instance, an example of a simple Vector. A unit test might create a new Vector, put in one element, remove that element, and then check that the Vector is empty. This is a perfectly valid test of the Vector's ability to remove elements, but if it is changed to use an existing Vector that has accumulated state (and elements), then this test would fail because after adding and removing one element, the Vector is obviously not going to be empty. However, this test could easily be modified for in vivo testing by changing the emptiness check to simply see if the removed element no longer exists. The goal is the same (checking that the Vector's method to remove el-

ements works correctly) but now this test will work even if it uses a Vector that already has elements in it. On the other hand, leaving the unit test as is, *i.e.* allowing it to create a new object instead of using an existing one, still has utility within in vivo testing because it can be used to test underlying APIs and system library calls, such as those that allocate memory and maintain pointer references.

It is important to note that this step does not require any modification or special constraints on the design of the software application itself; it may merely require moderate changes to the test code, which would be done *a priori* by the vendor who plans to distribute an in vivo-testable system, and not by the customer in whose environment the tests run.

To demonstrate the type of minor modification that would be necessary to use the in vivo testing approach, Figure 1 shows a simple Java class, and Figure 2 shows its corresponding test class in the JUnit [4] style, specifically that a separate class is used to hold all the test methods, and uses a "setUp" method to create new objects to test. Figure 3 shows the modified application code in the in vivo testing style: (1) the test methods have been moved into the same class; (2) the names of the test methods have been changed to match that of the method each is testing; (3) the return type of the test methods has been changed; and (4) the reference to the object being tested (in this case, the DataHolder) in the test methods is now "this" (implied) instead of a newly-created object.

```
public class DataHolder {
   public int largestSoFar = 0;
   public void update(int x) {
      if (x > largestSoFar) largestSoFar = x;
   }
   public void reset() {
      largestSoFar = 0;
   }
}
```

Figure 1: Original application code

```
public class DataTest {
   private DataHolder d;
   @Before public void setUp() {
      d = new DataHolder();
   }
   @Test public void testUpdateData() {
      d.update(5);
      d.update(3);
      assertTrue(d.largestSoFar == 5);
   }
   @Test public void testResetData() {
      d.update(5);
      d.reset();
      assertTrue(d.largestSoFar == 0);
   }
}
```

Figure 2: Original test code in JUnit style

Most importantly, note that no changes were necessary to the DataHolder class (aside from adding the new test methods) or to the application logic (the "update" and "reset" methods). If the "update" or "reset" methods created new

```
public class DataHolder {
   public int largestSoFar = 0;
   public void update(int x) {
      if (x > largestSoFar) largestSoFar = x;
   }
   public void reset() {
      largestSoFar = 0;
   }
   public boolean testUpdate() {
      update(5);
      update(3);
      return (largestSoFar == 5);
   }
   public boolean testReset() {
      update(5);
      reset();
      return (largestSoFar == 0);
   }
}
```

Figure 3: Modified application code

objects, that would be totally fine in the world of in vivo testing and would not necessitate modification; we are only concerned with changing references to the objects used by the test methods. Although this is clearly a toy example, it demonstrates that an application's test code can be rewritten in the in vivo testing format with minimal effort, and no significant changes to the application code are necessary.

Step 2. Instrument classes. After making any necessary code modifications, the vendor must then select one or more Java classes in the application under test for instrumentation, such that all method calls in the class will be points at which a unit test could be run. Aside from acting as jumping off points for the unit tests, the instrumented classes are also the same ones that will be tested by the Invite system, and should be selected according to which ones the vendor wants to test (this could certainly be all of the classes, of course). The list of classes is specified in a plain-text file. To achieve this instrumentation, Invite uses a Java component written in the aspect-oriented programming language AspectJ [2], which is woven into the instrumented classes. This does not require any modification of the original source code; it only calls for recompilation, though this restriction could be lifted by use of a system like [22], which would dynamically insert the test harness code into the application without recompilation.

