Automating Object Transformations for Dynamic Software Updating via Online Execution Synthesis

Tianxiao Gu
State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, China
tianxiao.gu@gmail.com

Xiaoxing Ma[1]
State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, China
xxm@nju.edu.cn

Chang Xu
State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, China
changxu@nju.edu.cn

Yanyan Jiang
State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, China
jyy@nju.edu.cn

Chun Cao
State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, China
caochun@nju.edu.cn

Jian Lu
State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, China
lj@nju.edu.cn

Abstract

Dynamic software updating (DSU) is a technique to upgrade a running software system on the fly without stopping the system. During updating, the runtime state of the modified components of the system needs to be properly transformed into a new state, so that the modified components can still correctly interact with the rest of the system. However, the transformation is non-trivial to realize due to the gap between the low-level implementations of two versions of a program. This paper presents AOTES, a novel approach to automating object transformations for dynamic updating of Java programs. AOTES bridges the gap by abstracting the old state of an object to a history of method invocations, and re-invoking the new version of all methods in the history to get the desired new state. AOTES requires no instrumentation to record any data and thus has no overhead during normal execution. We propose and implement a novel technique that can synthesize an equivalent history of method invocations based on the current object state only. We evaluated AOTES on software updates taken from Apache Commons Collections, Tomcat, FTP Server and SSHD Server. Experimental results show that AOTES successfully handled 51 of 61 object transformations of 21 updated classes, while two state-of-the-art approaches only handled 11 and 6 of 61, respectively.

2012 ACM Subject Classification Software and its engineering → Software evolution

Keywords and phrases Dynamic Software Update – Program Synthesis – Execution Synthesis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.19

[1] Corresponding author.

32nd European Conference on Object-Oriented Programming (ECOOP 2018).

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Acknowledgements We are grateful to the anonymous reviewers for their insightful comments. This work was supported by the National Natural Science Foundation of China (Grant Nos. 61690204, 61472177), the 973 Program of China (Grant No. 2015CB352202), and the Collaborative Innovation Center of Novel Software Technology and Industrialization.

1 Introduction

Today's industry definitely requires high availability of software systems. One of the major losses of availability is caused by system shutdowns for installing software updates that fix bugs and security vulnerabilities. *Dynamic Software Updating* (DSU) can eliminate this loss by updating running software systems without stopping them.

Modern operating systems and programming language virtual machines provide powerful runtime code manipulation facilities such as dynamic linking [8], dynamic class loading [28], on stack replacement [37] and live patch [2]. With these facilities, it is not difficult to update the code of a running program. In addition to code replacement, DSU also needs to ensure that the new loaded code can execute properly with the existing runtime state in the memory (*e.g.*, heap objects) after the dynamic update.

Existing DSU supporting systems, *e.g.*, Ginseng [33], Jvolve [41] and Javelus [15], ensure only syntactical correctness, *i.e.*, no type error would be caused by the update. To preserve semantics correctness, the DSU system should apply an additional state transformation that maps the runtime state left by the old code to a proper state with which the new code continues. To realize the state transformation, developers usually specify a manually-prepared state transformation function, *i.e.*, *state transformer*.

However, it is difficult and error-prone to develop and test state transformers. Software is seldom developed with DSU in mind but is assumed to start from scratch. The internal states between different versions of a program can be incompatible, although the external behavior of the two versions is similar. For example, the internal representation of a container may be an array in the old version but a linked list in the new version. As a result, transformer programming not only requires a thorough understanding of implementation details in both versions, but also has to break the principle of information hiding and manipulate low-level data representations.

In this paper we aim at automating the state transformations for DSU. In theory, it is not possible to automatically generate correct transformers for dynamic updates of arbitrary programs [17]. Nevertheless, in many practical cases, for particular software patches and particular dynamic update points, state transformations can be automatically derived with sophisticated program analysis under some proper assumptions. This kind of techniques can help reduce service disruption caused by software updates, and are useful for application domains where high availability is the major concern and occasional errors are tolerable or compensable.

Our approach, named AOTES, is designed for DSU of object-oriented programs, or more specifically, Java programs. In object-oriented programming, an important principle is to use *information hiding* and *encapsulation*. An object should be interacted with only via its methods, where methods are closely related to the behavior of the object. Based on this principle, we have the following observations. First, the current state of an object is a conclusion of its past method invocations. Second, the current state is also the basis of the future method invocations. Third, the behavior of an object, or specifically the history of method invocations, is mostly unchanged during updating, especially when the patch does not include new behaviors. Thereby, the new state of a stale object (*i.e.*, an instance of an

updated class) can be synthesized by replaying its past method invocations with the new version of methods. In this way we can avoid direct mapping of concrete states between two program versions with different implementations.

Specifically, AOTES abstracts the runtime state of a stale object as a *history of method invocations* (*i.e.*, invocation history for short) that can produce the current state from an *initial (object) state*. For example, suppose that an array based container object with three elements e_1, e_2 and e_3 is created by a history of `add(`e_1`)`, `add(`e_2`)`, `remove(`e_2`)`, `add(`e_3`)` and `add(1,`e_2`)`. Now a dynamic update requires transforming the array based container into a linked list based one. We can easily know that the new linked list based container can be naturally acquired by applying the invocation history with the new version of methods on an empty linked list based container.

The main challenge of this approach is to obtain the invocation history for a stale object. Recording every method invocation on every potentially updated object is prohibitively expensive. Moreover, the actual history may contain redundant elements, *e.g.*, `remove(`e_2`)` and `add(1,`e_2`)`. To address these problems, we try to synthesize an equivalent but more compact invocation history from the current state. For the previous example, we can synthesize a history of `add(`e_1`)`, `add(`e_2`)` and `add(`e_3`)` instead of the actual one.

Unfortunately, it is hardly feasible to synthesize an invocation history using methods of real world programs. First, synthesizing a single method invocation is difficult because a method invocation generally requires arguments, which usually lie within a large value space (*e.g.*, $[-2^{31}, 2^{31}-1]$ for `int` in Java). Second, searching for a valid invocation history is time-consuming because method invocations need to be ordered properly to form a valid invocation history. In the scenario of dynamic updating, an additional challenge is that the synthesis is performed online and must be completed very quickly.

AOTES addresses these challenges by combining the power of symbolic execution, program synthesis and execution synthesis. Specifically, AOTES first distills a set of promising execution paths for each method by an offline symbolic execution technique. During dynamic updating, AOTES uses the selected execution paths only to realize a backward online execution synthesis technique. To reuse techniques of forward execution synthesis, AOTES synthesizes an inverse method for each selected execution path. We tried out AOTES on 21 real updates of widely used open source software. AOTES correctly handled 51 of 61 different transformations, while two state-of-the-art methods handled 11 and 6 of 61, respectively.

The paper makes the following primary contributions:

- We propose a mechanism to synthesize method invocation histories that can be used to recreate objects.
- We use the object recreating mechanism to automate object transformations for DSU.
- We implement the mechanism and evaluate it with updates taken from widely used open source systems.[2]

The rest of this paper is organized as follows. We first give an introduction to DSU and AOTES using an illustrative example in Section 2 and then a detailed overview of AOTES in Section 3. Next, we describe the offline analysis in Section 4 and online synthesis in Section 5. Then, we illustrate the implementation of AOTES in Section 6 and evaluate AOTES with updates from real-world software in Section 7. We summarize related work in Section 8 and conclude in Section 9.

[2] All source code and tests are publicly available at `http://moon.nju.edu.cn/dse/aotes`.

```
class DefaultSshFuture {
  SshFutureListener firstListener;
  List otherListeners;
  void addListener(SshFutureListener listener) {
    if (firstListener == null) {
      firstListener = listener;
    } else {
      if (otherListeners == null) {
        otherListeners = new ArrayList(1);
      }
      otherListeners.add(listener);
    }
  }
}
```

(a) The old version of `DefaultSshFuture`

```
class DefaultSshFuture {
  Object listeners;
  void addListener(SshFutureListener listener) {
    if (listeners == null) {
      listeners = listener;
    } else if (listeners instanceof SshFutureListener) {
      listeners = new Object[]{listeners,listener};
    } else {
      // Check the array bound
      // Expand the array if necessary
      // Append the listener
    }
  }
}
```

(b) The new version of `DefaultSshFuture`

Figure 1 An update (rev. b98694) of class `DefaultSshFuture` of Apache SSHD Server.

2 Illustrative Example

In this section, we present an introduction to DSU and AOTES using an illustrative example.

2.1 Dynamic Software Updating and Its Challenges

Software is subject to changes and evolution: Bugs are fixed and new features are introduced by applying *software updates*. Figure 1 shows a real-world motivating example of software update, which will be discussed throughout the paper. The update is from the Apache SSHD Server. Class `DefaultSshFuture` provides a method `addListener` to add listeners (Figure 1a). For most cases, there is only one or two listeners but the implementation should support adding more. To save memory, the old version saves the first-added listener to `firstListener` and others into `otherListeners`, which is an auto-expanding list container (`ArrayList`). The new version (Figure 1b) only uses a single field `listeners` and a raw array to handle all situations.

To allow long-running programs to receive timely updates without restart, dynamic software updating migrates the running program from the old version to a new version. Specifically, a DSU system takes over the execution of a running program, transforms the runtime state at a properly determined update point (*e.g.*, when no updated method is active) to a new state conforming to the new version, and then continues executing with the new version [23, 41].

A major challenge in DSU is the state transformation at the update point. A runtime system's state consists of the *code*, the *stacks* and the *heap*. As new code can be easily dynamically re-loaded and stacks mostly remain unchanged at the update point, the main challenge is the state transformation of the heap. We restrict our discussions to object-oriented programming languages (*e.g.*, Java) and thus the heap state transformation is particularly referred to as the *object transformation*.

An object transformation takes the current state of a stale object as input and produces the new state as output. The transformation must be *consistent*: The future execution must be able to continue from the transformed state and take over the ongoing business smoothly. None of existing approaches [41, 42, 31, 38] is capable of automatically conducting object transformations beyond trivial cases. Most state-of-the-art DSU systems [41, 42, 38] provide *default transformations* that simply copy the values of unchanged fields from a stale object to its corresponding new object, and initialize all new fields with *type-specific default values*, *e.g.*, `0` for `int`.

