Synthesizing Object State Transformers for Dynamic Software Updates

Zelin Zhao*, Yanyan Jiang*, Chang Xu*, Tianxiao Gu†, Xiaoxing Ma*
*State Key Laboratory for Novel Software Technology and
Department of Computer Science and Technology, Nanjing University, China
†Alibaba Group, Sunnyvale, CA, USA

***Abstract*—There is an increasing demand for evolving software systems to deliver continuous services of no restart. Dynamic software update (DSU) aims to achieve this goal by patching the system state on the fly but is currently hindered from practice due to non-trivial cross-version object state transformations. This paper revisits this problem through an in-depth empirical study of over 190 class changes from Tomcat 8. The study produced an important finding that most non-trivial object state transformers can be constructed by reassembling existing old/new version code snippets. This paper presents a domain-specific language and an efficient algorithm for synthesizing non-trivial object transformers over code reuse. We experimentally evaluated our tool implementation PASTA with real-world software systems, reporting PASTA's effectiveness in succeeding in 7.5× non-trivial object transformation tasks compared with the best existing DSU techniques.**

***Index Terms*—Software maintenance and evolution, dynamic software update, object transformation, program synthesis.**

I. Introduction

Dynamic software update (DSU, updating software at runtime without restarting) [1] is a trending feature in modern software systems. DSU keeps systems up-to-date with security patches, bug fixes, and feature upgrades without hurting the systems' availability. DSU has become increasingly practical and compelling [1–8]: Linux Kernel [9–12] and Microsoft Windows [13] are already dynamically updatable to some extent; Java Virtual Machine (JVM) has been modified to partially support application updates [14–18]; live-upgradable components are also emerging in databases [19–21], servers [22–24], and even mission-critical systems [25].

Despite that *code* can be hot upgraded in emerging systems [24, 26], automatically updating *runtime states* for seamless system evolution remains a major research challenge [15, 27, 28]. Software updates may include a field of a class being added, removed, or semantically changed in a new version. In DSU, such field values (if not removed) should be (re)computed to match the new version code's semantics. This is known as the *object transformation* problem, whose solution typically relies on a key mini-program (*a.k.a.* a *transformer*) for computing these field values at update time.

To understand the challenges in object transformation, this paper empirically studied 100 uniform-randomly sampled commits (consisting of 190 class changes) from Apache Tomcat 8 [23], one of the most popular Web backend systems. Our major findings include:

1) Almost all (187, or 98.4%) class changes can be updated dynamically, indicating that DSU is broadly applicable. Even for a few cases (3, or 1.6%) that DSU is impossible over existing programs, proper refactoring could still make them updatable [4, 29, 30].
2) Most (166, or 87.4%) updates involve trivial object transformations over simple predefined rules. Existing DSU systems [15, 17, 18] are already capable of automatically updating these changes without developers' intervention.
3) The rest, not many but a non-negligible portion of class changes (21, or 11.1%) require non-trivial object transformations. Software developers without a DSU background would have substantial difficulties in specifying them, as the required transformers have to carefully manipulate two versions of program states simultaneously.

The empirical study results suggest that the key obstacle that hinders the continuous and automatic deployment of DSU in practice is probably how to obtain *non-trivial* object transformers. Unfortunately, this circumstance has not been seriously recognized by existing research. In fact, our later experiments show that state-of-the-art techniques, like TOS [27] and AOTES [28], could only succeed in 0 and 2 out of 25 non-trivial object transformation tasks. Their apparent high success rates in past experiments might be due to mixing non-trivial transformers with many trivial ones.

In this paper, we leverage another key empirical finding that *object transformers can be constructed by reassembling existing old/new version code* to establish an algorithm for synthesizing object transformers in the DSU of Java applications. The algorithm exhaustively and heuristically enumerates all possible combinations of extracted code snippets, producing both test-passing and developer-readable object transformers. A key advantage over existing techniques [27, 28] is that an application developer can easily verify synthesized transformers' correctness because application code is their major constructs.

We implemented our algorithm as the PASTA (PATCH STATES) tool for DSU of Java programs. The evaluation results over a set of non-trivial class changes (including those in the empirical study and more) were encouraging: PASTA synthesized 7.5× correct non-trivial object transformers (60.0%) compared to the best existing techniques TOS [27] and AOTES [28] (0.0% and 8.0%, respectively).

In summary, this paper's major contributions are recognizing the non-trivial object transformer synthesis as a critical problem

in DSU and providing it with an effective approach. The rest of the paper is organized as follows. Section II gives the necessary background knowledge of DSU with a motivating example. Section III presents a comprehensive study on DSU of 190 class changes in Tomcat 8. Our DSL and synthesis algorithm are elaborated on in Sections IV and V, respectively. The evaluation of PASTA against real-world updates is described in Section VI, followed by threats to validity discussions in Section VII, related work in Section VIII, and conclusion in Section IX.

II. Background and Motivation

A. DSU Systems and Object Transformation

This paper focuses on the DSU of Java programs[1], which consists of the following four steps:

1) *Pause the program under update at a safe point* [10, 31], *e.g.*, when all updated code is popped off the stack [15, 17].
2) *Upgrade the changed code* [32, 33] via dynamic linking [34], live patching [10], or hotswap [26].
3) *Transform stale (old-version) objects in the heap to their new state* [27, 28].
4) *Resume the updated program's execution*. The new version is now ready to serve.

Object transformation (the third step) is this paper's primary focus. When a program is paused at an update-safe point with code being upgraded, the heap may contain *stale* objects whose values are inconsistent with the new-version code. A DSU system must for each such object invoke its *transformer* to migrate to its corresponding new-version.

B. Motivating Example

Figure 1 lists a class change to `SocketProcessor`, which requires a non-trivial transformation. This class change replaces the `socket` field by `ka` with a type change from `NioChannel` to `KeyAttachment` (Lines 2–3). We correspondingly provide an object transformer `DSUHelper.transform` (Lines 16–28).

The `status` field undergoes a *default* (or *trivial*) transformation: it inherits its value from the old-version (Line 18). A default transformation copies the old-version value for a type-unchanged field or assigns a default value (*e.g.*, 0 for `int` and `null` for references) to a newly-added field [15, 17].

However, the `ka` field requires a *non-trivial transformation*[2]. If we leave `ka` with a default `null` reference, the program will quickly crash after DSU. Our transformer in Figure 1 leverages the program's implicit invariant that there is a 1-to-1 mapping between `NioChannel` objects and `KeyAttachment` objects in the heap. Lines 21–26 invoke a chain of I/O channel APIs to find `stale.socket`'s corresponding `KeyAttachment` object.

Providing non-trivial object transformers is considerably challenging even for experienced developers: it requires expertise in both the application logic and DSU system, where the latter is typically lacking for most application developers. Sometimes, a DSU system may automatically synthesize a non-trivial object transformer, however, our empirical study results in Section III show that existing techniques fall short on most real-world cases. For this motivating example, TOS [27] incorrectly falls back to the default `null` assignment because the non-trivial transformer is beyond TOS's search capability. AOTES [28] also fails in synthesizing a method history for such complex objects.

```
1 class SocketProcessor {
2 - private NioChannel socket = null;
3 + private KeyAttachment ka = null;
4   private SocketStatus status = null;
5   public void run() {
6 +   NioChannel socket = ka.getSocket();
7     SelectionKey key = socket.getIOChannel().keyFor(
8         socket.getPoller().getSelector());
9 -   KeyAttachment ka = null;
10 -  if (key != null)
11 -    ka = (KeyAttachment)key.attachment();
12    ... } ...
13 }
14
15 class DSUHelper {
16   static void transform(SocketProcessor* obj, SocketProcessor stale) {
17     // trivial default transformation for status
18     obj.status = stale.status;
19     // non-trivial transformation for ka
20     obj.ka = null;
21     NioChannel socket = stale.socket;
22     if (socket != null) {
23       SelectionKey key = socket.getIOChannel().keyFor(
24           socket.getPoller().getSelector());
25       if (key != null)
26         obj.ka = (KeyAttachment)key.attachment();
27     }
28   }
29 }
```

Fig. 1: A class change in Tomcat-8 (commit `#f4451c`) whose object transformation is non-trivial. `DSUHelper` is our manually provided object transformer. At update time, the DSU system for each (stale) object `sp` (of type `SocketProcessor`) creates its corresponding uninitialized new-version object `sp*` (of type `SocketProcessor*`, the same class after update) and invokes the object transformer `DSUHelper.transform(sp*, sp)`.

