# Towards In Vivo Testing of Software Applications

Christian Murphy, Gail Kaiser, Matt Chu
*Dept. 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. In this novel approach, unit tests execute within the current state of the program (rather than by creating a clean slate) without affecting or altering that state. Our approach has been shown to 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, the testing framework we have developed for Java applications, classes of bugs our approach can discover, and the results of experiments to measure the added overhead.*

## 1. Introduction

Thorough testing of a commercial 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. Furthermore, it is possible that the test code itself may have flaws in it, possibly 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 solution to this problem has been to continue testing the application in the field, even after it has been deployed. The theory 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 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. In this novel approach, tests execute within the current state of the program without affecting or altering that state. Here, we show that our 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).

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.

Although we only present our initial findings thus far, 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

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. Furthermore, bugs may exist in the unit tests themselves, not just in the application code. So a unit test that passes in development may not necessarily pass when executed after deployment, and the fact that a unit test passes does not mean that the piece of code is without flaw.

In vivo testing is an approach 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 arbitrary 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 caused by the test code rather than their own actions. This is ensured 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 is a defect 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, but should not crash or otherwise fail (since in vivo testing would not be needed to detect such gross failures). 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 will typically be one in the application code, or in the unit test code, or both.

## 2.2. Categories and motivating examples

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, OSCache[5] version 2.3, 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. 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. 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.

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. [1] 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, system administrators of such applications may have numerous runtime configuration options, and not all combinations may have been tested before release. This is especially true in the case of open source software, such as the applications we considered here.

Another OSCache bug 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. [2] 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

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

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

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.

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

| | |
|---|---|
| 1 | Corresponding unit test assumes a clean slate |
| 2 | Field configurations that were not tested in the lab |
| 3 | A legal user action that puts the system in an unexpected state |
| 4 | An unanticipated user action breaks the system |
| 5 | Those that only appear intermittently |

The third types of defects 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 are shared between users, and one user's activities modify that 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 in Apache Tomcat, in which a particular method used in the creation of a session is not threadsafe.[3] 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 (and not be caught) 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.

The fourth types of defects 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, since the unit tests call the methods at arbitrary times and the error may be detected before it affects any user. 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 we considered is one in which there is a resource leak in the database connection pool.[4] 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. 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 could conceivably reveal this bug.

The fifth and final type of defect is one that only appears intermittently. These defects may be discovered by a continuous testing approach during the development phase [41], 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, whereby flushing the cache, adding an item, and attempting to retrieve the item can occasionally result in an error.[5] A unit test that tries this sequence of actions may simply never encounter the error by chance during testing in the development environment. But by having instances of the application repeatedly execute this test in the in vivo testing approach, it may eventually appear.

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 can greatly improve 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.

## 3. Related work

Our work is principally inspired by the notion of "perpetual testing" [35, 40, 39, 48], 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

[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

product, in the development environment (the lab) as well as the deployment environment (the field). 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" [36]. 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 [36, 31] or by comparing the coverage to specifications [32]. However, this work does not consider the actual execution of unit tests in the deployment environment, as we describe here.

Also related to perpetual testing is "continuous testing", which refers to round-the-clock execution of tests, though typically in the development environment [41, 42]. However, the Skoll project [19, 28] 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, or performance testing, in vivo testing is different in that it seeks to execute unit tests within the application while it is running under normal operation.

While the notion of "self-checking software" is by no means new [47], 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 [22] to record testing and execution history and make the information available to a software tester, and "just-in-time testing" [21] to check component compatibility with client software. Work in "built-in-testing" [45] has included investigation of how to make components testable [11, 27, 10, 12], and frameworks for executing the tests [15, 29, 26], including those in embedded systems [37] and Java programs [16], or through the use of aspect-oriented programming [25]. 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 [7, 44] or enforcing a rearchitecture of the application [9, 30].

Other approaches to perpetual testing include the monitoring and profiling of deployed software, as surveyed in [17]. One of these, the GAMMA system [33, 34], uses software tomography for dividing monitoring tasks and reassembling gathered information; this can then be used for onsite modification of the code (for instance, by distributing a patch) to fix defects. Clause [13] has looked at methods of recording, reproducing, and minimizing failures to enable and support in-house debugging, and Baah [8] 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. Invite: The in vivo testing framework

The preliminary in vivo testing framework, which we call Invite (IN VIvo TEsting framework), is developed in Java and has been designed to entirely separate the testing code from that of the application under test. In order to use Invite, the software vendor must first ensure that the test classes follow the JUnit [4] conventions, specifically that the class has "setUp" and "tearDown" methods, and all test methods start with the word "test", take no arguments, and return void. These methods must all be public.