TOS [31] is the only known approach to automating object transformations, which embodies the idea of learning-by-example. A transformation example consists of an old-version object, which is collected during running a test over the old version of the program, and a new-version object, which is collected during running the same test over the new version of the program at the corresponding time point [31]. After collecting sufficient examples, TOS inductively composes a function following a set of predefined rules until the composed function can realize the transformations between all examples. However, TOS relies on the high quality tests in terms of covering transformations not only in testing but also in production. Even though there are sufficient good examples, TOS may easily fail due to its poor predefined rules.

```
1  void update(DefaultSshFuture1 o, DefaultSshFuture2 n) {
2    if (o.firstListener != null) {
3      if (o.otherListeners == null) {
4        n.listeners = new Object[] {o.firstListener};
5      } else if (o.otherListeners.size() > 0) {
6        int length = o.otherListeners.size() + 1;
7        n.listeners = new Object[length];
8        n.listeners[0] = o.firstListener;
9        for (int i = 0; i < o.otherListeners.size(); i++) {
10         n.listeners[i + 1] = o.otherListener.get(i);
11       }
12     }
13   }
14 }
```

Figure 2 A user-defined transformer for the example in Figure 1.

Both default transformations and TOS do not work for our motivating example because they only use *matched fields* (*i.e.*, fields with the exactly same name and type) to transfer information from the stale state to the new state. In other words, neither of them can find the relation between *unmatched fields*, *i.e.*, old fields (*e.g.*, `firstListener` and `otherListeners` and new fields (*e.g.*, `listeners`) that have different names or types. The only solution before this paper is to ask the developer to provide an object state transformer, which is a non-trivial procedure tightly coupled with program semantics and low-level implementations. Figure 2 presents a manually prepared transformer for the update in Figure 1. Even though there may be only a single stale state at the updating point, the transformer has to handle various stale object states and produces the new object states accordingly by directly manipulating

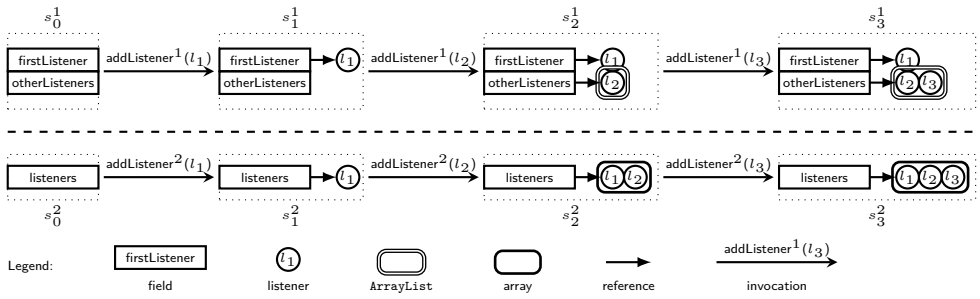

Figure 3 Object state evolution of `DefaultSshFuture`. s_i^v denotes the i-th state of the object in version v. Each state is depicted with a graphical representation of its data structure.

the data structure of the object.

2.2 Object Transformation Using Method Invocation History

Object transformations will be easy if the method invocation histories of objects are available. A *method invocation* consists of a method and a sequence of arguments, which may be empty if the method requires no arguments. A method invocation usually accepts some specific *input state* of the receiver object and produces an *output state* accordingly.

A *method invocation history* (invocation history for short) is a sequence of method invocations. Similarly, an invocation history accepts some specific *initial state*, *i.e.*, the input state of the first invocation, and produces the *final state*, *i.e.*, the output state of the last invocation. During replaying an invocation history, every method invocation must produce a valid output state as the input state for the consecutive method invocation in the history.

Two invocation histories are *equivalent* if they can yield the same final state when applied to the same initial state. For every object, there is a unique *actual invocation history*, including all method invocations applied to the object in the chronological order. An invocation history is *complete* if it can yield the current state of an object from the empty state, *e.g.*, the actual history. Note that nested methods are not included in an invocation history. For example, method `add` in Figure 4 can be included in an invocation history but nested methods such as `ensureCapacityInternal` cannot.

We have the following two *assumptions* for our approach:

1. The current state of an object is a summary of its past past method invocations. We can also recreate the current state of an object from its past method invocations, *i.e.*, replaying every method invocation on an object from the initial state.
2. The "role" or the behavior of an object is not changed during update [31]. The method invocation history usually keeps unchanged for such objects during updates that introduce no new functionalities, *e.g.*, bug fixes or performance improvements. Hence, the invocation history of the new state can be easily derived form the invocation history of the stale state.

Now, the idea can be explained in Figure 3 by an update of `DefaultSshFuture`. The program invokes `addListener`^{1} (In this paper superscripts denote program versions). with l_1, l_2 and l_3, respectively, and the update point (s_3^1) is reached. The transformed new-version object is synthesized by applying the invocation history, *i.e.*, invoking the new-version `addListener`^{2} with l_1, l_2 and l_3 on a newly allocated new-version object (s_0^2). State s_3^2 contains exactly the same sequence of listeners as s_3^1, indicating that this is a semantically correct state transformation. In contrast to default transformations and TOS, which cannot

connect unmatched fields, AOTES finds the relations between them by matching arguments of matched methods.

One limitation of our approach is that methods in the invocation history should be available in both versions. There is always a method with the same name and signature in the new version of an object whose interface is *not* changed. These methods are named *matched methods* for later discussion. AOTES does not insist that *every changed* class should preserve the binary compatibility. If some changed classes are binary incompatible, their callers must fix the incompatibility. In practice, there must be a direct or an indirect binary-compatible caller within a small scope, including one or two callers, since dynamic updating is often used for evolutionary changes (*e.g.*, build-to-build) rather than revolutionary changes (*e.g.*, release-to-release), and build-to-build changes usually do not introduce large patches. Suppose that a stale object's interface is changed. The invocation of an unmatched method on the object must be eventually enclosed in an invocation of a matched caller. We can enlarge the scope of the state being transformed to include all objects subject to the invocation of the matched caller and use the matched caller for history synthesis.

2.3 Synthesizing the Equivalent Invocation History

Recording method invocation histories for object transformations is *impractical* for long-running programs, because (1) we could not predict which objects were to be updated, thereby all method invocations on all objects would have to be recorded, which would introduce significant runtime overhead; (2) the log would be prohibitively expensive to store; and (3) replaying a long history could lead to a large service disruption during updating.

Alternatively, we try to find an *equivalent invocation history* that yields the same object state as the actual invocation history when applied to an object in the initial state, but is more *compact*, in terms of as few as possible redundant method invocations. For example, listeners can be added and removed for millions of times for a `DefaultSshFuture` object. However, at a specific execution point, only limited listeners are expected in the data structure. Any invocation history that yields exactly the same set of listeners suffices for a consistent object transformation.

AOTES synthesizes an object's equivalent invocation history from its current state without any logging. No history data need to be kept at runtime and no overhead is introduced when the program is not being updated. However, the synthesis of an invocation history is non-trivial because we need to first derive the arguments for a single method invocation and then find a valid history of method invocations. That means each method invocation must produce a valid output state as the input state for its consecutive method invocation in the synthesized history and the final state must be the current state of the object.

A naïve approach would enumerate all possible combinations of methods and arguments to determine the set of all possible method invocations. Since this searching space of invocation histories is huge, it is expensive to find an invocation history that can realize the state transition from the empty state to the current state of a given object. To narrow down the searching space for arguments [44, 5], execution synthesis techniques leverage on symbolic execution and constraint solvers. However, these approaches aim at searching for an execution path that reaches a particular statement, while AOTES aims at searching for an execution path that produces the given output state on a given input state of the receiver. In addition, these approaches are used for offline scenarios such as crash reproduction and search in the space of all execution paths, which may lead to a potentially unbounded searching time for real-world programs.

We observed that multiple execution paths of a method actually have the same purpose.

```
1  class ArrayList<E> {
2    int size; E[] elements = {};
3    public boolean add(E e) {
4      ensureCapacityInternal(size + 1);
5      elements[size++] = e;
6      return true;
7    }
8    private void ensureCapacityInternal(int minCapacity) {
9      if (minCapacity - elements.length > 0)
10       grow(minCapacity);
11   }
12   private void grow(int capacity) {...}
13   public boolean addAll(Collection<? extends E> c) {...}
14 }
```

Figure 4 A simplified version of class `ArrayList` in JDK.

To derive arguments of a method invocation, AOTES applies an offline analysis to populate a set of promising execution paths before dynamic updating, and during dynamic updating applies an online execution synthesis that considers these execution paths only. Typically, a method with multiple paths usually has a *fast path* that handles the most common situation and many *slow paths* that handle the rest cases. Moreover, the length of the execution path is usually guided by some input. We found that first the fast path is sufficient during execution synthesis for some methods, and second a long execution path guided by a large input can be replaced by many short execution paths guided by a small input.

Take the program in Figure 4 as an example. Method `add` has a fast path that appends the added element directly into the array (at line 5) and many slow paths that need to additionally calculate the new array size and expand the array (at line 10). AOTES can use the fast path only to synthesize the invocation history as if the array is initially allocated in the current size without any expansion. By this way, we can exclude the slow path (*i.e.*, the call to `grow`) during execution synthesis. The fast path of `add` can be expressed by the method in Figure 5. Besides, an invocation of `addAll` with a large input collection in the actual history can be replaced by many invocations of `addAll` with a small input collection, or even many invocations of `add` with a single element.

```
1 public boolean add(E e) {
2   if (size + 1 < elements.length)
3       elements[size++] = e;
4   return true;
5 }
```

Figure 5 The simplified equivalent version of method `add`.

To avoid backtracking, AOTES conducts a greedy backward searching starting from the current state instead of a forward searching starting from the initial state. Instead of every step searching for a method invocation that is applicable to a given input state, AOTES searches for a method invocation that can produce a given output state. This is because the initial input state (*i.e.*, the empty state) has zero information to guide the search, while the final output state (*i.e.*, the current state) has fruitful information. For example, if we synthesize a history for an array list from the empty state, we may include many method invocations that add or remove irrelevant elements. But if we synthesize the history backward from the current state, we can require that every method invocation must at least

contribute to a field with a non-default value in the current output state.

To facilitate the backward searching, AOTES converts each execution path into a separated *inverse method* by the offline analysis. The benefit is that we can simply make use of existing symbolic execution techniques to realize backward execution synthesis. Figure 6 shows an inverse method $\overline{\texttt{add}}$ of the method `add` shown in Figure 5. Here, an inverse method generally takes no arguments but only the receiver as input, reverts the receiver to a previous state (*e.g.*, line 2 in Figure 6), and finally returns an array of values (*e.g.*, line 4 in Figure 6). The return values can be used as the arguments to replay the invocation history of original methods. For example, suppose that an object of `ArrayList` is in state o, which contains two elements e_1 and e_2. After invoking $\overline{\texttt{add}}$ on the object, the state o becomes o', which contains only a single element e_1, and the return value of $\overline{\texttt{add}}$ is e_2. If we invoke `add` (in Figure 4) on o' with e_2, the state will be updated from o' to o again. Here, the input of a method consists of both the arguments and the state of the receiver, and the output of a method consists of both the return value and the final state of the receiver.

```
1 Object[] add() {
2   size--;
3   assert(size + 1 < elements.length);
4   return new Object[] { elements[size] };
5 }
```

Figure 6 The inverse method of `add` shown in Figure 5.

