# Improving Reliability of Dynamic Software Updating Using Runtime Recovery

Tianxiao Gu Zelin Zhao Xiaoxing Ma* Chang Xu Chun Cao Jian Lü
Department of Computer Science and Technology, Nanjing University, Nanjing, China
State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, China
{tianxiao.gu, zelinzhao1105}@gmail.com, {xxm, changxu, caochun, lj}@nju.edu.cn

***Abstract*—Dynamic software updating (DSU) is a technique that can update running software systems without stopping them. Most existing approaches require programmer participation to guarantee the correctness of dynamic updating. However, manually preparing dynamic updating is error-prone and time-consuming. Therefore, other approaches prefer to aggressively perform updating without programmer intervention, which may definitely lead to unanticipated runtime errors. To reduce human effort and enhance the reliability for dynamic updating, we leverage automatic runtime recovery (ARR) techniques to recover runtime errors caused by improper dynamic updating. This paper presents ADSU, a fully automatic DSU system using ARR. We evaluate ADSU with real updates from widely used open source software systems, *i.e.*, Apache Tomcat, Apache FTP Server and jEdit. The preliminary results have shown that ADSU succeeds in automatically applying 11 of 16 real-world updates that existing counterparts cannot.**

***Keywords*-dynamic software updating; automatic runtime recovery; Java virtual machine;**

## I. Introduction

Deployed software systems in the field are always subject to software updates. To apply updates, a running software system should be stopped. Dynamic software updating (DSU) can help to preserve the software availability from service interruption caused by updating. Specifically, a DSU system takes over the execution of a software system, properly determines an *update point*, applies transformations to the current runtime state and after that resumes the execution with the new version.

However, general-purpose DSU systems cannot automatically achieve correct dynamic updating without programmer intervention. Gupta *et al.* proposed a formal framework for DSU and proved that in general the validity of dynamic updating is undecidable [1]. Other approaches [2], [3], [4] can give sufficient conditions to validate DSU but all demand their applicable software to fit a specific model, *e.g.*, transaction model. As a result, various human efforts must be involved in preparing a patch (*i.e.*, dynamic patch) for dynamic updating, particularly in preparing state transformations [5], [6], [7].

In practice, state of the practice of DSU systems either put limitations to an update that can be dynamically applied in order to reduce the need of state transformations [8], [9], [10], [11], or demand programmers to provide state transformations [12], [13], which all have the risk that improper dynamic updating may lead to unexpected runtime errors.

*Corresponding author: Xiaoxing Ma.

Manually preparing state transformations is error-pone and time-consuming. Programmers should have a good knowledge about programming interfaces of the DSU system and the runtime representations of both the old and the new version of a program. In addition to these inherently difficulties, state of the art testing and verification tools for dynamic updating are still far from effective [14].

While most existing DSU systems delegate the responsibility of correctness to programmers, others prefer to aggressively perform dynamic updating without user intervention, based on predefined rules [15], [16] and program synthesis [17]. Apparently, they can succeed but may also lead to unexpected runtime errors. The benefit of no human effort is not enough to compensate the risk of service unavailability. Such aggressive approaches may be worthy of consideration if the software system can recover from runtime errors.

Recently, numerous approaches to automatic runtime recovery (ARR) have been published [18], [19], [20]. Many of them have shown promising effectiveness in recovering software systems from unexpected runtime errors. This inspires us to leverage existing ARR techniques to enhance the reliability of aggressive dynamic updating. Specifically, we prefer to rescue running software systems from runtime errors caused by improper automatic dynamic updating using ARR techniques, based on the inherent resilience among different components of a software system, rather than avoid runtime errors by increasing the burden to programmers.

In this paper, we present the design and implementation of ADSU, a fully automatic DSU system using a light-weight ARR technique. ADSU is based on Javelus [21], a flexible, efficient yet low disruptive dynamic-updating-enabled JVM, which is built on top of the Java HotSpot VM in OpenJDK. The light-weight approach to ARR makes ADSU also efficient and be able to be deployed in production environments, as the running performance is a major concern in these situations.

By using ARR techniques, ADSU can improve the reliability of dynamic updating, either manually or automatically prepared. Note that due to the limitation of existing ARR techniques, we do not aim to turn improper dynamic updating into proper one but instead mitigate the harmful effects of improper updating. The preliminary results have shown that ADSU successfully mitigates improper dynamic updating of 11 of 16 real-world updates from Apache Tomcat, Apache FTP Server and jEdit.

The main contributions of this paper are listed as follows.

- We propose and implement ADSU, a fully automatic DSU system using a light-weight ARR technique to enhance its reliability.
- We evaluate ADSU with real updates from Apache Tomcat, Apache FTP Server and jEdit.

The rest of this paper is organized as follows. In Section II, we give an overview of dynamic software updating and automatic runtime recovery. Section III depicts ADSU in detail and Section III presents our evaluation of ADSU. We briefly discuss related work in Section V and conclude in Section VI.

## II. Background

This section presents an introduction to the background of dynamic software updating and automatic runtime recovery.

### A. Dynamic Software Updating

Implementing a DSU system needs to consider when and how to perform the dynamic updating. Apparently, there do exist trade-offs between these two matters. Namely, an update point determines what needs to be transformed and what cannot be transformed also precludes time points at which insuperable obstacles exist.

