Tedsuto: A General Framework 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 patching *running* programs, to fix bugs or add new features. DSU avoids the downtime of stop-and-restart updates, but creates new risks—an incorrect or ill-timed dynamic update could result in a crash or misbehavior, defeating the whole purpose of DSU. To reduce such risks, dynamic updates should be carefully tested before they are deployed. This paper presents Tedsuto, a general testing framework for DSU, along with a concrete implementation of it for Rubah, a state-of-the-art Java-based DSU system. Tedsuto uses system-level tests developed for the old and new versions of the updateable software, and systematically tests whether a dynamic update might result in a test failure. Very often this process is fully automated, while in some cases (e.g., to test new-version functionality) some manual annotations are required. To evaluate Tedsuto's efficacy, 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, multithreaded systems. We used three large test suites, totalling 446 tests, and we found a variety of update-related bugs quickly, and at low cost.**

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, or 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, which is valuable for in-memory databases, gaming servers and other systems that rely on the relatively low expense and high performance of commodity RAM to maintain large in-memory datasets. 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]. The research community has developed general-purpose DSU systems that support release-level changes to substantial applications [2], [3], [4], [5], [6], [7].

DSU is not a panacea; it must be done with care. Program code assumes the execution state adheres to a certain format, so changing the code at run-time requires corresponding changes to the execution state, both *control* (i.e., thread call stacks and program counters), and *data* (i.e., contents and format of heap objects). As described in Section II, it is often the programmer's responsibility to define such state changes, e.g., by providing *migrations* that map between the old representation of an object and a new one, and between an old control state and a new one.

```
1  Object g = null;
2  void f( ) {
3    process();
4    cleanup();
5  }
6  void process() {
7    ...
8  }
9  void cleanup() {
10   g = new Object();
11   ...
12   g.hashCode( );
13 }
```

Version 0

```
1  Object g = null;
2  void f( ) {
3    process();
4    cleanup();
5  }
6  void process() {
7    g = new Object();
8    ...
9  }
10 void cleanup() {
11   ...
12   g.hashCode( );
13 }
```

Version 1

Fig. 1. Java program update example. Code changes are highlighted.

One challenge with writing migrations is *timing*: the migration code may assume certain invariants, and these must hold for all the circumstances under which an update could take place. Mistakes in migrations or their timing can result in crashes, corruption, and other misbehavior. For instance, consider the program example introduced in Figure 1. Version 1 moves the initialization code from line 10 to line 7. Suppose, while running version 0, we dynamically update the `process` and `cleanup` methods after returning to `f` from the call on line 3. We specify no data or control migration because we observe that no data formats or running methods (i.e., `f`) have changed. However, once the program resumes, `f` will call version 1's `cleanup`, which crashes on line 12 because the global variable `g` was not initialized by version 0's `process`.

This paper presents *Tedsuto*,[1] a framework for testing that dynamic updates are correct before we deploy them on live systems. As described in Section III, the basic idea is simple: Tedsuto runs existing system tests many times each, systematically exploring what happens when an update is applied at different moments during the test's execution. Each such moment when an update can be applied is an *update opportunity*. Which moments constitute opportunities depends

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

on the DSU system under test. Many systems apply updates just prior to calling a changed method [4], [5], [8], [9]; as such, for the example in Figure 1, method `f` has two update opportunities: before calling method `process`, and before calling method `clean`. A system test that passes when running either software version should also pass when being dynamically updated, no matter which update opportunity is taken. Tedsuto systematically tries different opportunities; for our example, it would discover that applying the update at the second opportunity in `f` would fail.

Tedsuto can also use system tests that pass only for the new version, e.g., because they exercise a new feature or bugfix. In this case, the tester simply indicates (typically via a one-line annotation in the test code) the last possible moment at which an update can occur while still passing the test; Tedsuto will only explore update opportunities to this point.

Finally, Tedsuto employs two novel techniques that test particular aspects of the updating process: *update-point synchronization* ensures that preparing the application state for an update does not lead to hangs or wrong behavior (e.g., due to synchronization problems), and *control-flow reboots* ensure that DSU-specific code in the old program that supports control-state transformation works properly.

We have applied our implementation of Tedsuto, described in Section IV, to the Rubah DSU system for the Java programming language [6]. Using Tedsuto required modifying Rubah in a straightforward manner; many other state-of-the-art DSU systems can be handled just as easily. We evaluated Tedsuto's ability to find DSU-related errors by using it to test dynamic updates previously developed (and tested in an ad hoc manner) for H2 [10], an SQL database server, and CrossFTP [11], an FTP server — two real-world multi-threaded systems. As described in Section V, with three suites of system tests—two for H2 and one for CrossFTP—Tedsuto was able to find 8 new update-related bugs. Adapting the test suites required little effort: 17 lines in total to support 17 update-specific tests, and no manual effort to support the remaining 446 backwards-compatible tests. Our experimental evaluation found that update-synchronization and control-flow reboots were particularly effective in finding errors.

