Automated Support for Testing Dynamic Software Updates

Luís Pina
Imperial College London
London, UK
l.pina@imperial.ac.uk

Michael Hicks
University of Maryland
College Park, MD, USA
mwh@cs.umd.edu

***Abstract*—Dynamic software updating (DSU) is a technique for updating running programs by changing their code and execution state without stopping them. To avoid crashes and errors induced by incorrect or ill-timed updates, we need to systematically test applying dynamic updates. This paper presents Tedsuto, a framework for doing just this, applied to the Java-based Rubah DSU system. In particular, we present three techniques to support DSU testing, all of them adapting existing system tests, that range from fully-automated to requiring minimal manual effort. We applied Tedsuto to dynamic updates previously developed (and tested in an ad hoc manner) for the H2 SQL database server and the CrossFTP server— two real-world, multithreaded systems— and we found several update-related bugs in short order, and at low cost. We believe our testing framework forms part of a practical approach to providing assurance for DSU.**

I. Introduction

As on-line services go global, an increasing number of systems require constant availability, and as a matter of convenience many other systems would prefer it. An approach for ensuring high availability is *dynamic software updating* (DSU). This technique works by updating a process *in place*, patching the existing code and transforming the existing in-memory execution state. DSU preserves active, long-running connections (e.g., to databases, media streaming, FTP and SSH servers), which can immediately benefit from important program updates (e.g., security fixes). It also preserves in-memory server state. Doing so is extremely valuable for in-memory databases, gaming servers and many other systems, that rely on the relatively low expense and high performance of commodity RAM, to maintain large in-memory data sets. This problem is acute enough that Facebook uses a custom version of memcached that keeps in-memory state in a ramdisk to which it reconnects on a post-update restart [1].

In the research community, general-purpose DSU systems have been shown to work to dozens of realistic applications, tracking changes according to those applications' release histories [2], [3], [4], [5], [6]. These results are penetrating the mainstream; e.g., Linux 4.0 supports "rebootless patches" to a running kernel, for security patches and other fixes [7], [8].

DSU is not a panacea, of course; it must be done with care. The program code assumes the execution state adheres to a certain format and invariants, so changing the code at run-time requires corresponding changes to state, and it is often the programmer's responsibility to define these changes. One particular challenge is *timing*: the transformation code may assume certain invariants, and these must hold under all the circumstances in which an update might take place. There is also the challenge that the semantics of the program may change due to the update, and this change may only make sense at certain points in execution. Mistakes in timing or transformation can result in crashes, corruption, or other misbehavior, defeating the whole purpose of DSU, that is, 24/7 service. As such, we need a reliable way to *test* that dynamic updates are correct before we attempt to deploy them on live systems.

This paper presents *Tedsuto*,[1] a new DSU testing framework implemented for the Rubah [6] Java DSU system. To use Tedsuto, the programmer annotates existing *system tests*, which check the end-to-end behavior of a program. Tedsuto uses the annotated tests to assert the correctness of an update by checking if the program's behavior before, during, and after an update makes sense. The basic idea is to systematically repeat each system test automatically, performing updates at different points in each re-execution.

Tedsuto can either test updates *intensively*, exhaustively updating at every possible opportunity in a small window; or *extensively*, across a broad space of possible opportunities while striving to explore updates that happen at different points during the program's execution (in essence, maximizing coverage). Tedsuto also eases the burden of developing updatable software by stressing update-related application code without actually performing an update.

Tedsuto is a complete solution to testing DSU that can check both backwards-compatible behaviors, i.e., those externally visible behaviors that the update does not change, and new behaviors, e.g., that a bug is fixed correctly or that a newly added feature functions properly. Even though we implemented it for Rubah, the basic concepts can be adapted to other state-of-the-art DSU systems such as Kitsune [3], JVolve [5], or the DCE-VM [4].

We evaluated Tedsuto by applying it to updates previously developed (and tested in an ad hoc manner) for H2, an SQL database server, and CrossFTP, an FTP server — two real-world multi-threaded systems. Tedsuto was able to (more quickly) find three bugs we already knew about, from our ad hoc testing efforts (after we reverted the code that fixes it) along with six new update-related bugs.

Adapting a significant portion of the system tests available

[1] This is a play on the name of the University of Maryland mascot, Testudo.

for these two programs required changing only 2% of the total lines of code of the existing system tests. This effort reduced the number of re-runs per intensive test by one order of magnitude for H2, making this approach feasible. The annotations on system tests also allowed extensive testing to systematically explore different updates and find two bugs out of three possible that have a narrow window of opportunity. Extensive update testing could also find bugs that require multiple threads to happen just by performing updates at random points during the system test. In this case, the annotations were still useful to characterize the update that caused the bug.

While prior work has briefly considered the problem of dynamic software update testing, Tedsuto represents a substantial step forward as it is applicable in more realistic situations (e.g. updating multi-threaded programs), and supports testing not only backward compatible updates, but also updates that add new features. We believe Tedsuto is a promising step toward practical assurance for real-world DSU.

II. Background and Problem Statement

This section begins by presenting some background on DSU in general, and on Rubah, the Java-based DSU system we used to implement Tedsuto, in particular. Then we describe the sorts of failures that could arise due to incorrect dynamic updates that Tedsuto aims to uncover.

A. Dynamic Software Updating

To implement a *dynamic* software update, which takes effect at run-time, we must have means not only to load and enable the new version's code, but also means to update the existing *execution state*. This state consists of *data*, like linked tree and list structures that store an in-memory database, and *control*, like the execution stacks of active threads. The program code assumes the state adheres to a certain format and invariants, so changes to code require corresponding changes to state.

Consider an example. If in the updated program the entries of a hash table are extended with a timeout field, then a dynamic update needs to convert in-memory hash table entries to now contain a timeout field; otherwise, when the updated code goes to access that field, it may behave incorrectly. Or, if the new version refactors some of a function `foo`'s code into another function `bar` that is then called by `foo`, a dynamic update that occurs while `foo` is running may require transforming the program counter and stack to be as if the `foo` had called `bar`. Changes to in-memory data, like the first example, we call *data migrations*, while changes to control, like the second example, we call *control migrations*.

