Automating Object Transformations for Dynamic Software Updating

Anonymous

Abstract

Dynamic software updating (DSU) systems eliminate costly downtime by dynamically fixing bugs and adding features to executing programs. Given a static *code* patch, most DSU systems can construct the run-time code changes automatically. However, a dynamic update must also specify how to change the running program's execution *state*, e.g., its stack and heap, to be compatible with the new code. Constructing such *state transformations* correctly and automatically remains an open problem. This paper presents a solution called *Targeted Object Synthesis* (TOS). TOS first executes the same tests on the old and new program versions separately, observing the program state at key points. Given two corresponding states, TOS *matches* corresponding objects between the two versions, and *synthesizes* the simplest-possible function to transform old version objects to their corresponding new versions. We show the efficacy of TOS by inferring transformation functions for actual updates to four open-source server programs.

1. Introduction

Suppose you are running an on-line service and a memory leak in your server software causes it to regularly run out of memory and crash. Eventually you discover the one-line fix: the Connection class's close method should unlink some metadata when a connection closes. To apply this fix in a standard deployment you stop your server and restart the patched version, but in doing so disrupt any active users. With *dynamic software updating* (DSU) support in an extended Virtual Machine such as LiveRebel [16] you can do better. You apply a *dynamic* patch to the Connection class of your *running* system to prevent further leaks without disrupting current users. In some DSU-enhanced VMs, such as Jvolve [14], you can do better still. You include *state transformation* code in your dynamic patch that traverses the heap and unlinks the useless metadata left reachable by the bug.

An important goal toward furthering the adoption of DSU systems is to make them easy to use, i.e., to minimize the effort required to produce a correct dynamic patch from two versions of a system. As a step in this direction, many DSU systems employ simple *syntactic*, *type-based* tool support for constructing a dynamic patch from the old and new program versions [1, 8, 13, 14]. For example, if class C's method m's bytecode has changed, Jvolve will include C.m in the dynamic patch. If C's field definitions change, in type or number, Jvolve creates a default *object transformation function* that it applies to all C objects when it applies the patch. This function retains the values of unchanged fields and initializes the rest with a default value, e.g., **null**.

While tool support for identifying changed code is highly effective, existing support for constructing state transformation code is rarely sufficient, and the programmer must therefore modify the generated code. For example, in Jvolve the programmer must add code that unlinks the leaked metadata in our example. Unfortunately, the cases that require manual intervention are often challenging to get right. For our example, the Connection transformer cannot simply unlink all connection metadata unconditionally. Instead, it must use appropriate context by examining the running program's heap and stack to identify and unlink only the metadata that is logically dead. Transformations that require moving objects between collections or partitioning single objects into several objects—examples we have observed in practice—require similar care in their construction. Thus, writing state transformation code for DSU systems is a programming task unique to DSU, and it can be a time-consuming, error-prone process.

This paper presents a general-purpose approach for synthesizing object transformers for dynamic patches, to ease the burden on programmers. Our *Targeted Object Synthesis* (TOS) approach was developed for the Java-based DSU system Jvolve, but our techniques should adapt readily to other DSU systems that support state transformation.

TOS works in two phases, *matching* and *synthesis*. The matching phase begins by running both versions of the program on the same inputs and taking heap snapshots at corresponding program points. Given a class C for which to synthesize a transformer function (i.e., because the class changed between versions), the goal is to find corresponding C objects in each pair of corresponding snapshots. TOS matching first attempts to match objects according to *key fields*, which are fields in an object that (1) uniquely identify the object in a given heap (i.e., each object of class C differs on the values of its key fields) and (2) there exist objects in both heaps with the same values for these fields. We use a greedy algorithm which, in our experience, very often succeeds in finding a set of key fields. In the case that no key fields exist, matching uses the most distinguishing set of fields it can find to pair up most of the objects, and then applies a lightweight form of synthesis (described next) to find a function that pairs up objects that remain.

With pairs of corresponding objects (o, o') in hand, TOS *synthesis* searches for functions that are consistent with the examples, i.e., functions δ for which $\delta(o) = o'$ for all matched pairs. Functions δ assign an expression to each new-version object field one at a time, where the expressions may reference any of the old object's fields (or fields reachable from them). These expressions may also contain constants, simple functions (e.g., the concatenation or partitioning of string expressions), and conditionals (e.g., if the value to assign to a field depends on the current value of another field). For

Mystery'12, Rome, Italy.

fields that are collections, we recursively invoke synthesis on the objects that make up each collection, mapping the resulting function over the old collection objects to produce the new one. When many functions are possible for a given set of examples, synthesis chooses the simplest. The language for expressing transformation functions δ was carefully chosen so that important operations (like intersecting a set of candidate functions) are efficient, while still being expressive enough to handle real examples.