Prior work, discussed in Section VI, has briefly considered the problem of dynamic software update testing. Tedsuto represents a substantial step forward. In short, we make the following contributions:

- Present the design of the first testing framework for DSU that is generalizable to state-of-the-art systems, re-uses existing system tests with low effort, and is able to test both backwards-compatible updates and new features;
- Introduce five novel and highly effective techniques for testing updates;
- Describe how to implement Tedsuto for state-of-the-art DSU systems and provide the details of an implementation for Rubah;
- Evaluate Tedsuto with an extensive experimental evaluation, by using three large test suites with two production-ready servers, and show that Tedsuto is effective by reporting new update-related bugs.

```
class Session {
  User user;
  String userName; // Added in version 1
  // Invariant:  userName == user.name
}
```

Fig. 2. Example of a class update

We believe Tedsuto is a promising step toward practical assurance for real-world DSU.

II. The Need for DSU Testing

This section presents background on dynamic software updating, and identifies ways in which dynamic software updates could fail. Then it shows how existing unmodified system tests can be used to find update-related errors.

A. Dynamic Software Updating Failures

Dynamic software updating systems work by accomplishing two tasks. First, they *load code* into a running program to add to and/or replace existing classes, methods, etc. Second, they *migrate* the running program's *execution state* to an equivalent form that is compatible with the newly loaded code. 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. Depending on how the code changed, a migration may need to convert data representations, e.g., when the new code expects a hash table where the old expected a tree, and control states, e.g., when the new code refactors an old function into two. Control and data migrations are typically specified by the programmer, perhaps with some automated assistance [2], [3], [6], [12]. For instance, in Rubah, described in Section IV-A, data migrations are specified using a special "update class," and control migrations are specified indirectly by changing the way the program resumes after an update.

An important element of DSU is *timing*: The effect of a dynamic update's data and control migrations may differ depending on when the update is performed. To reduce the chances of update-related crashes, DSU systems often restrict the moments during a program's execution at which an update can take place. We call these moments *update opportunities*. Many DSU systems limit update opportunities to program points at which changed code is not *active*, i.e., not referenced from any thread's callstack [4], [5], [8], [9]. As explained in Section I with the example in Figure 1, this limitation still allows programs to crash due to the update process. Other DSU systems, including Rubah, limit opportunities to moments when each thread's execution has reached a programmer-specified point [2], [3], [6], [12], [13].

Unfortunately, programmers will make mistakes when changing their program to support updating, and/or when writing data or control migrations for particular updates. These mistakes can manifest as a hang, crash, or other misbehavior when a dynamic update is applied.

```
1  // Set-up
2  FTPClient c = new FTPClient();
3  // Test
4  c.sendVersion(0);
5  c.USER("user");
6  c.PASS("wrong");
7  c.PASS("right");
8  assert(c.isConnected());
9  assert(c.isLoggedIn());
10 // Tear-down
11 c.QUIT();
```

Fig. 3. Simple system test for an FTP server

Consider the example shown in Figure 2, adapted from the CrossFTP server [11]. The updated program reads the name of a user trying to authenticate from the field `userName`, whereas the old program used field `user.name`. A dynamic update must transform the running version's data from the old representation of class `Session` to the new; in particular, it should copy the value of `user.name` into the new `userName` field, to establish the invariant shown in line 4. Suppose the programmer forgets to specify such a migration. The new code will behave incorrectly when accessing field `userName`; e.g., it will crash with a null-pointer exception, deny access to valid users, or allow access to invalid users.

As another example, recall the update shown in Figure 1. Here, a control migration must be used to map the program counter of the taken update opportunity to the appropriate location in the new version. The mapping is trivial for function `f`: Lines 3 and 4 in version 0 map to lines 3 and 4 in version 1, respectively. For function `process`, line 7 in version 0 maps to line 8 in version 1, with the caveat that global variable `g` also needs to be (data-)migrated for the update to be correct. However, suppose the programmer specifies a mapping from line 10 in version 0 to line 7 is version 1 (as that is the line of code that moved). Doing so results in incorrect behavior because the program counter will be moved to version 1's `process` method, but the (version 0) `process` method was already executed. As a result, messages may get duplicated.

In short, the behavior of a dynamic update depends on the control and data migrations specified by the programmer, and the update opportunity at which these migrations take effect. Without care, a dynamic update may produce incorrect results.

B. Testing Dynamic Updates

How can we avoid update failures? Gupta et al. have shown that, in general, establishing that an update is correct is undecidable [14]. Normal program properties are undecidable too (e.g., termination), so typical software development uses tests to ensure that a software system will behave as it should when deployed. Likewise, we can use testing to give us confidence that a dynamic update, when applied in deployment, will behave as expected. We can test the correctness of a dynamic update by running a test and checking the program's behavior before, during, and after an update. Since DSU is a whole-program operation, we can perform DSU tests by starting from *system tests*, which check the end-to-end behavior of a program. For instance, consider the test in Figure 3, and suppose that both the old and new versions (when run without updating) pass the test. It is possible that, due to bugs in data/control migrations or other program changes, dynamically updating from the old to new version during the test run will cause the test to fail. For instance, consider the update to the `Session` class of Figure 2, and suppose that the data migration fails to initialize the `userName` field. Dynamically updating the server on any of the update opportunities generated by lines 5 or 6 of the test will cause it to fail,[2] since these commands cause the FTP server to read the `userName` field. Updating the server at other opportunities during the test run will pass because the new code either initializes the field (when handling the request at line 4) or never refers to it again (when handling requests at line 7 and 11).