Control and data migrations are typically specified manually by the programmer (perhaps with some automated assistance). As such, mistakes could mean that when the dynamic update is applied the running program will hang, crash, or otherwise misbehave. Such an outcome is particularly problematic for DSU, since the whole point of updating while running is to avoid service disruption. Therefore, the developer needs some way to test that her dynamic update, when deployed, will produce the correct behavior.

```
1  Transfer transfer; // Local to this connection
2  Session session;   // Local to this connection
3  boolean stop;
4
5  Response process(Request req);
6
7  public void run() {
8    if (!Rubah.isUpdating()) {
9      transfer.init();
10     // Parse client version
11     // Negotiate protocol params
12     transfer.flush();
13     session = new Session();
14   }
15   Selector s = new Selector();
16   try {
17     while (!stop) {
18       try {
19         Rubah.update("process");
20         Request req = transfer.readRequest( s );
21         Response resp = process(req);
22         transfer.writeResponse(resp);
23       } catch (SQLException e) {
24         transfer.writeException(e);
25       } catch (UpdateRequestedException e) {
26         continue;
27       }
28     }
29   } catch (UpdatePointException e) {
30     throw e;
31   } catch (Throwable e) {
32     logError(e);
33   } finally {
34     if (!Rubah.isUpdateRequested()) {
35       s.close();
36       transfer.close();
37       session.close();
38     }
39   }
40 }
```

Fig. 1. Example adapted from H2 TcpServerThread featuring logic related with update points (gray highlight) and control migration (black highlight).

B. Rubah

Tedsuto is implemented using Rubah, a DSU system for dynamically updating Java programs.[2] Therefore, in this subsection we present some background on how Rubah works and how the programmer specifies a dynamic update.

Overview. To use Rubah to perform dynamic updates, the programmer must perform three tasks. First, prior to deployment, she must modify the host program to include *update points*, which are program points at which dynamic updates are permitted to take effect [9]. Limiting updates to a few well-chosen program points considerably simplifies the task of writing a dynamic update. Second, she must also modify the program to be able to perform control migration if it is restarted in a special *updating mode*; we say more about this shortly. Third, she must provide an *update class* that defines how objects whose classes have changed should be updated during data migration (many aspects of this class are automatically generated).

Example. We have used Rubah to dynamically update the

[2]We believe our framework would also work for other Java-based DSU systems, such as JVolve [5], with very little modification.

H2 database management system, so we will use it as a running example. Figure 1 shows a simplified version of a connection-handling method from H2. The changes we made to support DSU are highlighted—ignore them for now. The method starts by parsing the client data and negotiating the protocol parameters (lines 9–13). Then, it enters a loop (lines 17–28) that reads each client command (line 20), executes it in method `process` (line 21), and sends the response back to the client (line 22). The server keeps state about the client using the `session` object, declared in line 2.

Note the complex handling of exceptions, typical in server methods. The server sends recoverable exceptions back to the client (line 24), and logs non-recoverable exceptions (line 32). A **`finally`** block ensures that the connection is closed when the server method exits (lines 33–39).

Update Points. The programmer specifies update points as calls to method `Rubah.update`. When this method is called, if a dynamic update is available, then the Rubah run-time system will initiate (or continue) the process of applying the update. A good place to put an update point is at a point in a long-running loop at which a thread is *quiescent*, meaning that it has finished processing a unit of work and has not started to process the next one. State relevant to an update is not in the middle of being modified, which will make writing the update class easier. The `Rubah.update` method takes a string as its sole argument, which serves as a kind of label—update points across versions that share the same label are in some sense equivalent. For the example in Figure 1, the code related to update points is in gray. An update point is placed on line 19.

When an update is underway, calling `Rubah.update` throws an `UpdatePointException`; unhindered, this exception will ultimately reach a Rubah-provided wrapper for a thread's `run` (or `main`) method, where it is caught and the throwing thread is paused. Of course, the exception may be caught by intervening **`catch`** blocks in the application, so the developer may need to make changes to avoid this (line 30). The developer also needs to ensure that the exception does not change any state by being propagated, therefore actions within finally blocks must be guarded to account for possible updates (line 34). When all threads have been paused after reaching update points, the next stage of updating may begin, involving control flow migration and data migration.[3]

Control migration. Once the new version's code is installed, the next step is to guide the paused threads to update points equivalent to (i.e., having the same label as) the ones at which they were originally stopped; this process is called *control migration*.

Rubah restarts each paused thread's (possibly updated) `run` (or `main`) method. When a thread executes this method normally, it typically performs actions that should not be re-performed during control migration. In our example, lines 9–13 negotiate protocol parameters with the client, which should not be repeated post update. To address this issue, Rubah provides API calls that the developer can use to determine whether a thread is running for the first time or as a result of an update. In our example, line 8 guards the initialization code with a call to `Rubah.isUpdating` which returns **`true`** if called while performing the control migration and **`false`** otherwise.

[3]One note: When an update becomes available, the program may be blocked waiting for some I/O operation. To avoid an undue delay, I/O operations should be interruptible. The implementation of class `Transfer`, omitted from this example, was retrofitted to use the API that Rubah provides to wait for I/O in a way that is interruptible when an update becomes available. When interrupted by an update, method `readRequest` throws an `UpdateRequestedException`. This exception is caught on line 25 and the loop soon reaches the update point on line 19.

```
1  class Session {
2    User user;
3    String userName; // Added in version 1
4  }
5
6  class UpdateClass {
7    void convert(v0.Session o0, v1.Session o1) {
8      // Automatically generated
9      o1.user = o0.user;
10     o1.userName = null;
11     // Customized
12     o1.userName = o0.user.name;
13   }
14 }
```

Fig. 2. Example adapted of an update class with a single instance conversion method.

Data migration. Prior to restarting each thread, Rubah performs *data migration* to convert the existing program's objects to use the updated classes. Conceptually, this happens by visiting each object in the heap that might have been affected by an update and *transforming* it to work with the new version's code.

Rubah finds outdated instances and converts them *lazily*, while the new program executes.[4] To start this off, Rubah transforms the root references; these are just the instances of class `java.lang.Thread` and all the static fields of loaded classes (no local variables need be considered, since all stacks are unrolled). When each object is transformed, pointers to child objects instead point to *proxies* that initiate the transformation when accessed for the first time, and then remove themselves from the object graph.

How to convert an object is determined by the *update class*. Figure 2 shows an example of a class that changed between versions, together with the update class which specifies its transformation. This example has a single *instance conversion method*[5] that transforms instances of class `Session` by taking an existing instance `o0` that belongs to version v_0 and using it to initialize the equivalent new instance `o1` that shall take `o0`'s place in v_1.

