Quality Assurance of Software Applications using the In Vivo Testing Approach

Christian Murphy, Gail Kaiser, Ian Vo, Matt Chu
Department of Computer Science, Columbia University, New York NY 10027
{cmurphy, kaiser, idv2101, mwc2110}@cs.columbia.edu

Abstract

Software products released into the field typically have some number of residual defects that either were not detected or could not have been detected during testing. This may be the result of flaws in the test cases themselves, incorrect assumptions made during the creation of test cases, or the infeasibility of testing the sheer number of possible configurations for a complex system; these defects may also be due to application states that were not considered during lab testing, or corrupted states that could arise due to a security violation. One approach to this problem is to continue to test these applications even after deployment, in hopes of finding any remaining flaws. In this paper, we present a testing methodology we call in vivo testing*, in which tests are continuously executed in the deployment environment. We also describe a type of test we call* in vivo tests *that are specifically designed for use with such an approach: these tests execute within the current state of the program (rather than by creating a clean slate) without affecting or altering that state from the perspective of the end-user. We discuss the approach and the prototype testing framework for Java applications called* Invite*. We also provide the results of case studies that demonstrate Invite's effectiveness and efficiency.*

1. Introduction

Thorough testing of a software product is unquestionably a crucial part of the development process, but the ability to faithfully detect all defects in an application is severely hampered by numerous factors. A recent report [47] indicates that 40% of IT companies consider insufficient prerelease testing to be a major cause of later production problems, and the problem only worsens as changes are rolled out into production without being thoroughly tested. Furthermore, it is possible that the test code itself may have flaws in it, too, perhaps because of oversights or incorrect assumptions made by the authors.

A key issue is that, for large, complex software systems, it is typically impossible in terms of time and cost to reliably test all configuration options before releasing the product into the field. For instance, Microsoft Internet Explorer has over 19 trillion possible combinations of configuration settings [14]. Even given infinite time and resources to test an application and all its configurations, once a product is released, the other software packages on which it depends (libraries, virtual machines, *etc.*) may also be updated; therefore, it would be impossible to test with these dependencies prior to the application's release, because they did not exist yet. A last emerging issue is the fact that, as multiprocessor and multi-core systems become more and more prevalent, multi-threaded applications that had only been tested on single-processor/core machines are more likely to start to reveal concurrency bugs [30].

One proposed way of addressing this problem has been to continue testing the application in the field, after it has been deployed. The theory of this "perpetual testing" [42] approach is that, over time, defects will reveal themselves given that multiple instances of the same application may be run globally with different configurations, in different environments, under different patterns of usage, and in different system states.

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

Our three main contributions are an approach (in vivo testing) to executing tests within the deployment environment, without altering that system's state; a new style of

tests (in vivo tests) that exercise parts of the application as the system is running, no matter what its current state; and a prototype implementation of the testing framework, called *Invite*, developed in Java. In [11], we briefly sketched an earlier version of the Invite framework, focused on distributed execution of the tests; in this work, we present for the first time the complete system in full, including a more detailed description of in vivo tests, case studies in which the approach reveals defects in real-world applications, and evidence that the performance overhead of the approach is reasonable and yet a single application instance can still execute millions of tests per day.

2. The In Vivo Testing Approach

The foundation of the in vivo testing approach is the fact that many (if not all) software products are released into deployment environments with latent defects still residing in them, as well as our claim that these defects may reveal themselves when the application executes in states that were unanticipated and/or untested in the development environment. The in vivo testing approach can be used to detect defects hidden by assumptions of a clean state in the tests, errors that occur in field configurations not tested before deployment, and problems caused by unexpected user actions that put the system in an unanticipated state; these flaws may also be due to corrupted states that could arise due to a security violation. Our approach goes beyond passive application monitoring (*e.g.* [40]) in that it actively tests the application as it runs in the field.

In vivo testing is a methodology by which tests are executed continuously in the deployment environment, in the context of the running application, as opposed to a controlled or blank-slate environment. Crucial to the approach is the notion that the test must not alter the state of the application from the user's perspective. In a live system in the deployment environment, it is clearly undesirable to have a test alter 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 section motivates and describes the in vivo testing approach.