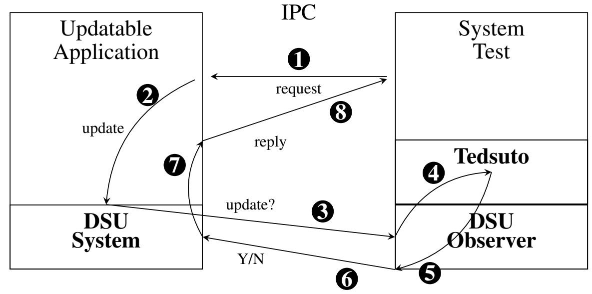

Fig. 4. Architecture of Tedsuto.

In summary, it is possible that a test passes when the target application is updated at some update opportunities, but not others. As such, a testing strategy for dynamic updates should explore the behavior of different update opportunities systematically. This idea provides the basis for the design of Tedsuto. But Tedsuto goes further, considering tests that do not necessarily pass for both versions, and it defines techniques that can test aspects of an update more efficiently than applying the update end-to-end. We describe Tedsuto in detail, next.

III. TEDSUTO

Tedsuto is a framework for systematic testing of Dynamic Software Updates. This section discusses Tedsuto's architecture and describes the kinds of testing methods it enables.

A. Architecture

Figure 4 shows Tedsuto's architecture. The updatable application and the system test run in separate processes that communicate through Inter-Process Communication (IPC). During its execution, the system test will interact with the updatable application, e.g., by sending it service requests (1). While processing each request, the execution of the updatable application triggers several update opportunities (2). Tedsuto assumes that the DSU system generates a small number of update opportunities per interaction. This may not be true for all DSU systems; we discuss in Section IV-C how Tedsuto can still be used with such systems. For now, let us assume that each interaction generates a small number of opportunities.

[2] Each test-server interaction may generate several update opportunities.

```
void allowUpdates();
void disallowUpdates();
void ensureUpdated();
void operation(Object threadID, String label);
```

Fig. 5. Tedsuto's API for adapting system tests.

A novel part of Tedsuto is that it employs an *update observer* that is queried at every update opportunity to decide whether to perform an update (3). The update observer, located in the same process as the system test, then notifies Tedsuto about the opportunity to perform an update (4). Based on the information that Tedsuto has about the system test in execution, it tells the observer whether to take the current opportunity, and what sort of update to perform (5). The update observer then sends the decision back for the DSU system (6) before returning back to the updatable application (7), which in turn sends the reply back to the system test (8), which can then continue.

The remainder of this section details the sorts of update tests that this architecture enables. They are differentiated by the decision process used at update opportunities (e.g., exhaustive or operation-specific), the kind of system test being performed (e.g., backwards-compatible or version-specific), and whether to perform a full update when the opportunity arises, or a more localized test (either an update synchronization or a control-flow reboot).

B. Exhaustive Tests

During a deterministic test, update opportunities can be matched between different re-executions by the order in which they happen. For instance, consider two executions of the test shown in Figure 3 on a server that performs updates only after FTP commands. Regardless of the update opportunity that Tedsuto explores, the second update opportunity will happen on line 5 in both executions. This allows Tedsuto to re-execute the test and systematically explore all update opportunities.

Of course, not all update opportunities need to be explored. In particular, in the example we are following, the update opportunities triggered while setting-up the test (lines 1–2) and tearing-down (lines 10–11) are not relevant to the test and may be skipped. Other FTP tests may require an authenticated user (e.g. to test file permissions) and move lines 4, 5, and 7 to the test set-up.

The developer can adapt existing system tests to skip exploring uninteresting update opportunities by retrofitting the tests with Tedsuto's API, shown in Figure 5. In particular, they can surround lines 3–9 in Figure 3 with calls to `allowUpdates` and `disallowUpdates` to explore opportunities only during the main body of the test. Tedsuto automatically supports tests written in the popular JUnit framework and skips all opportunities that happen during the `setUp` and `tearDown` methods. Section V describes how we evaluated Tedsuto using two JUnit frameworks without any effort to integrate them with Tedsuto's API.

```
AtomicInteger count;
int MAX; // Maximum clients allowed

// Launch several threads that do:
FTPClient c; // Thread-local
void run() {
  while (!stop) {
    Tedsuto.operation(c, "USER");
    c.USER("user");
    Tedsuto.operation(c, "PASS");
    c.PASS("right");
    assert(count.incAndGet() < MAX);
    Tedsuto.operation(c, "QUIT");
    C.QUIT();
    count.dec();
  }
}
```

Fig. 6. Example that tests the maximum number of connected clients to an FTP server. The highlighted code represents the code added to integrate this test with Tedsuto.

C. Update-Specific Tests

So far, we have assumed that system tests should pass both program versions when not performing an update. However, this may not always be the case. For example, the new version may add support for new features and fix bugs. As such some tests for the new version may not pass on the old version.

For instance, let us consider an example taken from version 1.0.7 (or, v_0 for short) to version 1.0.8 (v_1) of CrossFTP. When a client fails authentication in v_0, CrossFTP closes the connection at the first wrong password attempt. In v_1, CrossFTP allows the client to retry another PASS command reusing the same connection. The system test shown in Figure 3 passes in v_1 but fails in v_0. This is an important test because it checks that a new feature works as expected; we would like to adapt it so we can confirm that the dynamically updated program supports this new functionality.

Let us assume that the test generates one update opportunity per line on lines 4–7. Updating at either of the first two update opportunities, after lines 4 and 5, passes the test. Waiting for any update opportunity afterwards results in the connection being closed at line 6 and failing the test: Tedsuto performs the update too late.