Update classes have one instance conversion method for each class that has a different set of fields from version v_0 to version v_1. Even if the set of fields is the same, with regards to name and type, the developer can define instance conversion methods to account for fields whose semantics changes. If a field has changed neither name nor type, Rubah copies its value from the old to the new version by default. In the example that we are following, Rubah automatically copies field `user`.

The arguments of the conversion method in Figure 2 are *skeleton classes*, which as the name implies, have been stripped

[4]Rubah also supports eager migration, while the threads are stopped, in the style of a parallel garbage-collector, but it can induce a long pause.

[5]Update classes can also have *static* conversion methods, for static fields.

of a lot of the original's contents: all methods are removed, and all fields are made public (so as to be accessible to the update class's code). Each class is placed in a distinct namespace, depending on its version, allowing the developer to refer to version v_0 or v_1 unambiguously and still use the regular Java compiler to build the update class.

In most cases, the logic required for transformation is simple; e.g., version v_0 of a class has two fields while version v_1 has three, and the newly added field is initialized with its default value. Rubah provides a tool that generates a stub update class by analyzing two versions; matching fields by owner class name, field name, and field type; and generating a conversion method for each class with unmatched/changed fields. In the cases that a transformation is more involved, the programmer must specify what to do.

C. DSU Failures

There is room for error in each of the three steps discussed in the previous subsection, and so we need to test that a updatable program and actual updates to it are correct. For example, the developer might have done something wrong regarding update points, e.g., by not placing them in sufficiently many places in the program, or by failing to properly propagate the `UpdatePointException` to the top of the call-graph, or by failing to ensure that relevant I/O operations are interruptible. There is also room for error in the control migration code; e.g., the program fail to avoid previously taken initialization actions. Finally, the update class may be incorrect, e.g., due to newly added fields not being initialized properly, or due to new invariants among existing fields failing to be established.

One particular challenge with all of this is *timing*: Updates might work under some, but not all, circumstances. For example, it could be the control migration code leading up to update point A is correct, but is incorrect leading up to point B. This means our testing framework must exercise the program to try out updates at various points under different circumstances.

We conclude this section with two example bugs we experienced when developing updates using Rubah. We will return to these and other bugs later in the paper to show how our testing framework was able to find them.

Buggy Update Class. When a class changes its fields between versions, e.g., by adding a field or changing an existing field's type, it is easy for the developer to see that some customization of the update class may be needed. However, in some cases, the program state retain the same structure between versions but still require transformation because the new program version interprets the same state differently.

Consider the `TransactionCommand` class in Figure 3, which is adapted from H2. The field `type` is the type of the command, as defined by the enumeration that precedes it; the `execute` method switches on the type when handling a command. Version 1 introduces a new command, together with new constant `SHUTDOWN_COMPACT`, which shifts the value of the constant that represents another command, `BEGIN`, from 14 to 15. As such, the update class must change existing instances of `TransactionCommand` that have type 14 to type 15; otherwise executing a `BEGIN` command, which should start a database transaction, instead shuts down the database and compacts it.

```
1  class TransactionCommand {
2    // Version 0
3    static final int SHUTDOWN_IMMEDIATELY = 13;
4    static final int BEGIN = 14;
5    // Version 1
6    static final int SHUTDOWN_IMMEDIATELY = 13;
7    static final int SHUTDOWN_COMPACT = 14;
8    static final int BEGIN = 15;
9
10   int type;
11
12   void execute() {
13     switch (type) {
14       ...
15     }
16   }
17 }
```

Fig. 3. Change to the `TransactionCommand` class in which values are interpreted differently between versions.

```
1  while(iterator.next()) {
2    Rubah.update("query-group");
3    // Conditionally add row to result set
4  }
```

Fig. 4. Example adapted from H2 TcpServerThread featuring logic related with update points (gray highlight) and control migration (black highlight).

Buggy Control Migration. Correct control migration requires that a restarted thread makes its way back to the equivalent update point, and does neither redundant nor insufficient work in the process. Figure 4 shows an example of a subtle control migration bug. Here, the code iterates over a set of rows on a table and adds some to a result set. Unfortunately, if an update takes place, the value returned by `iterator.next()` is not processed. We might attempt to fix this problem by moving the update point to the end of the loop, after the element is processed. But even this is wrong: If this is the last iteration of the loop (i.e., the iterator is now empty), then on restart, the thread will not re-enter the loop in the new version. Therefore, the correct fix is to *also* place an update point, with the same label, just before the loop, so that control migration completes even when the loop is not entered.

III. Tedsuto — A Framework for DSU Testing

DSU is a whole-program operation, and the correctness of an update can be determined by checking that the program's behavior before, during, and after an update makes sense. To use Tedsuto, developers make small changes to existing *system tests* which check the back-to-back behavior of a program. Tedsuto will then repeat each system test automatically, performing an update at different points in each re-execution. Tedsuto easily tests backwards-compatible behaviors, i.e., those externally visible behaviors that the update does not change, and new behaviors, e.g., that a bug is fixed correctly or that a newly added feature functions properly. It is thus a complete solution for testing DSU. Even though we implemented Tedsuto using Rubah, the basic concepts can be adapted to other DSU systems such as Kitsune [3], JVolve [5], or the DCE-VM [4].

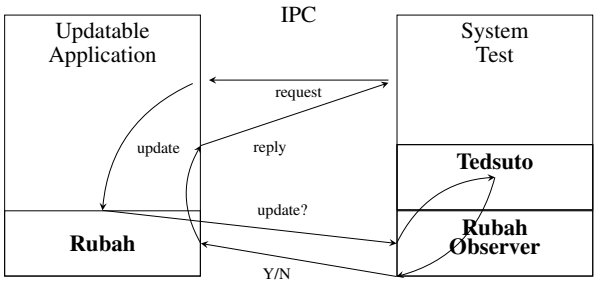

Fig. 5. Architecture of the testing framework.

```
1  class Tedsuto {
2    static void allowUpdates();
3    static void disallowUpdates();
4    static boolean updated();
5    static void operation(String label);
6  }
```

Fig. 6. Tedsuto's API for adapting system tests.

A. Architecture

Figure 5 shows Tedsuto's architecture. The updatable application and the system test run in separate processes that communicate through Inter-Process Communication (IPC). During its workload, the system test performs several requests to the updatable application. While processing each request, the execution of the updatable application triggers several update opportunities. In Rubah, these happen when the updatable application reaches update points.