C. Discussions

Interestingly, the key non-trivial step in our manually provided transformer, which retrieves the `SelectionKey` from an `NioChannel` object in Lines 23–24, is identical to the code in Lines 7–8. The null-check in the transformer (Lines 25–26) can also be found in the old-version code (Lines 10–11), which is removed in the new version because the local variable `ka` is available through the newly added field (Line 3).

This should not be considered completely incidental. If there is a code snippet for computing an object's property that reflects an internal invariant (potentially useful for object transformation like the code that finds the `SelectionKey` for a given `NioChannel` object), the code snippet might also be useful to other parts of the program and is likely to exist in the source code.

[1]DSU and object transformation for unmanaged heaps (*e.g.*, C/C++) are considerably different and are out of this paper's scope. However, arguments in this paper can also be applied to other managed runtime systems.

[2]An object transformer is considered *trivial* if it contains only default transformations, otherwise is *non-trivial*.

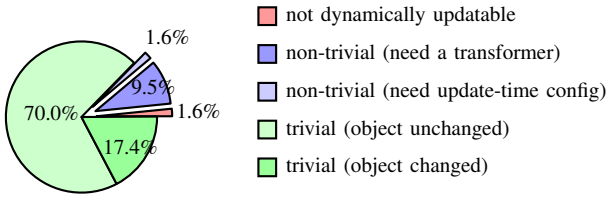

Fig. 2: Taxonomy of the 190 changed classes in Tomcat 8.

This observation motivates us to explore the possibility of automatically synthesizing object transformers by reassembling existing code snippets, including both old and new versions of a program. This observation is further validated in our empirical study in Section III, and then implemented as a heuristic search algorithm in Section V.

III. Empirical Study

In this short empirical study, we seek insights for understanding the challenges of object transformation in DSU over a set of randomly sampled real-world class changes.

A. Methodology

We empirically studied the applicability of DSU in the evolution of Apache Tomcat 8 [23], one of the most popular Java Web servers. Tomcat 8 is still under active maintenance upgrades since its first release in 2013, making it a suitable subject for studying DSU. We uniform-randomly sampled 100 commits from all 2,114 Tomcat 8 commits in its entire maintenance history by the paper was written (from 8.0.0 to the latest release 8.0.53). The sampled commits consist of in total 190 class changes[3].

For each changed class, we manually inspected the program state at a hypothetical update-safe point in which all changed methods of the class are popped off the stack. We determine whether object transformation is possible (*i.e.*, whether DSU is applicable) at that point and try to provide each of 2,957 fields in the 190 changed classes a transformer. Given a class change that can be dynamically updated, its object transformer is considered *trivial* if all of its field transformations are default (explained in Section II). Otherwise, the *non-trivial* object transformer has at least one field that requires non-trivial transformation (like `ka` in Figure 1).

To validate our observation that non-trivial transformers can be constructed by reassembling existing code snippets, we preferred reusing old/new version code statements with minor revisions. We collect and analyze the statistics of those reused statements in constructing transformers.

B. Results and Findings

The statistics in Figure 2 first indicate that DSU can be broadly applicable in a program's maintenance lifetime:

[3]Commits that do not change the Tomcat-core source code (*e.g.*, documentation or test case updates) are excluded from the study because they are irrelevant to DSU. 190 are all class changes because changes to Tomcat 8 are mainly maintenance upgrades.

Finding 1. *Almost all changed classes (187/190, or 98.4%) are dynamically updatable using either trivial default or non-trivial provided object transformers.*

In the three failing cases, two of them added new fields whose values are only available in an already popped stack frame. Another one is a fix for a resource leak in which whether an object is leaked cannot be effectively determined. Fortunately, refactoring the program to discard partial states at a component level [4, 29, 30] can make them updatable.

Furthermore, we found that simple default object transformation suffices in most cases:

Finding 2. *Most class changes (166/190, or 87.4%) can be dynamically updated via trivial object transformers.*

133 out of the 166 class changes (80.1%) involve only code logic upgrades that do not affect the concerned objects' data representations, *i.e.*, field values are unchanged. A typical example is a security patch. The rest 33 (19.9%) class changes can be automatically handled by a DSU system's default policy [15, 17, 18], *e.g.*, assigning a newly created field with a default value or garbage collecting a removed field's referred objects.

Finally, class changes that require non-trivial object transformers are of particular research interest:

Finding 3. *Not many but non-negligible class changes (21/190, or 11.1%) require non-trivial object transformers*[4]*. These changes substantially hinder the application of DSU in practice.*

For these updates, the upgrade maintainer can manually provide an object transformer to enable DSU over such non-trivial class changes. However, this is not an easy task because non-trivial object transformers usually exploit a program's implicit invariants or object state constraints (like the example in Figure 1). Automatic transformer synthesis [27, 28] can be a promising and highly-preferred solution. Unfortunately, our later experiments show that even the best state-of-the-art technique produces correct transformations in $<$10% of these non-trivial class changes.

Therefore, the general unavailability of non-trivial object transformers should be recognized as a key obstacle in making DSU practical. To address this challenge, we examined the characteristics of our manually crafted object transformers to find potentially useful guidance for automatic object transformer synthesis. Figure 3 summarizes the basic constructs in our manual transformers, which can be concluded by the following finding:

Finding 4. *Default transformations and existing code snippets are the major constructs of a non-trivial object transformer.*

The basic constructs of the 21 non-trivial object transformers are: 42 right-hand side expressions of assignments, 15 if-then-else branch conditions, and 2 for-each loop conditions. Understanding the characteristics of these basic constructs is critical to the development of an automatic transformer

[4]Dynamically updating such a class with a default transformer will result in a crash or broken application logic.

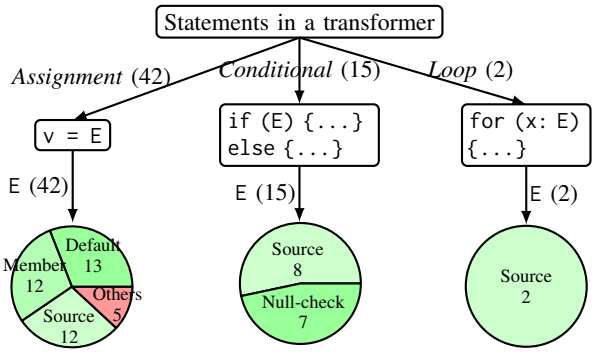

Fig. 3: Statistics of the basic constructs in the studied non-trivial object transformers.

Transformer	t	::=	s^*
Statement	s	::=	v `=` e`;`
			| `obj.`f `=` e`;`
			| `if (`c`) { `s^*` } else { `s^*` }`
			| `while (`c`) { `s^*` }`
Condition	c	::=	e | e `==null` | `!`c
Expression	e	::=	v | $g(v^*)$
Gadget	g	::=	extracted gadgets
Variable	v	::=	`stale` | v_1 | v_2 | ...

Fig. 4: Syntax of basic constructs in object transformers. f is a field subject to transformation; `stale` and `obj` are the stale object and its corresponding new-version object; a^* denotes zero or more repeats of a.

synthesis mechanism. As shown in Figure 3, the vast majority (54/59, or 91.5%) of the basic constructs are either:

1) a trivial *default* behavior (13/59, or 22.0%),
2) a single *member* method call (12/59, or 20.3%),
3) a simple *null-check* (7/59, or 11.9%), or
4) a minor revision of an existing *source code snippet* (22/59, or 37.3%) like Lines 23–24 in Figure 1.

Such a result motivated us to synthesize object transformers by assembling source code *gadgets* (extracted expressions from old/version source code) upon a domain-specific language designated for the object transformation in DSU.

For the remaining a few (5/59, or 8.5%) basic constructs, three of them are `boolean` configuration-related constants whose values are determined by an update-time configuration. The other two expressions need a reference that is not reachable from stale objects. Since this paper focuses on the automatic synthesis of object transformers, we leave these relatively rare cases to future work.

IV. Domain-Specific Language for Object Transformation

This section explains our design goals and choices in our domain-specific language (DSL) for describing object transformers. The DSL design and gadget extraction are described in Sections IV-A and IV-B, respectively.

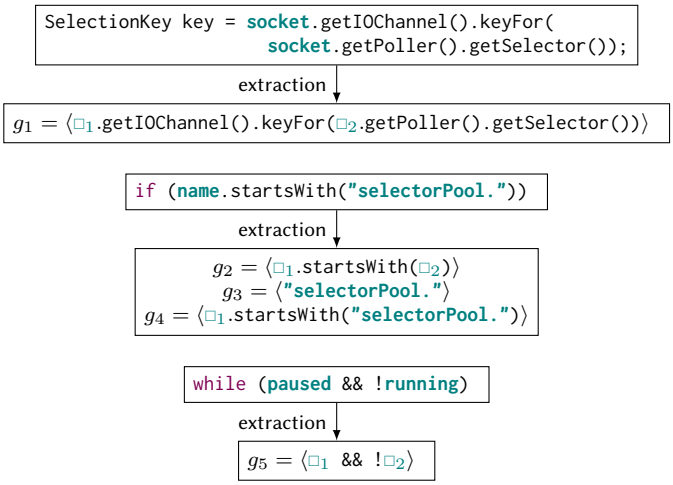

Fig. 5: Examples of extracted gadgets.