3 Approach Overview

Figure 7 presents an overview of our approach. AOTES aims at automatically constructing the new state s_n based on the current state s_c only. There must be an actual history $\mathcal{H}_a^1$ that leads to s_c. Instead of recording $\mathcal{H}_a^1$ from scratch, AOTES tries to synthesize a history $\mathcal{H}_s^1$ that can also lead to s_c. As the state of an object is a summary of its past method invocations, $\mathcal{H}_a^1$ and $\mathcal{H}_s^1$ should encode the same behavior accordingly. We assume that the role and the behavior of an object is unchanged during updating. Therefore, $\mathcal{H}_s^1$ can be used to recreate the new state s_n.

AOTES first conducts an offline analysis, which takes two versions of a program (P^1 and P^2) as input, and tries to produce the following output:

- A number of *execution summaries* (e_m) for each matched method m in P^1. Using

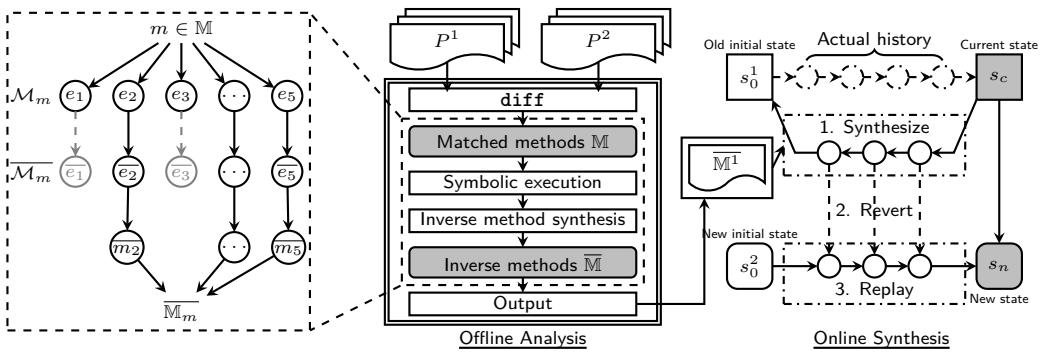

Figure 7 System overview.

symbolic execution, AOTES builds a map $\mathcal{M}_m : \mathcal{S} \times \mathcal{P} \rightarrow \mathcal{S}$ from the symbolic pre-state $\Sigma_{pre} \in \mathcal{S}$, *i.e.*, symbolic object state before applying this method, and the symbolic arguments $\Psi \in \mathcal{P}$, to the symbolic post-state $\Sigma_{post} \in \mathcal{S}$, *i.e.*, symbolic object state after applying this method.

- A number of *inverse execution summaries* $\overline{e_m} \in \overline{\mathcal{M}_m} : \mathcal{S} \rightarrow \mathcal{S} \times \mathcal{P}$, each of which corresponds to an execution summary in $\mathcal{M}_m$. An inverse execution summary takes a concrete state s aligned with the symbolic post-state Σ_{post} and computes a concrete state s' aligned with the symbolic pre-state Σ_{pre} and concrete arguments p such that applying m with p on an object in state s' will change its state to s.

To facilitate the online searching, AOTES serializes an inverse execution summary into an *inverse method*. An inverse method $\overline{m}$ takes only the receiver as input and reverts the state of the receiver to the state just before invoking the original method m. Moreover, it also returns all arguments required by the invocation of the original method m. For example, suppose that $\overline{\texttt{addListener}^1}$ is an inverse method of $\texttt{addListener}^1$. We can obtain s_2^1 if we apply $\overline{\texttt{addListener}^1}$ on s_1^1 with l_2. Conversely, we can obtain s_1^1 and l_2 if we apply $\texttt{addListener}^1$ on s_2^1.

Next, AOTES attempts to synthesize an *inverse method invocation history* (*inverse invocation history* for short) for the object and revert the object to the initial state. An inverse invocation history is a sequence of inverse methods and return values (*i.e.*, the corresponding reverted arguments). For example, $\langle \overline{\texttt{addListener}^1}/l_2, \overline{\texttt{addListener}^1}/l_1 \rangle$ is a synthesized inverse invocation history for s_2^1. The inverse invocation history can be synthesized by concatenating inverse methods as all inverse methods only take the object as input. Specifically, the inverse invocation history $\overline{\mathcal{H}_s^1}$ is synthesized by searching for a sequence of inverse execution summaries $\overline{e_{m_i}}, \overline{e_{m_{i-1}}}, \cdots, \overline{e_{m_1}}$, such that $\overline{e_{m_1}}(\overline{e_{m_2}}(\cdots(\overline{e_{m_i}}(s_c)))) = s_0$ and as well $e_{m_i}(e_{m_{i-1}}(\cdots(e_{m_1}(s_0)))) = s_c$.

Finally, AOTES constructs a new invocation history by inverting the inverse invocation history, substituting every inverse method with the new version of its original method, and using the return value of each inverse method as the arguments. The new state can be reified by applying the new invocation history on the new initial state. For example, an invocation history $\langle \texttt{addListener}^2(l_1), \texttt{addListener}^2(l_2) \rangle$ can be constructed by reverting the inverse invocation history $\langle \overline{\texttt{addListener}^1}/l_2, \overline{\texttt{addListener}^1}/l_1 \rangle$.

If the input of the original method cannot be derived by executing the inverse method, AOTES introduces a fresh symbolic variable for the input and leverages symbolic execution techniques to derive its value during online execution synthesis. AOTES considers a set of *short* execution paths as the promising candidates for execution synthesis. This is because long execution paths generally produce long path constraints that may not be solved by a constraint solver. Moreover, short paths can also help to mitigate the problem of long-running loop and deep recursive methods whose executions are guided by some input [43]. A long execution path guided by a very large input is replaced by many short execution paths, each of which is guided by a small input. An example about loop and recursion with detailed explanation is available in Section 4.4.

Figure 8 shows three inverse methods of $\texttt{addListener}^1$ (in Figure 1a) generated by AOTES. Note that the generated code has been simplified for brevity. We can obtain s_0^1 and l_1 when applying $\overline{\texttt{addListener1}^1}$ to s_1^1. Specifically, $\overline{\texttt{addListener1}^1}$ first loads l_1 from `firstListener` at line 2 and returns it at line 5, and then updates `firstListener` with a fresh symbolic value (denoted by a wild-card `*`) at line 3. The assertion at line 4 restricts the fresh symbolic value to be `null`, which can be derived by a constraint solver. Thereby, `firstListener` is `null` and the state becomes s_0^1 if the previous state is s_1^1.

```
1  Object[] addListener1^1() {
2    v0 = firstListener;
3    v1 = firstListener = *;
4    assert(v1 == null);
5    return { v0 };
6  }
7  Object[] addListener2^1() {
8    v0 = firstListener;
9    v1 = otherListeners;
10   v2 = v1.size;
11   v3 = v1.elements;
12   v4 = v3.length;
13   v5 = v3[0];
14   v3[0] = *;
15   assert(v4 == 1);
16   assert(v2 == 1);
17   v6 = otherListeners = *;
18   assert(v0 != null);
19   assert(v6 == null);
20   return { v5 };
21 }
22 Object[] addListener3^1() {
23   v0 = otherListeners;
24   v1 = v0.size;
25   v2 = v0.elements;
26   v3 = firstListener;
27   v4 = v1 - 1;
28   v0.size = v4;
29   v5 = v2[v4];
30   v2[v4] = *;
31   assert(v0 != null);
32   assert(v3 != null);
33   return { v5 };
34 }
```

Figure 8 Inverse methods of `addListener` in Figure 1a generated by AOTES. The execution traces of lines 5 and 6, lines 5, 8, 9, and 11, and lines 5, 8, and 11 in Figure 1a are postfixed with 1, 2 and 3, respectively. Note here we have simplified the generated code for brevity.

Similarly, we can obtain s_1^1 and l_2 by applying $\overline{\texttt{addListener2}^1}$ to s_2^1. Let's first analyze the original execution trace, *i.e.*, lines 5, 8, 9, and 11 in Figure 1a. Specifically, the argument `listerner` is l_2, and the input state is s_1^1, in which `firstListener` references l_1 and `otherListeners` is `null`. Then, `otherListeners` is assigned to a newly allocated `ArrayList`, in which `size` is initialized to 0 and `elements` is assigned to an empty array. After executing `otherListeners.add`, `elements` is expanded to an array of length 1, `size` is updated to 1, and the argument l_2 is placed at the first (index 0) slot of `elements`. Note that the `ArrayList` is the simplified implementation shown in Figure 4. Now we analyze the inverse method $\overline{\texttt{addListener2}^1}$ shown in Figure 8. Lines 8 and 18 assert that `firstListener` should not be `null`. Line 17 reverts `otherListeners` to a fresh symbolic value, which is further assigned to `null` by the assertion at line 19. Line 13 retrieves l_2 from `elements` by index 0 and line 20 returns l_2. We can easily verify that the state of the receiver is updated to s_1^1 at last, where `firstListener` still references l_1 and `otherListeners` is `null`. The third inverse method $\overline{\texttt{addListener3}^1}$ is similar, where the return value l_3 is retrieved from `elements` by index `size - 1`.

AOTES can sacrifice the completeness because it does not aim at synthesizing all possible transformations. The objects that AOTES cannot handle may be disposed at a later update point. On the other hand, a fixed number of inverse methods are sufficient for all transformations in practice. In Section 5.1, we will show that three inverse methods are sufficient for all transformations of `DefaultSshFuture`.

4 Inverse Program Synthesis

The insight of AOTES is to synthesize an inverse method from a symbolic execution trace not from all traces. We first give a high level overview of the symbolic execution technique of AOTES followed by a detailed description before illustrating the details.

4.1 Symbolic Execution of AOTES

In general, *symbolic execution* [25] is a technique to interpret a program with symbolic values instead of concrete values. A symbolic value is a formula over a set of *symbolic inputs*, and can be evaluated to a concrete value by substituting symbolic inputs with concrete values and then evaluating the formula.

AOTES populates a certain number of symbolic execution traces from a matched method to generate inverse methods. The symbolic execution technique of AOTES needs to allocate objects with explicit types, because dynamic method dispatching should know the type of each object. This requirement makes it non-trivial to symbolically execute an arbitrary method of an object, because the heap, or at least the receiver, must be instantiated in to a proper shape before execution. We name this heap *pre-heap* (Π_{pre}).

For example, suppose that a symbolic execution trace of `addListener` in Figure 1a explores lines 5, 8 and 11. Invoking `add` at line 11 should trigger a `NullPointerException` (NPE) if `otherListeners` does not reference an object. Otherwise, we have no idea about exploring which `add` method. To avoid the NPE and continue the execution, one can allocate an object in the pre-heap for `otherListeners` before the execution. However, we have no idea about the type for object allocation as there may be numerous subclasses of `List`. The type for object allocation should be as exact as possible. Here, the type must be `ArrayList` not any other type.