A novel part of our approach is that the DSU framework queries an *update observer* at every update opportunity to decide whether to install an update or not. The update observer, located on the same process as the system test, then notifies Tedsuto about the opportunity to perform an update.

Tedsuto requires manual effort to adapt each system test, so that it can then use information about the system test to explore different update opportunities systematically, repeating the test and relaunching the updatable application at every re-execution.

Once Tedsuto decides whether to update or not, the observer sends the decision back to Rubah, which reacts accordingly before returning the control to the updatable application.

Eventually, the updatable application finishes processing the request and sends the response back to the system test. At this point, the system test can check if the response is correct. Note that the system test can query Tedsuto to see if the update took place or not, so that it can test for old or new behavior.

B. Adapting Existing Tests

Tedsuto requires adapting existing system tests. Figure 6 shows the API available to do so. Methods `Tedsuto.allowUpdates` and `Tedsuto.disallowUpdates` delimit portions of the test where the testing framework should explore different update opportunities. Method `Tedsuto.updated` returns **`true`** if an update has taken place on the current test, **`false`** otherwise.

```
1  byte[] testData     = ...;
2  byte[] testDataHash = md5(testData);
3  login(USERNAME, PASSWORD);
4  TestUtil.writeDataToFile(FILE1, testData);
5  String fileName = FILE1.getName();
6
7  Tedsuto.allowUpdates();
8  int resp = sendCommand("MD5 " + fileName);
9  Tedsuto.disallowUpdates();
10
11 if (Tedsuto.updated()) {
12   assertEquals(251, resp);
13   assertEquals(testDataHash, getReplyBytes());
14 } else {
15   assertEquals(502, resp);
16 }
17
18 logout();
19 FILE1.delete();
```

Fig. 7. Example that tests the implementation of the MD5 command on an FTP server. The highlighted code represents the code added to integrate this test with our testing framework.

To understand the need for these methods, consider the example that Figure 7 shows, which tests the implementation of the MD5 command on an FTP server. The test starts by generating some state (set-up, lines 1–5), then perform the testing logic on the generated state (line 8) and checks if obtained results matches the expected (lines 11–16). Finally, the test rests the state (tear-down, lines 18–19).

The structure of this test is representative of many system tests. Methods `Tedsuto.allowUpdates` and `Tedsuto.disallowUpdates` delimit the portion of the test where it is interesting to explore different update opportunities.[6] In this particular test, the new version adds support for the MD5 command, which was not supported on the old version. The developer can use method `Tedsuto.updated` to adjust how the test interprets the result.

The test that Figure 7 shows is simple, in the sense that it is composed by a single logical operation, which is to test the MD5 command. System tests can be more sophisticated. For instance, consider the example that Figure 8 shows. This is a system test for SQL databases that implement a serializable level of isolation between concurrent transactions. It launches several threads, each incrementing the same row a number of times. Each increment takes place inside its own database transaction. At the end, the test checks the contents of the row to ensure that all transactions were in fact executed under serialized semantics.

The isolation level test in Figure 8 is more complex than the MD5 test in Figure 7. In particular, the it uses multiple threads and is composed of different logical operations: Each thread starts a transaction, reads the current value, writes the increment, and commits the transaction. The developer can use method `Tedsuto.operation`, to inform Tedsuto about when each of these operations starts. Tedsuto is thus able to differentiate update opportunities by when the test triggers them. Furthermore, Tedsuto can explore updates that happen on different combinations of operations on multithreaded sce-

[6]This FTP test assumes that updates cannot happen while processing an FTP command.

```
1  int n_threads; int inc;
2
3  // Launch n_threads that do:
4  void run() {
5    for (int i = 0 ; i < inc ; i++)
6      Tedsuto.operation("BEGIN");
7      sql("BEGIN TRANSACTION");
8      Tedsuto.operation("INC");
9      int c = sql("SELECT v FROM count");
10     c++;
11     sql("UPDATE count SET v="+c);
12     Tedsuto.operation("COMMIT");
13     sql("COMMIT");
14 }
15
16 // After all threads join, check result
17 int total = sql("SELECT v FROM count");
18 assertEquals(total,n_threads * inc);
```

Fig. 8. Example that tests if concurrent SQL transactions execute under serializable semantics. The highlighted code represents the code added to integrate this test with our testing framework.

narios, e.g. thread 1 writing the incremented counter while thread 2 is reading the value of the counter.

C. Exploring Update Opportunities

Update bugs can depend on the timing of the update. Finding the particular timing that causes an update to pass or fail the test is hard. Tedsuto thus executes each test multiple times and installs an update at different points during each re-execution.

Re-executing each system test multiple times changes the question about when to install an update to how many different update opportunities should be explored for each test. The trivial answer is to exhaustively explore all possible update opportunities for each system test. Given that each system test can only install one update per execution, this approach has two drawbacks. First, the sheer number of update opportunities that each system test generates may require an infeasible number of re-executions. Second, to explore different update opportunities over multiple test executions, the testing framework needs to be able to map update opportunities from one execution to the next. Otherwise, it might get stuck exploring the same set of update opportunities or miss important update opportunities.

Another approach is to use sampling, i.e. probabilistically deciding when to take an update, until enough update opportunities have been explored. This approach solves the two drawbacks of exhaustively exploring all opportunities, but has a drawback of its own: Finding when enough update opportunities have been explored is hard (or impossible), and getting it wrong means potentially missing bugs.

D. DSU Testing

Tedsuto provides three strategies to explore opportunities during a system test. The first — *control-flow reboots* — explores all update opportunities during a single execution of a system test by performing only control migration without changing the program code or transforming the program state. The second — *intensive update testing* — exploits the typical structure of system tests (set-up, test, tear-down) to repeat each system test until it has explored all update opportunities on just the relevant portion. The third — *extensive update testing* — uses the information about the operations that make up the test to explore update opportunities that happen during different test operations.

Control-flow reboots perform control migration at every possible update opportunity while the program is running without actually installing a new version or performing any data migration. It is our experience when using Rubah and Kitsune, that the control migration code does not change much between versions. Retrofitting is mostly a one-time effort on the first version that supports DSU. This observation is the basis of control-flow reboots, which stress the retrofitted code and the control migration thoroughly and systematically. This technique can be used together with manual testing during development time to systematically test the control migration at each different update point. Control reboots are thus a no-effort and low-cost solution to test control migration during development time.