A. The Language Design

Following the empirical findings that default transformations and existing code snippets are the major constructs of a non-trivial object transformer, we made the following choices in the DSL design:

1) *Providing a mechanism for code reuse.* Particularly, we provide a DSL construct named *gadget*, as denoted by $g(\vec{\square})$, a textural expression extracted from source code with all variable references being replaced by a placeholder. We use angled brackets to enclose a gadget, *e.g.*, $g(\square_1, \square_2, \square_3) = \langle \square_1\texttt{.foo(}\square_2\texttt{,}\square_3\texttt{)} \rangle$. Applying a gadget to an object transformer would reuse the entire expression with the flexibility for placeholders to be filled with transformer-specific values.
2) *Providing no arithmetic, logical, or bitwise operator.* We argue that when such operators (`+`, `&&`, `|`, ...) should appear in an object transformer, they will also likely to exist in old/new version source code and can be extracted as gadgets. Therefore, we do not have to include them in the DSL, yielding a minimal, concise DSL.
3) *Providing limited expressiveness for branch/loop conditions.* Branch/loop conditions in object transformers are also likely in the existing source code. Therefore, negations, nested branches, and while-loops provide sufficient expressiveness for constructing object transformers.

Figure 4 lists the syntax of our DSL. An object transformer t is a sequence of statements s^*, in which each field of the new-version object `obj` is assigned with a value. For each statement s, it can define a new variable v_i by applying a gadget g[5] and filling its placeholders with existing variables (a previously defined v_i or `stale`), assign a field to be transformed (`obj.`f) with a value, or use `if` branches or `while` loops with a condition c.

Readers may notice that the statements in a transformer t describe a *skeleton*, which specifies the targeted transformer's

[5]We allow a `void`-typed expression to be assigned to a variable, *i.e.*, g can be a `void` method invocation.

control flow (branches and loops) and data flow (variables and their dependencies). All concrete transformation operations are performed by gadgets extracted from the source code. Such a separation of concerns not only maximally reuses existing source code, but also gives considerable flexibility to implement diverse transformers. This design also facilitates our later heuristic synthesis algorithm to prioritize likely relevant object transformers by measuring both the structural complexity of a synthesized transformer and its "naturalness" in gadget use.

One final note is that we restrict branch/loop conditions to be either of e, `!`e, e `== null`, or e `!= null`, where expression e is either a variable or a gadget application. Theoretically, any condition can be expressed by negation ($\neg$) and nested branch ($\wedge$), but both our DSL and synthesis algorithm favor simple branch/loop conditions that reuse existing source code snippets.

B. Gadget Extraction

In gadget extraction, trade-offs must be made to balance the DSL's expressiveness and its synthesis difficulty. In an impractical extreme, one can include all Java language constructs as gadgets. This allows our DSL to be essentially equivalent to the vanilla Java. However, synthesizing Java programs directly for object transformation is considerably difficult and not practical.

The key trade-off we made is to only extract *complete source-code statements* as gadgets. We argue that no matter how many times method invocations, arithmetic/logical operations, *etc.* are performed in a statement, they should either be *all used* or *entirely not used* in an object transformer. The intuition behind this treatment is simple: each statement should contain a logically inseparable action in a well-maintained project for best readability and maintainability[6].

Gadgets are extracted by iterating over all statements in all application classes in both the old and new version source code using the following rules:

1) For a statement's associated expression (*i.e.*, the right-hand side of an assignment or an `if`/`while` condition), we parse it into an abstract syntax tree (AST) and replace each variable or constant node with a placeholder $\square_i$ to be a gadget. This rule yields g_1, g_2, and g_5 in Figure 5.
2) For each statement, we consider its contained *constants* potentially useful in synthesizing object transformers. Therefore, each constant literal in the statement is also extracted as a gadget. This rule yields g_3 in Figure 5.
3) For each statement, we also extract it into a gadget where only variable nodes are replaced by placeholders, *i.e.*, keeping all constants as-is in the gadget compared with the first rule. This is because constants may be inseparable from the statement's computational logic. This rule yields g_4 in Figure 5.
4) For each class in both old and new versions of the source code, we extract class field gadgets $\langle \square$`.fieldName`$\rangle$, class method gadgets $\langle \square$`.methodName(`$\vec{\square}$`)`$\rangle$, static field gadgets $\langle$`ClassName.fieldName`$\rangle$, static method gadgets $\langle$`ClassName.methodName(`$\vec{\square}$`)`$\rangle$, and object creation gadgets $\langle$`new ClassName(`$\vec{\square}$`)`$\rangle$. These rules are also additionally applied for the Java Standard Library for extracting potentially useful API calls, *e.g.*, container operations.

[6]There can be occasions that a statement consists of multiple actions, *e.g.*, a chain of method invocations. We optimistically believe that the desired action will independently appear elsewhere in the codebase.

$$g_1(\square_1) = \langle \square_1\texttt{.socket}\rangle$$
$$g_2(\square_1, \square_2) = \langle \square_1\texttt{.getIOChannel().keyFor(}\square_2\texttt{.getPoller().getSelector())}\rangle$$
$$g_3(\square_1) = \langle\texttt{(KeyAttachment)}\ \square_1\texttt{.attachment()}\rangle$$

```
1 class DSUHelper {
2   static void transform(SocketProcessor* obj, SocketProcessor stale) {
3     v1 = (NioChannel) g1(stale);
4     if (v1 != null) {
5       v2 = (SelectionKey) g2(v1, v1);
6       if (v2 != null) {
7         v3 = (KeyAttachment) g3(v2);
8       }
9     }
10    obj.ka = v3;
11  }
12 }
```

Fig. 6: A field transformer for field `ka` in the motivating example written in our DSL, and extracted gadgets. All used variables were initialized with default values, *e.g.*, `v3 = null`.

Algorithm 1: The transformer synthesis framework

```
1 Function SYNTHESIS(G)
2     Q ← {|▷};
3     while Q ≠ ∅ do
4         p ← arg min_{p'∈Q} cost(p');
5         if |▷ ∉ p then
6             yield p;
7         Q ← Q ∪ D_G(p) \ {p}
```

Figure 6 gives a transformer example for field `ka` in our motivating example (Figure 1). Three used gadgets g_1, g_2, and g_3 are extracted using rules #4, #1, and #1, respectively. This transformer is equivalent to the manually provided one in Figure 1.

One could expect that our DSL and gadget extraction rules suffice for object transformer synthesis. Unfortunately, there can be millions of gadgets extracted from a large codebase. Certainly, not all gadget combinations are equally relevant to a given upgrade. The *relevance* of a gadget to the class change and the *similarity* between a gadget combination and existing source code would serve as the guidance for efficient object transformer synthesis, which is described as follows.

V. Automatic Synthesis of Object Transformers

Conceptually, object transformer synthesis is simple: a systematic enumeration of all syntactically correct programs will eventually find a correct transformer (Section V-A). This section presents our heuristic search algorithm for efficiently pri-

oritizing correct and developer-readable (simple) transformers to make the search procedure practical (Sections V-B to V-D).

A. Synthesis Framework

Given a set of gadgets G, our object transformer synthesis algorithm listed in Algorithm 1 is essentially a straightforward syntax-directed search. It maintains a work list Q consisting of candidate synthesized programs. A program $p \in Q$ is an object transformation DSL program (syntax defined in Figure 4) with zero or more insertion marks $|\triangleright$ in which more statements can be filled[7]. Starting from the initial program that consists of a single insertion mark $|\triangleright$ in Q (Line 2), the algorithm iteratively pops the program p of the minimum cost in Q for a step of expansion (Lines 3–4). If p contains no insertion mark, we find a potentially useful transformer for further validation (Lines 5–6). Otherwise, there must be an insertion mark $|\triangleright$ in p and the algorithm expands the first insertion mark to obtain more candidate programs (Line 7).