2.1. In Vivo Tests

Although existing unit and integration tests can be used with in vivo testing without any modifications (for instance, to address configurations or environments not tested prior to release, as in [34]), developers may find it desirable to create in vivo tests that are able to take advantage of the approach. These tests ensure that properties of the application (or of subsystems or even units) hold true no matter what the application's state is. In the simplest case, they can be thought of as program invariants and assertions [12], though they go beyond checking the values of individual variables or how variables relate to each other, and focus more on the conditions that must hold after sequences of variable modifications and method calls, without worrying about side effects visible to the user.

A simple example is that of the functionality of an implementation of the Set interface (such as a Vector or ArrayList) in Java. One of its properties is that, if an object is added to the Set and then removed, a subsequent call to the "contains" method must return false. This condition must hold no matter what the state of the Set, and no matter what sort of object had been added. A traditional unit test may investigate this property by first creating a new, empty Set, but it would not be possible to conduct such a unit test on arbitrary states of the Set, after it has been used in a real, running application for some amount of time. Thus, an in vivo test would be useful in this case.

A more complex example can be found in Mozilla Firefox. One of the known defects is that attempting to close all other tabs from the shortcut menu of the current tab may fail on Mac OS X when there are more than 20 tabs open.[1] In this case, an in vivo 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.

It is important to note that in vivo tests are not intended to replace unit or integration tests; rather, we introduce a new type of test designed to run within the context of an executing application, which may be in a previously untested or unanticipated state. As in vivo tests are individual methods run inside the application, our approach is like unit testing in the sense of calling individual methods with specified parameters, but it is also like integration testing in that we use the integrated code of the whole application rather than stubs and drivers.

2.2. Conditions

In order for in vivo testing to be useful in practice, for a given test and a corresponding piece of software to be tested, three conditions must be met. First, the test must pass in the development environment, even though there are unknown defects in the software under test (if the 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,

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

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 not tested in the lab
3	Deployment environments not simulated in the lab
4	A user action puts the system in an unexpected state
5	Those that only appear intermittently

i.e. have a bug. Lastly, for some process state or condition of use, the 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 test code, or both.

2.3. Categories and Motivating Examples

To examine the feasibility of our testing approach, we investigated the documented defects (mostly caught by end-users after deployment) of some open-source applications to see which of them could have been discovered using in vivo testing. We considered OSCache [41], a multi-level caching solution designed for use with JSP pages and Servlet-generated web content, as well as another caching solution, Apache Java Caching System (JCS) [1], and Apache Tomcat [2], 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 types of defects 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 [24]. In these cases, the code may pass unit tests coincidentally, but not work properly once executed in the field, revealing defects in both the test code and the code itself. State-based testing [48] or static analysis [19] 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.

One of the OSCache bugs 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.[2] In this case, the corresponding in vivo test for testing cache removal may simply add something to the cache, remove it, and then check that it is no longer there; this sequence of operations should work consistently regardless of the state of the cache. A unit test that assumes an empty or new cache would pass, however; but when the cache is full, the in vivo test would fail, revealing a defect that may not have been caught until it affected a user.

Another example of this type of defect can be found in Apache JCS. Here, the method that returns the number of elements in the cache is off by two when the cache is at full capacity.[3] A unit test that simply creates a new cache, adds some number of elements, and checks the size may pass in the development environment if the number of added elements is smaller than the capacity. But an in vivo test that is executed in the field would detect this defect when it tries to add those elements and thus meets the cache's capacity.

The second category targeted by our approach includes **those defects that come about from field configurations that were not tested in the development environment**. For instance, Apache Tomcat has over 60 different parameters that can be configured, many of which allow for free-text input or unbounded integer values, meaning the entire potential configuration space is huge. We note that a testing approach using a system like Skoll [25] [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.

OSCache has around 20 configurable parameters, and one of its bugs falls into this category, too. 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.[4] 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 unit test would pass. However, if a system administrator sets the capacity to a large number in the properties file, an in vivo test would fail when it tries to set the cache capacity to a smaller value, revealing the defect.