Intensive update testing is applicable to system tests that focus on a single feature, such as the MD5 system test shown in Figure 7. This test is small and self-contained, which means it generates a small number of update opportunities inside the allowed window that the developer specified. Besides, this particular test is deterministic. That is, the nth update opportunity on a particular test run is the same as the nth update opportunity on a different test run. Given these two properties, it is feasible to exhaustively explore all update opportunities that this test triggers.

Intensive update testing is not suitable for all multithreaded system tests due to the non-determinism present in thread scheduling. For instance, consider the transaction isolation test shown in Figure 8. It is not possible to match update opportunities during different test runs because thread execution may have been interleaved in a different order. Some tests, however, use several threads and fix their interleaving. For instance, consider a test for lock timeouts that consists of a thread that acquires a lock and then sleeps, and another that tries to acquire the same lock and times out. Given that this test has to fix the thread scheduling in advance, we can use intensive update testing to ensure its correctness in the presence of updates.

Extensive update testing is applicable to more complex system tests, potentially multithreaded, that check a broader feature of the updatable program and generate a large number of update opportunities in doing so. The transaction isolation test, shown in Figure 8, is a good example of such tests.

Given the complexity of the system tests it uses, extensive update testing requires the developer to annotate the test with information about which logical operations the test performs. This information is useful for two reasons: The testing framework can use it to chose update opportunities that happen in different parts of the test; and, when the test fails, the testing framework can report when the update happened in terms of which operations the test was performing at the time of the update. Later, the developer can run the test limited to exploring opportunities only during the failed operation to reproduce the bug or test that it is fixed.

Extensive update testing groups the operations per thread to consider combinations of operations. For instance, in the trans-

action isolation test that we are following, all possible combinations for two threads are: BEGIN/BEGIN, BEGIN/INC, BEGIN/COMMIT, INC/INC, INC/COMMIT, COMMIT/COMMIT. The number of update opportunities in each may vary between executions. For instance, consider that reading a value is implemented by acquiring a lock and that, in combination INC/COMMIT, the thread that is trying to read spins while waiting for the lock and reaches an update point at every spin. The number of update points reached in this combination depends on how long the commit operation takes.

Furthermore, for higher numbers of threads, the sheer number of different combinations may make it infeasible to even explore them all. As a consequence, sampling is the best technique to explore different update opportunities between executions while performing extensive update testing.

IV. Experimental Evaluation

This section presents experiments that show Tedsuto:

1) **Requires Low Effort** The manual effort required to adapt existing system tests to use Tedsuto, in lines of code, is 2% of the size of the existing system tests;
2) **Has Acceptable Overhead** Whilst overhead is not our main concern, observing Rubah while it is being tested cannot make the tests take an infeasible amount of time to complete. Instrumenting Rubah to be controllable by an external process slows down the original application by a factor of 2 at most;
3) **Provides Practical Coverage** Adapting existing system tests to perform intensive update testing only during specified periods reduces the total number of re-executions to a feasible number;
4) **Finds Bugs Effectively** Extensive update testing can find timing-sensitive update bugs by performing updates at random times. Systematically exploring different update times finds timing-sensitive update bugs that have a narrower window of opportunity by exploring less common operations. Both approaches find bugs that require several threads to trigger. Of the total six new bugs we discovered, three were originally found by control-flow reboots, one by extensive update testing, and two by intensive update testing.

A. Experimental Configuration

All experiments that we describe in this section were run on a machine equipped with an Intel Core i7-4770 CPU (4 physical cores, 8 logical) and 16GB of RAM, running GNU/Linux Ubuntu 14.04.2 LTS with kernel version 3.13.0-45-generic. All tests were conducted with Oracle's JVM version 1.7.0_75-b13 (HotSpot version 24.75-b04).

We performed the experimental evaluation using two applications previously retrofitted to support DSU through Rubah [6]: H2, which is a SQL DBMS written in Java; and CrossFTP, which is an FTP server. We updated H2 from version 1.2.121 to version 1.2.123 and CrossFTP from version 1.07 to version 1.11.

As for system tests, H2 ships with a comprehensive testing framework that includes suitable tests. We also used the TPC-C benchmark shipped with the DaCapo benchmark suite [10] as a multi-threaded system test for H2. CrossFTP does not have a testing framework of it own, so we adapted the testing framework of another Java FTP server (Apache's MINA project[7]) to work with CrossFTP. We also used the FTP benchmark *FTP-bench* that we implemented to measure the performance of CrossFTP [6] as a multi-threaded system test.

Framework	Total			Adapted		
	Tests	Files	LOC	Tests	Files	LOC
H2	417	122	32287	48	16	389
MINA	188	50	7590	110	23	154
TPC-C	-	-	8563	-	-	32
FTP-bench	-	-	392	-	-	31

TABLE I. Effort required to adapt existing testing frameworks. LOC stands for Lines Of Code.

B. Manual Effort

Table I shows the effort required to adapt the existing testing frameworks to use Tedsuto. Lines were counted with CLOC, modified lines were counted with DIFFSTAT on the patch that adds support. The results show that adapting existing system tests to use our technique required modifying 2% or less of the total lines of code.

It was not possible to adapt all system tests in each framework for several reasons. MINA tests features that CrossFTP either does not support, or supports in a non RFC compliant way. Other tests depended on the particular implementation of the MINA FTP server they were designed to test. Some tests required the server to be run with slightly different configurations and Tedsuto does not yet support that.

H2 can be configured to work on several different modes: Memory vs disk-based storage, and different types of disk-based stores, recovery modes, and indexes; lock vs multi-version concurrency control; networked vs in-process databases; encrypted vs plain database; ssl vs plain client-server channels; etc. Some tests require very specific combinations of these modes. We only support memory storage, lock concurrency control, networked database server, plain database over a plain channel. All tests that required a different configuration were skipped. We also skipped tests do not generate any update opportunity because they test client-only behavior.

The results in table I show that we mostly reused existing code, only modifying up to 2% of it. However, we do not support all available system tests. Still, if we assume that adding support for all available system tests requires a comparable amount of effort, supporting all H2 and CrossFTP system tests would require modifying 11% and 3% of the respective testing framework. Of course, these are worst case numbers. Still, we argue these values are acceptable.