The expansion step defines $D_G(p)$, the descendant programs of p over a set of gadgets G. Let σ_p be the first occurrence of insertion mark in p. A step of expansion either closes (*i.e.*, removes) the insertion mark σ_p to obtain

$$p_\epsilon = p[\sigma_p \mapsto \epsilon],$$

or prepends it with a statement `stmt` to obtain

$$p_{\texttt{stmt}} = p[\sigma_p \mapsto \texttt{stmt}\ |\triangleright].$$

In the latter case, let $V(\sigma_p) = \{\texttt{stale}, v_1, v_2, \ldots\}$ be all variables within the lexical scope of the insertion mark σ_p in p. For a gadget $g \in G$ of n placeholders, the set of its all possible applications at the insertion mark is defined by

$$E_g(\sigma_p) = \{g(v_1, v_2, \ldots, v_n) \mid v_i \in V(\sigma_p) \text{ for } 1 \le i \le n\}.$$

All syntactically valid expressions at σ_p are thus

$$E_G(\sigma_p) = V(\sigma_p) \cup \left(\bigcup_{g \in G} E_g(\sigma_p) \right).$$

To prepend a statement `stmt` at σ_p, it must be either of the following four patterns according to the syntax in Figure 4:

1) (variable assignment) v `=` e`;`
2) (field transformation) `obj.`f `=` e`;`
3) (if-branch) `if (`c`) { `s^*` } else { `s^*` }`
4) (while-loop) `while (`c`) { `s^*` }`

Also recall that we limit the form of a condition c to be either of $\{e,\ \texttt{!}e,\ e\ \texttt{==}\ \texttt{null},\ e\ \texttt{!=}\ \texttt{null}\}$. Therefore, `stmt` must consist of exactly one expression. Enumerating the expressions $e \in E_G(\sigma_p)$ and filling e into the above code patterns yields S, the set of all possible statements to prepend to σ_p. For an assignment v `=` e, the left-hand side variable v is also enumerated (over $v \in V(\sigma_p) \setminus \{\texttt{stale}\}$). Then, the descendants of p can be defined as

$$D_G(p) = \{p_\epsilon\} \cup \{p_{\texttt{stmt}} \mid \texttt{stmt} \in S\}.$$

Though conceptually simple, it is a challenge to scale the search. The key treatments we made to boost the search include decomposing an object transformer into independent field transformers (Section V-B), pruning likely irrelevant gadgets (Section V-C), and boosting the search by measuring the "naturalness" of p (Section V-D).

B. Object Transformer as Independent Field Transformers

An object transformer is responsible for transforming *all* fields in an object, and an object transformer for large classes may be lengthy and difficult to synthesize. Fortunately, the program is frozen at update-time and each field's value should be computed by a pure function over the update-time heap snapshot.

Therefore, the transformation for fields in an object can be *independently conducted* and the object transformation problem is essentially equivalent to the *field transformation* ones. In practice, a developer or upgrade maintainer simply specifies which fields may require a non-default transformer, and the synthesis algorithm will produce a series of candidate field transformers for further validation (*e.g.*, independently tested). This is a standard treatment of existing techniques [15, 27, 28].

C. Pruning Irrelevant Gadgets

There can be millions of gadgets for a large program, resulting in a huge $|E_G(\sigma_p)|$. Fortunately, we observed that most of the gadgets are irrelevant to the field to be transformed in terms of the upgrade. Particularly, class B is *relevant* to A if either: B is a (sub)class of A, B contains a field of type A, or a method in B refers to A (*e.g.* in the parameter list or a local variable, *etc.*). To synthesize a field transformer for field f in class A (*e.g.*, $A =$ `SocketProcessor` and $f =$ `ka` in Figure 1), we restrict the concerned gadget set G to be:

1) all gadgets in A,
2) all gadgets in any class relevant to A, and
3) all gadgets in any class relevant to f`.class`.

Finally, almost all classes are relevant to primitive types (*e.g.*, `int`, `bool`, *etc.*) and `java.lang.String`. We do not apply the third rule in synthesizing field transformer for these types, as otherwise, the search would be intractable.

D. Boosting the Search

A naive implementation of Algorithm 1 will be overwhelmed by the huge search space. For example, $|E_g(\sigma_p)|$ grows exponentially over increased $|V|$ and $|\vec{\square}|$. In our search implementation, we only fill a placeholder $\square_i$ with a type-compatible variable in the scope that will not trivially throw an exception (*e.g.*, invoking a `null` reference's method). This treatment yields a manageable $|E_g(\sigma_p)|$ in practice.

The key to the success of our search algorithm is a cost function *cost* to prioritize: (1) *simple* programs with few basic

[7]Given a program and an insertion mark in it, one can prepend a statement before the insertion mark to obtain a new program. $|\triangleright$ is equivalent to s^* in the syntax.

constructs, and (2) *natural* programs that maximally reuse existing method-local data flows in the source code.

To measure the naturalness, we perform an *intra-procedural* forward slicing [35] for each extracted gadget g to obtain

$$slice(g) = [g_1, g_2, \ldots],$$

the sequence of gadgets (appearing in their statement order) in which each gadget $g' \in slice(g)$ data-depends on g.

Given a program p and its used gadgets $G_p = [g_1, g_2, \ldots, g_n]$, we argue that the programs that better reuse consecutive gadgets in a slice (*i.e.*, existing data flow) are more likely to be natural. Formally, for each gadget $g_i \in G_p$, we find its *maximal* containing slice $S_p(g_i)$, the $slice(g_j)$ of maximum length satisfying that for $g_j \in G_p$ and $slice(g_i) \subseteq slice(g_j)$. The set of all maximal containing slices is thus $\mathcal{S}_p = \bigcup_{1 \le i \le n} S_p(g_i)$.

We measure the naturalness of p by calculating the average data-flow similarity for all maximal containing slices:

$$\delta(p, G) = \frac{1}{|\mathcal{S}_p|} \sum_{S \in \mathcal{S}_p} \max_{1 \le i \le j \le |S|} \frac{\mathrm{LCS}(G_p \upharpoonright S, S_{i:j})}{\max\{|G_p \upharpoonright S|, |S_{i:j}|\}},$$

in which LCS is the longest common subsequence, and $G_p \upharpoonright S$ is the subsequence of G_p obtained by removing any element not in S. The intuition behind this formula is that a program p of a high δ should look like a "blending" of gadgets in short slice sequences.

Finally, we add the measurement of simplicity to the cost function $cost$. Recall that our synthesis algorithm (Algorithm 1) either closes an insertion mark, or expands it by prepending a statement in an iteration:

$$p_\epsilon = p[\sigma_p \mapsto \epsilon] \text{ or } p_{\texttt{stmt}} = p[\sigma_p \mapsto \texttt{stmt } \ |\triangleright].$$

Thus we define the cost function to be $cost(p_\epsilon) = cost(p)$ and

$$cost(p_{\texttt{stmt}}) = \left(\sum_{p' \in H} |p'| \cdot rank(p') \right) \cdot \frac{1}{\delta(p_{\texttt{stmt}}) + \lambda_{\texttt{stmt}}},$$

where H is all partial programs in the process of generating p (by applying the two rewriting rules of the insertion mark), $|p'|$ denotes the number of statements (assignment, branch, or loop) in p', and $rank(p')$ is the rank of p' in terms of naturalness (δ) within all generated siblings in the expansion step in obtaining p'. Using $rank$ in the cost function reflects the intuition that one should not only favor short (and thus simple) synthesized programs, but also favor those programs whose generation processes are mostly natural.

The cost function also contains a mechanism (λ in the formula) for giving likely more relevant statements with extra credits. Our implementation adopts a simple rule that lets $\lambda_{\texttt{stmt}} = 0.2$ if `stmt` involves any changed field (*e.g.*, Lines 6–8 in Figure 1) or the corresponding gadget is from the changed code (*e.g.*, Line 11 in Figure 1). Future use of this mechanism can be assigning human-provided gadgets with a higher priority.

VI. Evaluation

We implemented the synthesis algorithm as the PASTA tool, which consists of ∼15,000 lines of Java code[8]. The source code parsing and extracting gadgets was implemented over JavaParser [36] and Javassist [37]. In this section, we experimentally compare PASTA with two state-of-the-art techniques using real-world updates from Apache Tomcat 8 and Apache FtpServer. We elaborate on the experimental setup in Section VI-A and present the evaluation results in Section VI-B.

A. Experimental Setup

The three transformer generation techniques under evaluation are: our PASTA, program-synthesis-based TOS [27], and trace-synthesis-based AOTES [28]. We evaluate these techniques against a set of class changes that require non-trivial object transformation from widely-used server applications undergoing long-term development and maintenance. As discussed in Section V-B, each field in an object can be independently transformed. Therefore, the evaluation subjects (first two columns in Table I) consist of:

1) The 22 Apache Tomcat 8 field updates studied in the empirical study (Section III);
2) Another 4 Apache FtpServer field updates selected following the same collection methodology of our empirical study.