Step 3. Configure framework. Before deployment, the vendor would configure Invite with values representing, for each method with a unit test in the instrumented classes, the probability p with which that method's unit test will be run. This configuration is specified in a plain-text file, where each line contains the name of the class, the name of the method, and the percent of calls to that method that should result in execution of the corresponding unit test. For ease of use, the list of methods that have unit tests is auto-generated by Invite as part of the instrumentation phase; the vendor need only specify the p values. The file is read at run-time (not at compile-time) so it can be modified by a system administrator at the customer organization if necessary. A "DEFAULT" value can be specified as well: any method not explicitly given a percentage will use that global default. If the global default is not specified, then

the default percentage is simply set to zero, which provides an easy way of disabling all in vivo testing for all but the specified methods. To disable testing for all methods in the application, the administrator can simply put "DISABLE" in the first line of the file. Note that if method "foo" is called twice as frequently as method "bar", and both have equal p values, then "testFoo" is going to be called twice as frequently as "testBar", which we feel is desirable since that method should be tested more often since it is called more often.

Step 4. Deploy application. It is assumed that the application vendor would ship the compiled code including the unit tests and the configured testing framework as part of the software distribution. However, the customer organization using the software would not need to do anything special at all, and ideally would not even notice that the in vivo tests were running.

4.2 Implementation Details

Whenever a method of an instrumented class is invoked, the AspectJ code weaved in by Invite uses the percentage value p for that method to decide whether to execute a test. If Invite decides that a test is to be run, it uses Java Reflection to see if the method has a corresponding "test" method (for performance reasons, however, we cache the results of previous checks to see if the test method exists). This is the unit test that will then be executed. The purpose of running a method's corresponding "test" method is so that the unit test is executed at the same point in the program (the same state) as the method itself. This makes it possible to see how the test performs in the same state in which the method performs, which is preferable to arbitrarily choosing a random test to execute, since there may be states when such a test is *not* expected to work correctly.

If the test method exists and it is determined that a test should be run, Invite then forks a new process (which is a copy of the original) to create a sandbox in which to run the test code, ensuring that any modification to the local process state caused by the unit test will not affect the "real" application, since the test is being executed in a separate process with separate memory. As Invite is currently implemented in Java, and there is no "fork" in Java, we have used a JNI call to a simple native C program which executes the fork. Performing a fork creates a copy-on-write version of the original process, so that the process running the unit test has its own writable memory area and cannot affect the in-process memory of the original. Once the test is invoked, the application can continue its normal execution, while the unit test runs in the other process. Note that the application and the unit test run in parallel in two processes; the test does not pre-empt or block normal operation of the application after the fork is performed.

In the current implementation of Invite, unit test modifications to files, network I/O, the operating system, external databases, *etc.* are not automatically undone; the sandbox only includes the in-process memory of the application. Though this somewhat limits the type of testing that can be performed currently, there are still many categories of defects (listed in Section 2) that can be detected when considering tests that only utilize and affect the state of the process in memory. Furthermore, if a "tearDown" method exists in the class in which the unit test was run, that method is executed upon completion of the test, allowing for any programmatic clean-up that needs to be done (though, as described previously, it is not necessary to restore in-process memory to its original state, only that of external systems). To further address this limitation in a more automatic fashion, we are currently integrating Invite with DejaView [25], an application which creates a virtual execution environment that isolates the process running the unit test and gives it its own view of the file system, so that it will not affect files used by the original process.

When the unit test is completed, Invite logs whether or not it passed, and the process in which the unit test was run is terminated. Invite provides a tool for analyzing the log file and providing simple statistics like the number of tests run, the number that passed/failed, and a summary of the success/failure of each instrumented method's unit test. We have also implemented a "client-server" version of Invite [15] in which all errors are reported back to a central server (presumably this would be set up at the vendor's location), and could be processed as in [40] or [34], wherein configuration parameters (like the frequency of test execution or even the list of classes to test) could then be modified.

Unlike other testing approaches that test the application as it is running, such as [18] or [36], Invite avoids the "Heisenberg problem" of having the test alter the state of the application it is testing. This is the major contribution and differentiating characteristic of the in vivo testing approach.