The third category of defects concerns **those that come about from deployment environments that were not simulated in the lab prior to release**. Java applications may require testing on multiple platforms with multiple JDK versions and multiple revisions of the application code, possibly with multiple third-party libraries or application servers; this is not always feasible for testing in a single test lab. Additionally, a new JDK, OS, or library may be released after the software is deployed, making testing prior to deployment impossible. For instance, in OSCache certain func-

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

[3] http://issues.apache.org/jira/browse/JCS-16

[4] http://jira.opensymphony.com/browse/CACHE-158

tionality works fine with Solaris 8 but not Solaris 9, which was released after the version of OSCache in question.[5] By extending testing into the various deployment environments, in vivo testing would detect such defects.

The fourth type of defects targeted by in vivo testing are **ones that stem from a user action that puts the system in an unexpected state** that would not have been tested. These actions may be legal ones that were simply unanticipated, or illegal actions, *e.g.* a security violation. This could also 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.

For example, in OSCache, an uncaught NullPointerException would appear only after a particular sequence of operations that involves attempting to flush cache groups that do not exist.[6] In this case, an in vivo test that checks the operation of the flush method would detect this invalid state because the test would fail, even though that test would succeed in normal "expected" states.

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

Concurrency bugs are a very common type of defect in this category. We noticed one of the concurrency bugs 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 uncaught exception would occur because one of the objects being used in the session creation could be set to null by the invalidator.[7] A unit test that is simply testing the creation of sessions is not likely to detect this defect 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.

We found at least ten such examples in the listing of known OSCache defects. For instance, in one of them, 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.[8] 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.

Note that, in all these cases, in vivo testing helps find defects in poorly designed unit tests as much as it does in the applications themselves. Software testers may not anticipate these types of defects when they write their tests, but we hope that by using in vivo testing, they will consider a different approach that allows them to test functionality of the application, regardless of its state or environment.

Also, it is conceivable that the defects 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 tests on an application in the field greatly improves the chances of the errors being detected before affecting an end-user. More importantly, certain defects will in practice only manifest themselves in the field (because of limited time and resources in the testing lab, or because they are heavily dependent on the state), and these are the ones for which in vivo testing is most useful.

2.4. 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 its 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 a corresponding 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 test runs to completion in a separate process, and the results of the test are logged. Note that the 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 those that produce calculations or results that may not be obviously wrong, and do not otherwise make the error obvious, such as crashing. For instance, in most of the caching examples above, the user would not notice that the cache is acting incorrectly, as the data would still be usable and may appear to be correct; however, in those examples the caches are not working correctly and/or as efficiently as they should. Applications that include machine learning may also benefit from in vivo testing because

[5] http://jira.opensymphony.com/browse/CACHE-193
[6] http://jira.opensymphony.com/browse/CACHE-173
[7] http://issues.apache.org/bugzilla/show_bug.cgi?id=42803
[8] http://jira.opensymphony.com/browse/CACHE-175

the user may not know whether the calculations are obviously incorrect, but defects in the implementation could cause slightly erroneous results. Systems that have complex states that perhaps could not be anticipated in advance are other good candidates for in vivo testing, which is designed to execute tests in such situations. We are also considering the approach for use with "non-testable programs" [52] that have no test oracle: although it may not be possible to know overall results in advance, the application may have certain properties that should hold true regardless of its state, and these are the exact aspects of a program that in vivo testing is designed to exercise.

3. Related Work

While the notion of "self-checking software" is by no means new [53], our work is principally inspired by the idea of "perpetual testing" [42] [44] [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 these 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 tests are executed from within the context of the running application and do not alter the application state from the user's perspective.

In vivo testing is also a form of "residual testing" [43]. 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 on measuring the coverage provided by this approach by looking at untested residue [38] [43] or by comparing the coverage to specifications [37]. However, this work does not consider the actual execution of 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.

The Skoll project [25] [34] has extended the idea of "continuous" [45], round-the-clock testing 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 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 executes 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.