For Apache FtpServer, we uniform-randomly sampled 30 commits (consisting of 75 class changes) from all 244 commits with class changes in the entire maintenance history of Apache FtpServer (from 1.0.0 to the latest 1.0.6). In the 75 class changes, 71 can be dynamically updated via a trivial object transformer. The rest four non-trivial cases (4 field updates) were all used as our experimental subjects.

To validate a transformer, we for each changed class provide sufficiently strong DSU test cases that can cover all locations writing to the changed field (by revising existing test cases or manually providing one whenever necessary). For each test case, we also specify an active-safe [10, 15, 17, 31] dynamic update-safe point, at which all changed methods in the changed class are popped off stack. Each test case also checks the object state consistency after the update point. These checkers take the heap snapshots before and after object transformation as inputs and determine whether the object transformation is successful. We use the same test cases for all the evaluated techniques.

The evaluation settings are:

1) For PASTA, the search depth is set to 10 (sufficiently large to handle all studied transformers). We automatically validated each synthesized transformer by running the DSU test cases and manually checked the test-passing ones until a semantically correct field transformer is produced.
2) For TOS, we followed the evaluation steps in its paper [27]. TOS requires heap snapshots for field transformer synthesis. Thus we used our test cases for field transformation correctness validation to produce these snapshots. The

[8]Available at https://zelinzhao.github.io/pasta.

TABLE I: Evaluation results on the real-world updates in Apache Tomcat 8 and Apache FtpServer.

ID	Subject `ChangedClass.changedField (FieldType)`	#Gadgets	#Tests	PASTA	AOTES	TOS
1	Tomcat-6a940d `StandardContext.path (String)`	2,373	2	**1** / 336 (20.0^m) ✓	I×	$M{\to}S$×
2	Tomcat-ec8dff `ContextConfig.context (Context)`	28,232	3	×	$I{\to}H$×	$M{\to}S$×
3	Tomcat-f84800 `WsHttpUpgradeHandler.wsSession (WsSession)`	3,098	2	×	$I{\to}H$×	$M{\to}S$×
4	Tomcat-a752f3 `AsyncContextImpl.request (Request)`	14,927	2	×	$I{\to}H$×	$M{\to}S$×
5	Tomcat-c0d4f7 `WsHandshakeResponse.headers (CaseInsensitiveKeyMap)`	633	3	**1** / 316 (7.6^m) ✓	$I{\to}H$×	$M{\to}S$×
6	Tomcat-c0d4f7 `PojoMethodMapping.onMessage (List)`	38,475	2	**1** / 14 (0.3^m) ✓	$I{\to}H$×	$M{\to}S$×
7	Tomcat-dbb784 `SenderState.memberStates (ConcurrentHashMap)`	37,234	3	**1** / 7 (0.2^m) ✓	$I{\to}H$×	$M{\to}S$×
8	Tomcat-a8d16b `MemberImpl.msgCount (AtomicInteger)`	6,962	2	**1** / 13 (0.2^m) ✓	✓	$M{\to}S$×
9	Tomcat-358f94 `StandardContext.parameters (ConcurrentHashMap)`	19,805	2	**1** / 45 (1.8^m) ✓	$I{\to}H$×	$M{\to}S$×
10	Tomcat-ad012e `FutureToSendHandler.result (AtomicReference)`	1,468	2	**1** / 4 (0.1^m) ✓	$I{\to}H$×	$M{\to}S$×
11	Tomcat-c0d4f7 `WsOutputStream.used (boolean)`	76	2	⊘	$I{\to}H$⊘	$M{\to}S$⊘
12	Tomcat-c0d4f7 `WsWriter.used (boolean)`	77	2	⊘	$I{\to}H$⊘	$M{\to}S$⊘
13	Tomcat-db1a6e `StandardContext.useRelativeRedirects (boolean)`	2,413	2	⊘	$I{\to}H$⊘	$M{\to}S$⊘
14	Tomcat-69196d `StoreConfigLifecycleListener.oname (ObjectName)`	12,783	2	**1** / 1 (0.1^m) ✓	$I{\to}H$×	$M{\to}S$×
15	Tomcat-2e7c68 `Nio2Endpoint.threadGroup (AsynchronousChannelGroup)`	3,116	2	**1** / 144 (14.0^m) ✓	I×	$M{\to}S$×
16	Tomcat-d8ad3c `DefaultServlet.showServerInfo (boolean)`	1,000	2	**1** / 3 (0.1^m) ✓	$I{\to}H$×	$M{\to}S$×
17	Tomcat-5952de `WebappServiceLoader.context (Context)`	32,325	2	×	$I{\to}H$×	$M{\to}S$×
18	Tomcat-6b64bb `StandardContext.noPluggabilityListeners (Set)`	40,592	3	**1** / 77 (8.7^m) ✓	I×	$M{\to}S$×
19	Tomcat-6b64bb `StandardContext.noPlug...Context (NoPlug...Context)`	12,865	2	×	$I{\to}H$×	$M{\to}S$×
20	Tomcat-766c9e `AprSocketWrapper.endpoint (AprEndpoint)`	124,569	2	×	$I{\to}H$×	$M{\to}S$×
21	Tomcat-f4451c `SocketProcessor.ka (KeyAttachment)`	2,430	3	**1** / 4 (0.3^m) ✓	$I{\to}H$×	$M{\to}S$×
22	Tomcat-4355ed `StandardContext.applicationEventListenersList (List)`	50,274	3	**1** / 27 (1.4^m) ✓	$I{\to}H{\to}R$×	$M{\to}S$×
23	FtpServer-faa153 `MinaListener.acceptor (SocketAcceptor)`	469	2	**1** / 14 (0.3^m) ✓	$I{\to}H$×	$M{\to}S$×
24	FtpServer-32ed0b `FtpServer.serverContext (FtpServerContext)`	1,789	2	**1** / 7 (0.2^m) ✓	$I{\to}H$×	$M{\to}S$×
25	FtpServer-1b2ea6 `PropertiesUserManager.isConfigured (boolean)`	249	2	×	$I{\to}H$×	$M{\to}S$×
26	FtpServer-afffc8 `FileIpRestrictor.file (File)`	1,634	2	**1** / 154 (2.5^m) ✓	✓	$M{\to}S$×
	Summary	**439,868**	**58**	**16** (61.5%)	**2** (7.7%)	**0** (0.0%)

For PASTA, "m / n" denotes that (1) the first semantically correct transformer is in the m-th place among all test-passing transformers; (2) there are n test-runs before the first semantically correct transformer is generated. The number in the bracket (t^m) indicates that the first correct transformer is produced after t minutes. For AOTES, "I×", "$I{\to}H$×", and "$I{\to}H{\to}R$×" denote failing at mutator generation, execution history synthesis, and trace replay, respectively. For TOS, "M×" and "$M{\to}S$×" denote failing at object matching and transformer synthesis, respectively. ⊘ denotes that this update requires a human-provided configuration (thus is out of the scope of automatic transformer synthesis). We included these cases in the evaluation for completeness: they are indeed non-trivial and may potentially be addressed in future work (*e.g.*, via providing a gadget by the upgrade maintainer).

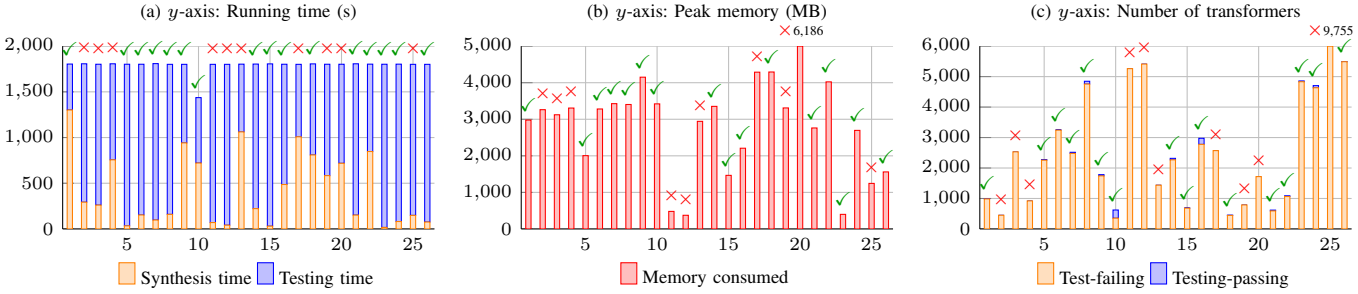

Fig. 7: Statistics of running time, peak memory, and statistics of testing results. The x-axis denotes subject IDs in Table I.

test cases provide sufficiently informative traces for a human DSU expert to derive correct field transformers. We also manually checked each synthesized transformer for semantic correctness.