4.3 Comparison to JUnit Conventions

We originally considered maintaining the JUnit style for the test code, in particular that the test code resides in a different class. However, a major limitation of this approach is that, in JUnit, new objects to test are created (for instance, in the "setUp" method [5]), but this defeats the point of in vivo testing, which is designed to test within the confines of the state of the running application, and not on new or clean objects. Additionally, because JUnit tests reside in classes separate from the ones they are testing, this results in the added difficulty of providing arbitrary access to existing objects (which is necessary for in vivo testing to be effective) from separate classes without modifications, possibly major ones, to the application code. For instance, if a new DataTest object (Figure 2) is created and its test methods are invoked, there is no way for it to access an existing instance of the DataHolder class for its testing because the test methods do not take arguments, nor does the "setUp" method.

One alternative would have been to add a method to the JUnit classes for purposes of in vivo testing (or allow the test methods or "setUp" to take arguments), but this still brings up the issue of the absence of a clear mapping in terms of names between a method and its corresponding unit test in the JUnit style, in which test methods can be arbitrarily named. For these reasons, we abandoned the JUnit conventions, and require that the test code reside in the same class as the application code. A desirable side effect of this is that private methods can now be tested, as well, since all code is in the same class.

When converting from the JUnit style to the in vivo testing style, sometimes a JUnit test may be designed to test a number of methods, not just one, in order to test the state of the object after a particular sequence of actions. In the cases where there is not a clear one-to-one mapping between the methods to be tested and the unit tests, it is necessary

```
public class DataHolder {
   public int largestSoFar = 0;
   public void update(int x) {
      if (x > largestSoFar) largestSoFar = x;
   }
   /** This is the dispatch method **/
   public boolean testUpdate() {
      if (Math.random() > 0.5)
         return testUpdateIncreasing();
      else
         return testUpdateDecreasing();
   }
   /** Original JUnit test method **/
   public boolean testUpdateIncreasing() {
      update(5);
      update(10);
      return (largestSoFar == 10);
   }
   /** Original JUnit test method **/
   public boolean testUpdateDecreasing() {
      update(10);
      update(15);
      return (largestSoFar == 10);
   }
}
```

Figure 4: The number of methods to be tested is less than the number of test methods.

to use intermediate placeholder methods. For instance, if the number of methods to be tested is less than the number of test methods, then the existing JUnit test methods (with original names intact) can still be used, but the unit test named in the in vivo style should arbitrarily decide between the JUnit tests and then dispatch accordingly. In Figure 4, there is only one method to test ("update") but it has two corresponding unit tests: "testUpdateIncreasing" and "testUpdateDecreasing". The "testUpdate" method, which would be invoked by Invite, then must choose between the two test methods.

On the other hand, in the situation when a single test is used to test numerous methods, there may be more methods to be tested than there are unit tests. Such an example is shown in Figure 5, in which the JUnit test method "testSequenceOfUpdatesAndResets" is meant to test *both* the "update" and "reset" methods. In this case, the "testUpdate" and "testReset" methods simply pass along the return value of the one, singular unit test.

4.4 Scheduling Execution of Tests

We have also considered other policies for determining how frequently unit tests should be run, aside from the static configuration value. For instance, if it is desirable to have all the test cases run equally often, then the p value could be automatically adjusted to increase probability for a method that, empirically, runs rarely, and lowered for one that runs often. Another policy would be to multiply the weighting (which treats all essentially equally but considers how often they run in practice) by some factor that is larger for methods/classes where more bugs were found during lab testing and/or more field bugs were reported, so as to increase the likelihood of finding a bug in a potentially flawed method or class. One other idea is for Invite to automatically alter the

```
public class DataHolder {
   public int largestSoFar = 0;
   public void update(int x) {
      if (x > largestSoFar) largestSoFar = x;
   }
   public void reset() {
      largestSoFar = 0;
   }
   public boolean testUpdate() {
      return testSequenceOfUpdatesAndResets();
   }
   public boolean testReset() {
      return testSequenceOfUpdatesAndResets();
   }
   /** Original JUnit test method **/
   public boolean testSequenceOfUpdatesAndResets() {
      update(8);
      reset();
      update(10);
      reset();
      update(6);
      return (largestSoFar == 6);
   }
```

Figure 5: The number of methods to be tested is greater than the number of test methods.

p value based on the desired frequency of test execution, or an acceptable performance overhead. For instance, the value can be raised when there is less usage of the application, so that more tests will run but the system will not be under excessive load. The relative effects of these different policies are outside the scope of this paper.