A DSU system usually implements a consistency model to help automatically determine updating points. Most DSU systems follow the *strict consistency model*, which ensures that at any time only a single version of code is executing. A relaxed consistency model allows the coexistence of multi-versions of code. The dynamic updating can occur at any time but requires additional non-trivial manual work [22]. Although a strict consistency model may not be as timely as relaxed one, it is ease-to-use for programmers and thus implemented by most DSU systems.

A majority of existing DSU systems prefer to update only when there is no active updated method and delegate the only job, *i.e.*, preparing heap state transformation, to programmers [23], [24], [5], [21]. The timeliness may be poor if there is an active updated method with a long-running loop. Furthermore, a highly concurrent running software system can hardly reach an update point if there are too many updated methods that are also invoked frequently.

To solve the former problem, one could prepare transformations for the active updated method [25] or extract the loop body into another method [24]. The latter problem can be mitigated by suspending a thread to prevent it from executing an updated method and wait until all threads are ready. However, this approach may easily lead to deadlock. A safer approach is to use relaxed synchronization [26].

Preparing heap state transformations is the most error-prone and labor-consuming task for programmers. The challenges mainly come from two aspects. First, the heap is usually complex and tremendously large for modern software systems. Note that dynamic updating should not pause the running software for a long time. Thus, updating such a huge heap needs to fulfill strict timing constraints. An effective way to solve this problem is to update every stale object lazily when it is actually used in subsequent execution [21].

Second, programmers should figure out the relation between the current old state and the transformed new state. Obviously, this is non-trivial as programmers should not only have a comprehensive understanding of the old version but also the new one. In addition, programmers should learn the programming interfaces provided by the underlying DSU system to prepare valid state transformations.

Recently, an approach named TOS [17] can infer conditional object transformation functions that are able to realize various transformations based on the current state of a stale object. TOS first runs a same set of tests over the old and the new version of a program to collect a set of matched old and new objects, and then inductively applies a set of predefined rules to compose a transformation function until it can transform an old object into the corresponding new object.

However, TOS is still ineffective for non-trivial cases due to its poor predefined rules. Besides, the transformation functions inferred from objects in testing may not be proper to transform objects in production. Last, it requires programmers to provide a set of good quality tests and also specify how to collect objects. Therefore, TOS is actually not fully automated.

To mitigate the difficulty of preparing heap state transformation, most DSU systems implement a fully automated default transformation scheme based on a set of predefined rules. For Java programs, a default transformation preserves the value of an unchanged field and assigns a type-specific default value to a newly added field, *e.g.*, `0` for `int` and `null` for a reference.

The first rule contributes most to the success of a default transformation as most updates actually do not change fields and preserving the value of an unchanged field is just the right way. But for newly added fields, the type-specific default value is not as effective and may easily result in unexpected runtime errors such as `NullPointerException`. Hence, we leverage existing automatic runtime recovery techniques to further mitigate this problem.

### B. Automatic Runtime Recovery

In Java programs, a runtime error usually manifests itself via a *checked* or an *unchecked* exception. Checked exceptions are anticipated and mostly explicitly thrown by a `throw` statement. Programmers can predict their occurrences in advance and prepare error handlers for them accordingly. The Java programming language enforces that a method must either handle a checked exception that is thrown during its invocation or rethrow the exception up to the caller.

In contrast, unchecked exceptions are usually unanticipated, particularly those implicitly thrown via various bytecode instructions. For example, reading a field or invoking a method via a null pointer can both lead to a `NullPointerException`. Such an implicitly thrown exception is mostly related to a bug and must be precluded during software testing. However, software testing is non-trivial and there do exist leaked bugs.

Automatic runtime recovery mainly aims to rescue a running software system from unanticipated runtime errors that are

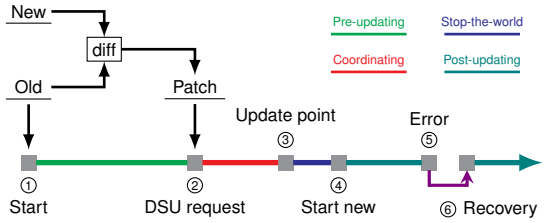

Fig. 1: Overview

caused by software bugs. In general, an ARR technique needs to synthesize a set of recovery actions, which are mostly based on redundancy or predefined rules, and use various validation techniques such as testing to preclude potential dangerous ones. Existing ARR techniques can be classified into two types, *i.e.*, heavy-weight [27], [28], [18] and light-weight [29], [30], [19], based on whether they make use of heavy-weight mechanisms such as *checkpoints* to amend the runtime and validate their recovery actions.

Heavy-weight approaches create checkpoints periodically during execution. During a recovery, the erroneous state must be rolled back to a previous checkpoint first. Next, a recovery action is picked out and applied to the running program, followed by a testing phase. If the testing fails, the runtime state is rolled back again for another recovery action. Finally, only actions passed their testing can be used to recover the error. However, tests for validating recovery actions must be prepared in advance for some heavyweight approaches. Besides, side effects that checkpoints cannot cover must be handled carefully, *e.g.*, writing to a database.