The developer can state the latest point in the test execution at which an update can be performed using method `ensureUpdated` from Tedsuto's API, shown in Figure 5. In this case, the developer would add a call to this method after line 5. We call this annotated test an *update-specific test*.

Oftentimes, update-specific tests start as existing tests written for the new version, which the developer then adapts by adding annotations or small code changes. Section V reports the effort required to write 17 update-specific tests for 2 updatable applications; mostly adding one single line of code per test.

D. Operation-oriented Testing

Exhaustive tests and update-specific tests match update opportunities between different executions, which means they only apply to deterministic tests. Tedsuto supports multi-threaded and nondeterminstic tests by employing a notion of coverage that is based on the high-level operations carried out by the test. To see this by example, consider the test in Figure 6, ignoring the highlighted code for now. This test uses multiple threads to check whether the server respects the limit on the number of authenticated clients. The test is composed of three high-level operations: (1) Sending the USER command; (2) sending the PASS command, thus becoming authenticated; and (3) sending the QUIT command to reset the connection for another iteration of the test. The developer annotates the test with information about the operations, exposing them to Tedsuto through method `operation` on its API. In this case, it means adding lines 8, 10, and 13, highlighted in Figure 6.

Method `operation` takes two arguments. The second one is simply a label that distinguishes the operation from all others. The first is some object that identifies the thread in the test. In this case, each thread uses a dedicated `FTPClient` object to interact with the server; such an object identifies each thread unambiguously. With this information, Tedsuto can reason about *combinations of opportunities* during multi-threaded testing. In this case, for 2 threads, Tedsuto can explore the following combinations: USER/USER, USER/PASS, USER/QUIT, PASS/PASS, PASS/QUIT, and QUIT/QUIT. Tedsuto can now rerun the same test repeatedly until all combinations are explored. Currently, Tedsuto does not force any particular thread scheduling; we plan to explore Tedsuto in this direction in the future.

Operation annotations are useful even for single threaded tests: Essentially, they suggest to Tedsuto to take one update opportunity per annotated operation, even if many more are available. This gives the tester a way to shrink the testing space, taking advantage of domain knowledge.

E. Update-point Synchronization

Most DSU systems require all threads to *synchronize* before performing an update [2], [3], [4], [5], [6], [12]. That is, each thread should reach some known program point (e.g., a function call, a thread/GC safe point, or a manually annotated point). When all threads reach such *update points*, the DSU system performs the update, resuming all threads afterwards.

For instance, consider the code in Figure 6. Consider that there are several threads executing the loop and that the FTP server internally uses a lock to implement a maximum number of clients. Consider also that one thread acquires the lock and blocks for an update. In the meantime, another thread blocks waiting for the lock. At this point, the program is in deadlock: No thread makes progress and the update never gets performed. And this happened because an update became available; regular program operation would never result in such a deadlock.

Tedsuto finds this class of update-related bugs by simply requiring all threads to synchronize at all update opportunities during a system test, just to release them immediately after. We call this technique *update-point synchronization*. It allows Tedsuto to explore all the update opportunities in a single test execution, without performing an actual update. Update-point synchronization proved very effective: It found 3 of the 8 new update-related errors that we found using Tedsuto.

F. Control-flow Reboots

In Section II we discussed an example of control migration, performing the update that Figure 1 shows while modified code is active. Some DSU systems simply forbid such updates [2], [5], providing a simpler update model at the cost of flexibility. More recent DSU systems—UpStare [12], Kitsune [3], and Rubah [6]—overcome this limitation and provide support for updating active code. These systems require the programmer to migrate the control state of the program between versions, effectively mapping PC positions and stack frames, which is an error-prone proposition.

Kitsune and Rubah implement control migration by stopping all active threads and then restarting them in the new-version code. As shown in Figure 7, described in the next section which presents our implementation of Tedsuto for Rubah, the control migration code is effectively embedded in the startup code of each thread. This code, as it executes, regenerates the stack by taking an alternative path, provided by the programmer, to the one it would normally take during startup. If this alternative startup code is erroneous, it could leave the updated program in an incorrect control state.

It turns out we can test this migration code without performing a complete update. In particular, Tedsuto can initiate a *null update* by performing an update synchronization, and then "rebooting" each thread, causing it to restart. But instead of restarting at the new version, we restart it at the current version, and thereby test this alternative startup path. If the control migration code is correct, the program will return to the state it was in before the null update, and things will proceed as expected. Pleasantly, this technique requires just a single test run to reboot at each update opportunity, rather than having to run the whole test many times, once per opportunity. Control-flow reboots are very effective: We found 2 of the 8 new update-related errors using it; Section V presents all the details.

IV. Implementation

We have implemented Tedsuto for Rubah [6], a DSU system for dynamically updating Java programs. Therefore, in this subsection we present some background on how Rubah works and how the programmer specifies a dynamic update. We then explain how the concepts of Rubah map to Tedsuto.

A. Rubah

To implement DSU, Rubah uses a strategy called *whole program updates*, pioneered by Makris et al.'s UpStare system [12] and also adopted by Kitsune [3]. In a nutshell, the scheme has three parts. First, the code that corresponds to the new version (constituting the "whole program") is dynamically loaded. This takes place only after all program threads reach an *update point*, which is a code location designated (in advance) by the programmer [13]; each time an update point is reached

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

Fig. 7. Example adapted from H2 TcpServerThread featuring (gray highlighted) logic related to update points and control migration.

during execution constitutes an update opportunity. Once the code is loaded, control migration is performed by unwinding the stacks of all active threads and then "rewinding" them, starting from the new code, until they reach the same logical update point they had reached in the old code. Third, data changed by the update is migrated as it is accessed during the new program's execution. Rubah is implemented using bytecode transformation; no changes to the VM are required.