4.5 Configuration Guidelines

In order to help a system administrator or vendor understand the configuration's impact on performance and testing, Invite periodically records to a log file the total number of unit tests that have been run, the average time each test takes, and the number of tests run per second. All of these statistics are tracked globally, but also for the separate methods, since they may have different p values. From this data, it is then possible to estimate how altering the value of p will affect the system's performance and number of tests executed.

Specifically, the rate of tests run per second is proportional to p: for instance, to double the frequency of execution of a particular test, simply double the method's p value. This simple calculation will help guide how to adjust p so as to execute more (or fewer) unit tests for a given method.

To estimate the performance overhead caused by the unit tests, one can multiply the number of unit tests by the average time each takes to see what additional time is being spent running those tests. Then, by calculating the effect that p has on the number of tests being run per unit time, one can then calculate the additional overall time cost of increasing or decreasing p. We surmise that, in practice, the p values would presumably be very small (perhaps around 1%). However, these are heavily dependent on the number of instrumented methods, the frequency with which they are called, the desired amount of testing to be performed, and the acceptable performance degradation. We discuss more performance issues in Section 6.

5. CASE STUDY

Given the numerous motivating examples listed in Section 2, we sought to apply Invite to a different publicly-available application, in order to determine whether the approach would work to detect more defects. The application we instrumented for testing was the Jetty WebServer 6.1 [3], a Java HTTP server that also supports the Java Servlet API. We chose it primarily because it is open source and provides its own unit tests.

We first selected 13 classes from the Jetty distribution, and to those classes we added the corresponding test methods (approximately 50 in total), taken from the JUnit tests that are included with Jetty. We needed to modify some of the tests as described above; to get references to the object in the running application, rather than creating new ones, we changed those references to point to the object in which the test was being run, or to other objects directly accessible through the current one. Note that these changes were only necessary because Jetty uses the JUnit style, and the tests had already been written. However, these changes took less than eight man-hours to complete, and no changes were needed for the application code, nor were any other restrictions imposed.

To ensure that there would be at least one defect to potentially find, we then planted a bug in Jetty in the "copyThread" method of an I/O utility class. This method is given an input stream and an output stream as arguments, creates a new thread, reads from the input stream in its entirety, and writes to the output stream. We removed the call to the output stream's "flush" method after the copy had been completed; this omission is a very easy mistake to make, but will not always cause an error in the application because the output stream may send the bytes anyway without explicitly being flushed. Its unit test, which shipped with the Jetty distribution as opposed to being written by us, creates and initializes a byte array input stream, invokes the "copyThread" method, waits 1.5 seconds, and then reads from a byte array output stream to see if the data were correctly copied.

When we executed all of the tests outside of the running program to ensure that the tests would pass under normal circumstances, *i.e.* outside of the in vivo testing framework, the unit test for the "copyThread" method also passed, even though a bug was present. This is exactly the type of defect that in vivo testing is designed to find, as it falls into the category of "bugs that only appear intermittently", as described in Section 2.

In order to attach Invite to Jetty, we then used AspectJ to instrument the 13 classes, most of which are used in every page request, and then weave in the Invite code, so that every page request had a chance of causing one or more unit tests to be invoked. To simulate user activity on the Jetty web server, we used The Grinder [7], a load testing tool, to request a series of static and dynamic web pages.

The unit test for "copyThread" generally passed during in vivo testing, but occasionally (approx. 15% of the time) failed when there was load on the web server, because the byte array of the output stream would sometimes be empty at the end of the test. We speculated that the 1.5 second waiting time in the unit test was not always enough to copy over the bytes from the input stream to the output stream, and increased the value to 10 seconds but still the error occasionally appeared (there were only 44 bytes being copied and this certainly should not take 10 seconds). When we restored the call to flush the output stream, the error disappeared. Thus, it was the failure to flush the output stream that was intermittently causing the bytes not to appear there, and in vivo testing discovered this bug in the code.

This example demonstrates one type of intermittent bug that may not be revealed in traditional unit testing in the development environment, but could appear in the deployment environment, and would be detected with in vivo testing. Although we did not find any unexpected bugs in the 13 classes we instrumented, aside from the one we planted, our testing is continuing, and we expect to find others in the future. More importantly, this case study demonstrates the technical feasibility of our approach and is indicative of its efficacy in such situations.