As far as we are aware, no prior work attempts TOS matching, i.e., mapping heap objects from different program version executions. The TOS synthesis step is similar to recent work on synthesizing string and Excel table data transformation functions from input and output examples [5, 6]. TOS matching creates examples *automatically*, whereas the prior work requires users to provide examples. TOS functions are a superset of string transformations. Excel table functions focus on filters and numerics, where as TOS data transformations focus on objects, and furthermore, TOS generates the examples from unstructured heap snapshots.

We demonstrate our approach by synthesizing transformation functions for updates to several open-source Java servers, including JavaEmailServer (a POP and SMTP server), CrossFTP (an FTP server), Azureus (a Bittorrent client), and JEdit (a graphical text editor). We show that TOS produces correct transformation functions for a variety of actual program updates, whose changes require transformers that handle object field additions, string partitioning, partitioning a collection based on a predicate, and deleting objects due to memory leaks. In fact, to our knowledge no prior evaluation of a DSU system has considered the need to correct the residual effects of a memory leak, and our synthesized functions are the first demonstration of this capability.

In summary, this paper's contribution is *Targeted Object Synthesis*, a method for automatically generating data transformation functions needed by dynamic patches, which we have shown to be practical by considering a variety of real-world examples.

2. Overview

This section presents an overview of synthesizing state transformation functions for dynamic software updates using TOS. We begin with some background on DSU, present an example dynamic update taken from an actual program change, and show how TOS synthesizes this dynamic update automatically.

2.1 Dynamic software updating

Suppose an old version of a program is actively running, and a new version becomes available that fixes some bugs or adds some new features. Since the running program contains useful program state (e.g., active connections), we would like to update it without shutting it down. To use a typical DSU system to do so, we must construct a *dynamic patch* [8] that specifies the changed code and a *state transformation function*, which modifies heap objects and other program state, as necessary, to be compatible with the new code. For example, if the old program version maintains a list of Connection objects and the new version adds some fields to the Connection class, the state transformation function must initialize the values of the new fields for the existing objects. In some systems, the state transformer may also update the *control state* of the program, e.g., examining and modifying the existing stack and program counter as necessary [10, 14]. Our TOS design generalizes across DSU systems, and we implement TOS for Jvolve [14], which performs DSU in a Java Virtual Machine (based on Jikes RVM).

Figure 1 depicts a dynamically updated program's execution. The circles represent the program's state, the labels a_1, a_2, a_3, a_4 represent *actions*, e.g., messages sent to and from client applications. Each gray circle represents a state in which a dynamic update is permitted by the system, if one is available (discussed more below). In this trace, the program starts executing at version 1, and after executing actions a_1 and a_2, it applies a dynamic patch. As a result, the code of the program is updated to version 2, and the state transformation function δ is applied to transform the current state. A patch could have been applied in the initial state or the one after a_3, too, but no patch was available.

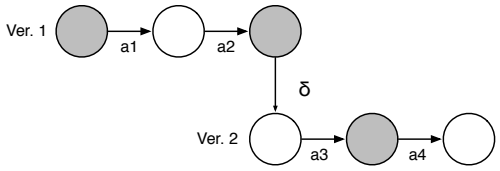

Figure 1. Trace of an updated program at update point after a2.

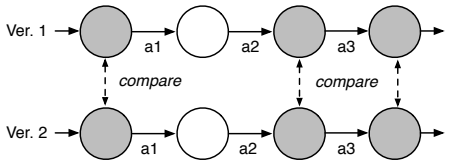

Figure 2. Comparing old & new version heaps at update points.

Most DSU systems work in three steps. First, they dynamically load the new and changed code. Second, they redirect existing references to the new definitions. Finally, they execute the state transformation function to update the existing state. Jvolve implements these steps within a modified virutal machine. It uses standard classloading to load new versions of classes. For classes whose only change is to the code of methods, Jvolve simply modifies the metadata for that class to point to the new method definitions (which the JIT may subsequently optimize). For each class whose objects' state requires modification (e.g., because the new version adds fields), the patch must include an *object transformation method*. At update time, the garbage collector finds all objects that require transformation. It executes the object transformation method on each old object, creating a new corresponding object that conforms to the new class's type specification and initializing its state.