In order for the test classes to use objects that exist in the running application, it is necessary for the constructor or the "setUp" method to get references to existing objects, rather than creating new ones, as is typical in a JUnit test suite. This allows Invite to execute the unit tests in the context of the current state of the system by using the objects that have been modified over the course of the program's execution, rather than creating a blank slate. However, because the Invite code that invokes the JUnit test does not have references to the objects in the application, and the test methods do not take arguments, modification of the application may be necessary to provide arbitrary access to the objects that are necessary for testing.

One possible approach is to add singleton instance references so that the test code can statically get access to an object of that type (*i.e.*, the one being used in the application) without explicitly being passed a reference. This requires modification of the source code, though, unless singletons are already being used. Only a few extra lines need to be added to each class that must expose one object of that type as a singleton,

and the modification should not affect the normal execution of the system, since the reference to the singleton need not be used anywhere else.

However, this approach has limitations in the cases where multiple objects of the same class are used in the application (such as in a resource pool), and the test needs to somehow pick amongst them, or where objects are created using a factory class, and arbitrary object access may be impossible. It is important to note, however, that any source code modification 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.

The vendor must then select one or more Java classes in the application under test for instrumentation, such that all method calls into the class will be points at which a unit test could be run. To achieve this, 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 [18]. Note that the unit tests will not necessarily be those written for the instrumented classes; the instrumented classes merely provide "jumping-off points" where tests may be executed.

Lastly, the vendor would configure Invite with a list of JUnit test classes. Invite is also configured with the percentage of method calls in the instrumented classes on which to execute the unit tests. In practice, this number would presumably be very small, but is heavily dependent on the number of instrumented methods, the frequency with which they are called, and the desired amount of testing to be performed.

It is assumed that the application vendor would ship 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 at all, and ideally would not even notice that the in vivo tests were running; Section 6 explores the performance overhead caused by Invite.

At system startup, Invite uses Java Reflection to find all the "test" methods in the JUnit classes, and stores the names for later use. Whenever a method of an instrumented class is invoked, Invite uses the percentage value to decide whether to execute a test. If Invite decides that a test is to be run, it randomly chooses one method from amongst those in the configured JUnit test classes (alternatively, a planned schedule of tests could be implemented). It 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 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 native C program which executes the fork.

Once the test is invoked, the application can continue its normal execution, while the unit test runs in the other process. In the current implementation of Invite, unit test modifications to files, I/O, the operating system, *etc.* cannot be undone; the sandbox only includes the in-process memory of the application (this limitation is discussed in Section 7). When the unit test is completed, Invite logs whether or not it passed, and that process is terminated. The testing results currently would need to be manually inspected, but this could be automated, and errors could be reported back to a central server as in [33] or [28].

Unlike other testing approaches that test the application as it is running, such as [30] or [16], 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.

## 5. Empirical Study

After considering the numerous motivating examples listed in Section 2, we sought to apply Invite to a publicly-available application, in order to determine whether the approach would work to detect more defects.

The application we instrumented for testing was Jetty WebServer 6.1 [3], an open-source Java HTTP server that also supports the Java Servlet API. We chose it primarily because it provides unit tests in the JUnit style, which we could use for our in vivo testing.

Before conducting any tests using the in vivo testing approach, we selected 15 unit test classes from the ones that ship with the Jetty distribution, and executed all of them outside of the running program to ensure that the tests would pass under normal circumstances, *i.e.* outside of the in vivo testing framework.

In order to attach Invite, we then instrumented Jetty's HttpConnection class, which is used in every page request, so that every request had a chance of causing a unit test to be invoked. We configured Invite to use the 15 different unit test classes. Where appropriate, we needed to modify some of the tests to get references to certain objects in the running application, rather than creating new ones. Specifically, we added a reference to a singleton instance (three lines of code) to the Server class so that

unit tests could access the same object used in the running application. Doing so has no effect on the user, since this just creates a static reference that is unused by the rest of the application during normal execution.

We then modified the seven classes that need access to the Server so that they accessed it via the singleton reference instead of creating a new instance. Any other objects required by these tests were then accessed via the methods of the Server object (no code change was necessary). To simulate user activity on the Jetty web server, we used The Grinder [6], a load testing tool, to request a series of static and dynamic web pages.

Our testing appears to have revealed one new, unreported bug in the "copyThread" method of the IO 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. The corresponding unit test 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.

This unit test 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. We speculated that the 1.5 second waiting time in the unit test was not 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).