6. PERFORMANCE EVALUATION

We are concerned with the performance impact of our approach, particularly in using aspect-oriented programming to instrument potentially numerous method calls (perhaps all of them), and the overhead incurred by forking a process through a native method call to create a new process in which the test would be run. We conducted some performance tests to determine the additional overhead introduced by the Invite framework.

6.1 Test Setup

For our performance testing, we instrumented Jetty WebServer 6.1 [3], running on Java 1.5.0 on a Linux RedHat 2.6.9 server with four 3.2 GHz CPUs and 1 GB of memory. Only minimal background system processes were executing during our tests. To place load on the web server, we used The Grinder [7] installed on a Microsoft Windows XP system with a single 3 GHz processor and 1 GB of memory. The server and the client machines were connected over our department's gigabit LAN.

6.2 Baseline Testing

We first tested Jetty in our configuration without the in vivo testing framework attached, to determine a baseline. The test consisted of 10,000 requests for a JSP page of 20 kilobytes, which is approximately the average size of an HTML page [26]; the page was dynamically generated to avoid any caching by Jetty. The mean time for complete page requests was 6.35ms, the mean time to receive the first byte of the response was 3.59ms, and the throughput of HTTP response bytes was 3910kBps.

We then instrumented one Java class in Jetty (HttpConnection), but for each method set the probability of running a test to 0. In this case, we could measure the overhead of the instrumentation itself from the inserted AspectJ code, which still has to check that probability on each method call, since the instrumentation of the code is done at compile-time but the configuration is checked at run-time. In this case, though, we did not need to consider the forking of new processes or parallel execution of any test code, since Invite would never execute any tests. This time, the mean time for page requests was 6.43ms (1.2% increase), the mean time to receive the first byte was 3.62ms (0.1% increase), and the throughput was 3800kBps (2.8% decrease), which indicated very little impact overall and is consistent with the small overhead caused by calls to weaved-in AspectJ code [23].

6.3 Performance Impact of In Vivo Testing

Next we configured Invite so that the probability p of running a unit test was set to 1% for one method that is called exactly once per page request; all other methods still had p set to 0. This meant that only 1% of the page requests would invoke a unit test, or that each page request had a 1% chance of causing a test to be run. Using the same test setup as above, we saw that the mean time to complete a page request was 6.65ms (4.7% increase), though the overall throughput stayed the same at 3800kBps; this performace degradation includes the overhead from the framework itself, as well. The most telling statistic was the mean time to the first byte, which rose to 3.89ms (8.3% increase) compared to the baseline. The reason for the increase is that the class we instrumented is used before any bytes are sent back to the client, so any unit tests would be forked into new processes and launched during that time, hence the initial overhead.

Table 2: Load tests with pages of 20kB

Percent of page requests that execute tests	Mean time to serve page (ms)	%diff	Throughput of response data (kBps)	Avg time to start sending response (ms)	% diff
Baseline	6.35	-	3910	3.59	-
0%	6.43	1.2	3800	3.62	0.1
1%	6.65	4.7	3800	3.89	8.3
10%	7.98	25.6	3030	5.04	40.3
100%	13.6	114	1810	10.4	189

We then configured Invite to execute a unit test on 10% of the page requests. In this case, the mean time to complete a page request rose to 7.98ms (25.6% increase), the mean time to the first byte increased to 5.04ms (40.3% increase), and the throughput was 3030kBps (22.5% decrease). We also ran a test in which 100% of the page requests launched one unit test. As shown in Table 2, the differences in the mean times to complete a page request are mirrored in the mean times to the first byte.

Despite the large overhead incurred by running tests very frequently, we note that there is less than 5% overhead when running unit tests on 1% of the page requests. We contend that 1% is probably sufficient for detecting defects on a heavily-used application. In this particular test, setting p = 1% yielded a total of 100 tests in just over one minute of execution, which (assuming sustained traffic) would be well over 100,000 unit tests run per day.