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 [28] to record testing and execution history and make the information available to a software tester, and "just-in-time testing" [29] to check component compatibility with client software. Work in "built-in-testing" [50] has included investigation of how to make components testable [8] [9] [10] [33], and frameworks for executing the tests [15] [32] [35], including those in Java programs [16], 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 test code, rather than requiring extensive modification to the original source to provide special functional and testing interfaces [4] [49] or enforcing a rearchitecture of the application to allow for the use of testers and controllers/handlers [6] [36] [49]. 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 [17]. One of these, the GAMMA system [40], uses software tomography for dividing monitoring tasks and reassembling gathered information. 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

[13] has looked at methods of recording, reproducing, and minimizing failures to enable and support in-house debugging, and Baah [5] uses machine learning approaches to detect anomalies in deployed software. All of these strategies could take advantage of in vivo testing as part of their implementation.

Lastly, in vivo tests themselves can be considered "extended assertions" or "extended program invariants". Others have looked at the automatic detection of invariants, *e.g.* DIDUCE [21] and Daikon [18], and of checking them at runtime [46]. However, those approaches will only perform runtime checks that have no side effects, whereas in vivo tests necessarily allow for side effects, but they are hidden from the end user.

4. The In Vivo Testing Framework

The prototype in vivo testing framework, which we call Invite (IN VIvo TEsting framework), has been implemented for Java applications and has been designed to reuse existing test code and to allow for the creation of in vivo tests, while not imposing any 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 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. It is important to note that these steps do not require any modification or special constraints on the design of the software application itself; the development of any new test code and the configuration of the framework 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.

Step 1. Create test code. If unit and integration tests already exist, it is certainly possible to use the Invite framework without writing a single line of new code. By shipping these tests with the application and then running them in vivo as the application executes in the field, it is clear that defects that appear infrequently are much more likely to be revealed purely by increasing the number of times the tests are executed. Furthermore, it is also clear that this approach will help find defects that only appear in certain configurations or environments, since the tests will run in a broad variety of settings, as in [34]. Thus, one can take advantage of in vivo testing even without writing any new code.

To get the most out of in vivo testing, however, application developers should create in vivo tests as described in Section 2. These tests are designed to check properties of the application that should hold true regardless of its state, and these are most likely to reveal defects that were not found (or could not have been found) in the development environment.

To create in vivo tests, the software vendor must ensure that the test methods reside in the same class as the code they are testing (or in a superclass). Also, the in vivo 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). The parameters to "testFoo" should be the same as those to the original method "foo", so that the actual arguments can be used when testing. Additionally, rather than create new objects to test in the in vivo 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 whose method invocation triggered the testing, which has been modified over the course of the application's execution.

Figure 1 shows a simple in vivo test that could be used in a Set implementation. Upon invocation of the "add" method with an Object parameter, for instance, "testAdd" is called and the argument is passed to it as testObj. Because this test method resides in the same class that defines the "add", "remove", and "contains" methods, it uses the object reference "this" to call methods on itself.

```
public boolean testAdd(Object testObj) {
   this.add(testObj);
   this.remove(testObj);
   return (this.contains(testObj) == false);
}
```

Figure 1. Example of in vivo test

We are currently investigating mechanisms for automatically detecting application properties that could be used in the creation of in vivo tests, and of course developers may have their own notion of what properties of the application should hold, regardless of its state.

Another approach to creating in vivo tests is to build upon already-created unit tests, modified so that they use existing objects, rather than creating new ones. Figure 2 shows such a unit test in the JUnit [23] style. It is clear that there are only small changes required to convert this into the in vivo test in Figure 1: (1) the test method has been moved into the same class as the one it is testing; (2) the name of the test method has been changed to match that of the method it is testing; (3) the parameter to the test method matches that of the original method, and the parameter is used in the testing; (4) the return type of the test method has been changed, and a return statement is used instead of an assert; and (5) the reference to the object being tested (in this case, the Set) in the test method is now "this" instead

of a newly-created object. Future work could look into the potential of automating this conversion.

```
private Set set;
@Before public void setUp() {
   set = new SetImpl();
}
@Test public void testAddRemoveContains() {
   Object testObj = new Object();
   set.add(testObj);
   set.remove(testObj);
   assert(set.contains(testObj) == false);
}
```