3) For AOTES, as it does not produce any human-readable transformer, we consider it correct if all test cases passed even if it might fail in other online transformations[9]. We also inspected the synthesized method histories for manually diagnosing the root cause in case of a failure.

For each field transformation, we set a moderate 30-minute time limit. TOS and AOTES did not time out for all experimental subjects. All experiments were conducted on a commodity PC with a quad-core Intel Core i7-4770 CPU and 32 GB RAM running Ubuntu Linux 18.04.

[9]Conversely, AOTES may have a chance to correctly transform the heap in practice even if it fails on some test cases. We argue that using AOTES in this case is considerably risky because AOTES is a black-box technique that silently transforms the heap.

B. Evaluation Results

Overall Results. The major evaluation results are shown in Table I. PASTA produced 16/26 (61.5%) correct field transformers, or 15/25 (60.0%) correct object transformers (because Subjects #18 and #19 are from the same class). All succeeded cases are amazingly top-1 hits, *i.e.*, the most "natural" test-passing transformer is semantically correct. This result supports our previous claim that the test cases are sufficiently strong. Compared with the best existing techniques TOS (failed

for all cases) and AOTES (2/26, 7.7%), PASTA made automatic non-trivial object transformation in DSU significantly more practical.

Like PASTA, TOS is also a syntax-guided synthesis. However, the TOS DSL contains only a subset of basic Java language constructs. Consequently, expressing a practical non-trivial object transformer would require an unrealistically large search depth. For example, the most complex transformer produced by PASTA (#23) consists of a field access, an `if` condition, two different string literals, two constructor calls, and four instance method calls. There is also a transformer (#18) with a branch in a loop. It is of no surprise that TOS failed on these practical subjects even with the aid of test traces.

AOTES succeeded only for Subjects #8 and #26 because their inverse histories happened to be relatively simple (AOTES's inherent requirement on successful history synthesis). The major failure cause for AOTES is generating an incorrect history (20/26, 76.9%), which is the major drawback hard to avoid for a runtime state transformer.

Unlike TOS and AOTES in which a correct transformer may be too complex to correctly synthesize, PASTA "shortcuts" the solution by gluing gadgets using a simpler DSL focused on code reuse. For example, AOTES generated a long method call history for Subject #8, whereas PASTA found a constructor call to fulfill the same functionality. The evaluation results suggest that the code-reuse in PASTA could become an effective approach to object transformer synthesis.

Detailed Analysis. Based on the type of code change, a field change can be either of:

A *field type change* with field name unchanged (7 cases; #5–10 and #26), which was handled best by PASTA. All 7 cases were successfully transformed. A type-changed field usually plays a similar role in both old and new versions. They can likely be used interchangeably somewhere in the source code. Thus, there may exist code snippets to retrieve the information contained in this field, which can be used in a transformer.

A *field value change* with field type and name unchanged (6 cases; #1–4 and 23–24). PASTA succeeded in 3/6 (50%) of them. Value change indicates that the semantics of an object is changed in the update. Since our approach is semantics-unaware, PASTA has to perform a brute-force search across all potentially useful transformers.

A *new field* (13 cases; #11–22 and #25). This includes renaming a field with a type change, which cannot be objectively distinguished from adding a new field (*e.g.*, `ka` in Figure 1). Excluding the three out-of-scope cases (explained in the footnote of Table I), PASTA succeeded in 6/10 (60%) of them. For a similar reason of the "field value change" category, PASTA is essentially an exhaustive enumeration for this category of field change.

For all failing cases excluding the out-of-scope ones, we found that our DSL and PASTA's extracted gadgets suffice to construct a correct transformer. However, PASTA was not able to do sufficiently many explorations to identify them within the given time limit: the failing cases on average tested only 2,677 candidate transformers. It is considerably challenging to assemble gadgets scattered in different parts of a program (*i.e.*, with a relatively low naturalness score), which is required in synthesizing field transformers in these cases. Nevertheless, PASTA as a prototype implementation points out a promising research direction that reuses existing code in object transformation.

One may also wonder whether our treatment for pruning irrelevant gadgets for primitive types and `String` (Section V-C) should be considered proper. Excluding the three out-of-scope cases (#11–13), PASTA successfully synthesized 2/3 (#1,16,25) of the cases. For the only failing case (#25), the semantically correct transformer sets the field to be `true` only when two conditions are simultaneously satisfied. These two conditions were correctly identified as gadgets by PASTA, however, it failed to find them within the time limit. Therefore, our aggressive policy for pruning irrelevant gadgets should be considered proper for primitive types.

Considering the resource consumption for conducting object transformation, a 30-minute time limit should be considered reasonable for production use[10]. Figure 7 (a) and (b) display the statistics of running time and memory consumption, respectively. Among all succeeded cases, 9/16 (56.2%) returned the first semantically correct transformer within one minute, and 14/16 (87.5%) were within 10 minutes. Both the search algorithm and test validation can also be parallelized to further accelerate the implementation. However, these engineering issues are not the major focus of this paper. In terms of memory, PASTA used less than 4 GB memory for 22/26 (84.6%) of the subjects. Subject #20 consumed the most memory (∼6 GB) on time out due to its large number of gadgets (124,569). The overall results can be considered acceptable for production use.

Finally, to our surprise, our search algorithm even found a simpler field transformer for field `ka` of the motivating example in Figure 1 (Subject #21):

```
1 class DSUHelper {
2   static void transform(SocketProcessor* obj, SocketProcessor stale) {
3     obj.ka = null;
4     if (stale.socket != null)
5       obj.ka = stale.socket.getAttachment(false);
6   }
7 }
```

The gadget $\square_1$`.getAttachment(`$\square_2$`)` in Line 5 was originally used for creating an `NioChannel` object's corresponding `KeyAttachment`, which was passed to by a `true` argument (thus was considered irrelevant in our manual transformer construction). However, its behavior of returning an existing `KeyAttachment` object is perfectly correct for our expected object transformation. The transformer in Figure 1 was ranked in the 5th position, which also passed the tests within the time limit.

Implications. Both the empirical study and evaluation results show that the unavailability of *non-trivial object transformers* should be recognized as a major obstacle that hinders the application of DSU in practice. Considering updating Tomcat 8

[10]Gadget extraction time is less than 5 minute for both Tomcat and FtpServer. We did not count such pre-processing time.

and FtpServer with state-of-the-art techniques before PASTA, roughly 18% updates (commits) still require a restart. However, if all non-trivial cases can be provided with a proper transformer, the restart rate would decrease to ∼2%.

This paper opens a promising research direction towards automated object transformation in DSU on code reuse. Considering the distribution of all updates (commits) in our evaluation, PASTA can reduce the restart rate by 60.5%, compared with the best state-of-the-art technique. Considering the rapid advances in the program synthesis and repair community [38–41], we are optimistic that most and more non-trivial transformers can be automatically synthesized in the near future.

VII. Threats to Validity

A major threat concerns the generalization of our empirical study results because Tomcat 8 is the only investigated subject. Since Tomcat is a mature, actively developed, and widely used subject extensively studied by existing literature [17, 28, 42–45], we should consider that the empirical study results reflect real-world software evolution to a large extent.

There is a minor chance that we erroneously marked a non-trivial object transformer as trivial in the empirical study because reasoning about a program's runtime state is labor-intensive. (This is also the major reason that we did not include more evaluation subjects.) Taking such potential error into consideration, the applicability of DSU in practice may potentially be more challenging. On the other hand, it also suggests that object transformation should draw more serious research efforts.

Another threat to our evaluation results' validity is that the experimental subjects partially overlap the ones used in the empirical study, and the high success rate may be due to overfitting. We argue that this is not likely the case because: (1) the design of PASTA follows the general principles of software systems, and (2) the experiments on Apache FtpServer also show significant improvements over existing techniques. Therefore, PASTA should be recognized as useful in conducting DSU for similarly long-running server applications, which are actually the major focus of dynamic software updating.

VIII. Related Work

Dynamic Software Updates. To dynamically update a running system, one must determine:

What to update, *i.e.*, specifying the parts of the system to be dynamically upgraded, *e.g.*, via a source code patch. This paper's focus is the DSU of long-running Java server programs over maintenance upgrades.

When to update, *i.e.*, monitoring the system execution until an update-safe point is reached [10, 31]. Existing strategies include activeness safety [15, 17, 46, 47], con-freeness safety [47], and transactions version consistency [48]. This paper assumes the most popular activeness safety criterion.