6.4 Tests with Large Web Pages

We suspected that the process forking was the cause of much of the overhead in this implementation of the in vivo testing framework. However, we conjectured that the overhead of executing the tests is not related to the size of the web page being requested; it should be more or less constant, according to how fast the fork can be executed.

To demonstrate this, we conducted another test using a large (static) web page of 600kB. Although the pages could be cached, this did not affect the execution of unit tests, since the instrumented method was invoked before looking for the pages in the cache. As shown in Table 3, with no test instrumentation, the average time to serve a page request of this size was 61.8ms. This number rose only to 62.0ms (0.8% increase over baseline) if 1% of the page requests were resulting in a unit test being executed; 62.3ms (1.3% increase) when 10% resulted in unit tests; and only 64.2ms (4.4% increase) in the case where 100% of the page requests caused a unit test to run. This average overhead is still only a few milliseconds, as in the test with small dynamic pages, but is very small compared to the total time to serve the page.

Table 3: Load tests with pages of 600kB

Percent of page requests that execute tests	Mean time to serve page (ms)	%diff	Throughput of response data (kBps)	Avg time to start sending response (ms)	% diff
Baseline	61.5	-	9770	0.960	-
0%	61.8	0.4	9340	0.961	0.1
1%	62.0	0.8	9770	1.20	25.0
10%	62.3	1.3	9770	1.49	55.2
100%	64.2	4.4	9340	3.22	235

6.5 Areas for Performance Improvement

We have investigated ways to reduce the overhead by distributing the testing load across multiple instances of the application under test. In our initial findings [15], we discovered that it is possible to share the testing load across a small "application community" [30] in a software monoculture of only 450 instances of the application to reduce the overhead to just 1%, yet still achieve the same number of tests executed globally. Another solution may be to use a tool like the GAMMA system [39] [40] for distributing the tests and determining which tests should be run under different circumstances, such as system load.

7. LIMITATIONS AND FUTURE WORK

The most critical limitation of the current Invite framework implementation is that anything external to the application process itself, *e.g.* files, database tables, network I/O, *etc.*, is not replicated by forking the process and modifications of those made by a unit test may therefore affect the external state of the original application. As described previously, we do allow for the execution of a "tearDown" method for programmatic cleanup, and are currently integrating Invite with DejaView [25], though DejaView only provides a limited sandbox that addresses local file system issues and does not address any concerns related to external databases or network I/O. We will be addressing these limitations in future work.

Also, we have not yet finalized what action to take once a unit test fails and a defect is found. A simple approach would be to use something akin to an online crash reporting system like the Mozilla Quality Feedback Agent or Microsoft XP Error Reporting. An advantage of using the DejaView system for creating a virtual execution environment for the in vivo tests is that DejaView creates a "snapshot" of the process execution state and file system state, so that when a test fails, the snapshot could be sent back to the vendor, who could then try to reproduce, debug, and fix the problem. This could conceivably raise privacy and security issues, however. Another option would be to simply report failed unit tests to a central server, without any user state or environment information so as to avoid privacy issues, as we explored in the distributed in vivo testing approach [15].

Currently the Invite framework has only been implemented

in Java and is designed to work with Java applications. Porting it to C or C++ could present a challenge because the framework uses Java Reflection techniques to discover and execute the unit test methods (though it could conceivably be easily implemented with aspect-oriented programming and reflection in other managed languages like C#). Additionally, it may not always be desirable or even possible to recompile the target source code, as made necessary by our use of aspect-oriented programming. An approach to dynamically instrumenting the compiled code, such as in Kheiron [21] [22], could be used instead.

To date we have not made efforts to determine the *adequacy* [52] of our testing approach, for instance by measuring path/statement coverage or percentage of defects reliably found, and establishing success criteria. Further work could also more precisely categorize the prospective defects that could be found.

Future work could also investigate which classes to instrument, the percentage of method calls that should launch unit tests, or the optimal timing for when tests should be run, since the current framework only uses a percentage value to choose when to execute tests, based on each method call of the instrumented classes. This would vary greatly depending on the type of application and the defects that are being targeted, however. A further enhancement could consider the automatic selection of test cases at the time of execution, depending on the current system state and load.