Figure 2. Example of JUnit test

As noted above, the modification of unit tests into the style of in vivo tests is *not* strictly a requirement for using the Invite framework. Existing unit and integration tests can be used without any modifications whatsoever, and the different types of tests are not mutually exclusive. However, our intention is to demonstrate that it is possible to create in vivo tests only with small changes to existing unit tests.

Step 2. Instrument classes. In the next step, the vendor must then select the methods in one or more Java classes in the application under test for instrumentation. Aside from acting as jumping off points for the tests, the instrumented methods 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 methods, of course). The list is specified in an XML file. To achieve this instrumentation, a component written in the aspect-oriented programming language AspectJ [3] 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 [20], which would dynamically insert the test harness code into the application after it is compiled.

Step 3. Configure framework. Before deployment, the vendor would then configure Invite with values representing, for each method with a test in the instrumented classes, the probability ρ with which that method's test(s) will be run. This configuration is specified in an XML file, which for each test specifies 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 tests (if a method is associated with multiple tests, these are all specified separately). 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 ρ 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 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 Invite framework uses the percentage value ρ 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, Invite remembers the results of previous checks to see if the test method exists). This is the in vivo test that will then be executed. The purpose of running a method's corresponding test method is so that the test is executed at the same point in the program (the same state) as the method itself, 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 a 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 in vivo 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 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 test runs in the other process. Note that the application and the in vivo 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, test modifications to network I/O, the operating system, external databases, *etc*. are not automatically undone; the sandbox only includes the in-process memory of the application

(through the copy-on-write forking). To address this limitation, we are currently integrating Invite with DejaView [26], an application which creates a virtual execution environment that isolates the process running the unit test and gives it its own isolated view of the system. Furthermore, if a "tearDown" method exists in the class in which the 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).

When the test is completed, Invite logs whether or not it passed, and the process in which the 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 corresponding test(s). We describe in [11] the mechanism by which all errors are reported back to a central server (presumably at the vendor's location), and could then be processed as in [40], wherein configuration parameters (like the frequency of test execution or even the list of methods to test) could then be modified and sent back to the application instance.

Unlike other testing approaches that test the application as it is running, such as [16] or [36], Invite avoids the "Heisenberg problem" of having the test alter the state of the application it is testing. This is one of the major contributions and differentiating characteristics of the in vivo testing approach.

4.3. 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. The relative effects of these different policies, however, are outside the scope of this paper. 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. Another solution may be to use a tool like the GAMMA system [39] [40] for determining which tests should be run under different circumstances, such as system load.

4.4. 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 in vivo 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 ρ values. From this data, it is then possible to estimate how altering the value of ρ will affect the system's performance and number of tests executed.

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

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 ρ has on the number of tests being run per unit time, one can then calculate the additional overall time cost of increasing or decreasing ρ. We surmise that, in practice, the ρ values would presumably be very small (less than 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 Studies

Given the motivating examples listed in Section 2, we sought to apply Invite to some of those applications to demonstrate that in vivo testing would have quickly detected those defects, even assuming the presence of sufficient unit tests that could be used in the development environment.

We first investigated OSCache 2.1.1, which contained three of the known defects listed in Section 2. Unfortunately the unit tests that are distributed with that version of OSCache do not cover the methods in which those defects are found, so we asked a student (who was not aware of the goals of this work) to create unit tests that would reasonably exercise those parts of the application. As expected, those tests passed in the development environment during traditional unit testing, primarily because the student had created the tests assuming a clean state which he could control. This took a total of two hours.