In Rubah, control migration is handled by the programmer. This is done by writing the startup code of each thread to be cognizant of an "updating mode". When starting in this mode, the thread is being rewound following an update; otherwise it is starting from scratch. Rubah performs data migration automatically, following a manually-defined transformation logic: The programmer specifies an *update class* that defines how a new version instance is initialized, given an old version instance. And, as mentioned already, the programmer is responsible for designating update points at which threads synchronize prior to the update taking place. All of these choices are opportunities for errors that can be uncovered during testing. Therefore, to make the process more clear, we present an example that shows all three in action.

Example. We have used Rubah to dynamically update the H2 database management system. Figure 7 shows a simplified version of a connection-handling method from H2. The changes made to support DSU are highlighted—ignore them for now. The method starts by parsing the client data and negotiating the protocol parameters (lines 7–11). Then, it enters a loop (lines 15–26) that reads each client command (line 18), executes it in method `process` (line 19), and sends the response back to the client (line 20). The server keeps state about the client using the `session` object, declared on line 2.

Note the complex handling of exceptions, typical in server methods. The server sends recoverable exceptions back to the client (line 22), and logs non-recoverable exceptions (line 30). A `finally` block ensures that the connection is closed when the server method exits (lines 31–37).

Update Points. The programmer specifies update points as calls to method `Rubah.update`. This method will simply return, doing nothing, when an update is not available. Otherwise, it initiates (or continues) the update process, as described below. 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 simplifies writing the update class. 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 logically equivalent. For the example in Figure 7, the code related to update points is in gray. An update point is placed on line 17.

When an update is available, calling `Rubah.update` throws an `UpdatePointException` to unwind the calling thread's stack. 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 manually propagate it (line 28). 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 32). When all threads have been paused after reaching update points, the new code is loaded, and the control flow and data migration process is initiated.

Control migration. Rubah restarts each paused thread from its (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 in Figure 7, lines 7–11 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 (i.e., in "updating mode"). In our example, line 6 guards the initialization code with a call to `Rubah.isUpdating` which returns `true` if called while performing the control migration and `false` otherwise.

Data migration. Prior to restarting each thread, Rubah initiates *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 according to the logic on the update class, so that it work with the new version's code. Rubah provides a tool that generates a stub update class by analyzing two versions and matching fields by owner class name, field name, and field type. The generated code automatically copies each matched field; the programmer must specify what to do for unmatched fields. In the example shown in Figure 2, Rubah automatically copies field `user` and leaves a comment reminding the developer that field `userName` was not matched.

B. Tedsuto for Rubah

Tedsuto requires the target DSU system to provide support for an update observer, as we explained in Section III-A. Then, at each update opportunity, Tedsuto interacts with the observer to decide if it should perform an update or not. In Rubah, update opportunities map directly to calls to method `Rubah.update`, which we redirect to the observer process. The observer process performs a full update for exhaustive and update-specific tests. For update synchronization tests, the observer initiates quiescence, but then allows threads to continue in the same version without throwing an `UpdatePointException`. For control-flow reboots, the observer initiates quiescence and then restarts each thread in updating mode (without updating the code) to test that the control migration code does not incorrectly corrupt the program's state.

C. Tedsuto for other DSU Systems

Although the current implementation of Tedsuto targets Rubah, Tedsuto can be applied to other update systems with explicit update points, which are among the most practical systems yet built and evaluated. These include Kitsune [3], Ekiden [15], UpStare [12], and Ginseng [2]. Exhaustive tests, update-specific tests, and update synchronization tests would all apply; for Kitsune, control-flow reboots would as well.

Some systems define update points *implicitly*, rather than explicitly [4], [5], [8], [9]. For example, updates may be permitted at any point as long as a changed function or method is not active. All testing techniques would also apply to these systems (with some engineering effort), with the exception of control-flow reboots. In particular, for tractability, Tedsuto should not test updates at all (implicit) opportunities, but at a representative set of opportunities. To compute this set without loss of effectiveness we can use a technique from Hayden et al. [16] which determines, through program analysis, which update opportunities would have provably the same outcome for a given update, i.e., the updated program would execute the same code. Hayden et al. found that for typical updates, the number of opportunties can be reduced by between 87% and 95%. To work with Tedsuto, we would simply have to make these opportunities manifest in the program, and then coordinate them with the Tedsuto observer.

V. Experimental Evaluation

We evaluated Tedsuto by testing two programs for which we previously added support for DSU through Rubah: *H2*, an SQL database; and *CrossFTP*, and FTP server. We show that Tedsuto requires *low effort* to use existing tests: We used 3 test suites, comprising a total of 446 backwards compatible tests and 17 new update-specific tests that we adapted from existing tests by changing 5% of the code on 5 files. We also used a complex benchmark for H2 as a multi-threaded system test, adapted to use Tedsuto by changing 0.5% of its code. We also show that Tedsuto is *effective*: We report 8 new update-related bugs that we found. Finally, we show that Tedsuto is *efficient*: Update-point synchronization and control-flow reboots can be used as development time tools; exhaustive testing requires more time to complete each test suite but can still be used to ensure that a new release can be correctly deployed as a DSU. This section describes the experiments in detail.