C. Overhead

To measure the base overhead that any of the testing techniques impose, we used the benchmarks to measure the time H2 and CrossFTP take to finish a fixed-sized workload (TPC-C size *default* and FTP's download of 160 files of 1MB each, both using 4 threads) under two settings: Executing with

[7] https://mina.apache.org/ftpserver-project/index.html

Program	Regular (ms)	Observed (ms)	Overhead
H2	11,181	15,268	1.37
CrossFTP	9,008	17,806	1.98

TABLE II. OVERHEAD INTRODUCED BY ATTACHING AN EXTERNAL PROCESS AS AN OBSERVER FOR RUBAH. COLUMNS *regular* AND *observed* SHOW THE TIME TO COMPLETE A FIXED-SIZE WORKLOAD.

regular Rubah and with observed Rubah. Both versions do not perform any update. We ran each version 5 times. Table II shows the average completion time and overhead.

Running an updatable program while Tedsuto is attached to Rubah introduces significant overhead. Tedsuto is thus not applicable to performance sensitive tests. Still, the performance penalty was acceptable for the set of system tests we considered, even considering all the re-executions that our Tedsuto performs.

D. Intensive Update Testing: Practical Coverage

Intensive system testing supports precise update testing (considering most or all update opportunities) within a window specified by the tester using annotations. To measure how much these annotations reduce the total number of update opportunities that an intensive test would otherwise explore, we ran a version of our technique that counts all update opportunities during a test run, and how many of those are explorable according to the annotations we added.

Figure 9 shows the average results for 5 runs of this experiment. Each bar is a different test. Bars are horizontally sorted by the total number of update opportunities each test generates (light bars). Dark bars report the explorable update opportunities during that test. The difference between bars shows the reduction. Originally, each H2 test had in average 454 opportunities (3590 max) and each FTP test had in average 7 opportunities (108 max, bottom bar not fully shown). Only considering the annotated portions of each test reduced the average opportunities to 58 for H2 (188 max) and 4 for FTP (15 max).

Note that we report 110 tests for MINA in table I but 145 in Figure 9. This happens because we counted the tests statically and some tests dynamically repeat others on a different setting (e.g. repeating RETR/STOR/LIST in active/passive modes). H2 tests are not as easy to run separately as MINA's because earlier tests set results that later tests re-use. As a result, we had to repeat each file as a whole. Figure 9 thus shows the results per file for H2.

The manual effort that intensive update testing requires maximizes its applicability. These results show how that effort reduces the space of possible update opportunities to explore. It may be acceptable to consider all update opportunities for simple system tests, such as the ones on MINA'S FTP testing framework. However, in the general case and for more complex system tests such as H2's, the gains in annotating the interesting portions of the tests are clear.

E. Extensive Update Testing: Sampling Effectiveness

Extensive update testing explores update opportunities during system test, repeating the test until enough opportunities have been explored. This section describes an experiment

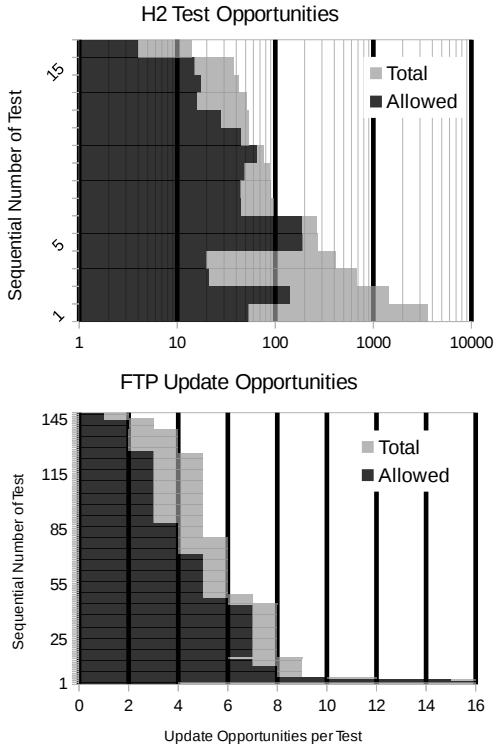

Fig. 9. Updates performed per test. Each bar represents a single test, its size represents how many update opportunities that test had, and the black part represents how many updates were tested. The vertical axis just counts the bars.

to compare how different techniques for selecting update opportunities detect known bugs.

The experiment starts with three profiling runs that do not perform any update, during which the tool gathers data about the possible update opportunities. Update opportunities are identified by the label of the last call to `Tedsuto.operation`. On multi-threaded scenarios, update opportunities are identified by the label on all threads.

The tool only considers update opportunities that happen during all three runs, and uses three different techniques to explore them: (1) *Random* points during the test run, (2) perform a number of updates (2 in this case) per opportunity starting from the *most common* and moving towards the least common, and (3) the same but from the *least common* to the most common.

We performed this experiment on CrossFTP and H2, introducing some errors to check if extensive update testing can find them. For CrossFTP, we manually skipped a conversion method that migrates the user name to its own field, as shown in Figure 2, which causes the server to reject a valid sequence of commands USER–PASS if an update happens in between. We also used the wrong update-point bug explained in the next section. For H2, we added the bug that Figure 3 describes, which causes a BEGIN statement to close the database after an update. All these bugs were tested using a single thread. For multi-threaded scenarios, we used the lock timeout bug described in the next section.

We used TPC-C and the FTP benchmark as system tests for

Bug	Random		Least Common		Most Common	
	Found	Faulty	Found	Faulty	Found	Faulty
CrossFTP						
TOTAL	1.33		10		10	
User/Pass	0	0	1	1	1	1
Wrong update	0	1	0	1	0	1
H2 1 Thread						
TOTAL	15.33		25		25	
Begin/Shutdown	2	3	2.33	3	2	4
H2 2 Threads						
TOTAL	41.67		25		25	
Wait for lock	1.6	16	0	5.67	0.67	10.33
H2 4 Threads						
TOTAL	50		25		25	
Wait for lock	24.67	46.67	12	22.33	21.67	25

TABLE III. COMPARISON OF THE DIFFERENT TECHNIQUES TO EXPLORE UPDATE OPPORTUNITIES.

H2 and CrossFTP, respectively. TPC-C was modified to issue an explicit BEGIN statement to start each transaction, thus creating a window for one of the bugs we tested. We than run each technique for 50 updates on each bug, in isolation.