8. CONCLUSION

We have presented *in vivo testing*, a novel testing approach that supports the execution of unit tests within a running application in the deployment environment, without affecting that application's state. We have classified the types of defects that could be found by our approach, and described a Java implementation of the Invite framework. Through our initial findings and investigation, we have presented some real-world examples of bugs that could be detected, and shown the usefulness of the approach with a case study of a real-world system. Additionally, we 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 [11], and we expect that in vivo testing will provide a foundation for future work in this field.

9. ACKNOWLEDGMENTS

Murphy and Kaiser are members of the Programming Systems Lab, funded in part by NSF CNS-0717544, CNS-0627473, CNS-0426623 and EIA-0202063, and NIH 1 U54 CA121852-01A1.

10. REFERENCES

[1] Apache Tomcat. http://tomcat.apache.org/.

[2] AspectJ. http://www.eclipse.org/aspectj/.

[3] Jetty WebServer. http://www.mortbay.org/.

[4] JUnit. http://www.junit.org/.

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

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

[7] The Grinder. http://grinder.sourceforge.net/.

[8] C. Atkinson and H.-G. Gross. Built-in contract testing in model-driven, component-based development. In *Proc. of ICSR Workshop on Component-Based Development Processes*, 2002.

[9] G. Baah, A. Gray, and M.J. Harrold. On-line anomaly detection of deployed software: a statistical machine learning approach. In *Proc. of the 3rd International Workshop on Software Quality Assurance*, pages 70–77, 2006.

[10] F. Barbier and N. Belloir. Component behavior prediction and monitoring through built-in test. In *Proc. of the 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 17–12, April 2003.

[11] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *Proc. of ICSE Future of Software Engineering (FOSE)*, pages 85–103, May 2007.

[12] S. Beydeda. Research in testing COTS components - built-in testing approaches. In *Proc. of the 3rd ACS/IEEE International Conference on Computer Systems and Applications*, 2005.

[13] S. Beydeda and V. Gruhn. The self-testing cots components (STECC) strategy - a new form of improving component testability. In *Proc. of the Seventh IASTED International Conference on Software Engineering and Applications*, pages 222–227, 2003.

[14] D. Brenner and C. Atkinson et al. Reducing verification effort in component-based software engineering through built-in testing. *Information System Frontiers*, 9:151–162, 2007.

[15] 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.

[16] J. Clause and A. Orso. A technique for enabling and supporting debugging of field failures. In *Proc. of the 29th ICSE*, pages 261–270, 2007.

[17] G. Denaro, L. Mariani, and M. Pezz'e. Self-test components for highly reconfigurable systems. In *Proc. of the International Workshop on Test and Analysis of Component-Based Systems (TACoS'03), vol. ENTCS 82(6)*, April 2003.

[18] D. Deveaux, P. Frison, and J.-M. Jezequel. Increase software trustability with self-testable classes in Java. In *Proc. of the 2001 Australian Software Engineering Conference*, pages 3–11, August 2001.

[19] 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.

[20] R. Fairley. Static analysis and dynamic testing of computer software. *Computer*, 11(4):14–23, April 1978.

[21] R. Griffith and G. Kaiser. Adding self-healing capabilities to the common language runtime. Technical Report CUCS-005-05, Columbia University, Dept. of Computer Science, January 2005.

[22] R. Griffith and G. Kaiser. A runtime adaptation framework for native C and bytecode applications. In *3rd IEEE International Conference on Autonomic*

Computing, pages 93–103, June 2006.

[23] E. Hilsdale and J. Hugunin. Advice weaving in aspectj. In *Proceedings of the 3rd international conference on aspect-oriented software development (AOSD)*, pages 26–35, 2004.

[24] 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.

[25] 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.

[26] S. Lawrence and C.L. Giles. Accessibility of information on the web. *Intelligence*, 11(1):32–39, Spring 2000.

[27] 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 (RAMSS '04)*, pages 57–62, May 2004.

[28] C. Liu and D.J. Richardson. RAIC: Architecting dependable systems through redundancy and just-in-time testing. In *ICSE Workshop on Architecting Dependable Systems (WADS)*, 2002.

[29] C. Liu and D. Richardson. Software components with retrospectors. In *Proc. of International Workshop on the Role of Software Architecture in Testing and Analysis*, pages 63–68, June 1998.