A. Experimental Configuration

All experiments that we describe in this section were run on a machine equipped with an an Intel Xeon E31280 machine, 3.5 GHz CPU (8 logical cores, 4 physical), 16GB of RAM, with GNU/Linux Ubuntu 14.04 (kernel 3.13.0-39). All tests were conducted with Oracle's JVM version 1.7.0_79-b15 (HotSpot version 24.79-b02).

We performed the experimental evaluation using two applications previously adapted to support DSU through Rubah [6]: H2 [10], which is a mature, SQL DBMS written in about 40K lines of Java; and CrossFTP [11], which is an FTP server written in about 18K lines of Java. We updated H2 from version 1.2.121 to version 1.2.123, spanning version 1.2.122; and CrossFTP from version 1.07 to version 1.11, spanning versions 1.08 and 1.09. We manually collapsed all releases into a single update.

We evaluated Tedsuto with a total of 3 test suites and 1 performance benchmark. We used 2 test suites for H2: (1) The tests that ship with it, called *H2-test* (14K LoC, 39 tests); and (2) a JUnit test suite taken from another Java SQL database called *HSQLDB* [17] (15K LoC, 300 tests). We adapted a JUnit test suite for Apache's MINA FTP server [18] to use with CrossFTP, called *FTP-test* (2.7K LoC, 107 tests). Finally, we used the *TPC-C* performance benchmark (8K LoC) shipped with the DaCapo benchmark suite [19] as a multi-threaded system test for H2, given that TPC-C verifies the invariants of the benchmark at the database level after its completion. All values reported are the average of 3 executions.

Adapting each test suite to run with only a subset of its tests (e.g., to ignore unit tests) independently required some effort, but this is not directly related with Tedsuto.

B. Manual Effort

Backward-compatible system tests (i.e., those that pass the old and new software versions) can be used with Tedsuto with no extra effort. Update-specific tests do require some manual adjustment (e.g., to indicate the latest point at which an update can be applied). Operation-oriented testing also

TABLE I
EFFORT REQUIRED TO WRITE UPDATE-SPECIFIC TESTS. EACH TEST WAS FIRST EXTRACTED FROM ITS ORIGINAL SUITE—WE REPORT NUMBER OF TESTS PER EXTRACTED FILE. TPC-C IS A BENCHMARK THAT PERFORMS 8 DIFFERENT OPERATIONS, WHICH WE MANUALLY ANNOTATED.

Test	Program	Extracted		Modified
		LOC	Tests	LOC
Login	CrossFTP	17	2	2
MD5	CrossFTP	128	10	10
DROP admin	H2	36	1	1
SELECT	H2	32	1	1
SCOPE_ID	H2	108	3	3

Benchmark	Program	Extracted		Modified
		LOC	Operations	LOC
TPC-C	H2	8231	8	42

TABLE II
TIME REQUIRED TO RUN EACH TEST SUITE TO COMPLETION UNDER SEVERAL TESTING CONFIGURATIONS. *Baseline* MEANS THAT TEDSUTO DOES NOT PERFORM ANY UPDATE. *UP-S* MEANS UPDATE-POINT SYNCHRONIZATION AND *CF-R* MEANS CONTROL-FLOW REBOOTS.

Suite	Original time	Tedsuto time		# Opportunities
		Baseline	UP-S + CF-R	
HSQLDB	4s	6s	216s	1905
H2-test	12s	19s	2541s	23651
FTP-test	6s	13s	29s	257

requires annotating the operations. Table I shows the effort required to extract tests that fail on the old version, and adding annotations to turn them into update-specific tests. It also tabulates the effort required to identify operations performed by the TPC-C benchmark. As we can see, the effort is very low, we modified under 0.5% of the total number of lines.

Each update-specific test checks whether a feature, introduced by the new version, works as expected after the update: *Login* checks that the same connection supports several login attempts without dropping; *MD5* checks the MD5 command; *DROP admin* checks that an administrator can delete herself; *SELECT* checks a particular idiom of select queries with a select sub-query; and *SCOPE_ID* checks IDs automatically generated by triggers.

C. Performance

Moving the decision whether to explore updates to another process requires an IPC at every update opportunity, which adds performance overhead. To measure that overhead, we first executed all test suites to completion without Tedsuto. Then we computed a *baseline* overhead which uses Tedsuto, but without performing any updates. We also measured the overhead when performing an update synchronization and control-flow update at every possible update opportunity during a test run. Table II shows the results together with the number of update opportunities per test suite.

Table III reports the time to perform exhaustive testing on the HSQLDB and FTP-test suites. For these runs, we only performed updates at opportunities that occurred after the set-up phase, and before the start of the clean-up phase for each JUnit test. The table shows how many total opportunities were explored, how many were avoided, and the overall time to complete each suite.

TABLE III
TIME REQUIRED TO EXHAUSTIVELY TEST EACH JUNIT TEST SUITE AND NUMBER OF UPDATE OPPORTUNITIES GENERATED AND EXPLORED. EACH EXPLORED OPPORTUNITY REQUIRES AN INDIVIDUAL TEST RUN.