How to update, *i.e.*, applying the dynamic patch [27, 28]. Conducting the update includes replacing the changed code [32, 33] and updating the stale objects [15, 46]. *Inter-process state transformation* (multi-versioning) maintains multiple execution flows of the same program during the update, and DSU is implemented by process replacement [49–51]. A more lightweight approach is the *intra-process* strategy, where dynamic software update happens in-place [15, 17, 18].

Object Transformation in DSU. Both inter- and intra-process DSUs require object transformation. Object transformation can be eager [15] (all stale objects are transformed at update time) or lazy [17] (object is transformed upon access). PASTA works for both cases.

Our empirical study reveals that a default transformer (either copies the old-version value for an unchanged field, or assigns a default value to a newly-added field) suffices for most dynamic updates. Therefore, it is not a surprise that default transformation is widely adopted in existing DSU systems [15, 17, 18, 22, 46, 52]. To perform non-trivial object transformations, existing DSU work [15, 17, 22, 46] suggests that object transformers should be shipped along with the patch to enable DSU in practice.

TOS [27] took the first step in the automatic synthesis of object transformers. Given paired old/new version objects, transformer synthesis can be regarded as a programming by example (PBE) problem, in which syntax-guided search is usually adopted. However, TOS adopts a Java-alike DSL for specifying transformers, yielding a huge search space and subsequent search failures. In contrast, we kept DSL constructs to a minimal extent and let extracted gadgets to perform the actual object state computation.

AOTES [28] took a fundamentally different approach to object transformation. Instead of synthesizing a transformer, AOTES for each object synthesizes a method invocation history that brings the object to its current state. The synthesized history is then replayed (executed) on the new version code to obtain the transformed object. AOTES has the potential to scale out (though its current implementation frequently fails on large systems like Tomcat). However, it is extremely difficult for a developer to validate the correctness of a transformation, leaving it risky to use in practice.

Finally, some DSU systems [53, 54] entirely eliminate object transformations by restricting the update timing, *e.g.*, updates can only be applied when there is no unsafe event [53] or object state [54]. Existing object transformation techniques (PASTA, TOS, and AOTES) are orthogonal to these systems. Generally speaking, better transformers allow more update timings, and fewer update timings tolerate simpler transformers.

Program Synthesis. Object transformer is a piece of program. Thus, PASTA belongs to the family of program synthesis [55–58]. TOS is inspired by the PBE approaches [59, 60], which focus on spreadsheet data transformations. Foofah [61, 62] adopted PBE to synthesize data transformation programs for data analysis tasks. PASTA aims at generating non-trivial object transformers, which is a more challenging task.

The design of placeholders in PASTA originates from *program sketching* [63–65] in which placeholders are filled with synthesized code snippets. However, program synthesis for PASTA is quite different in nature compared with program

sketching: the latter is given a program sketch and placeholders are filled with limited language constructs (*e.g.*, expressions). PASTA faces a more open-ended world consisting of a large number of gadgets without a sketch.

The design of code reuse in PASTA resembles *component-based program synthesis* [66–69] in which components (usually APIs) are assembled to perform designated tasks. Gadgets from existing code can be regarded as the "components" in PASTA. However, the search requires careful calibration for efficiency.

There exist also other techniques for improving the effectiveness of program synthesis (*e.g.*, interactive synthesis [70–74]). They are generally orthogonal to this paper and are out of this paper's scope.

IX. Conclusion

This paper recognizes the existence of non-trivial object transformers as a major obstacle to the DSU for practical systems. The paper also reveals that these non-trivial transformers can essentially be constructed by reassembling gadgets extracted from existing source code. This paper presents the PASTA tool and the experimental results show that PASTA can handle 7.5× non-trivial object transformers compared with the best existing techniques, advancing the state-of-the-art effectiveness on automated transformer synthesis for practical DSU.

Acknowledgments

The authors would like to thank the anonymous reviewers for their comments. This work is supported in part by National Key R&D Program (Grant #2017YFB1001801) of China, National Natural Science Foundation (Grants #61932021, #61802165, #62025202) of China, and the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China. Yanyan Jiang (jyy@nju.edu.cn) and Chang Xu (changxu@nju.edu.cn) are the corresponding authors.

References

[1] M. Hicks and S. Nettles, "Dynamic software updating," *ACM Transactions on Programming Languages and Systems (TOPLAS'05)*, vol. 27, no. 6, pp. 1049–1096, 2005.

[2] D. Gupta, P. Jalote, and S. Member, "A formal framework for on-line software version change," *IEEE Transactions on Software Engineering (TSE'96)*, vol. 22, pp. 120–131, 1996.

[3] J. Stanek, S. Kothari, T. N. Nguyen, and C. Cruz-Neira, "Online software maintenance for mission-critical systems," in *IEEE International Conference on Software Maintenance (ICSM'06)*, 2006, pp. 93–103.

[4] Y. Vandewoude, Heverlee, P. Ebraert, Y. Berbers, and T. D'Hondt, "Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates," *IEEE Transactions on Software Engineering (TSE'07)*, vol. 33, no. 12, pp. 856–868, 2007.

[5] C. Hayden, E. Smith, E. Hardisty, M. Hicks, and J. Foster, "Evaluating dynamic software update safety using systematic testing," *IEEE Transactions on Software Engineering (TSE'12)*, vol. 38, no. 6, pp. 1340–1354, 2012.

[6] R. A. Bazzi, B. Topp, and I. Neamtiu, "How to have your cake and eat it too: Dynamic software updating with just-in-time overhead," in *International Workshop on Hot Topics in Software Upgrades (HotSWUp'12)*. IEEE, 2012, pp. 1–5.

[7] J. Shen and R. A. Bazzi, "A formal study of backward compatible dynamic software updates," in *International Conference on Software Engineering and Formal Methods (SEFM'15)*. Springer, 2015, pp. 231–248.

[8] L. G. G. de Pina, "Practical dynamic software updating," Ph.D. dissertation, Instituto Superior Técnico, 2016.

[9] "Ksplice," http://www.ksplice.com, 2008.

[10] J. Arnold and M. F. Kaashoek, "Ksplice: Automatic rebootless kernel updates," in *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys'09)*, 2009, pp. 187–198.

[11] "kpatch," https://github.com/dynup/kpatch, 2014.

[12] "kGraft," https://www.suse.com/products/live-patching, 2014.

[13] "The benefits of windows dynamic update," https://techcommunity.microsoft.com/t5/Windows-IT-Pro-Blog/The-benefits-of-Windows-10-Dynamic-Update/ba-p/467847, 2019.

[14] A. R. Gregersen and B. N. Jørgensen, "Dynamic update of Java applications - balancing change flexibility vs programming transparency," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 21, no. 2, pp. 81–112, 2009.

[15] S. Subramanian, M. Hicks, and K. S. McKinley, "Dynamic software updates: A VM-centric approach," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*, 2009, pp. 1–12.

[16] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification, Java SE 8 Edition*, 1st ed. Addison-Wesley Professional, 2014.

[17] T. Gu, C. Cao, C. Xu, X. Ma, L. Zhang, and J. Lu, "Javelus: A low disruptive approach to dynamic software updates," in *Proceedings of 19th the Asia-Pacific Software Engineering Conference (APSEC'12)*, 2012, pp. 527–536.

[18] L. Pina, L. Veiga, and M. Hicks, "Rubah: DSU for Java on a stock JVM," in *Proceedings of the 2014 International Conference on Object Oriented Programming Systems Languages Applications (OOPSLA'14)*, 2014, pp. 103–119.

[19] F. Ferrandina, T. Meyer, R. Zicari, and G. Ferran, "Schema and database evolution in the O2 object database system," in *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB'95)*, 1995, pp. 170–181.

[20] "Oracle database 10g online data reorganization & redefinition," Oracle Inc., Tech. Rep., 2005.

[21] K. Saur, T. Dumitraş, and M. Hicks, "Evolving NoSQL databases without downtime," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'16)*, 2016, pp. 166–176.

[22] C. Giuffrida, C. Iorgulescu, and A. S. Tanenbaum, "Mutable checkpoint-restart: automating live update for generic server programs," in *Proceedings of the 15th International Middleware Conference (Middleware'14)*. ACM, 2014, pp. 133–144.

[23] "Apache Tomcat," http://tomcat.apache.org, 1999.

[24] "Spring boot hot swapping," https://docs.spring.io/spring-boot/docs/current/reference/html/howto-hotswapping.html, 2012.

[25] L. Alkalai and A. T. Tai, "Long-life deep-space applications," *IEEE Annals of the History of Computing*, no. 4, pp. 37–38, 1998.

[26] "Java Platform Debugger Architecture: Java SE 1.4 Enhancements," https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/enhancements1.4.html, 2002.