[30] M. Locasto, S. Sidiroglou, and A.D. Keromytis. Software self-healing using collaborative application communities. In *Proc. of the Internet Society (ISOC) Symposium on Network and Distributed Systems Security (NDSS 2006)*, pages 95–106, February 2006.

[31] C. Mao. AOP-based testability improvement for component-based software. In *31st Annual International COMPSAC, vol. 2*, pages 547–552, July 2007.

[32] C. Mao, Y. Lu, and J. Zhang. Regression testing for component-based software via built-in test design. In *Proc. of the 2007 ACM Symposium on Applied Computing*, pages 1416–1421, 2007.

[33] L. Mariani, M. Pezz'e, and D. Willmor. Generation of integration tests for self-testing components. In *Proc. of FORTE 2004 Workshops, Lecture Notes in Computer Science, Vol.3236*, pages 337–350, 2004.

[34] A. Memon and A. Porter et al. Skoll: distributed continuous quality assurance. In *Proc. of the 26th ICSE*, pages 459–468, May 2004.

[35] M. Merdes et al. Ubiquitous RATs: how resource-aware run-time tests can improve ubiquitous software systems. In *Proc. of the 6th International Workshop on Software Engineering and Middleware*, pages 55–62, 2006.

[36] M. Momotko and L. Zalewska. Component+ built-in testing: A technology for testing software components. *Foundations of Computing and Decision Sciences*, 29(1-2):133–148, 2004.

[37] L. Naslavsky and R.S. Silva Filho et al. Distributed expectation-driven residual testing. In *Second International Workshop on Remote Analysis and Measurement of Software Systems (RAMSS'04)*, 2004.

[38] L. Naslavsky et al. Multiply-deployed residual testing at the object level. In *Proc. of IASTED International Conference on Software Engineering (SE2004)*, 2004.

[39] 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 Conference*, pages 128–137, 2003.

[40] A. Orso, D. Liang, and M.J. Harrold. Gamma system: Continuous evolution of software after deployment. In *Proc. of ISSTA 2002*, pages 65–69, 2002.

[41] L. Osterweil. Perpetually testing software. In *The Ninth International Software Quality Week (QW'96)*, May 1996.

[42] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *Proc. of the 21st ICSE*, pages 277–284, May 1999.

[43] I. Pavlova, M. Åkerholm, and J. Fredriksson. Application of built-in-testing in component-based embedded systems. In *Proc. of the 2006 ISSTA Workshop on the Role of Software Architecture for Testing and Analysis*, pages 51–52, 2006.

[44] D. Richardson, L. Clarke, L. Osterweil, and M. Young. Perpetual testing project. http://www.ics.uci.edu/~djr/edcs/PerpTest.html.

[45] D. Rubenstein, L. Osterweil, and S. Zilberstein. An anytime approach to analyzing software systems. In *Proc. of the 10th International FLAIRS Conference (Florida Artificial Intelligence Research Society)*, pages 386–391, May 1997.

[46] D. Saff and M.D. Ernst. Reducing wasted development time via continuous testing. In *Proc. of the 14th International Symposium on Software Reliability Engineering*, page 281, 2003.

[47] D. Saff and M.D. Ernst. An experimental evaluation of continuous testing during development. In *Proc. of ISSTA 2004*, pages 76–85, 2004.

[48] StackSafe, Inc. IT Operations Research Report: Testing Maturity. Technical report, 2008.

[49] C. Turner and D. Robson. State based testing and inheritance. Technical Report TR-1/93, Durham, GB, 1993.

[50] J. Vincent, G. King, P. Lay, and J. Kinghorn. Principles of built-in-test for run-time-testability in component-based software systems. *Software Quality Journal*, 10(2), September 2002.

[51] Y. Wang et al. On built-in test reuse in object-oriented framework design. *ACM Computing Surveys*, 32(1), March 2000.

[52] E. Weyuker. Axiomatizing software test data adequacy. *IEEE Trans. Software Eng., SE-12*, pages 1128–1138, December 1986.

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

[54] M. Young. Perpetual testing. Technical Report AFRL-IF-RS-TR-2003-32, Univ. of Oregon, February 2003.