Suite	Time	# Opportunities	
	(sec)	Total	Explored
HSQLDB	16374	77521	2431
FTP-test	1654	696	408

The H2-test suite generates an enormous number of update opportunities and has multi-threaded tests. We thus did not perform exhaustive testing with it. We did, however, perform operation-oriented testing with the H2-test suite. Operation-oriented testing can be configured with a budget of update opportunities to explore per test, which allows the developer to control how long tests take. However, we discovered that our other testing techniques found many more bugs, more quickly, as we discuss further in Section V-E.

Given a test suite with small, modular, and deterministic tests, such as HSQLDB and FTP-test, the low cost of update-point synchronization and control-flow reboots allows developers to use these two techniques to quickly find bugs during development of a new version. Exhaustive testing has a higher cost that forbids using it during development, but still low enough to be used for every version deployed as a DSU. Performing update-point synchronization and control-flow reboots can also be used with larger, non-deterministic tests, such as H2-test, with similar costs to exhaustive testing.

When performing exhaustive testing, each re-execution required restarting the target program from scratch, including launching a new JVM instance. This happens because Rubah does not support reverting updated code back to its old version. As a result, each re-execution of a test on the FTP-test suite took around 4.5 seconds; 3 of which just launching a JVM and starting the CrossFTP server; and around 1 second performing the update. The test itself took the remaining 0.5 seconds. Time ratios for the HSQLDB suite are similar.

A possible solution for this problem is to launch several tests on the same program version, interacting with the same target server; stop all at the nth opportunity; perform an update when the last test reaches the nth opportunity; allow all to complete; and repeat for the n+1th opportunity. This would require a single execution for all tests for each opportunity explored. We implemented an early prototype of this idea and noticed that the overall time to perform exhaustive testing reduced drastically. However, it required extensive changes to the original test suite so that tests run concurrently would not interact (we never got it to work correctly). We leave this for future work.

D. Bugs found

When developing dynamic updates for H2 and CrossFTP to use Rubah [6], we performed extensive manual testing and debugging. Despite this, applying Tedsuto to these programs revealed 8 new bugs, all of which would have serious consequences (discussed below) if they manifested during deployment. Table IV reports the name of each bug together

TABLE IV

BUGS FOUND, GROUPED BY TECHNIQUE AND TEST SUITE. *a)* AND *b)* GROUP SEVERAL BUGS IN RUBAH; WE LIST THEM MULTIPLE TIMES.

	H2-Test	HSQLDB	FTP-test
Update-Point Sync	a), d) , e)	a)	a)
Control-Flow Reboots	b)	b)	b), c), h)
Update-Specific	—	—	g)
Exhaustive	—	f)	—

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

Fig. 8. Example adapted from CrossFTP that shows a badly placed update point on line 4.

with the test suite and the technique that found it most quickly. In several cases, exhaustive and operation-oriented testing found the bug, too. We discuss why each technique found each error at the end of this sub-section.

In the following we describe each bug in detail:

a) **Internal Data Races in Rubah:** Rubah had internal data races that would manifest only in rare circumstances. For instance, when launching a thread just before an update took place, that thread would not stop for the update but keep executing while Rubah performed program-state transformation. The update would eventually crash due to old code accessing transformed data. Another example is when performing two updates in tight sequence, the second update could start before all threads finished control migration for the first update. In this case, Rubah would fail to stop all threads for the second update; the left-out threads would then crash due to accessing wrong-version data.

b) **Resource Leak:** Updatable programs should use interruptible I/O so that an update can be readily applied even when the program is waiting for I/O. Rubah provides a drop-in API for interruptible Java I/O calls[3] that requires the developer to provide, and manage the lifetime of, selector objects. In Figure 7, method method `readRequest` throws an `UpdateRequestedException` when interrupted by an update. This exception is caught on line 23 and the loop soon reaches the update point on line 17. When we added support for Rubah to H2 and CrossFTP, we re-opened each selector between updates without closing the one used in the previous version, i.e. we did not add line 33. After some updates, the program reached the maximum number of selectors and terminated.

c) **Wrong Update Point:** When retrofitting CrossFTP with Rubah, we added support for updates to happen while transferring files. Figure 8 shows how. After the update, the control migration restores the offset already transferred (line 2), which was saved before the update (line 11), and then reaches an update point to complete 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.

d) **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 fail the operation after the time-out expires and only then reach an update point. This bug then manifests itself as transactions aborting due to either (1) mis-detecting concurrent operations or (2) triggering a lock timeout, depending on the particular SQL statement waiting for the lock.

e) **Exclusive Mode:** The H2 database supports a feature called *exclusive mode* in which a single client has exclusive access to a particular database. All other clients that try to connect to that database have to wait until the connected client exits exclusive mode. Performing an update in this setting leads to a deadlock: The threads belonging to the exclusive client reach an update point, and thus stop executing until the update starts; while the other threads keep waiting for the exclusive mode to be released without ever reaching an update point, and thus preventing the update from starting.

f) **Function ID Transformation:** Internally, H2 represents prepared *CALL* statements, which invoke built-in functions on the database, using a distinct integer for each possible function. One version of H2 added a new built-in function *SCOPE_ID* to retrieve the IDs generated through database triggers for each statement. However, this new function had, in the new version, the same ID — 154 — as another function *AUTOCOMMIT* in the old version, which checks whether the auto-commit flag is set for the current session; *AUTOCOMMIT* was given ID 155 in the new version. When an update is performed after preparing a *CALL AUTOCOMMIT* statement, a prepared statement could invoke the wrong function *SCOPE_ID* after the update. We were able to observe this bug on other test case that checks whether the current session is read-only through function *READONLY*.