Further inspection of the "copyThread" code revealed that the output stream was never being "flushed"; this could possibly be the error, though we have not yet verified this (adding the "flush" call seems to have made the error go away, but we are concerned that it may reappear if the test were run for a longer time). This has been the only new bug we have found in Jetty to date, but our testing is continuing, and we expect that we may find others in the future. This is one example of an intermittent bug that would not be revealed in traditional unit testing in the development environment, but could appear in the deployment environment, and may be detected with in vivo testing.

# 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 sandbox in which the test would be run. We conducted some performance tests to measure the feasibility of such an approach.

## 6.1. Test setup

For our performance testing, we instrumented Jetty WebServer 6.1 [3] with 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 [6] 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 [23]; the page was dynamically generated to avoid any caching by Jetty. The mean time for page requests was 6.35ms and throughput of HTTP response bytes was 3910kBps.

We then instrumented one Java class in Jetty (HttpConnection), which is used on every page request, but did not specify any unit tests to run. In this case, we could measure the overhead of the instrumentation itself (from the inserted AspectJ code), but did not need to consider the forking of new processes or parallel execution of any test code, since there were no unit tests from which to choose. This time, the mean time for page requests was 6.43ms (1.2% increase) and a throughput of 3800kBps, which indicated very little impact overall.

## 6.3. Performance impact of in vivo testing

Next, we configured Invite to use 15 JUnit test classes (part of the Jetty distribution) with a total of 52 test methods. We instrumented the HttpConnection class, in which there are approximately 50 method calls on each page request, meaning 50 possible chances to launch a unit test. Note that the JUnit test classes did not necessarily test the HttpConnection class; it was simply used as a launching point for the test methods.

We first configured Invite to execute unit tests on only 0.02% of the method calls in the instrumented class, so that each page request would have about a 1% chance of executing a unit test (no precautions were taken to ensure that a single page request did not result in the execution of more than one test, however). Using the same test environment 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. 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 test would be launched during that time, hence the initial overhead.

We then configured Invite to execute unit tests on 0.2% of method calls to the instrumented class, which meant that each page request would have approximately a 10% chance of executing a unit test. In this case, the mean time to complete a page request rose to 7.98ms (25.6% increase), and the throughput was 3030kBps. The mean time to the first byte increased to 5.04ms (40.3% increase), demonstrating that this value was the one most affected by having the unit test execute at the beginning of a response to a page request. We also ran a test in which 100% of the page requests launched unit tests, to get an idea of the worst case overhead. 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.

**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 |

Despite the 25% overhead between the uninstrumented baseline and the configuration to run unit tests on 10% of the page requests, we note that there is less than 5% overhead when running unit tests on 1% of the page requests, and contend that 1% is probably sufficient for detecting defects on a heavily-used application. Of course, there is a tradeoff between executing more tests (and increasing the likelihood of finding bugs) and performance, and careful planning would need to be employed when configuring the testing framework.

## 6.4. Tests with large web pages

Clearly the process forking is the cause of much of the overhead in this implementation of the in vivo testing framework. We measured an average of 4.34ms for the completion of the fork call during the setup with 10% of the requests resulting in unit test execution. However, it should be noted that the overhead of executing the tests is not related to the size of the web page being requested; it is more or less constant, according to how fast the fork can be executed.

**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 |

To demonstrate this, we conducted another test using a large (static) web page of 600kB. 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 unit tests 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 requests caused unit tests to run. This average overhead is still on the order of a few milliseconds, but is very small compared to the total time to serve the page.

## 6.5. Areas for performance improvement

We are continuing to seek ways in which to reduce the overhead of the Invite testing framework. Forking a process is programmatically simple but incurs a large cost, and a transactional rollback strategy may be preferable [43]. With any strategy, though, care must always be taken to ensure that the test does not affect the system state.

It may be possible to reduce the overhead by distributing the testing load across multiple instances of the application. One solution may be to use a tool like the GAMMA system [33, 34] for distributing the tests and determining which tests should be run under different circumstances. Another approach would be similar to the "application communities" idea [24], in which application instances in a software monoculture share information. This would allow in vivo testing to

distribute the testing load in space as well as in time. As part of future work, we intend to consider these solutions and develop a new, distributed implementation of the Invite framework.

## 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 made by any unit test therefore cannot be undone. 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, of course. However, in order to add more robustness to the testing approach, changes must be made to the Invite framework so that it does not modify the state of external systems at all.

One possible way to solve this problem is to use speculative execution at the system level, such as in [43], with the OS kernel modified to allow for the rollback of system calls. An alternative would be to use a virtual machine with a copy of the entire operating system state, and then run the test in the virtual machine [38]. This has the advantage of seeing how the unit tests fare in the context of the entire system state, rather than just the process state, though it does not address any concerns related to external databases or network I/O.