During the symbolic execution, an object is either *pre-allocated* in the pre-heap before execution or *newly allocated* during execution. The type of a newly allocated object is known at its allocation site. For pre-allocated objects, AOTES maintains a shared dictionary $\mathbb{S}$ that maps an *access path* (*e.g.*, `this.otherListeners`) to a set of types. A type is randomly picked out for the pre-allocated object if there are multiple types for an entry. AOTES will produce inverse methods for each randomly chosen type. During runtime execution synthesis, only inverse methods that match the actual type are applicable.

The dictionary $\mathbb{S}$ is empty at first and updated by traversing the heap at the end of every successful symbolic execution, which is named *post-heap* (Π_{post}). Note that at any time, only *live* objects in a heap are of interest. The execution trace that explores lines 5, 8 and 11 depends on the type at `this.otherListeners` in $\mathbb{S}$. Hence, the execution should be first suspended at line 11 and resumed until some other symbolic execution trace updates the entry. Fortunately, the execution that explores lines 5, 8, 9 and 11 can update the entry. Line 9 allocates an `ArrayList` for `otherListeners`. The entry at `this.otherListeners` in $\mathbb{S}$ is updated by `ArrayList`.

To update missing entries in $\mathbb{S}$, AOTES dynamically collects extra methods to execute. If the missing entry is rooted at the receiver, all methods of the receiver are added. If the missing entry is rooted at an argument of the entry method of the symbolic execution, all callers of the entry method are added. Callers of a method are determined by a call graph. AOTES constructs a static call graph at first and refines it when invoking a method during symbolic execution.

4.2 Program and Execution Definitions

This subsection gives a detailed description of the symbolic execution technique of AOTES. A program in AOTES is a set of classes. A class $\mathbb{C}$ is a set of fields $\mathbb{F}$ and a set of methods $\mathbb{M}$, which also include those inherited from super classes. Every method has a *receiver* and an optional sequence of *parameters*. A method is a sequence of Java virtual machine bytecode instructions [29]. A bytecode instruction may allocate new objects, create new values, copy

Table 1 Bytecode instructions.

Type	Instructions
Stack & Local	`ldc` c, `load` i, `store` i
Array Access	`aload`, `astore`
Field Access	`getfield` f, `putfield` f
Allocation	`new` $\mathbb{C}$, `newarray`
Binary Operator	`add`, `sub`, `mul`, `div`, `rem`
Branch	`ifgt` ρ, `ifeq` ρ, `iflt` ρ, `goto` ρ
Invoke & Return	`invoke` m, `return`

or move existing values, and evaluate branch conditions and change control flow accordingly.

We group all bytecode instructions into seven groups, which are shown in Table 1. A bytecode instruction may have one operand encoded with it. In a nutshell, this kind of operand may be an array index i, a field f, a method m, a class $\mathbb{C}$, a constant c, or an offset ρ of instruction index. AOTES can handle almost all bytecode instructions, except `invokedynamic`. This is because `invokedynamic` usually needs to execute a piece of custom code to resolve the callee. We list a single `invoke` instruction only without showing various method dispatching semantics (*e.g.*, `invokevirtual` and `invokespecial`), because AOTES tracks the type of every object and method dispatching for these invoke instructions is straightforward if we know the receiver type.

An object, *i.e.*, an *instance* of a class or an *array*, is defined as a tuple of $(\mathbb{C}, \mathbb{L})$, in which $\mathbb{C}$ is its type and $\mathbb{L}$ is its heap locations. $\mathbb{C}$ is either the class of the instance, or a generic array type, which means that we do not distinguish array element types. A variable that can appear in symbolic input and output (actually as a fresh symbolic value) is represented by a *location* θ, which may be a *named location* or a *heap location* of an object. A named location is either the receiver or a method parameter of the entry method of the symbolic execution. A heap location is either an *object field* or *array element* and denoted by (o, α), in which o is the reference of the object, α is either a field f or a symbolic value i representing the array index.

AOTES organizes symbolic values into a *value graph*. A node in the value graph represents a symbolic value, and is a tuple of (t, P, a), in which t is the *type* of the node, P is a set of predecessors, and a is an optional type-specific attribute associated with this node. There are six types of nodes.

1. *Constant*: $(\textsc{Const}, \varnothing, c)$, where c is the constant literal in an `ldc` instruction.
2. *Reference*: $(\textsc{Ref}, \varnothing)$. The heap is a mapping between reference values and objects. A `new` or `newarray` instruction allocates a new object and creates a reference value for the object to retrieve the object from the heap,
3. *Expression*: $(\textsc{Expr}, \{v_1, v_2\}, op)$, where v_1 and v_2 are two operand values and op is a binary operator, *i.e.*, one of `+`, `-`, `*`, `/`, and `%`.
4. *Assertion*: $(\textsc{Assert}, \{v_1, v_2\}, op)$, where v_1 and v_2 are two operand values and op is a relational operator, *i.e.*, one of `>`, `>=`, `==`, `!=`, `<` and `<=`. An *opposite* operator (*e.g.*, `<=` for `ifgt`) is used when a `false` branch is taken.
5. *Input*: $(\textsc{Input}, \varnothing, \theta)$, where θ is a location in the pre-heap or a method parameter.
6. *Output*: $(\textsc{Output}, \{v_\theta\}, \theta)$, where θ is a location in the post-heap and v_θ is the value of θ.

Figure 9 summarizes the effects of each bytecode instructions in terms of the modification of a *configuration*. A configuration is a refection of the runtime of a running Java program, and is composed of the following components, denoted as $(\mathcal{F}, \Pi, \Phi, \Sigma)$ for brevity.

- $\mathcal{F}$, the stack for method frames.
- Π, the symbolic heap, a mapping from values to objects.
- Φ, the path condition, actually a sequence of ASSERT.
- Σ, the symbolic state, a mapping from variable (locations) to values (nodes).

A method frame is denoted by a tuple $(m, pc, \mathcal{L}, \mathcal{E})$ of the method m, the current bytecode index pc, the local variables array $\mathcal{L}$, and the expression stack $\mathcal{E}$ for bytecode instructions [29]. Since most instructions are intra-procedure, we ignore the method m and a configuration is also denoted by a sextuple $(pc, \mathcal{L}, \mathcal{E}, \Pi, \Phi, \Sigma)$.

Rules in Figure 9 actually define an structural operational semantics [35] of each bytecode instruction over the node (t, P, a). The detailed semantics of each bytecode instruction can be found in [29]. Table 2 summarizes the symbols used in describing every rule.

Table 2 Symbols used in the rules

Symbol	Description
σ, σ'	a configuration $(\mathcal{F}, \Pi, \Phi, \Sigma)$
$\langle$`ldc` c$, \sigma\rangle \Rightarrow \sigma'$	a rule for the instruction `ldc` c
v, o, i	a generic value, a reference value and an index value, respectively
$\Pi[\![o]\!]$	obtain the object referenced by o
$\Pi[\![o = (\mathbb{C}, \mathbb{L})]\!]$	update the heap and make o reference the object $(\mathbb{C}, \mathbb{L})$
$\Sigma[\![(o, i)]\!]$	read the value of the location (o, i)
$\Sigma[\![(o, i) = v]\!]$	update the value of the location (o, i)
$\mathcal{F} \cdot (m, pc, \mathcal{L}, \mathcal{E})$	push a frame $(m, pc, \mathcal{L}, \mathcal{E})$ to the method frame stack $\mathcal{F}$
$\mathcal{E} \cdot v$	push a value v into the expression stack

Not all bytecode instructions produce values, *e.g.*, an `invoke` only copies arguments from the caller to the callee. For the entry method, AOTES creates a CONST and allocates a pre-allocated object for its receiver, and creates an INPUT for each method parameter of it. INPUT in pre-allocated objects are created when first used. At the end of a *normally terminated* execution, AOTES creates an OUTPUT for every heap location in objects reachable from the receiver. Exceptional executions are abandoned.

Note that in symbolic execution, which branch (*i.e.*, `true` or `false`) is taken is not determined by evaluating the condition to a concrete value but by a strategy. AOTES takes a random strategy to explore branches and collect path conditions. First, it randomly takes an *unvisited* branch. After all branches have been visited, it then randomly takes a *visited* branch. For any method, we only collect a path condition of a limited length. Loop and recursion are discussed in detail in Section 4.4.

Finally, we create a value graph to summarize an execution. There are two kinds of edges between nodes, representing *value dependency* or *location dependency*. The location dependency tracks values in heap locations of INPUT and OUTPUT (*e.g.*, object reference and array index), and is used to align the symbolic post-heap to a concrete heap. The value graph is constructed as follows. Initially, the value graph is empty. Then, all INPUT, OUTPUT, ASSERT are first added to the value graph. Other nodes are recursively added by following the two kinds of dependency edges.

Figure 10 depicts three value graphs of `addListener` in Figure 1a. Note that we simplify the implementation of method `add` of class `ArrayList` for brevity but AOTES can handle the actual one. Lets take the left-most graph as an example to illustrate the semantics of a value graph. The value graph contains the following nodes and edges.

Stack & Locals

$$\langle \texttt{ldc}\ c, (pc, \mathcal{L}, \mathcal{E}, \Pi, \Phi, \Sigma)\rangle \Rightarrow (pc+1, \mathcal{L}, \mathcal{E}\cdot(\textsc{Const}, \varnothing, c), \Pi, \Phi, \Sigma)$$
$$\langle \texttt{load}\ i, (pc, \mathcal{L}, \mathcal{E}, \Pi, \Phi, \Sigma)\rangle \Rightarrow (pc+1, \mathcal{L}, \mathcal{E}\cdot\mathcal{L}[\![i]\!], \Pi, \Phi, \Sigma)$$
$$\langle \texttt{store}\ i, (pc, \mathcal{L}, \mathcal{E}\cdot v, \Pi, \Phi, \Sigma)\rangle \Rightarrow (pc+1, \mathcal{L}[\![i=v]\!], \mathcal{E}, \Pi, \Phi, \Sigma)$$

Array Access

$$\langle \texttt{aload}, (pc, \mathcal{L}, \mathcal{E}\cdot o\cdot i, \Pi, \Phi, \Sigma)\rangle \Rightarrow (pc+1, \mathcal{L}, \mathcal{E}\cdot\Sigma[\![(o,i)]\!], \Pi, \Phi, \Sigma)$$
$$\langle \texttt{astore}, (pc, \mathcal{L}, \mathcal{E}\cdot o\cdot i\cdot v, \Pi, \Phi, \Sigma)\rangle \Rightarrow (pc+1, \mathcal{L}, \mathcal{E}, \Pi, \Phi, \Sigma[\![(o,i)=v]\!])$$