We then asked the same student to develop 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. Although we did not have a real-world application based on OSCache for our testing, we created a driver that used the OSCache API to randomly add, retrieve, and remove elements of random size from a cache, and randomly flushed the cache. All three defects were revealed by Invite in less than two hours. The last to reveal itself was the one that only happened when the cache was at full capacity, which happened rarely in our test because the random adding, removing, and flushing did not allow it to reach capacity often; however, this defect may have revealed itself more quickly in a real-world application.

A similar experiment was conducted with Apache JCS version 1.3. Here we were looking for a defect that only appeared when the cache was at full capacity, and this defect was revealed in less than one hour, but again may have appeared sooner in the real world.

Although these defects were discovered in our own testing environment (as opposed to a deployment environment), these examples demonstrate that certain intermittent defects or those that only are revealed under certain circumstances may not be revealed in traditional unit testing, but would be detected with in vivo testing. More importantly, these case studies 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. This section describes some of the design considerations to address performance, and the results of some tests we conducted to determine the additional overhead introduced by the Invite framework.

6.1. Addressing Performance Concerns

The first and perhaps most obvious measure we took is to allow the administrator to limit the number of simultaneous tests that are being executed, so that test processes are not created so frequently as to flood the CPU. When the maximum number of test processes are executing, the Invite framework is temporarily disabled so that no more tests are started. This gives the administrator a mechanism for keeping the number of processes under control.

The maximum allowed number of simultaneous test processes would ideally be less than or equal to the number of CPUs/cores in the machine. To take advantage of multiprocessor/multicore architectures, it is possible to configure Invite so that each process runs independently and does not interrupt the others. Each process is assigned to a separate CPU/core using an affinity setting (this is not supported in Java but is possible through a JNI call), thus ensuring that the tests do not run on the same CPU/core as the main process and limiting the overall impact on the application. For instance, on a quad-core machine, one core could be executing the application, allowing for up to three simultaneous tests, each on a separate core, so that none of them would pre-empt the original application process.

We have also investigated ways to reduce the overhead by distributing the testing load across multiple instances of the application under test, as described in [11]. However, here we only discuss the case in which a single instance performs the tests.

6.2. Test Setup

We measured the performance impact during our testing of OSCache 2.1.1, using Java 1.6.0 on a Linux Ubuntu 2.7.1 server with a dual-core 3.0GHz CPU and 1 GB of memory. Only minimal background system processes were executing during our tests.

We first executed the test in our environment without the in vivo testing framework attached, to determine a baseline. The test consisted of 100,000 random calls to add, retrieve, and remove items from a cache, as well as to flush the entire cache. The time to complete the benchmark with no Invite instrumentation was 1062ms.

We then instrumented the four appropriate methods in the GeneralCacheAdministrator class and created simple in vivo tests for each; we then 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 test completed in 1080ms (1.6% increase), which indicated very little impact overall and is consistent with the small overhead caused by calls to weaved-in AspectJ code [22].

6.3. Performance Impact

Next we configured Invite so that the probability ρ of running a test (for each of the four methods) was set to 0.1%. To demonstrate the effects of more frequent testing, we then repeated the tests with larger ρ values; the results are shown in Table 2. All tests ran on a different core from the original process.

Table 2. Results of performance testing

Percent of methods that execute tests (ρ)	Total time (ms)	%diff	Number of tests	Tests per second
Baseline	1062	-	-	-
0%	1080	1.6	0	0
0.1%	1115	4.9	39	34.9
1%	1140	7.3	51	44.7
10%	1276	20.1	58	45.5
100%	1299	26.6	72	55.4

Note that the number of tests executed does not increase by an order of magnitude just because the percent probability of running a test does: depending on how long the test takes to run, and the allowed maximum number of concurrent tests, only a certain number of tests could be fit in before the program runs to completion. It is possible that an in vivo test that takes a very long time to run will reduce the overall number of tests run, but it will also reduce the overhead (since new tests are not starting), and we expect that the ratio of overhead to number of tests run would stay about the same even in those cases.

We also ran similar experiments on a quad-core machine, in which we allowed for two and then three simultaneous tests to be run. The results were as expected: the performance overhead increased because more test processes were being forked, but the number of tests run during the experiment also increased with the number of allowable simultaneous tests. For instance, with ρ set to 100% and allowing for up to three simultaneous tests, we were able to achieve rates of over 200 tests per second, though at a higher performance cost (around 60%). See Table 3.