Lightweight approaches usually have negligible influence on the running program during execution and are triggered on-demand when an error occurs. FOC [29] can handle errors caused by invalid memory access in server applications. It just discards invalid write and synthesizes a type-specific default value for invalid read. However, its simple yet aggressive nature cannot convince people of its effectiveness. Recently, RCV [19], an extension to FOC, has shown promising effectiveness in repairing real-world security bugs. This inspires us to use light-weight ARR techniques such as FOC and RCV to make automatic dynamic software updating for more practical.

There are mainly two considerations for why we have not used heavy-weight ARR techniques. First, checkpoints bring extra complexity and overheads to the system, which may require more implementation efforts. Second, a runtime error caused by improper dynamic updating is raised in the execution with the updated new version. To recover it, the runtime state may be reverted into a checkpoint that is created with the old version. From another perspective, the dynamic updating itself is rolled back and discarded. As dynamic updating is generally used to fix security vulnerabilities in emergency, a more severe failure than the improper dynamic updating may occur.

## III. Aggressive Dynamic Software Updating

This section presents the design and implementation of ADSU.

### A. Overview

ADSU is based on Javelus, a dynamic updating system for mission-critical Java applications. Figure 1 gives an overview of ADSU. The entire execution of a program with dynamic updating on ADSU is roughly divided into four phases, the *pre-updating* phase, the *coordination* phase, the *stop-the-world* phase, and the *post-updating* phase.

ADSU is fully automated. The input to ADSU is only the currently deployed *old* version and the to-be-updated *new* version of a program, or specifically, only binary Java class files of each version. Users use a program differentiating tool to create a *patch* that lists all changed classes. ADSU provides an interface to let users send the patch to a running program to request dynamic updating.

In the pre-updating phase, the execution is the same as that in an unmodified JVM. When receiving a dynamic updating request, ADSU performs an eager and quick system check that needs to suspend all application threads. ADSU follows a strict consistency model to determine an update point. To achieve type safety, there must be no active updated method in any thread. If the current runtime state fulfils this condition, the execution transits into the stop-the-world phase directly. Otherwise, the execution turns into the coordination phase, in which many techniques such as return barrier [5] are adopted to speed up reaching an update point.

The running program may finally reach an update point and turn into the stop-the-world phase. Apparently, ADSU may also fail to reach an update point in a period of coordinating execution and then discards the dynamic updating. In the stop-the-world phase, stacks are unchanged and only code is patched. ADSU resumes the execution with stale objects left in the heap. In the post-updating phase, these stale objects are identified and updated lazily using default transformations.

We embed a lightweight automatic runtime recovery subsystem in ADSU in case default transformations lead to unexpected runtime errors. If such runtime errors occur, ADSU attempts to recover them automatically.

### B. Default Transformations

ADSU detects stale objects and updates them using default transformations in the post-updating phase. In practice, default transformations are effective for most trivial updates, particularly those without changes to fields. However, we can hardly determine whether default transformations will succeed ahead of updating. There do exist risks that a default transformation leads to unexpected runtime errors.

In fact, an improper transformation can result in two kinds of errors, *runtime error* and *inconsistency error*. A runtime error can manifest itself as a Java exception, which is usually raised by failed sanity checks or invalid memory access. Take an example shown in Fig. 2. Tomcat update 5eb244 introduces a new field `errorState` that is a reference to an object of `AtomicInteger`. A default transformation of a stale `Response` object leaves `errorState` to be of `null`.

A later recycle of the `Response` object attempts to set `0` to the `errorState` blindly, although `errorState` is use-

```
1  class Response {
2  - boolean error = false;
3  - boolean errorAfterCommit = false;
4  + AtomicInteger errorState = new AtomicInteger(0);
5    public void recycle() {
6  -   error = false;
7  -   errorAfterCommit = false;
8  +   errorState.set(0);
9      /* omitted */
10   }
11 }
```

Fig. 2: Tomcat update 5eb244.

less if the response is successfully composed. This useless operation leads Tomcat to fail to respond to the entire request. Specifically, the method `recycle` invokes the method `set` on `errorState` of `null` (at line 8), which results in a `NullPointerException` that cannot be handled by existing exception handlers of Tomcat.

In contrast, an inconsistency error is hard to detect. Ideally, there must be sanity checks everywhere in a program. A sanity check should identify such inconsistency and convert it into a detectable runtime error. However, the sanity check may not be designed with dynamic updating in mind and fails to identify inconsistency caused by improper dynamic updating.

In this paper, we focus on runtime errors that can be manifested via *unchecked exceptions* whose classes are subclasses of `RuntimeException` in Java. Inconsistency errors may only be precisely identified via user-provided specifications, which would further require human efforts.

### C. Automatic Runtime Recovery

Nevertheless, both automatic and manually-prepared state transformations may not be thoroughly tested or verified and thus lead to unexpected unchecked exceptions. ADSU only takes over the responsibility to handle unchecked exceptions that have no exception handler or only a *trivial* one.

Specifically, we treat exception handlers that declare to catch `Throwable` or `Exception` as trivial in this paper. An unchecked exception often has no effective exception handler, as one cannot predict its occurrence, *i.e.*, when and where it will be raised. In contrast, if there is an exception handler that declares to catch `RuntimeException` or any super type of the type of the unchecked exception, we treat this handler as non-trivial and let it handle the unchecked exception.