Field Access

$$\langle \texttt{getfield}\ f, (pc, \mathcal{L}, \mathcal{E}\cdot o, \Pi, \Phi, \Sigma)\rangle \Rightarrow (pc+1, \mathcal{L}, \mathcal{E}\cdot\Sigma[\![(o,f)]\!], \Pi, \Phi, \Sigma)$$
$$\langle \texttt{putfield}\ f, (pc, \mathcal{L}, \mathcal{E}\cdot o\cdot v, \Pi, \Phi, \Sigma)\rangle \Rightarrow (pc+1, \mathcal{L}, \mathcal{E}, \Pi, \Phi, \Sigma[\![(o,f)=v]\!])$$

Allocation

$$\langle \texttt{new}\ \mathbb{C}, (pc, \mathcal{L}, \mathcal{E}, \Pi, \Phi, \Sigma)\rangle \Rightarrow (pc+1, \mathcal{L}, \mathcal{E}\cdot o, \Pi[\![o=(\mathbb{C},\mathbb{L})]\!], \Phi, \Sigma) \wedge o \leftarrow (\textsc{Ref}, \varnothing)$$
$$\langle \texttt{newarray}, (pc, \mathcal{L}, \mathcal{E}\cdot v, \Pi, \Phi, \Sigma)\rangle \Rightarrow (pc+1, \mathcal{L}, \mathcal{E}\cdot o, \Pi[\![o=(\mathbb{A},\mathbb{L})]\!], \Phi, \Sigma[\![(o,\iota)=v]\!]) \wedge o \leftarrow (\textsc{Ref}, \varnothing)$$

Binary Operator

$$\langle \texttt{add}, (pc, \mathcal{L}, \mathcal{E}\cdot v_1\cdot v_2, \Pi, \Phi, \Sigma)\rangle \Rightarrow (pc+1, \mathcal{L}, \mathcal{E}\cdot(\textsc{Expr}, \{v_1, v_2\}, +), \Pi, \Phi, \Sigma)$$

Branch

$$\langle \texttt{ifgt}\ \rho, (pc, \mathcal{L}, \mathcal{E}\cdot v_1\cdot v_2, \Pi, \Phi, \Sigma)\rangle \Rightarrow (pc+\rho, \mathcal{L}, \mathcal{E}, \Pi, \Phi\cup(\textsc{Assert}, \{v_1, v_2\}, >), \Sigma), \text{if } \texttt{true}$$
$$\langle \texttt{ifgt}\ \rho, (pc, \mathcal{L}, \mathcal{E}\cdot v_1\cdot v_2, \Pi, \Phi, \Sigma)\rangle \Rightarrow (pc+1, \mathcal{L}, \mathcal{E}, \Pi, \Phi\cup(\textsc{Assert}, \{v_1, v_2\}, <=), \Sigma), \text{if } \texttt{false}$$
$$\langle \texttt{goto}\ \rho, (pc, \mathcal{L}, \mathcal{E}, \Pi, \Phi, \Sigma)\rangle \Rightarrow (pc+\rho, \mathcal{L}, \mathcal{E}, \Pi, \Phi, \Sigma)$$

Invoke & Return

$$\langle \texttt{invoke}\ m, (\mathcal{F}\cdot(m', pc', \mathcal{L}', \mathcal{E}'\cdot o\cdot v_1\cdots v_n), \Pi, \Phi, \Sigma)\rangle \Rightarrow (\mathcal{F}\cdot(m, 0, \mathcal{L}[\![0=o]\!][\![1=v_1]\!]\cdots[\![n=v_n]\!], \varnothing), \Pi, \Phi, \Sigma)$$
$$\langle \texttt{return}, (\mathcal{F}\cdot(m', pc', \mathcal{L}', \mathcal{E}'\cdot o\cdot v_1\cdots v_n)\cdot(m, pc, \mathcal{L}, \mathcal{E}\cdot v), \Pi, \Phi, \Sigma)\rangle \Rightarrow (\mathcal{F}\cdot(m', pc', \mathcal{L}', \mathcal{E}'\cdot v), \Pi, \Phi, \Sigma)$$

Figure 9 Rules describing effects of bytecode instructions. Each rule is in the format $\langle \texttt{inst}, \sigma\rangle \Rightarrow \sigma'$, where `inst` is a bytecode instruction, σ and σ' are the configurations before and after execution the instruction, respectively.

- Two Const nodes *w.r.t.* `this` and `null`.
- Two Input nodes *w.r.t.* `firstListener` and the parameter.
- An Output *w.r.t.* `firstListener`. This Output has a location dependency edge from `this` (Const) and a value dependency edge from the parameter (Input).
- An Assert *w.r.t.* the `if` statement at line 5 in Figure 1a. This Assert has value dependency edges from `firstListener` (Input) and `null` (Const).

AOTES aims at deriving values for the two Input nodes from a given concrete object. The derivation is conducted by traversing the graph. First, AOTES derives the concrete value for Output by loading `firstListener` after aligning `this` to the concrete object. Next, AOTES derives the value for the parameter (Input) using the value of `firstListener` (Output) directly. Note that we need to check whether the Assert satisfies. The Input (*w.r.t.* `firstListener`) has been overridden during forward execution. We then create a fresh symbolic value for the Input and try to use the constraint solver to derive a value for it. All deriving steps are serialized into a method to facilitate the online execution synthesis, which will be discussed in the next subsection.

4.3 Inverse Method Synthesis

We say that a node is *resolved* if its concrete value has been derived. An inverse method is created by resolving all Input. An Output can be directly resolved if its symbolic location can be *aligned* to a concrete location. `this` can be directly aligned to the receiver. An object field can be aligned if its object reference is resolved. Thus, all fields accessed from `this` can be aligned. An array element can be aligned if both the object reference and index are

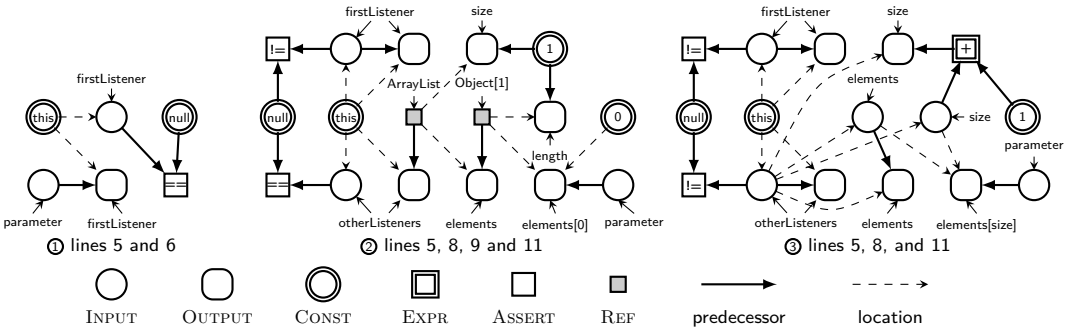

Figure 10 Value graphs of three execution traces of `addListener` in Figure 1a. Their inverse methods are in Figure 8.

resolved. AOTES translates every resolution and alignment into a statement. All statements finally make up the inverse method. AOTES supports four kinds of resolution methods.

1. *Direct Resolution*: All Const and aligned Output can be directly resolved.
2. *Forward Resolution*: A node is resolved if all predecessors are resolved. For example, if $a = b \times c$, and b and c are resolved, then a can also be resolved by evaluating $b \times c$ again.
3. *Backward Resolution*: If an Expr and one of its predecessors are resolved, we can resolve the other predecessor by these two nodes. For example, if $a = b - c$, and a and b are resolved, then c can be resolved by evaluating $b - a$. We treat `+`, `-`, `*`, and `/` *invertible* due to the aggressive nature of AOTES.
4. *Aggressive Resolution*: As an inverse method is used for execution synthesis, we can aggressively guess a value for an Input by assigning a fresh symbolic value to it. Besides, we can guess an index for an array element if its object reference has been resolved.

Algorithm 1 aims at resolving all nodes of a value graph. The algorithm maintains a sequence of statements $\overline{m}$, and two sets of nodes, R and U, *i.e.*, the sets of resolved and unresolved nodes, respectively. At first, R and $\overline{m}$ are empty, and U contains all nodes in the value graph. The four resolution methods try to apply rules defined in Figure 11 and return `true` if there is an applicable rule, which means some nodes have been resolved. Every successful resolution appends a statement to $\overline{m}$ (denoted by $\uplus$). In theory, we can continue to apply aggressive resolution to resolve every Input and then use forward resolution to resolve all unresolved nodes in the value graph. The algorithm can finally terminate when R is fixed, since a value can never be moved from R to U. $\overline{m}$ is successfully generated only if U is empty. We then decorate $\overline{m}$ into a valid Java method. This method has no parameter and returns all reverted arguments.

Every node is indeed converted into a variable with a unique name. Every resolved Input must be aligned first and its concrete location is also updated with the resolved value. For presentation, this requirement is not expressed in the rules. A fresh symbolic value is denoted by `*` but in fact produced by a runtime method. We also provide a runtime method `guess` that chooses an index in a given array.

Figure 11 presents rules that are used to resolve a node. A rule takes a node from the value graph and the currently visiting status (*i.e.*, the tuple $(R, U, \overline{m})$) as input to update the visiting status for next visiting and produce a statement for the inverse method $\overline{m}$ as output. Each rule has a precondition that should be checked first. Basically, the precondition at least ensures that each node is resolved once by a rule. Take rules of direct resolution as an example. To resolve a Const, the rule only checks whether the node being resolved has

Algorithm 1: Resolution of a value graph.

Input: (V, E), the value graph.
Output: $(R, U, \overline{m})$, where R is the set of resolved nodes, U is the set of unresolved nodes, and $\overline{m}$ is the inverse method.

```
1  (R, U, m̄) ← (∅, V, ∅) foreach v ∈ U do DIRECT(v, (R, U, m̄))
2  repeat
3  |  repeat
4  |  |  foreach v ∈ U do FORWARD(v, (R, U, m̄))
5  |  |  foreach v ∈ R do BACKWARD(v, (R, U, m̄))
6  |  until R is fixed
7  |  foreach v ∈ U do
8  |  |  if AGGRESSIVE(v, (R, U, m̄)) then break
9  until R is fixed
10 return (R, U, m̄)
```

been resolved. To resolve an OUTPUT, the two rules further check whether the location has been aligned.

Figure 8 shows the inverse methods created by resolving nodes from value graphs in Figure 10. We have simplified the output, *e.g.*, remove redundant variables for CONST. Algorithm 1 and rules in Figure 11 ensure that a node is only resolved once. We randomly visit nodes and thereby, a node can be resolved via different rules and nodes. For consistency, AOTES attempts to resolve every node using a different method at last and adds assertions to ensure that all resolved concrete values must be equal, *e.g.*, lines 15 and 16.