Table III shows the results of this experiment. Rows titled *TOTAL* report the total number of different operations in which each technique explored performing updates. Columns titled *faulty* report how many of those can potentially display the bug if an update is performed at the right time and with the right data. Columns titled *found* report how many of the potential faulty operations actually displayed the bug during the test run. We repeated the experiment three times and the table reports the average values.

The data from CrossFTP shows that extensive update testing failed to detected the wrong update bug because the timing window for this bug is very narrow. Exploring update opportunities systematically, either starting by the least common or most common, could find the user/pass bug while random updates could not.

The FTP system test only had 10 different operations, and systematic exploration could reach full operation coverage. On H2, systematic exploration always tries 25 operations, each twice over 50 updates. Random exploration tries less operations by repeating more times the few operations it finds. By increasing the thread count, random exploration finds more operations than systematic exploration because the total number of available operations increases exponentially.

For H2, the begin/shutdown window is large enough for random updates to perform as well as systematic updates. On multi-threaded scenarios, random exploration outperforms systematic exploration by finding more bugs.

These results show that taking the effort to annotate existing tests generally improves the results. Even when random sampling outperforms systematic sampling, the information about what operations the test was performing at update time when a bug was detected allows the developer to understand what went wrong.

F. Bugs Found

This section describes the bugs we found using Tedsuto. We provide a description of each bug, the technique that found it first and how other techniques that confirmed it.

```
1  if (Rubah.isUpdating())
2    transferredOffset = saved.transferredOffset;
3
4  Rubah.update("transfer");
5
6  while (transferredOffset != file.size()) {
7    // transfer a block and increase the offset
8    try {
9      Rubah.update("transfer");
10   } catch (UpdateRequestedException e) {
11     saved.transferredOffset = transferredOffset;
12     throw e;
13   }
14 }
```

Fig. 10. Example adapted from CrossFTP that shows a badly placed update point on line 4.

Resource leak Retrofitting Rubah means turning all blocking I/O operations into non-blocking. A possible way of doing that is to use Rubah's API, that requires a selector as input for each blocking I/O operation. We created one selector per thread after each update without closing it before the next update. As a result, after a number of updates, the program reached the maximum number of file descriptors and terminated. This bug was found with control-flow reboots and no other technique could find it.

Wrong update point We retrofitted CrossFTP to support installing an update while transferring a file. Figure 10 shows how. After the update, the control migration resets the offset already transferred (line 2), which was saved before the update (line 11), and then reaches an update point to stop the control migration (line 4). An update that takes place after starting the transfer but before sending any data could reach the update point on line 4 without setting the state. That update point should be guarded by line 1. This bug was first discovered by control-flow reboots and confirmed by intensive update testing.

Internal data-races in Rubah Rubah had internal data-races on previously untested scenarios. For instance, installing an update while only some of the threads were released after an update. Control-flow reboots were able to find these data-races. No other technique could repeat them.

Lock Timeout H2 implements transaction isolation through row locks. When attempting to grab an already locked row, threads spin until the lock becomes available or the operation times-out. If an update happens at this point, the thread that holds the lock reaches an update-point, and thus stops executing, while other threads are waiting for the lock. The other threads will eventually time-out, fail the operation with a time-out, and only then reach an update-point. This bug this manifests as increased latency on some updates. It was first found by extensive updates, and then confirmed by intensive updates.

We originally detected this bug and fixed it after a laborious manual debugging process, in an ad hoc way, before the first Rubah submission. Providing a simple way to detect such bugs was part of our original motivation to perform this work.

New FTP Command (MD5) CrossFTP adds support for the MD5/MMD5 commands[8] in one of the versions we retrofitted.

[8]These commands allow a client to request the checksum of a remote file.

However, the server failed to detect that the command was available after the update because the map of available commands was not transformed during the update. This bug was fixed by adding two extra conversion methods to the update class that add the command to the map and re-load a properties file that describes how to format the reply for this command. This bug was only found by intensive update testing.

Batch FTP commands We retrofitted CrossFTP in a way that did not support receiving FTP commands in batch. When several commands were concatenated in a single message, CrossFTP would process the first and then wait for more commands directly on the network socket without checking if a command buffer was empty. This bug was only found by intensive update testing.

Slow SQL Query We originally retrofitted H2 to reach update points only between queries. A slow query can thus delay an update until it completes. We fixed this bug by adding an update point to the loop that iterates over rows and conditionally adds them to the result set. This bug was only found by intensive update testing.

Early update-point When fixing the previous bug, we added an update point to the top of the loop that iterates over all rows as Figure 4 shows, which causes the row being processed at the time of the update to be skipped. This issue can remain undetected if that row would be filtered out either way. We only detected it when an update interrupted the last iteration of this loop. After the update, the program never enters the loop and thus never reaches the same update point. This bug was found by extensive update testing and confirmed by control-flow reboots.

Discussion These results show that Tedsuto is effective at finding bugs. Six of the seven found bugs were completely new to us. We were able to isolate and reproduce these bugs using different techniques. When an intensive test found a bug by taking the nth update opportunity, we could reproduce that bug by re-running that test and taking the same nth opportunity. When an extensive test found a bug by taking an update opportunity on a particular operation (or combination of operations, in multithreaded cases), we could reproduce the bug by re-running the same extensive test and only take update opportunities during the same operations that had triggered the bug. Once the bug was found, we were able to re-use the test to check whether the fix actually worked as expected.

V. Related Work

Testing updates is a way of ensuring their correctness. The question of what constitutes a correct dynamic update has been the subject of prior work. Kramer and Magee [11] propose that updates are correct if they are "backward compatible," i.e., the updated program preserves all observable behaviors of the old program. Bloom and Day [12] observed that this is too restrictive because it forbids updates that fix bugs or add features. Gupta et al. [13] propose that an update is correct if the updated program eventually reaches some state of the new program. Hayden et al. [14] argue that any attempt to define update correctness generally is flawed as update correctness depends on the particular semantics of each updatable program. Tedsuto is applicable to whatever notion is chosen—backward compatible updates follow directly from existing system tests, and ones that change semantics are easily checked as well, as described in Section III-B.

Tedsuto is implemented for Rubah [6], a DSU system that requires the programmer to make manual changes to the program to support updating. Rubah's use of *update points* is critical to practical assurance of DSU, since allowing updates at nearly arbitrary program points—e.g., any time that changed classes are "inactive" [19], [21], [7], [22], [23], [2], [5], [4], [24]—would produce a huge blowup of update opportunities, and many more chances for failure. Many other DSU systems also address this issue by requiring update points [3], [2], [15], [16], [17], [18], [9], and prior work demonstrates their effectiveness [25].