Modern real-world programs usually has a *trivial* exception handler on the stack for all kinds of exceptions raised at any location within the context. This exception handler indeed cannot recover an unexpected exception but provides a better feedback about an exception. For example, Tomcat may catch an unchecked exception anyway and send to the client an HTTP 500 error indicating the existence of a server internal error instead of letting the thread crash and resulting in a connection loss.

Apparently, a real non-trivial exception handler can declare to throw `Throwable` or `Exception` and a real trivial exception handler may also declare to throw `RuntimeException`. The previous way to determine trivial exception handlers may make ADSU reject to recover a runtime error. Another rational way may be to treat an exception as non-trivial only when the declared exception type is exactly the type of the raised exception. However, we found that in practice this may lead ADSU to mistakenly take over to handle numerous unchecked exceptions and finally mess the execution.

We leverage an existing lightweight ARR technique in JVM, named *early return* [20], to handle both implicitly (*e.g.*, null-dereference) and explicitly (*e.g.*, failed sanity check) thrown unchecked exceptions. If an unchecked exception is captured in the current active method, no matter whether the exception is raised in the method or re-thrown by a callee, we force the method to return prematurely with a type-specific default value, *e.g.*, `0` for `Integer`.

Recall that at line 8 in Fig. 2 a `NullPointerException` is raised due to an improper default transformation. ADSU can recover the system from this exception as follows. The method `set` not method `recycle` is treated as the current active method as all arguments for invoking `set` are loaded on the stack when the exception is raised. The return type of `set` is `void`. Thus, we have no need to synthesize a type-specific default value for simulating a premature return. Instead, we can just remove all loaded arguments and begin to execute the next instruction of `recycle`. The effect of this recovery is similar to changing line 8 into the following one.

```
if (errorState != null) errorState.set(0);
```

A recovery of an exception may not solve the problem thoroughly at the first place. There may be *cascaded* unchecked exceptions. We repeatedly apply a recovery for each of them. A discussion of the update Tomcat 4a10ae with cascaded exceptions can be found in Section IV-B.

One may doubt that existing ARR techniques are actually proposed for software bugs not improper dynamic updating. In fact, FOC or early return only fixes broken data and control flow blindly. The ability of FOC or early return that can recover errors comes from the inherent resilience among different components. Besides, many automated runtime recovery or program repair techniques cannot invent new functionalities but reuse existing functionally-equivalent implementations [27], [28], [18] or delete functionalities [31].

### D. Implementation

ADSU is based on Javelus, a dynamic-updating-enabled JVM on top of the Java HotSpot VM released in OpenJDK[1]. The implementation of Javelus can be found in [21], [7]. We implement the early return described in [20] by modifying the interpreter and compiler of the HotSpot JVM.

Due to implementation challenges, we have not adopted the FOC as the ARR technique. In FOC or RCV, the execution should be continued at the next instruction after a recovery. Thus, we should first repair the context for the next bytecode instruction in JVM for Java programs. However, a bytecode instruction may be translated into a number of native instructions. The exact context for the next bytecode instruction cannot be precisely rebuilt in an industrial-strength JVM.

[1] http://openjdk.java.net/groups/hotspot/.

To ease implementation, the JVM clears the expression stack when an exception is raised, before jumping to a common routine. This common routine is mainly used to look up for an exception handler. We leverage this routine to add extra logic that identifies exceptions of interest. When the callee frame is just removed, the expression stack of the caller frame is intact and arguments for invoking the callee lie at the top of the expression stack. To support executing the next instruction of the caller, ADSU preserves this intact expression stack before jumping to the common routine.

If ADSU determines that the exception can be recovered, it removes all arguments for the callee and pushes a type-specific default value as the return value, which are similar to a premature return. We directly modify the interpreter and the compiler to perform the simulated premature return and arguments removal. In other words, we actually do not synthesize bytecode instructions and insert them into existing code of a method.

ADSU should have no extra performance overhead during normal execution, as the code that we added to the JVM is only triggered when there is an exception and exceptions are rare in steady-state execution.

## IV. Experiments

This section presents our experiments of ADSU on real-world updates taken from widely used open source software systems. We evaluated ADSU on its *effectiveness* in automatically dynamic updating in comparison with default transformations using Javelus. We have not compared with TOS as it is not fully automated and requires tests and manually specified update points at which object examples are collected.

### A. Effectiveness in Updating Real-world Applications

We first evaluated the effectiveness of ADSU with real-word updates taken from two server applications, Apache Tomcat and Apache FTP Server, and a GUI application, jEdit[2]. We have successfully built and deployed 815 updates of Tomcat and 88 updates of FTP. Each update has either an issue number with respect to a bug report or a record in the change log. Hence, such an update is meaningful and worthy of dynamic updating. For jEdit, we further required that an update of interest must at least introduce a new field to a class and finally obtained 80 updates. However, not all updates were suitable for our evaluation.

The updates in our evaluation were determined as follows. First, we only took updates in which at least an updated class was loaded and executed during evaluation. In fact, 336 of 815 updates of Tomcat have no executed updated class during our evaluation. Second, we only took updates that Javelus failed using default transformations, as ADSU is indeed an extension to it with ARR support. Finally, we obtained 16 updates that Javelus failed (5 from Tomcat, 3 from FTP and 8 from jEdit). ADSU succeeded in performing 11 of 16 updates. The details are shown in Table I.