When a dynamic patch becomes available the system may choose not apply it immediately. A policy adopted by many DSU systems is to delay updates while changed code is actually executing or referenced by the call stack. While this delay makes sense, it is not sufficient to avoid trouble. Hayden et al. [7] studied several years' worth of changes to three server programs and found that dynamic updates derived from actual releases sometimes fail even while adhering to this "activeness" restriction. Other work [8] has suggested that simply asking programmers to specify a few program points (dubbed *update points*) at which updates are permitted makes the system easier to reason about. Hayden et al.'s study finds this approach to be effective: updates were applied promptly (e.g., roughly every 10 ms) and never failed. Jvolve and several other systems [8, 10, 13] support this approach, and TOS takes advantage of it, as we will see in Section 2.3.

2.2 JavaEmailServer example

We now present an example Jvolve dynamic update and will subsequently show how TOS can synthesize it. Figure 3 illustrates code from versions 1.3.1 and 1.3.2. of JavaEmailServer (JES), a simple SMTP and POP e-mail server, that we obtained from the JES open-source repository. In the old version of the User class, forwardedAddresses is an array of strings. In the new version,

```
public class User {
  private final String username, domain, password;
  private String [] forwardAddresses;
  public User(...) {...}
  public String [] getForwardedAddresses() {...}
  public void setForwardedAddresses(String[] f)
  {...}
}
public class ConfigurationManager {
  private User loadUser(...) {
      ...
      User user = new User(...);
      String [] f = ...;
      user.setForwardedAddresses(f);
      return user;
  }
}
```

(a) Version 1.3.1

```
public class User {
  private final String username, domain, password;
  private EmailAddress[] forwardAddresses;
  public User(...) {...}
  public EmailAddress[] getForwardedAddresses() {...}
  public void setForwardedAddresses(EmailAddress[] f)
  {...}
}
public class ConfigurationManager {
  private User loadUser(...) {
      ...
      User user = new User(...);
      EmailAddress[] f = ...;
      user.setForwardedAddresses(f);
      return user;
  }
}
```

(b) Version 1.3.2

```
public class EmailAddress {
  public EmailAddress(
    String username,
    String domain
  ) {
    _isEmpty = false;
    _username = username;
    _domain = domain;
  }
  ...
  private String _username = "";
  private String _domain = "";
  private boolean _isEmpty = true;
}
```

(c) Common code

Figure 3. An update to JavaEmailServer (JES) User and ConfigurationManager classes

```
public class v131_User {
  private final String username, domain, password;
  private String [] forwardAddresses;
}
public class JvolveTransformers {
  ...
 public static void
  jvolveObject(User n, v131_User o) {
    n.username = o.username;
    n.domain = o.domain;
    n.password = o.password;
    int len = o.forwardAddresses.length;
    n.forwardAddresses = new EmailAddress[len];
    for (int i = 0; i < len; i++) {
      String [] parts = o.forwardAddresses[i].split("@");
      n.forwardAddresses[i] = new EmailAddress(parts[0], parts[1]);
}}}
```

Figure 4. User object transformer for updating JES 1.3.1 to 1.3.2

forwardedAddresses is an array of EmailAddress objects. This difference requires a corresponding change to the types of other methods in User, and to the loadUser method code of the ConfigurationManager class, which sets the field by calling setForwardedAddresses.

A Jvolve dynamic patch for this update contains the new versions of the User and ConfigurationManager classes. No object transformer is needed for the ConfigurationManager class because only its methods have changed, not its fields. Figure 4 illustrates the object transformer method for User objects. The transformer is a **static** method jvolveObject in the class JvolveTransformers. The method takes the old-version object and an allocated uninitialized new-version object as arguments. Both have the same class name, so to distinguish them Jvolve renames the old object's class to v131_User. The transformation method copies the first three fields from the old to the new version.[1] The object transformer allocates and populates an array of EmailAddress objects to replace the existing array of String objects.

Given two program versions, Jvolve and other systems can automatically construct the code portion of a dynamic patch by syntactically comparing the old and new class files. But generating the object transformer in Figure 4 is well beyond the reach of current techniques. Jvolve produces the first three lines, but then inserts the line n.forwardedAddresses = **null**. Ginseng [13], POLUS [4], and DLpop [8] do slightly better: they generate the loop, but the loop's body simply contains the statement n.forwardedAddresses[i] = **null**. Next we show how TOS generates this function in its entirety.