g) **New FTP Command — MD5:** CrossFTP adds support for the MD5/MMD5 commands in one of the versions we retrofitted. However, the server failed to detect that the command was available after the update because an in-memory map structure of available commands was not updated during the update to contain the new command.

h) **Batch FTP commands:** We retrofitted CrossFTP in a way that did not support receiving FTP commands in a batch. When several commands were included in a single message, CrossFTP would process the first command and then wait for more commands from the client, instead of checking if

[3] In Java, interrupting a socket operation closes the socket; Rubah's API is compatible with the socket API but can be safely interrupted for updates.

the received message had any commands left. The original code used a buffered stream, which only performs a socket read when empty. The retrofitted code also uses a buffer but it always reads commands from the socket, thus missing commands left in the buffer by a previous read.

Discussion: Update-point synchronization and control-flow reboots perform a large number of updates in tight sequence, thus revealing data races on corner-cases inside the DSU system itself (*a*) and erroneous thread interleaving on multi-threaded tests (*d* and *e*). The sheer number of updates that these two techniques perform also reveals resource leaks (*b*). Control-flow reboots would find the same errors as update-point synchronization, but at a slightly higher cost. Exhaustive and operation-oriented testing performs a single update per execution and would not find these errors. Some errors are simply caused by taking a rare and erroneous update opportunity (*c* and *h*). Exhaustive testing would also find these two bugs because it explores all possible update opportunities. Finally, some errors are due to incorrect data migration between versions and would only be found by exhaustive testing (*f*) or update-specific testing (*g*), depending if the error is on modified backwards-compatible code or new features.

E. Operation-Oriented Testing in Practice

We used operation-oriented testing in the adapted TPC-C benchmark suite. Operation-oriented testing requires specifying a budget of updates to explore per combination, and which combinations to consider. We applied this technique to H2 and TPC-C, exploring 20 combinations per operation on the most common, least common, and randomly selected combinations (we measured the number of update opportunities per combination in a pre-run). This technique also discovered bug (d), but found no additional bugs. One issue is that it is fairly inefficient: updating on uncommon combinations required several re-runs until the target combination would finally happen. We conjecture that a more efficient scheduling scheme (e.g., along the lines of an explicit state model checker like CHESS [20]) might make operation-oriented testing more efficient and effective.

VI. 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 [21] 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 [22] observed that this is too restrictive because it forbids updates that fix bugs or add features. Gupta et al. [14] propose that an update is correct if the updated program eventually reaches some state of the new program. Hayden et al. [23] argue that any attempt to define update correctness generally is flawed as update correctness depends on the particular semantics of each updatable program. Furthermore, they propose client-oriented specifications as small programs to specify properties that must hold before and after the update, and then propose techniques to verify updatable programs with regards to these specifications. Tedsuto, instead, uses system tests to ensure that updates are correct. They also introduce the concept of backwards-compatible specifications, which are similar to how Tedsuto uses system tests; and post-update specifications, which are similar to update-specific tests. They applied their technique to small, verifiable programs; Tedsuto is applicable to large, real-world applications.

Tedsuto is implemented for Rubah [6], a DSU system that requires the programmer to make manual changes to the program to support updating. We discuss other DSU systems, and how Tedsuto can be applied to them, in Section IV-C.

Previous work by Hayden et al. [16] considers systematic testing for the Ginseng DSU system for C programs [2]. This work does little to develop support for backward-incompatible tests (though they suggest the idea), and does not work with multi-threaded programs; Tedsuto handles both. Moreover, Hayden et al. have no notion of update synchronization tests or control-flow reboots, which are unique to Tedsuto and, as our experiments have shown, highly effective.

Our approach of repeating tests to explore different update opportunities systematically is related to multi-threaded testing tools [20], [24] that explore a subset of all the possible potential thread schedules systematically by repeating each test for each schedule.

VII. 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. Tedsuto re-uses existing system tests to find bugs induced by the update process that are dependent on the instant at which the update happens by automatically exploring different update opportunities during the execution of each system test. Tedsuto can check that, after an update, unmodified features are still supported and modified features behave correctly.

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 8 update-related bugs in short order and at low cost. We argue that Tedsuto is a general solution for testing DSU, readily applicable to other state-of-the-art DSU systems. We believe Tedsuto is an important step toward practical assurance for DSU.

Acknowledgments

We thank the anonymous reviewers for the helpful comments on early drafts of this paper. This work was supported by the partnership between UMIACS and the Laboratory for Telecommunication Sciences, and by the EPSRC through grant EP/L002795/1.

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

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

[10] "H2 Database Engine," http://h2database.com/html/main.html, [Online; accessed 22-January-2016].

[11] "CrossFTP server," http://www.crossftp.com/crossftpserver.htm, [Online; accessed 22-January-2016].

[12] 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

[13] 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.

[14] 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

[15] 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

[16] 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

[17] "HSQLDB Java Database," http://hsqldb.org/, [Online; accessed 22-January-2016].

[18] "Apache FtpServer," https://mina.apache.org/ftpserver-project/index.html, [Online; accessed 22-January-2016].

[19] 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

[20] 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

[21] 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

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

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

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