Previous work on testing DSU [25] for the Ginseng DSU system for C programs [2] exhaustively enumerates all possible update opportunities that happen during a test run and systematically tries each one, repeating the test multiple times. However, this approach assumes that both tests and updated program are completely deterministic, which allows the framework to collapse neighbor update opportunities into a single one that has the same effects and thus reduce the overall number of update opportunities that need to be explored. While some work-arounds are possible, e.g. for handling the system time, this approach does not support multi-threading and is largely unrealistic in practice. By contrast, Tedsuto is carefully integrated into testing practices and places no restriction on the updatable programs.

Our approach of repeating tests to explore different update opportunities systematically is related to multi-threaded testing tools [26], [27] that explore a subset of all the possible potential thread schedules systematically by repeating each test for each schedule.

VI. Conclusion

This paper presented Tedsuto, a practical framework for testing Dynamic Software Updates (DSU) that is able to test all aspects of DSU, from installing new code to transforming the program state. Requiring little developer effort to annotate existing system tests, Tedsuto can find bugs induced by the update process that are dependent on the timing at which the update happens by automatically exploring different update opportunities during the execution of each system test. Tedsuto can check that unmodified features are still supported and modified features behave as expected after the update.

We implemented Tedsuto using Rubah, our previous system for updating Java applications, and we applied it to dynamic updates previously developed (and tested in an ad hoc manner) for the H2 SQL database server and the CrossFTP server— two real-world, multi-threaded systems. We found several update-related bugs in short order and at low cost. We believe Tedsuto is a promising step toward practical assurance for DSU.

References

[1] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at facebook," in *USENIX NSDI*, 2013. [Online]. Available: https://www.usenix.org/conference/nsdi13/scaling-memcache-facebook

[2] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for C," in *PLDI*, 2006.

[3] C. Hayden, E. Smith, M. Denchev, M. Hicks, and J. Foster, "Kitsune: efficient, general-purpose dynamic software updating for c," in *OOPSLA*, 2012. [Online]. Available: http://doi.acm.org/10.1145/2384616.2384635

[4] T. Würthinger, C. Wimmer, and L. Stadler, "Dynamic code evolution for Java," in *PPPJ*, 2010. [Online]. Available: http://doi.acm.org/10.1145/1852761.1852764

[5] S. Subramanian, M. Hicks, and K. McKinley, "Dynamic software updates: a VM-centric approach," in *PLDI*, 2009. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542478

[6] L. Pina, L. Veiga, and M. Hicks, "Rubah: DSU for Java on a Stock JVM," in *OOPSLA*, 2014. [Online]. Available: http://doi.acm.org/10.1145/2660193.2660220

[7] J. Arnold and M. F. Kaashoek, "Ksplice: Automatic rebootless kernel updates," in *EuroSys*, 2009. [Online]. Available: http://doi.acm.org/10.1145/1519065.1519085

[8] "Linux 4.0 live patching infrastructure," https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=1d9c5d79e6e4385aea6f69c23ba543717434ed70.

[9] M. Hicks and S. M. Nettles, "Dynamic software updating," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 6, pp. 1049–1096, November 2005.

[10] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA*, 2006. [Online]. Available: http://doi.acm.org/10.1145/1167473.1167488

[11] J. Kramer and J. Magee, "The evolving philosophers problem: Dynamic change management," *IEEE TSE*, 1990. [Online]. Available: http://dx.doi.org/10.1109/32.60317

[12] T. Bloom and M. Day, "Reconfiguration and module replacement in argus: theory and practice," *Software Engineering Journal*, 1993.

[13] D. Gupta, P. Jalote, and G. Barua, "A formal framework for on-line software version change," *IEEE TSE*, 1996. [Online]. Available: http://dx.doi.org/10.1109/32.485222

[14] C. Hayden, S. Magill, M. Hicks, N. Foster, and J. S. Foster, "Specifying and verifying the correctness of dynamic software updates," in *VSTTE*, 2012.

[15] K. Makris and R. A. Bazzi, "Immediate multi-threaded dynamic software updates using stack reconstruction," in *USENIX ATC*, 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855807.1855838

[16] R. P. Cook and I. Lee, "Dymos: A dynamic modification system," in *SIGSOFT*, 1983. [Online]. Available: http://doi.acm.org/10.1145/800007.808034

[17] C. M. Hayden, E. K. Smith, M. Hicks, and J. S. Foster, "State transfer for clear and efficient runtime updates," in *HotSWUp*, 2011. [Online]. Available: http://dx.doi.org/10.1109/ICDEW.2011.5767632

[18] L. Pina and J. Cachopo, "DuSTM - Dynamic Software Upgrades using Software Transactional Memory," INESC-ID Lisboa, Tech. Rep. 32/2011, 2011.

[19] Oracle(TM), "Java SE 1.4 Enhancements," http://download.java.net/jdk8/docs/technotes/guides/jpda/enhancements1.4.html.

[20] J. Armstrong, *Programming ERLANG: software for a concurrent world*, ser. Pragmatic programmers, 2007.

[21] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew, "Polus: A powerful live updating system," in *ICSE*, 2007.

[22] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz, "Opus: Online patches and updates for security," in *SSYM*, 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251398.1251417

[23] T. Ritzau and J. Andersson, "Dynamic deployment of Java applications," in *Java for Embedded Systems Workshop*, 2000.

[24] A. Orso, A. Rao, and M. J. Harrold, "A technique for dynamic updating of java software," in *ICSM*, 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=876882.879757

[25] C. Hayden, E. Smith, E. Hardisty, M. Hicks, and J. Foster, "Evaluating dynamic software update safety using efficient systematic testing," *IEEE TSE*, 2012. [Online]. Available: http://www.cs.umd.edu/~mwh/papers/dsutesting-journal.pdf

[26] M. Musuvathi, S. Qadeer, and T. Ball, "Chess: A systematic testing tool for concurrent software," Microsoft Research, Tech. Rep. MSR-TR-2007-149, November 2007. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=70509

[27] W. Pugh and N. Ayewah, "Unit testing concurrent software," in *ASE*, 2007. [Online]. Available: http://doi.acm.org/10.1145/1321631.1321722