4.4 Loop and Recursion

AOTES takes a single-path symbolic execution and limits the length of the path condition. Hence, the loop and recursive method invocations are unrolled for a limited length. A set of short execution paths is considered as the promising candidates for online execution synthesis. Obviously, there do exist long-running loops and deep recursions. Thus, the symbolic execution trace may be infeasible for some inputs (pre-heaps).

Actually, the problem of loop and recursion may not be as critical as it seems to be. Recall that AOTES has no need to synthesize an inverse method for all execution traces. Besides, a fixed number of inverse methods are sufficient sometimes. In comparison with existing whole program execution techniques [44, 5], the insight of invocation history synthesis is that it infers the sequence from the state only and requires no complete control flow and call graph. Moreover, many loops and recursive methods are guided by some input [43]. AOTES can split a loop with a very large input in the actual history into many loops with a small input in the synthesized history.

Take the class in Figure 12 as an example. Method `addN` has a loop and also recursively calls itself. AOTES can easily populate the execution trace where `n` is `1` and also synthesize an inverse method for it, because `addN(a,1)` is equivalent to `elements.add(a)`. We have shown that AOTES can easily handle `ArrayList`. Hence, no matter how divergent the actual history is, AOTES can always guarantee to synthesize a history $\texttt{addN(e}_0\texttt{,1)}, \ldots, \texttt{addN(e}_i\texttt{,1)}, \ldots, \texttt{addN(e}_{k-1}\texttt{,1)}$, where $\texttt{e}_i$ is the i-th element in `elements` and k is the size of the list `elements`. For example, an actual history composed of an `addN([a,b],2)` would be replaced by the following synthesized history: `addN(a,1)`, `addN(a,1)`, `addN(b,1)`, `addN(b,1)`, where both of them fill the `ArrayList` referred to by `elements` with the sequence `[a, a, b, b]`.

Direct Resolution

$$\frac{v \notin R}{\langle v = (\textsc{Const}, \varnothing, c), (R, U, \overline{m})\rangle \Rightarrow (R \cup \{v\}, U \setminus \{v\}, \overline{m} \uplus \texttt{v = c;})}$$

$$\frac{v \notin R \wedge \theta = (o, i) \wedge o \in R \wedge i \in R}{\langle v = (\textsc{Output}, \{v_\theta\}, \theta), (R, U, \overline{m})\rangle \Rightarrow (R \cup \{v\}, U \setminus \{v\}, \overline{m} \uplus \texttt{v = o[i];})}$$

$$\frac{v \notin R \wedge \theta = (o, f) \wedge o \in R}{\langle v = (\textsc{Output}, \{v_\theta\}, \theta), (R, U, \overline{m})\rangle \Rightarrow (R \cup \{v\}, U \setminus \{v\}, \overline{m} \uplus \texttt{v = o.f;})}$$

Forward Resolution

$$\frac{v \notin R \wedge v_1 \in R \wedge v_2 \in R}{\langle v = (\textsc{Expr}, v_1, v_2, \texttt{+}), (R, U, \overline{m})\rangle \Rightarrow (R \cup \{v\}, U \setminus \{v\}, \overline{m} \uplus \texttt{v = v1 + v2;})}$$

$$\frac{v \notin R \wedge v_1 \in R \wedge v_2 \in R}{\langle v = (\textsc{Assert}, v_1, v_2, \texttt{>}), (R, U, \overline{m})\rangle \Rightarrow (R \cup \{v\}, U \setminus \{v\}, \overline{m} \uplus \texttt{assert(v1 > v2);})}$$

Backward Resolution

$$\frac{v_1 \notin R \wedge v \in R}{\langle v = (\textsc{Output}, \{v_1\}, \theta), (R, U, \overline{m})\rangle \Rightarrow (R \cup \{v_1\}, U \setminus \{v_1\}, \overline{m} \uplus \texttt{v1 = v;})}$$

$$\frac{v_2 \notin R \wedge v \in R \wedge v_1 \in R}{\langle v = (\textsc{Expr}, v_1, v_2, \texttt{+}), (R, U, \overline{m})\rangle \Rightarrow (R \cup \{v_2\}, U \setminus \{v_2\}, \overline{m} \uplus \texttt{v2 = v - v1})}$$

Aggressive Resolution

$$\frac{\theta = (o, i) \wedge o \in R \wedge i \notin R}{\langle i, (R, U, \overline{m})\rangle \Rightarrow (R \cup \{i\}, U \setminus \{i\}, \overline{m} \uplus \texttt{i = guess(o);})}$$

$$\frac{v \notin R \wedge \Pi_{pre}[\![v]\!] = \varnothing \wedge \theta = (o, i) \wedge o \in R \wedge i \in R}{\langle v = (\textsc{Input}, \varnothing, \theta), (R, U, \overline{m})\rangle \Rightarrow (R \cup \{v\}, U \setminus \{v\}, \overline{m} \uplus \texttt{v = o[i] = *;})}$$

$$\frac{v \notin R \wedge \Pi_{pre}[\![v]\!] = \varnothing \wedge \theta = (o, f) \wedge o \in R}{\langle v = (\textsc{Input}, \varnothing, \theta), (R, U, \overline{m})\rangle \Rightarrow (R \cup \{v\}, U \setminus \{v\}, \overline{m} \uplus \texttt{v = o.f = *;})}$$

Figure 11 Rules for resolving nodes and generating statements. Each rule is in the format $\frac{P}{\langle v,(R,U,\overline{m})\rangle \Rightarrow (R',U',\overline{m}')}$, where P is the precondition of applying the rule, v is the node that we attempt to resolve, R is the set of resolved nodes, U is the set of unresolved nodes, $\overline{m}$ is the sequence of generated statements, and P', U' and $\overline{m}'$ are new versions after applying the rule.

5 Execution Synthesis

This section depicts the online synthesis and replaying of invocation histories.

5.1 Online Synthesis of Invocation Histories

AOTES uses a greedy strategy to search for an inverse invocation history. As shown in Algorithm 2, AOTES first collects all applicable inverse method invocation ($\mathcal{T}$), and uses a heuristic method to rank them (by function Rank). Intuitively, a better inverse method should revert more Input in the Π_{pre} from non-default values to default values and preserve more locations in the Π_{pre}, which is the Π_{post} for the next step. Hence, AOTES prefers the inverse method with no aggressively resolved Input first, then more live locations after execution, and finally more reverted Input. The searching stops when the object is in the empty state or there is no applicable inverse method. AOTES executes an inverse method in two ways, *i.e.*, TestApply, which will restore the object state for applying next inverse method, and Apply, which will retain the modification.

For example, suppose that an object of `DefaultSshFuture` is in state s_3^1. $\overline{\texttt{addListener2}^1}$ is not applicable as line 16 in Figure 8 fails, *i.e.*, the `size` is not 2. Both $\overline{\texttt{addListener1}^1}$ and $\overline{\texttt{addListener3}^1}$ are applicable, but we prefer $\overline{\texttt{addListener3}^1}$ over $\overline{\texttt{addListener1}^1}$ as it reverts more locations and also preserves `otherListeners`. The object state then becomes s_2^1. $\overline{\texttt{addListener2}^1}$ is still not applicable on s_2^1 as line 15 fails, *i.e.*, the array has been expanded and its length is not 1. We then prefer $\overline{\texttt{addListener3}^1}$ for the same reason and apply it to obtain s_1^1. Now, only $\overline{\texttt{addListener1}^1}$ is applicable on s_1^1. $\overline{\texttt{addListener3}^1}$ is

```
1 class LoopAndRecursion {
2   List elements = new ArrayList();
3   void addN(Object o, int n) {
4     if (n < 1) {
5       return;
6     } else if (o instanceof Object[]) {
7       for (Object e : (Object[]) o) {
8         addN(e, n);
9       }
10    } else {
11      elements.add(o);
12      addN(o, n-1);
13 }}}
```

Figure 12 An example of loop and recursion.

Algorithm 2: History synthesis.

Input: o, the receiver object for history synthesis, $\overline{\mathbb{M}}$, the set of inverse methods.
Output: $\mathcal{H}$, the invocation history for o.

$\overline{\mathcal{H}} \leftarrow \varnothing$
while isNotEmptyState(o) **do** ▷ *Stop once the object is reverted into the empty state.*
  $\mathcal{T} \leftarrow \varnothing$ ▷ *The set of applicable inverse method invocation at this step.*
  foreach $\overline{m} \in \overline{\mathbb{M}}$ **do**
    $\mathbf{a} \leftarrow$ TestApply($\overline{m}, o$) ▷ *Apply $\overline{m}$ to o and restore o afterwards.*
    $\mathcal{T} \leftarrow \mathcal{T} \cup \{(\overline{m}, \mathbf{a})\}$
  if $\mathcal{T} = \varnothing$ **then**
    break ▷ *Stop when there is no applicable inverse method.*
  $(\overline{m}, \mathbf{a}) \leftarrow$ Rank($\mathcal{T}$) ▷ *Heuristic-based ranking.*
  $\overline{\mathcal{H}} \leftarrow \overline{\mathcal{H}} \cup \{(\overline{m}, \text{Apply}(\overline{m}, o))\}$ ▷ *Apply $\overline{m}$ to o without restore.*
return Revert($\overline{\mathcal{H}}$) ▷ *Revert $\overline{\mathcal{H}}$ and replace every $\overline{m}$ by its original method m.*

inapplicable on s_1^1 as line 28 attempts to revert `size` from `0` to `-1`. At last, there is no applicable inverse method and the synthesis terminates.

5.2 Realizing Object Transformations

AOTES realizes object transformations as follows. Given a stale object, we first try to synthesize an inverse invocation history for it. If the history is empty, then we fall back to default transformations. Otherwise, we apply a default transformation to the object after reverting its state by applying the inverse invocation history. Finally, we invert the inverse invocation history to build a new history and apply it to the object. Note that the synthesized history is not necessarily to be complete.

6 Implementation

We implemented AOTES, including the symbolic execution engine, inverse method synthesizer and invocation history synthesizer, in about 25K lines of Java code. AOTES is fully automated and only takes binary class files as input and thus requires no source code, no test case and no human specified update points.

We implemented a trivial single-variable solver. For example, a fresh symbolic value for `int` includes all integers in [`MIN_INT`, `MAX_INT`]. As such a symbolic value is mostly used in assertions and pre-states. Therefore, it narrows its range towards passing the assertion

and to the default value during evaluation. For example, suppose that a variable `v1` has a fresh symbolic value for `int`. After evaluating the assertion `assert(v1 == 0)`, its range is narrowed to $[0, 0]$, which means that this symbolic value can only be `0`. Currently, we are working on integrating Z3 [9] as the solver to further improve the effectiveness of AOTES.