TABLE I: ADSU succeeded in applying 11 of 16 updates.

| # | Application | Size (LOC) | Patch Size | | ADSU (11/16) |
|---|---|---|---|---|---|
| | | | +/- (LOC) | Class (#) | |
| 1 | Tomcat-4355ed | 405K | 19/11 | 1 | ✘ |
| 2 | Tomcat-4a10ae | 404K | 29/3 | 1 | ✔ |
| 3 | Tomcat-4e4bbc | 405K | 2/1 | 1 | ✘ |
| 4 | Tomcat-5eb244 | 402K | 96/47 | 4 | ✔ |
| 5 | Tomcat-db5bd6 | 405K | 52/20 | 3 | ✘ |
| 6 | FTP-8be4ff | 24K | 9/4 | 1 | ✔ |
| 7 | FTP-55cc6b | 24K | 82/8 | 5 | ✔ |
| 8 | FTP-4907aa | 25K | 78/59 | 2 | ✘ |
| 9 | jEdit-2adbd1 | 170K | 22/1 | 1 | ✔ |
| 10 | jEdit-49e44f | 171K | 21/14 | 1 | ✔ |
| 11 | jEdit-623155 | 169K | 92/2 | 3 | ✔ |
| 12 | jEdit-b0fed8 | 170K | 48/91 | 2 | ✔ |
| 13 | jEdit-c890ed | 171K | 86/4 | 5 | ✘ |
| 14 | jEdit-e1a63e | 170K | 46/2 | 2 | ✔ |
| 15 | jEdit-e4cd44 | 169K | 8/8 | 2 | ✔ |
| 16 | jEdit-f3c70a | 170K | 12/7 | 1 | ✔ |

To evaluate the effectiveness of ARR, we need an elaborate test that is close to real-world scenarios rather than a simple unit test. This is because that our ARR technique depends on the inherent resilience among different components of a software system. Hence, we deployed a DayTrader [32] web application in the Tomcat and used JMeter[3] to run a JMeter test plan released with the DayTrader as the client.

For FTP server, we manually designed a systematic test suite to simulate user interactions, including almost all supported FTP commands. For jEdit, we designed a set of GUI actions, including opening a file and making some editing.

Although some tests were manually created by ourself and thus not representative, they were only used to reveal improper default transformations and validate the effectiveness of a recovery. To further avoid infeasible recovery, we discuss each update of server applications in detail in Section IV-B.

***Tomcat and FTP*** For server applications, we performed all selected updates on both Javelus and ADSU. Additionally, we run the old version in a standard JVM to record some performance metric as the baseline. The server application and its corresponding test client were run in the same machine, *i.e.*, a Linux machine with Intel Quad-Core i7 3.4GHz CPU and 20 GB memory. We run eight clients that continuously sent a huge number of different requests to the server. Hence, there were also at least eight threads at the server side to handle those requests.

Details of all requests are shown in Table II. ADSU can succeed in applying four updates (*i.e.*, Update #2, #4, #6 and #7 in Table II). In addition, for the four successful updates, the performance degradation of ADSU, *e.g.*, requests per second, is not significant in comparison with the unmodified JVM. We have not manually validated any request. To avoid hidden improper recovery, we further discuss all updates and how ADSU ucceeded in detail in Section IV-B.

***jEdit*** For Javelus, we failed to edit the file, *e.g.*, make any typewriting, and even save the file after the dynamic updating for all eight updates. ADSU can successfully perform seven

[2]http://www.jedit.org.

[3]http://jmeter.apache.org.

TABLE II: Results of ADSU on Tomcat and FTP server.

| # | Application | Java | | ADSU | | | | Javelus | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Total (#) | Req./s | Total (#) | Req./s | Succ. (%) | Failed (#) | Total (#) | Req./s | Succ. (%) | Failed (#) |
| 1 | Tomcat-4355ed | 24,382 | 305.4 | 234,802 | 2942.5 | 61.2 | 93,518 | 53,896 | 675.4 | 19.2 | 43,540 |
| **2** | **Tomcat-4a10ae** | 24,846 | 311.4 | 24,020 | 300.8 | **100.0** | **0** | 29,887 | 274.5 | 25.7 | 19,206 |
| 3 | Tomcat-4e4bbc | 21,710 | 271.0 | 27,491 | 344.4 | 74.9 | 6,878 | 24,473 | 306.5 | 99.9 | 17 |
| **4** | **Tomcat-5eb244** | 24,015 | 300.8 | 24,515 | 307.1 | **100.0** | **0** | 32,915 | 412.5 | 29.2 | 23,311 |
| 5 | Tomcat-db5bd6 | 23,601 | 295.6 | 10,643 | 116.9 | 98.1 | 202 | 11,007 | 121.4 | 99.6 | 45 |
| **6** | **FTP-8be4ff** | 11,352 | 315.3 | 11,472 | 310.0 | **100.0** | **0** | 11,432 | 317.6 | 93.5 | 741 |
| **7** | **FTP-55cc6b** | 11,400 | 316.6 | 11,366 | 315.7 | **100.0** | **0** | 5,038 | 335.9 | 72.3 | 1,396 |
| 8 | FTP-4907aa | 11,402 | 316.7 | 7,457 | 392.4 | 78.9 | 1,577 | 7,393 | 389.1 | 78.3 | 1,604 |

```
class StandardContext {
- Object applicationEventListeners[] = new Object[0];
+ List applicationEventListeners = new
    CopyOnWriteArrayList();
  public Object[] getApplicationEventListeners() {
-   return applicationEventListeners;
+   return applicationEventListeners.toArray();
  }
}
class Request {
  void notifyAttributeAssigned(/* ommitted */) {
    Context context = getContext();
    Object listeners[] = context.
        getApplicationEventListeners();
    if ((listeners == null) || (listeners.length == 0)) {
      return;
    }
    /* ommitted */
  }
}
```