[27] S. Magill, M. Hicks, S. Subramanian, and K. S. McKinley, "Automating object transformations for dynamic software updating," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'12)*, 2012, pp. 265–280.

[28] T. Gu, X. Ma, C. Xu, Y. Jiang, C. Cao, and J. Lu, "Automating object transformations for dynamic software updating via online execution synthesis," in *32nd European Conference on Object-Oriented Programming (ECOOP'18)*, 2018.

[29] W. Cazzola, A. Ghoneim, and G. Saake, "Software evolution through dynamic adaptation of its oo design," in *Objects, Agents, and Features*. Springer, 2004, pp. 67–80.

[30] X. Ma, L. Baresi, C. Ghezzi, V. Panzica La Manna, and J. Lu, "Version-consistent dynamic reconfiguration of component-based distributed systems," in *Proceedings of the ACM SIGSOFT Symposium and the European Conference on Foundations of Software Engineering (ESEC/FSE'11)*, 2011, pp. 245–255.

[31] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz, "OPUS: Online patches and updates for security," in *Proceedings of the 14th Conference on USENIX Security Symposium (USENIX Security'05)*, 2005, pp. 19–19.

[32] T. Würthinger, C. Wimmer, and L. Stadler, "Dynamic code evolution for Java," in *Proceedings of the International Conference on the Principles and Practice of Programming in Java (PPPJ'10)*, 2010, pp. 10–19.

[33] T. Wurthinger, C. Wimmer, and L. Stadler, "Unrestricted and safe dynamic code evolution for Java," *Science of Computer Programming (SCP'13)*, vol. 78, no. 5, pp. 481–498, 2013.

[34] R. C. Daley and J. B. Dennis, "Virtual memory, processes, and sharing in MULTICS," *ACM Symposium on Operating System Principles (SOSP'67)*, vol. 11, no. 5, pp. 306–312, 1967.

[35] G. A. Venkatesh, "The semantic approach to program slicing," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI'91)*, 1991, p. 107119.

[36] "Javaparser for processing java code," https://javaparser.org/, 2011.

[37] "Javassist: Java bytecode engineering toolkit since 1999," https://www.javassist.org/, 1999.

[38] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, 2012, pp. 3–13.

[39] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering (ICSE'16)*. ACM, 2016, pp. 691–701.

[40] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE'17)*. IEEE, 2017, pp. 416–426.

[41] R. Alur, R. Singh, D. Fisman, and A. Solar-Lezama, "Search-based program synthesis," *Communications of the ACM*, vol. 61, no. 12, pp. 84–93, 2018.

[42] L. Qi, H. Jin, I. Foster, and J. Gawor, "Hand: Highly available dynamic deployment infrastructure for globus toolkit 4," in *15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing (PDP'07)*. IEEE, 2007, pp. 155–162.

[43] H. Li, B. Huang, and J. Lu, "Dynamical evolution analysis of the object-oriented software systems," in *2008 IEEE Congress on Evolutionary Computation (CEC'08)*. IEEE, 2008, pp. 3030–3035.

[44] S. Cech Previtali and T. R. Gross, "Aspect-based dynamic software updating: a model and its empirical evaluation," in *Proceedings of the 10th international conference on Aspect-oriented software development (AOSD'11)*. ACM, 2011, pp. 105–116.

[45] T. Gu, Z. Zhao, X. Ma, C. Xu, C. Cao, and J. Lü, "Improving reliability of dynamic software updating using runtime recovery," in *Proceedings of the 23rd Asia-Pacific Software Engineering Conference (APSEC'16)*, 2016, pp. 257–264.

[46] D. Gupta, P. Jalote, and G. Barua, "A formal framework for on-line software version change," *IEEE Transactions on Software Engineering (TSE'96)*, vol. 22, no. 2, pp. 120–131, 1996.

[47] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu, "Mutatis mutandis: Safe and predictable dynamic software updating," *ACM Transactions on Programming Languages and Systems (TOPLAS'07)*, vol. 29, no. 4, 2007.

[48] I. Neamtiu, M. Hicks, J. S. Foster, and P. Pratikakis, "Contextual effects for version-consistent dynamic software updating and safe concurrent programming," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL'08)*, 2008, pp. 37–49.

[49] C. M. Hayden, E. K. Smith, M. Hicks, and J. S. Foster, "State transfer for clear and efficient runtime updates," in *Proceedings of the International Conference on Data Engineering Workshops (ICDE'11)*, 2011, pp. 179–184.

[50] C. Giuffrida, C. Iorgulescu, A. Kuijsten, and A. S. Tanenbaum, "Back to the future: Fault-tolerant live update with time-traveling state transfer," in *Proceedings of the Large Installation System Administration Conference (LISA'13)*, 2013, pp. 89–104.

[51] L. Pina, A. Andronidis, M. Hicks, and C. Cadar, "Mvedsua: Higher availability dynamic software updates via multi-version execution," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*. ACM, 2019, pp. 573–585.

[52] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for c," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*, 2006, pp. 72–83.

[53] I. Neamtiu and M. Hicks, "Safe and timely updates to multi-threaded programs," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*, 2009, pp. 13–24.

[54] W. Cazzola and M. Jalili, "Dodging unsafe update points in java dynamic software updating systems," in *International Symposium on Software Reliability Engineering (ISSRE'16)*, 2016, pp. 332–341.

[55] Z. Manna and R. J. Waldinger, "Toward automatic program synthesis," *Communications of the ACM*, vol. 14, no. 3, p. 151165, Mar. 1971.

[56] Z. Manna and R. Waldinger, "A deductive approach to program synthesis," *ACM Transactions on Programming Languages and Systems (TOPLAS'80)*, vol. 2, no. 1, pp. 90–121, 1980.

[57] S. Gulwani, "Dimensions in program synthesis," in *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming (PPDP'10)*. New York, NY, USA: Association for Computing Machinery, 2010, p. 1324.

[58] S. Gulwani, O. Polozov, R. Singh *et al.*, "Program synthesis," *Foundations and Trends in Programming Languages (FTPL'17)*, vol. 4, no. 1-2, pp. 1–119, 2017.

[59] S. 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 (POPL'11)*, 2011, pp. 317–330.

[60] W. R. Harris and S. Gulwani, "Spreadsheet table transformations from examples," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*, 2011, pp. 317–328.

[61] Z. Jin, M. R. Anderson, M. Cafarella, and H. Jagadish, "Foofah: Transforming data by example," in *Proceedings of the 2017 ACM International Conference on Management of Data (MOD'17)*, 2017, pp. 683–698.

[62] Z. Jin, M. R. Anderson, M. Cafarella, and H. V. Jagadish, "Foofah: A programming-by-example system for synthesizing data transformation programs," in *Proceedings of the 2017 ACM International Conference on Management of Data (MOD'17)*, 2017, pp. 1607–1610.

[63] A. Solar-Lezama, "Program sketching," *International Journal on Software Tools for Technology Transfer (STTT'13)*, vol. 15, no. 5-6, pp. 475–495, 2013.

[64] L. D'Antoni, R. Samanta, and R. Singh, "Qlose: Program repair with quantitative objectives," in *International Conference on Computer Aided Verification (CAV'16)*, 2016.

[65] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Towards practical program repair with on-demand candidate generation," *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE'18)*, pp. 12–23, 2018.

[66] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*, 2010, pp. 215–224.

[67] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps, "Component-based synthesis for complex apis," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17)*, 2017, pp. 599–612.

[68] Y. Feng, R. Martins, J. V. Geffen, I. Dillig, and S. Chaudhuri, "Component-based synthesis of table consolidation and transformation tasks from examples," *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17)*, 2017.

[69] K. Shi, J. Steinhardt, and P. Liang, "Frangel: component-based synthesis with control structures," in *Proceedings of the ACM on Programming Languages (POPL'19)*, vol. 3, 2019, pp. 1–29.

[70] T. Gvero, V. Kuncak, and R. Piskac, "Interactive synthesis of code snippets," in *International Conference on Computer Aided Verification (CAV'11)*, 2011.

[71] J. Galenson, P. Reames, R. Bodík, B. Hartmann, and K. Sen, "Codehint: dynamic and interactive synthesis of code snippets," *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*, 2014.

[72] T. Gvero and V. Kuncak, "Interactive synthesis using free-form queries," *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE'15)*, vol. 2, pp. 689–692, 2015.

[73] C. Wang, A. Cheung, and R. Bodík, "Interactive query synthesis from input-output examples," *Proceedings of the 2017 ACM International Conference on Management of Data (MOD'17)*, 2017.

[74] H. Peleg, S. Shoham, and E. Yahav, "Programming not only by example," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE'18)*. IEEE, 2018, pp. 1114–1124.