Table 3. Results of performance testing when allowing for three simultaneous tests

Percent of methods that execute tests (ρ)	Total time (ms)	%diff	Number of tests	Tests per second
Baseline	1062	-	-	-
0%	1080	1.6	0	0
0.1%	1411	32.8	81	57.4
1%	1573	48.1	312	198.3
10%	1628	53.2	368	226.0
100%	1700	60.0	412	242.3

Despite the large overhead incurred by running numerous tests very frequently, the results indicate that incurring an overhead of just 5% still achieves over three million tests per day for a single application instance. 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.

7. Limitations and Future Work

The most critical limitation of the current Invite implementation is that anything external to the application process itself, *e.g.* database tables, network I/O, *etc*., is not replicated by forking the process and modifications of those made by an in vivo test may therefore affect the external state of the original application. As described previously, we are currently looking into integrating Invite with DejaView [26], 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 hope to address these limitations soon.

Also, we have not yet finalized what action to take once a test fails and a defect is found. We currently have an option to report failed in vivo tests to a central server, as we previously discussed in [11]. Another approach would be to create 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.

Currently the Invite framework has only been designed to work with Java applications, but we are now in the process of developing an implementation for C. Additional issues may come up with using a language that does not use managed code (*e.g.* our implementation uses Java Reflection to determine the test method names), but other types of defects may be revealed, particularly those related to the state of the environment; most of our work so far has only focused on issues related to the state of the application. 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 [20], could be used instead.

To date we have not made efforts to determine the *adequacy* [51] 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, or the types of applications for which the approach is best suited.

Future work could also investigate which classes to instrument, the percentage of method calls that should launch unit tests, the optimal timing for when tests should be run, or how to test code that is not in the execution path, since the current framework only uses a percentage value to choose when to execute tests, based on actual invocations of the instrumented methods. 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 tests in the deployment environment, without affecting that application's state. We have also presented *in vivo tests*, which execute within a running application and test properties of the application that must hold regardless of the state the process is in. Last, we have described a Java implementation of our framework, called *Invite*. Through our initial findings, we have presented some real-world examples of defects that could be detected, and have demonstrated that our approach and the current implementation add limited overhead in terms of system performance and code modification.

Testing in the deployment environment has been identified as a future challenge for the software testing community [7], and we expect that in vivo testing will provide a foundation for future work in this field.

9. Acknowledgments

The authors would like to thank Lori Clarke, Lee Osterweil, Simha Sethumadhavan and Junfeng Yang for their suggestions and assistance. Murphy and Kaiser are members of the Programming Systems Lab, funded in part by NSF CNS-0717544, CNS-0627473 and CNS-0426623, and NIH 1 U54 CA121852-01A1.

References

[1] Apache Java Caching Solution (JCS). http://jakarta.apache.org/jcs/.

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

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

[4] 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.

[5] G. K. 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] D. Brenner and C. Atkinson et al. Reducing verification effort in component-based software engineering through built-in testing. *Information System Frontiers*, 9(2-3):151–162, 2007.

[11] 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.

[12] 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.

[13] 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.

[14] M. B. Cohen, J. Snyder, and G. Rothermel. Testing across configurations: implications for combinatorial testing. In *Proc of the Second Workshop on Advances in Model-based Software Testing*, pages 1–9, 2006.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

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

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

[25] 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.

[26] 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.

[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. 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.

[29] 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.

[30] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Proc of the 12th international conference on architectural support for programming languages and operating systems*, pages 37–48, 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 runtime 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] OSCache. http://www.opensymphony.com/oscache.

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

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

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

[45] D. Saff and M.D. Ernst. Reducing wasted development time via continuous testing. In *Proc. of ISSRE 2003*, page 281.

[46] 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.

[47] StackSafe, Inc. IT Operations Research Report: Testing Maturity, 2008.

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

[49] 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.

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

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

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

[53] 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.

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