Fig. 3: Tomcat update 4355ed

```
class MessageBytes {
+ private Map encoders = new HashMap();
  void toBytes() {
    /* omitted */
+   Charset charset = byteC.getCharset();
+   CharsetEncoder encoder = encoders.get(charset);
+   if (encoder == null) {
+     encoder = charset.newEncoder();
+     encoders.put(charset, encoder);
+   }
    /* omitted */
  }
}
```

Fig. 4: Tomcat update 4a10ae.

updates. After errors were mitigated, we can continue to edit, save and close the editor. Although the need to dynamically update client applications like jEdit is not as urgent as server applications, we believe that DSU can at least help to make updating not annoying and enhance user experiences as well for such client applications.

### *B. Individual Update Analysis for Server Applications*

We discuss how ADSU mitigated runtime errors caused by improper dynamic updating for server applications, except Tomcat 5eb244, which has been discussed in Section III-B.

The effectiveness of lightweight ARR techniques has been widely studied in [29], [19]. During our evaluation, we have found that there are mainly three patterns that these ARR techniques could succeed, namely, (1) ignoring useless operations (*e.g.*, #4-Tomcat-5eb244 and #7-FTP-55cc6b), (2) lazy initialization (*e.g.*, #2-Tomcat-4a10ae) and (3) default behavior (*e.g.*, #7-FTP-8be4ff).

***Tomcat 4355ed*** As shown in Fig. 3, this update fixes a rare race bug by substituting an array with a copy-on-write container. A default transformation left the reference to the new container be null. Then, a later read of this field at line 6 incurred a `NullPointerException`.

ADSU cannot recover this exception at the first place when the exception is triggered. This is because that there is an exception handler for `RuntimeException`. Actually, this exception handler trivially re-throws the exception. However, ADSU cannot treat it as a trivial exception handler (described in Section III-C) and then take over to handle the exception. After the exception was re-thrown, ADSU captured it and made a premature return, which resulted in an almost empty response to the client but with a correct HTTP status code that deceived the JMeter validity checker. That is why ADSU resulted in more requests than others.

In fact, ADSU can recover this error if it can determine that the existing error handler for `RuntimeException` is trivial. As we can see at line 2, there is no event listener in default and an empty array is the same as a null reference at the call site (at line 13). We plan to conduct some program analysis to further identify such trivial error handlers in the future work. Additionally, this error can also be mitigated if programmers allocate the container in an on-demand way with the consideration of dynamic updating.

***Tomcat 4a10ae*** As shown in Fig. 4, this update introduces a map to cache the encoder for each character set. A default transformation assigned a null reference to `encoders`. There were two unchecked exceptions during the invocation of method `toBytes`.

The first one occurred at line 6 when method `get` is invoked to fetch the pre-saved encoder. ADSU recovered it by making `get` return `null`. The second was a cascaded exception that occurred at line 9, when `put` is invoked to save a new allocated encoder. We can always get a new encoder on an updated `MessageBytes` and sacrifice the performance improvement. Nevertheless, the performance improvement can be reflected on newly allocated `MessageBytes`.

***Tomcat 4e4bbc and db5bd6*** These two updates are similar to Tomcat 4355ed. The recovery was disabled by an exception handler that declares to catch `RuntimeException`. As a result, ADSU failed to update them dynamically.

***FTP 8be4ff*** As shown in Fig. 5, this update fixes a bug that violates the RFC, where a time stamp in UTC should be returned instead of one in local time for some command.

```
class DateUtils {
+ private static final TimeZone TIME_ZONE_UTC =
      TimeZone.getTimeZone("UTC");
  static String getFtpDate(long millis) {
    StringBuffer sb = new StringBuffer(20);
-   Calendar cal = new GregorianCalendar();
+   Calendar cal = new GregorianCalendar(TIME_ZONE_UTC);
    cal.setTimeInMillis(millis);
  }
}
```

Fig. 5: FTP update 8be4ff.

A default transformation left `TIME_ZONE_UTC` to be of a null reference. A later time formating operation invoked the method `getOffset` of `TimeZone` on the null reference and incurred a `NullPointerException`. Based on the API specification, the `getOffset` should return 0 for UTC and a non-zero number for any other local time zone. Hence, ADSU can perfectly recover this error by making `getOffset` prematurely return with 0.