The main limitation of AOTES in analyzing real-world applications is uninterpreted native methods. The current implementation of AOTES only handles a small part of native methods that we have encountered during evaluation, among which some are re-implemented using Java, *e.g.*, `arraycopy`, and others are manually marked as operators, *e.g.*, `sin` and `identityHashCode`. Operator methods are not interpreted during symbolic execution and their effects are recorded like an operator (*i.e.*, creating an Expr). During synthesis, they can be executed as all arguments are available at present. We allocate a phantom object for every class as the container for static fields. The reference of the phantom object is treated as a Const and thus a static field can be easily aligned.

7 Experiments

We evaluated AOTES's effectiveness with real-world updates and performance in synthesizing long histories using a micro benchmark, respectively. All experiments were conducted on an Intel Core i7 3.4GHz machine with 20 GB memory running 64-bit Windows 10. The offline synthesis was conducted on JDK 1.8.0_65 and the dynamic updating was carried out on Javelus. We forced AOTES to only explore at most 20 different traces for a method and 1000 branches for a trace.

7.1 Real-world Updates

We collected 21 updated classes from Apache Commons Collections, Apache FTP Server, Apache SSHD Server and Apache Tomcat, which are all widely used common libraries and server applications under years of active development. These updates were chosen for the following reasons. First, the two versions of all updates must be successfully compiled. Second, all updates *must* involve field changes otherwise would require no transformation. We classified these fields changes into the following four types:

1. Value Change: with no field added, but the values of some fields need to be updated.
2. Name Change: with a field renamed only.[3]
3. Type Change: with a type-changed field only.
4. Complex Change: any other changes.

Third, an updated class must not invoke uninterpreted native methods beyond those handled by the current implementation of AOTES. Finally, we also excluded rare cases in which the stale state does not contain sufficient information to determine the new state. In this situation, even a programmer may not be able to provide a transformer based on the stale state without additional information, not to mention TOS or AOTES.

In addition to existing test cases, we additionally wrote a few test cases for some updated classes under our test frame work designed for DSU, because most updated classes have no test case and some existing test cases were insufficient to detect improper object transformations. Every test created an object with one or a few method invocations before

[3] Note that if either the name or type of a field is changed, it is considered as deleted and a new field with the new name or type is added.

dynamic updating. Then, we triggered the dynamic updating and applied the transformation to the object. Every dynamic update was verified as follows [33, 30]. That is, the state after dynamic updating of the old version must be equivalent to a state that can be achieved by executing the same methods on the new version.

We ran all tests with dynamic updating for both AOTES and default transformations on Javelus. All results are shown in Table 3. AOTES succeeded in 51 (83.6%) updates and failed in other 10 updates due to *incomplete* or *inconsistent* synthesized histories. An incomplete history cannot update all fields as the searching also stops when no applicable inverse method is found. This is mainly because many native methods prevent AOTES from generating sufficient inverse methods. We plan to model more native methods in future. An inconsistent history leads to the same state as the actual history in the old version but different states in the new version. We will discuss this limitation with examples in the following paragraphs.

We did not run TOS with dynamic updating as TOS is not fully automated and requires extra training tests and manually specified update points. Instead, we used our validation tests to train TOS and hope that it could synthesize a conditional transformer for each update that can realize transformations for all test cases. TOS failed to synthesize a function for 16 of 21 updates (marked with N.A. in Table 3). For the rest 5 updates with 12 tests in total, TOS even failed in validating its output against 6 training tests.

As shown in Table 3, almost all updates have field name changes or type changes. These updates are the majority of updates that require transformations in practice. Default transformations and TOS failed to derive a valid transformation/transformer even for many name and type changes because they cannot find the relations between fields with different names or types. AOTES can infer their relations when changed fields used the same arguments in matched methods. Moreover, both default transformations and TOS used a set of predefined simple rules, and cannot handle transformations involving custom type conversions (*e.g.*, `ArrayList` to `ConcurrentHashMap`). AOTES leveraged program code to infer custom type conversions when objects of different types used the same arguments in matched methods for initialization.

We discuss the limitation and effectiveness of AOTES with the following four examples. AOTES failed in the first two examples and succeeded in the last two examples.

7.1.1 Value Change: Tomcat dd741c

```
1 - private String jmxNameBase = "pool";
2 + private String jmxNameBase = null;
```

This update only changes the initial value of `jmxNameBase` in the constructor. If the setter of `jmxNameBase` is not invoked, the transformation should update the value to `null`. AOTES failed in two test cases due to inconsistent histories. That is, both the constructor and the setter method can assign `"pool"` to `jmxNameBase` in the old version but `null` and `"pool"` in the new version. In fact, even a programmer cannot write a *general* transformer here because using the current state only cannot distinguish different actual histories that lead to the same current state.

7.1.2 Name Change: FTP 5d5592

```
1 - private int maxIdleTimeMillis = 10000;
2 + private int idleTime = 300;
3   public void setIdleTime(int idleTime) {
```

Table 3 Results of real-world updates.

Type	Update	Tests	AOTES	Default	TOS
Value Change	tomcat-dd741c	6	$4^{\alpha)}$	4	N.A.
Name Change	ftp-5d5592	4	$3^{\alpha)}$	0	N.A.
	sshd-6f8507	2	2	1	N.A.
	tomcat-951e08	2	2	1	N.A.
	tomcat-b75f5c	2	2	1	N.A.
Type Change	collections-0e1140	1	1	0	N.A.
	collections-cdacd4	1	1	0	N.A.
	ftp-f8110b	5	5	0	N.A.
	sshd-054334	3	3	1	N.A.
	tomcat-480edc	3	3	0	N.A.
Complex Change	collections-b7327a	2	2	0	N.A.
	ftp-43ff5f	4	$2^{\beta)}$	0	N.A.
	ftp-4907aa	4	$2^{\beta)}$	0	N.A.
	ftp-5df186	4	$2^{\beta)}$	0	N.A.
	sshd-009d83	5	5	0	N.A.
	sshd-1487b0	1	1	0	1
	sshd-2297b2	1	1	0	1
	sshd-b98694	7	7	2	2
	sshd-eeeec6	1	$0^{\alpha)}$	0	1
	tomcat-24bc4d	2	2	1	1
	tomcat-2db0f7	1	1	0	N.A
	Total	61	51/83.6%	11/18.3%	6/9.8%

a) α and β indicates inconsistent and incomplete synthesized history, respectively.

```
-   maxIdleTimeMillis = idleTime * 1000;
+   this.idleTime = idleTime;
  }
```

Except this update, we can just copy the value from a new field to the old field for all Name Change updates. AOTES failed in the only test for the same reason as Tomcat dd741c. `maxIdleTimeMillis` was set to `10000` by the constructor in the old version but in the new version `idleTime` should be 300.

7.1.3 Type Change: FTP f8110b

```
class DefaultFtpletContainer {
- private List ftplets = new ArrayList();
- class FtpletEntry { String name; Ftplet ftplet; }
+ private Map ftplets = new ConcurrentHashMap();
  public void addFtplet(String name, Ftplet ftplet) {
-   ftplets.add(new FtpletEntry(name, ftplet));
+   ftplets.put(name, ftplet);
  }
}
```

Type conversions between built-in types are easy, *e.g.*, `int` to `long`. However, for this update, we need the key to convert an `ArrayList` to a `ConcurrentHashMap`. AOTES can synthesize inverse methods for `addFtplet` and the constructor and also a history using them. The key can be inferred from the parameter `name` of the new version of `addFtplet`.

7.1.4 Complex Change: SSHD 009d83

```
class AgentImpl {
  private List keys = new ArrayList();
- private boolean closed;
+ private AtomicBoolean open = new AtomicBoolean(true);
  public void close() throws IOException {
-   closed = true;
-   keys.clear();
+   if (open.getAndSet(false)) {
+     keys.clear();
+   }
  }
  public void addIdentity(KeyPair key, String comment) {
    keys.add(new Pair(key, comment));
  }
}
```

This update changes both the type and the name of a field. Method `close` removes all elements in `keys`. Hence, the synthesized history is a constructor and a `close` if the last method in the actual history is a `close`. For example, suppose that the actual history includes a constructor of `AgentImpl`, an `addIdentity`, and a `close`. However, a critical field `modCount` in `ArrayList`, which is used to avoid concurrent modification during iterating the list, prevents AOTES from applying the inverse constructor of `AgentImpl` if its value cannot be reverted to 0. Fortunately, the inverse method of `clear` decrements `modCount`. As a result, the synthesized history is a constructor followed by two invocations of `close`.

7.2 Micro Benchmark

We selected five classes of commonly used collections and designed a micro benchmark to evaluate the synthesizing time of AOTES. Theoretically, the synthesizing time only directly depends on the current object state and the number of inverse methods but not the actual history. Thereby, we first conducted experiments with all synthesized inverse methods and then repeated the experiments with a small set of inverse methods. The results help us to reveal solutions that can optimize the synthesizing time of AOTES.

The micro benchmark created an object of each class, filled it with a number of elements (ranged from $0, 10, \ldots, 90$), and finally synthesized a history for it. We also repeated the procedure for 10 times first to warm up the JVM. Figure 13a shows the distribution of inverse methods generated by AOTES for all *public* methods of every class. We first ran the benchmark on *all* inverse methods and then repeated the benchmark with *only previously used* inverse methods.

The synthesizing time using all inverse methods is shown in Figure 13b. The time depends on the size of elements in a collection. AOTES spent more than 5s in the worst case for `Vector`. This is mainly because our implementation heavily uses reflections and exceptions. Besides, most of the inverse methods of these classes were indeed redundant.

The number of inverse method candidates has an impact to the searching time. Figure 13c shows the number of *different inverse methods* (not *different inverse method invocations*) appeared in each history. No more than four inverse methods were actually used for all histories. That means the online synthesis wasted a certain amount of time on trying out redundant inverse methods. Note that here an inverse method may be invoked for many times. The actual number of method invocations in total were mostly the same as the number of elements.

Pruning redundant inverse methods can speed up the online history synthesis. In practice, we can prune redundant inverse methods using an automatic random testing tool [36].

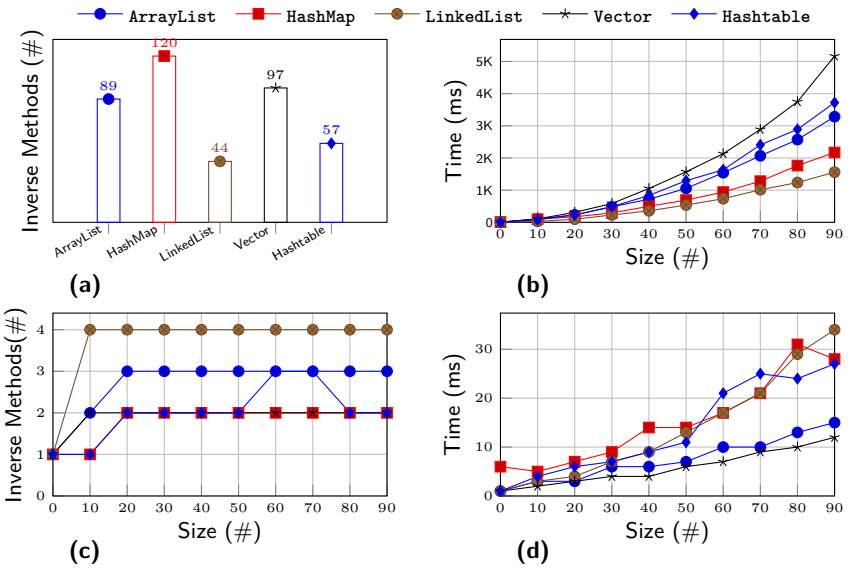

Figure 13 Results of micro benchmark.