The in vivo testing approach has been designed to call for minimal modification of the source code (both the unit tests and that of the software under test), but because most unit tests are written such that they create new instances of objects, modification would generally be needed so that they refer to existing objects within the running application. Whereas this often requires only small changes to existing test code, the more concerning issue is that the application source code may not be equipped to provide arbitrary access to objects, as explained in Section 4. In order to avoid major modification of the existing source code, one approach could be to inject code at runtime with a system like [18] so that references to objects can be stored in a central location within the application, accessed by the test code; however, this may require more modification of the unit tests.

Currently the Invite framework has only been implemented in Java and designed to work with Java applications. Porting it to C or C++ could present a challenge because the framework uses 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 modify the compiled code dynamically, such as in Kheiron/C [18], could be used instead.

To date we have not made efforts to determine the *adequacy* [46] of our testing approach, for instance by measuring path/statement coverage. Further work could more precisely categorize the prospective defects that could be found, and establish success criteria.

Also, we have not yet considered what action to take once a unit test fails and a defect is found. A simple approach would be to use an online crash reporting system like the Mozilla Quality Feedback Agent or Microsoft XP Error Reporting to gather state when the system crashes and send the data back to the development team. Another option would be similar to that of Skoll [19, 28], in which defects are reported to a central server, which manages the distribution of tests.

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 arbitrarily chooses random tests to execute 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, rather than just selecting unit tests randomly.

Lastly, one possible future direction for this work is to consider only the case of executing unit and/or regression tests after an automatic repair policy is invoked in self-healing systems, as in [14] and [23], to ensure that the repair did not adversely affect the system. The approach could also be applied to the domain of security testing: it could detect invalid states that are a result of an attack or intrusion attempt.

## 8. Conclusion

We have presented in vivo testing, a novel testing approach that allows for 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 framework called Invite used for implementation.

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. Additionally, we have demonstrated that our approach and the current implementation add limited overhead in terms of system performance and code modification.

As this is just a report of our initial work in this area, we expect that in vivo testing will provide a foundation for other work in perpetual testing.

## 9. Acknowledgements

The authors would like to thank Phil Gross for his assistance. Murphy and Kaiser are members of the Programming Systems Lab, funded in part by NSF CNS-0717544, CNS-0627473, CNS-0426623 and EIA-0202063, 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] OSCache: `http://www.opensymphony.com/oscache`

[6] The Grinder: `http://grinder.sourceforge.net/`

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

[8] G.K. Baah, A. Gray, 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*, Portland OR, 2006, pp. 70-77.

[9] 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*, April 2003, pp. 17-12.

[10] 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.

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

[12] D. Brenner, C. Atkinson, *et al.*, "Reducing Verification Effort in Component-Based Software Engineering through Built-In Testing", *Information System Frontiers vol. 9 issue 2-3*, 2007, pp. 151-162.

[13] J. Clause and A. Orso, "A Technique for Enabling and Supporting Debugging of Field Failures", In *Proc. of the 29th ICSE*, Minneapolis MN, 2007, pp. 261-270.

[14] B. Demsky and M. C. Rinard, "Automatic data structure repair for self-healing systems", In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2003.

[15] G. Denaro, L. Mariani, M. Pezz`e, "Self-test components for highly reconfigurable systems", In *Proceedings 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, J.-M. Jezequel, "Increase software trustability with self-testable classes in Java", In *Proc. of the 2001 Australian Software Engineering Conference*, Aug 2001, pp. 3-11.

[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*, Boston MA, 2004, pp. 65-75.

[18] R. Griffith and G. Kaiser, "A Runtime Adaptation Framework for Native C and Bytecode Applications", *3rd IEEE International Conference on Autonomic Computing*, June 2006, pp. 93-103.

[19] A. Krishna, *et al.*, "A Distributed Continuous Quality Assurance Process to Manage Variability in Performance-intensive Software", *19th ACM OOPSLA Workshop*, Vancouver, 2004.

[20] S. Lawrence and C.L. Giles, "Accessibility of Information on the Web", *Intelligence vol. 11 issue 1*, Spring 2000, pp. 32-39.

[21] C. Liu, D.J. Richardson, "RAIC: Architecting Dependable Systems through Redundancy and Just-In-Time Testing", *ICSE Workshop on Architecting Dependable Systems (WADS)*, Orlando FL, 2002.

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

[23] M.E. Locasto, G.F. Cretu, A. Stavrou, A.D. Keromytis, "A Model For Automatically Repairing Execution Integrity",

Tech Report CUCS-005-07, Department of Computer Science, Columbia University, January 2007.