***FTP 55cc6b*** This updates fixes a bug and also adds a logger to print debugging messages. During execution, the method `logger.debug()` is always invoked but actually logs nothing when debugging is disabled. A default transformation cannot allocate a logger. Hence, any call to `logger` incurs a `NullPointerException`. ADSU just ignored the call to `logger.debug()` and recovered the system from the error.

***FTP 4907aa*** As shown in Fig. 6, this update is an improvement of the implementation of passive ports. Users can assign a range of numbers as the candidates for passive ports. The old implementation uses a large `int` array to store all numbers in the range and a `boolean` array to indicate which number/port is used. The number 0 indicates that any port can be used without checking whether it is in the assigned range. The number -1 is used to indicate that no port is available. Hence, a client cannot connect in the passive mode.

A default transformation left `freeList` to be of `null` and then incurred a `NullPointerException` at line 7. ADSU recovered this error and made `freeCopy` reference an empty list. Thus, the following loop at line 8 is not executed and `reserverNextPort` returned -1. Finally, the server sent a 425 error to clients, indicating that the server currently cannot build connections in the passive mode. Hence, ADSU cannot succeed in applying this update during our evaluation.

In fact, ADSU can still mitigate this error. Without ARR, the client received an 500 error indicating that something internal was wrong. If the client received the 425 error, it may make another attempt and choose to build connections via another mode. In contrast, the client may stop connecting when it received an 500 error as it may think the entire server is out of order.

### C. Discussion

The current implementation of ADSU only uses a simple ARR technique. In our experiments, this technique is effective, because the default behavior is just the desired behavior for many updates. In other updates, the semantics of the old object may be lost after recovery and ADSU may fail. In fact, a semantics loss can be easily detected by interacting with the two versions of an object in the same way and then comparing their outputs and responses. Besides, we can leverage existing search based approaches to automatic runtime recovery and automatic program repair to develop techniques that are able to preserve semantics for dynamic updating, which we leave as the future work.

```
class PassPorts {
- private int[] passivePorts;
- private boolean[] reservedPorts;
+ private List<Integer> freeList;
+ private Set<Integer> usedList;
  public int reserveNextPort() {
+   List freeCopy = new ArrayList(freeList);
+   while (freeCopy.size() > 0) {
      /* omitted, return 0 if there is no free port */
      /* omitted, remove port occupied by others */
    }
    return -1;
  }
}
```

Fig. 6: FTP update 4907aa.

## V. Related Work

This section presents a brief survey of related work.

### A. Dynamic Software Updating

Dynamic software updating systems focus on making DSU more safe [26], efficient [5], flexible [15], low disruptive [7] on various programming languages such as C/C++ [24], [22], [6], [33] and Java [34], runtime environments [5], [15], [21], component based software systems [2], [35], [3], [4] and operating systems [12], [13], [10]. However, DSU systems are not easy to use. To make DSU practical, many researches on automating [17], [36], testing [14], [37] and verifying [38] DSU have been published. ADSU tackles this problem from a new angle by leveraging existing ARR techniques to enhance the reliability of DSU systems.

### B. Automatic Runtime Recovery

Automatic runtime recovery aims to mitigate the harmful effects of runtime errors [29], [27], [28], [30], [18], or even repair the defect [19]. Recently many work on automated program repair have been published [39], [40], [41]. By combining DSU and ARR, we may bring these techniques to deployed software systems, *i.e.*, dynamically generating and applying a patch to the running program.

## VI. Conclusion

This paper presents ADSU, a fully automatic DSU system using ARR. ADSU is efficient and built on an industry-strength JVM. Within our knowledge, ADSU is the first approach that leverages existing fault tolerance techniques to enhance the reliability of DSU systems. The preliminary results have shown that ADSU is able to mitigate improper automatic dynamic updating for 11 of 16 updates. In the future work, we plan to extend ADSU in two aspects, *i.e.*, developing more predefined rules for state transformations and applying other ARR techniques with the consideration of dynamic updating.

## ACKNOWLEDGMENT

This work was supported in part by National Basic Research 973 Program (Grant #2015CB352202), National Natural Science Foundation (Grants #61472177, #91318301, #61321491) of China. The authors would also like to thank the support of the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

## REFERENCES

[1] D. Gupta, P. Jalote, and G. Barua, "A formal framework for on-line software version change," *IEEE Transactions on Software Engineering*, vol. 22, no. 2, pp. 120–131, 1996.

[2] J. Kramer and J. Magee, "The evolving philosophers problem: Dynamic change management," *IEEE Transactions on Software Engineering*, vol. 16, no. 11, pp. 1293–1306, 1990.

[3] Y. Vandewoude, Heverlee, P. Ebraert, Y. Berbers, and T. D'Hondt, "Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 856–868, 2007.

[4] X. Ma, L. Baresi, C. Ghezzi, V. Panzica La Manna, and J. Lu, "Version-consistent dynamic reconfiguration of component-based distributed systems," in *Proceedings of the ACM SIGSOFT symposium and the European Conference on Foundations of Software Engineering*, 2011, pp. 245–255.

[5] S. Subramanian, M. Hicks, and K. S. McKinley, "Dynamic software updates: A VM-centric approach," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009, pp. 1–12.

[6] C. M. Hayden, E. K. Smith, M. Denchev, M. Hicks, and J. S. Foster, "Kitsune: Efficient, general-purpose dynamic software updating for c," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2012, pp. 249–264.