Figure 13d shows the synthesizing time using only previously used inverse methods. AOTES only spent 35ms in the worst case for `LinkedList` and only 12ms for `Vector`. We believe that this synthesizing time is acceptable for practical usage and can also be further reduced with a more efficient implementation of AOTES.

7.3 Discussion

AOTES can be used as a complement to existing techniques such as default transformations, TOS, and manual approaches, especially for name changes, type changes and complex changes. While both default transformations and TOS cannot find the relations between old fields and new fields that have different names or types, AOTES can find the relations by matching the arguments in matched methods. Different from TOS, which requires test cases and manual efforts during collecting transformation examples, AOTES is *fully* automated and works purely on binaries without source code and test cases. AOTES is the only approach that can leverage the program code to infer powerful transformations. Moreover, it is an on-demand dynamic approach and can avoid synthesizing transformation that are hard to be automated but may not be encountered during dynamic updating.

The two assumptions of AOTES (Section 2.2) may be a threat. Techniques such as random testing [36] can help to reveal the violation of assumptions before updating. The time of online execution synthesis may be another threat to AOTES, particularly when there are many stale objects. AOTES tackles this challenge by adopting a lazy updating mechanism and sharing searching strategies across different transformations. Specifically, AOTES first mitigates the disruption caused by synthesis using a lazy updating technique, which is naturally supported by Javelus. Second, AOTES can try out methods that have already been used only. The effectiveness has been demonstrated in the micro benchmark. In other words, AOTES shares the searching strategy across different transformations, while TOS and manual approaches share the transformer.

8 Related Work

We survey related work in this section, including dynamic software updating, and program and execution synthesis.

8.1 Dynamic Software Updating

In general, DSU systems can be divided in two types, *intra-process state transformation* and *inter-process state transfer*. Many challenges in implementing inter-process state transfer have made it not so popular in building DSU systems [21, 12]. Not all programs can run multiple instances of multiple versions simultaneously, particularly in a production environment [21], *e.g.*, the Linux kernel. In contrast, this approach has been extensively studied in live migration of virtual machines [6].

The majority of DSU systems apply transformations to the intra-process states. These approaches can generate default transformations and support programmers specified transformations as well [41, 13, 15]. Besides, they also provide a safety guarantee, *e.g. type safety*, to facilitate developing transformers [23, 34, 4, 32, 41, 14]. The transformations can be taken eagerly [41, 42], or lazily [14, 15], or both [38]. Although the size of a valid transformer may not be great [2], it should be delivered with extremely timing constraints, *e.g.*, for security patches.

Automated approaches such as TOS [31] and TTST [12] require a pair of matched objects as example and infer transformations from these examples. AOTES requires no example as it uses matched methods, which makes it able to handle non-trivial cases that TTST and TOS cannot handle. Other approaches for debugging prefer no user intervention by sacrificing the flexibility or validity [11, 24]. Gupta *et al.* have proved that the validity of general dynamic updating is undecidable [17]. Besides, existing programming techniques can also help transformer programming, *e.g.*, formalization and verification [20, 26, 45] and software testing [19, 22].

8.2 Program and Execution Synthesis

Program Synthesis and *Execution Synthesis* [44] have been extensively studied for years. Among them most related to AOTES are inverse program generation [10, 40] and data transformations [16, 18, 27, 39]. AOTES combines the program synthesis and execution synthesis. AOTES indeed makes use of reverse execution [3, 7, 1] over *symbolic execution traces* to generate an inverse program.

9 Conclusion

AOTES is an experimental approach to automating object transformations for dynamic software updating. It preserves the continuity of stateful behavior of objects whose classes are changed at runtime. The novelty of AOTES is to synthesize a method invocation history that can produce the current object state in the old version, and replay the history to get the desired state for the new version. Our preliminary evaluation shows that AOTES has the promising ability to handle software updates taken from real-world software systems. Although the current implementation of AOTES is for Java only, we believe that the general idea of AOTES can also apply to other object-oriented programming languages. In the future, we plan to improve AOTES by supporting more native methods and searching strategies, and also conduct a thorough evaluation of AOTES with more real-world updates.

References

1 Tankut Akgul and Vincent J. Mooney III. Assembly instruction level reverse execution for debugging. *ACM Transaction Software Engineering Methodology*, 13(2):149–198, 2004.

2 Jeff Arnold and M. Frans Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the 4th ACM European Conference on Computer Systems*, pages 187–198, 2009.

3 Bitan Biswas and R. Mall. Reverse execution of programs. *SIGPLAN Notices*, 34(4):61–69, 1999.

4 Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. POLUS: A powerful live updating system. In *Proceedings of the 29th International Conference on Software Engineering*, pages 271–281, 2007.

5 N. Chen and S. Kim. STAR: Stack Trace Based Automatic Crash Reproduction via Symbolic Execution. *IEEE Transactions on Software Engineering*, 41(2):198–220, 2015.

6 Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, pages 273–286, 2005.

7 Jonathan J. Cook. Reverse execution of Java bytecode. *The Computer Journal*, 45(6):608–619, 2002.

8 Robert C. Daley and Jack B. Dennis. Virtual memory, processes, and sharing in MULTICS. *Communications of the ACM*, 11(5):306–312, 1968.

9 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

10 Edsger W. Dijkstra. Program inversion. In *Program Construction*, volume 69, pages 54–57. 1979.

11 Mikhail 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.

12 Cristiano Giuffrida, Calin Iorgulescu, Anton Kuijsten, and Andrew S. Tanenbaum. Back to the future: Fault-tolerant live update with time-traveling state transfer. In *Proceedings of the 27th Large Installation System Administration Conference*, pages 89–104, 2013.

13 Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Safe and automatic live update for operating systems. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 279–292, 2013.

14 Tianxiao Gu, Chun Cao, Chang Xu, Xiaoxing Ma, Linghao Zhang, and Jian Lu. Javelus: A low disruptive approach to dynamic software updates. In *Proceedings of 19th the Asia-Pacific Software Engineering Conference*, pages 527–536, 2012.

15 Tianxiao Gu, Chun Cao, Chang Xu, Xiaoxing Ma, Linghao Zhang, and Jian Lü. Low-disruptive dynamic updating of Java applications. *Information and Software Technology*, 56(9):1086–1098, 2014.

16 Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 317–330, 2011.

17 Deepak Gupta, Pankaj Jalote, and Gautam Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131, 1996.

18 William R. Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 317–328, 2011.

19 Christopher M. Hayden, Eric A. Hardisty, Michael Hicks, and Jeffrey S. Foster. Efficient systematic testing for dynamically updatable software. In *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*, pages 9:1–9:5, 2009.

20 Christopher M. Hayden, Stephen Magill, Michael Hicks, Nate Foster, and Jeffrey S. Foster. Specifying and verifying the correctness of dynamic software updates. In *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments*, pages 278–293, 2012.

21 Christopher M. Hayden, Edward K. Smith, Michael Hicks, and Jeffrey S. Foster. State transfer for clear and efficient runtime updates. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering Workshops*, pages 179–184, 2011.

22 C.M. Hayden, E.K. Smith, E.A. Hardisty, M. Hicks, and J.S. Foster. Evaluating dynamic software update safety using systematic testing. *IEEE Transactions on Software Engineering*, 38(6):1340–1354, 2012.

23 Michael Hicks and Scott Nettles. Dynamic software updating. *ACM Transactions on Programming Languages and Systems*, 27(6):1049–1096, 2005.

24 Jevgeni Kabanov and Varmo Vene. A thousand years of productivity: the JRebel story. *Software: Practice and Experience*, 2012.

25 James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

26 V.P. La Manna, J. Greenyer, C. Ghezzi, and C. Brenner. Formalizing correctness criteria of dynamic updates derived from specification changes. In *Proceedings of the 2013 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 63–72, 2013.

27 Vu Le and Sumit Gulwani. FlashExtract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 542–553, 2014.

28 Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 36–44, 1998.

29 Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

30 Xiaoxing Ma, Luciano Baresi, Carlo Ghezzi, Valerio Panzica La Manna, and Jian 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*, pages 245–255, 2011.

31 Stephen Magill, Michael Hicks, Suriya Subramanian, and Kathryn S. McKinley. Automating object transformations for dynamic software updating. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 265–280, 2012.

32 Kristis Makris and Rida A. Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Proceedings of the Conference on USENIX Annual Technical Conference*, 2009.

33 Iulian Neamtiu and Michael Hicks. Safe and timely updates to multi-threaded programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24, 2009.

34 Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for c. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 72–83, 2006.

35 Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1 edition, 1992.

36 Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, pages 75–84, 2007.

37 Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot™ server compiler. In *Proceedings of the 2001 Symposium on Java™ Virtual Machine Research and Technology Symposium - Volume 1*, pages 1–12, 2001.

38 Luís Pina, Luís Veiga, and Michael Hicks. Rubah: DSU for Java on a stock JVM. In *Proceedings of the 2014 International Conference on Object Oriented Programming Systems Languages Applications*, pages 103–119, 2014.

39 Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In *Proceedings of the 24th International Conference on Computer Aided Verification*, pages 634–651, 2012.

40 Saurabh Srivastava, Sumit Gulwani, Swarat Chaudhuri, and Jeffrey S. Foster. Path-based inductive synthesis for program inversion. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 492–503, 2011.

41 Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates: A VM-centric approach. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2009.

42 Thomas Würthinger, Christian Wimmer, and Lukas Stadler. Dynamic code evolution for Java. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 10–19, 2010.

43 Xusheng Xiao, Sihan Li, Tao Xie, and Nikolai Tillmann. Characteristic studies of loop problems for structural test generation via symbolic execution. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 246–256, 2013.

44 Cristian Zamfir and George Candea. Execution synthesis: A technique for automated software debugging. In *Proceedings of the 5th European Conference on Computer Systems*, pages 321–334, 2010.

45 Min Zhang, Kazuhiro Ogata, and Kokichi Futatsugi. Formalization and verification of behavioral correctness of dynamic software updates. *Electronic Notes in Theoretical Computer Science*, 294(0):12–23, 2013.