[24] M.E. Locasto, S. Sidiroglou, A.D. Keromytis, "Software Self-Healing Using Collaborative Application Communities", In *Proceedings of the Internet Society (ISOC) Symposium on Network and Distributed Systems Security (NDSS 2006)*, San Diego CA, Feb. 2006, pp. 95-106.

[25] C. Mao, "AOP-based Testability Improvement for Component-based Software", In *31st Annual International COMPSAC, vol. 2*, July 2007, pp. 547-552.

[26] C. Mao, Y. Lu, J. Zhang, "Regression testing for component-based software via built-in test design", In *Proc. of the 2007 ACM Symposium on Applied Computing*, Seoul, South Korea, 2007, pp. 1416-1421.

[27] L. Mariani, M. Pezze, D. Willmor, "Generation of integration tests for self-testing components", In *Proceedings of FORTE 2004 Workshops, Lecture Notes in Computer Science, Vol.3236*, 2004, pp. 337-350.

[28] A. Memon, A. Porter, *et al.*, "Skoll: distributed continuous quality assurance", In *Proc. of the 26th ICSE*, Edinburgh, UK, May 2004, pp. 459-468.

[29] 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*, Portland OR, 2006, pp. 55-62.

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

[31] L. Naslavsky, *et al.*, "Multiply-Deployed Residual Testing at the Object Level", In *Proc. of IASTED International Conference on Software Engineering (SE2004)*, Innsbruck, Austria, 2004.

[32] L. Naslavsky, R.S. Silva Filho, *et al*. "Distributed Expectation-Driven Residual Testing", *Second International Workshop on Remote Analysis and Measurement of Software Systems (RAMSS'04)*, Edinburgh, UK, 2004.

[33] A. Orso, D. Liang, M.J. Harrold, "Gamma System: Continuous Evolution of Software after Deployment", In *Proc. of ISSTA 2002*, Rome, Italy, 2002, pp. 65-69.

[34] A. Orso, T. Apiwattanapong, M.J. Harrold, "Leveraging field data for impact analysis and regression testing", In *Proc. of the 9th European Software Engineering Conference*, Helsinki, Finland, 2003, pp. 128-137.

[35] L. Osterweil, "Perpetually Testing Software", *The Ninth International Software Quality Week (QW'96)*, San Francisco, May 1996.

[36] C. Pavlopoulou and M. Young, "Residual Test Coverage Monitoring", In *Proc. of the 21st ICSE*, Los Angeles CA, May 1999, pp. 277-284.

[37] I. Pavlova, M. Åkerholm, 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*, Portland ME, 2006, pp. 51-52.

[38] S. Potter and J. Nieh, "Reducing Downtime Due to System Maintenance and Upgrades", In *Proc. of the 19th Large Installation System Administration Conference (LISA 2005)*, San Diego CA, Dec 2005, pp. 47-62.

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

[40] D. Rubenstein, L. Osterweil, S. Zilberstein, "An Anytime Approach to Analyzing Software Systems", In *Proc. of the 10th International FLAIRS Conference (Florida Artificial Intelligence Research Society)*, Daytona Beach FL, May 1997, pp. 386-91.

[41] D. Saff, "Automated continuous testing to speed software development", Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge MA, Feb. 2004.

[42] D. Saff and M.D. Ernst, "An experimental evaluation of continuous testing during development", In *Proc. of ISSTA 2004*, Boston MA, 2004, pp. 76-85.

[43] S. Sidiroglou, M.E. Locasto, S.W. Boyd, A.D. Keromytis, "Building A Reactive Immune System for Software Services", In *Proceedings of the USENIX Annual Technical Conference*, April 2005, pp. 149-161.

[44] J. Vincent, G. King, P. Lay, J. Kinghorn, "Principles of Built-In-Test for Run-Time-Testability in Component-Based Software Systems", *Software Quality Journal vol. 10 no. 2*, Sept. 2002.

[45] Y. Wang *et al.*, "On built-in test reuse in object-oriented framework design", *ACM Computing Surveys vol. 32 no. 1*, March 2000.

[46] E. Weyuker, "Axiomatizing software test data adequacy", *IEEE Trans. Software Eng., SE-12*, Dec 1986, pp. 1128-1138.

[47] S.S. Yau and R.C. Cheung, "Design of self-checking software", In *Proc. of the International Conference on Reliable Software*, Los Angeles CA, 1975, pp. 450-455.

[48] M. Young, "Perpetual Testing", AFRL-IF-RS-TR-2003-32 Final Technical Report, February 2003, [http://handle.dtic.mil/100.2/ADA412542].