[7] T. Gu, C. Cao, C. Xu, X. Ma, L. Zhang, and J. Lü, "Low-disruptive dynamic updating of Java applications," *Information and Software Technology*, vol. 56, no. 9, pp. 1086–1098, 2014.

[8] M. Dmitriev, "Towards flexible and safe technology for runtime evolution of Java language applications," in *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution*, 2001.

[9] "Using hotpatching technology to reduce servicing reboots," https://technet.microsoft.com/en-us/library/cc787843(v=ws.10).aspx, accessed: 2016-07-10.

[10] "kGraft," https://www.suse.com/products/live-patching, 2016-07-07.

[11] J. Kabanov and V. Vene, "A thousand years of productivity: the JRebel story," *Software: Practice and Experience*, 2012.

[12] J. Arnold and M. F. Kaashoek, "Ksplice: Automatic rebootless kernel updates," in *Proceedings of the 4th ACM European Conference on Computer Systems*, 2009, pp. 187–198.

[13] "kpatch," https://github.com/dynup/kpatch, accessed: 2016-07-07.

[14] C. M. Hayden, E. A. Hardisty, M. Hicks, and J. S. Foster, "Efficient systematic testing for dynamically updatable software," in *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*, 2009, pp. 9:1–9:5.

[15] T. Würthinger, C. Wimmer, and L. Stadler, "Dynamic code evolution for Java," in *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, 2010, pp. 10–19.

[16] M. E. Kabir, H. Wang, and E. Bertino, "Efficient systematic clustering method for k-anonymization," *Acta Inf.*, vol. 48, no. 1, pp. 51–66, 2011.

[17] S. Magill, M. Hicks, S. Subramanian, and K. S. McKinley, "Automating object transformations for dynamic software updating," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2012, pp. 265–280.

[18] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezzè, "Automatic recovery from runtime failures," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013, pp. 782–791.

[19] F. Long, S. Sidiroglou-Douskos, and M. Rinard, "Automatic runtime error repair and containment via recovery shepherding," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 227–238.

[20] T. Gu, C. Sun, X. Ma, J. Lu, and Z. Su, "Automatic runtime recovery via error handler synthesis," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 684–695.

[21] T. Gu, C. Cao, C. Xu, X. Ma, L. Zhang, and J. Lu, "Javelus: A low disruptive approach to dynamic software updates," in *Proceedings of 19th the Asia-Pacific Software Engineering Conference*, 2012, pp. 527–536.

[22] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew, "POLUS: a powerful live updating system," in *Proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 271–281.

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

[24] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for c," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006, pp. 72–83.

[25] K. Makris and R. A. Bazzi, "Immediate multi-threaded dynamic software updates using stack reconstruction," in *Proceedings of the Conference on USENIX Annual Technical Conference*, 2009.

[26] I. Neamtiu and M. Hicks, "Safe and timely updates to multi-threaded programs," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009, pp. 13–24.

[27] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: Treating bugs as allergies—a safe method to survive software failures," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, 2005, pp. 235–248.

[28] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis, "Assure: Automatic software self-healing using rescue points," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009, pp. 37–48.

[29] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, Jr., "Enhancing server availability and security through failure-oblivious computing," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, 2004, pp. 21–21.

[30] V. Nagarajan, D. Jeffrey, and R. Gupta, "Self-recovery in server programs," in *Proceedings of the 2009 International Symposium on Memory Management*, 2009, pp. 49–58.

[31] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 691–701.

[32] "DayTrader—A more complex application," http://geronimo.apache.org/GMOxDOC22/daytrader-a-more-complex-application.html, accessed: 2016-06-15.

[33] G. Chen, H. Jin, D. Zou, Z. Liang, B. B. Zhou, and H. Wang, "A framework for practical dynamic software updating," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 4, pp. 941–950, 2016.

[34] L. Pina, L. Veiga, and M. Hicks, "Rubah: DSU for Java on a stock JVM," in *Proceedings of the 2014 International Conference on Object Oriented Programming Systems Languages Applications*, 2014, pp. 103–119.

[35] S. Ajmani, B. Liskov, and L. Shrira, "Modular software upgrades for distributed systems," in *Proceedings of the European Conference on Object-Oriented Programming*, 2006, pp. 452–476.

[36] Z. Zhao, T. Gu, X. Ma, C. Xu, and J. Lü, "Cure: Automated patch generation for dynamic software update," in *Proceedings of the 23rd Asia-Pacific Software Engineering Conference*, 2016, p. to appear.

[37] C. Hayden, E. Smith, E. Hardisty, M. Hicks, and J. Foster, "Evaluating dynamic software update safety using systematic testing," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1340–1354, 2012.

[38] C. M. Hayden, S. Magill, M. Hicks, N. Foster, and J. S. Foster, "Specifying and verifying the correctness of dynamic software updates," in *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments*, 2012, pp. 278–293.

[39] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 364–374.

[40] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013, pp. 802–811.

[41] S. Ding, H. B. K. Tan, and H. Zhang, "ABOR: an automatic framework for buffer overflow removal in c/c++programs," in *Proceedings of the 16th International Conference Enterprise Information Systems, Revised Selected Papers*, 2014, pp. 204–221.