[1] Jvolve relaxes the restrictions on private field accesses during an update.

2.3 Matching

TOS works in two steps, *matching* and *synthesis*. Matching begins by executing the old and new version of the program on the same inputs. Each time the program reaches an update point, we take a standard heap *snapshot*, which records all the objects—their types and field values. The acquire a meaningful matching it must be the case that if the program is run twice with a given input, then the i^{th} snapshot always contains the same set of objects of that class. In a sense, we require that the program behave deterministically with respect to the class of interest when provided with a particular input. The JES example illustrates this point. The email server itself is non-deterministic. Network events and the order in which requests are processed may vary from one run to the next. However the set of forwarded addresses behaves deterministically since it is read from a configuration file and thus does not vary across runs.

As shown in Figure 1, matching works by comparing snapshots taken at update points, with the goal of identifying a mapping between corresponding objects in the old and new snapshots. Each mapped pair will serve as input/output example used to synthesize an object transformation method. As such, we care to pair up objects whose class has changed, and sometimes objects in a collection referenced from the field of a changed object, as in our JES example. Informally, two objects o_{old}, o_{new} correspond if they serve the same role in the two heaps. Synthesis works best when objects of a given type correspond one-to-one. That is, for each object of type T in the old heap, there will be exactly one corresponding object of type T in the new heap, and vice versa. The JES update produces such a one-to-one mapping. Figure 5 illustrates an old and new JES heap with three User objects. (Objects in collections may not correspond one-to-one as we discuss later.)

Each User object in the old heap clearly has a corresponding User object in the new heap. To automatically match objects such as these, we observe that objects often contain *key fields*. The fields $\vec{f}$ of a class C are key fields if objects of class C have two properties: no pair of C objects in the same snapshot have the same values for all the fields $\vec{f}$, and for each object in the old-heap snapshot there is exactly one object in the new-heap snapshot that has the same values for fields $\vec{f}$.

Old Version Heap Objects JES Heap

user 1
username = john
domain = yahoo.com
password = poor
forwardedAddresses[] =
["john@att.com", "john−alice@yahoo.com"]

user 2
username = alice
domain = yahoo.com
password = poor
forwardedAddresses[] =
["john−alice@yahoo.com"]

user 3
username = pat
domain = gmail.com
password = poor
forwardedAddresses[] = NULL

New Version Heap Objects JES Heap

user 1
username = john
domain = yahoo.com
password = poor
forwardedAddresses[0] =
[_username = "john", _domain = "att.com"]
forwardedAddresses[1] =
[_username = "john−alice", _domain = "yahoo.com"]

user 2
username = alice
domain = yahoo.com
password = poor
forwardedAddresses[0] =
[_username = "john−alice",
_domain = "yahoo.com"]

user 3
username = pat
domain = gmail.com
password = poor
forwardedAddresses[] = NULL

Figure 5. JES heap example

For the example in Figure 5, username is a key field. The matching algorithm begins by seeing whether any field on its own is a key field. In this case, it will discover that neither domain nor password are key fields because multiple objects in the same snapshot have the same value for each field. It identifies username is a key field because it produces a one-to-one perfect correspondence of old to new objects of the changed class User.

However, if one key field does not exist, for example, if **user 3** were john@gmail.com instead of pat@gmail.com, then there is no *single* key field, since none of the primitive fields username, domain, or password have unique values. In this case, matching searches for *key pairs* of fields. For the modified **user 3**, matching tests key pairs and finds that username, domain together impose a one-to-one mapping on all the objects.

This process is quadratic in the number of fields and the example illustrates that good inputs are needed to generate input/output pairs for synthesis. It is possible that no such mapping exists, in which case, TOS matching relaxes the one-to-one requirement, but still aims to identify key fields. TOS also uses any reference object field values to match. Section 3 describes these alternative techniques in more detail.

2.4 Synthesis

Synthesis proceeds in two steps. First, we synthesize a set of candidate functions for each example pair. Second, we intersect these sets to find a function consistent with all examples.

The first step proceeds as follows. For each example object pair, synthesis seeks a *set* of functions Δ such that each $\delta \in \Delta$ is consistent with the example, i.e., $o_{new} = \delta(o_{old})$ for the example (o_{old}, o_{new}). Each $\delta_i \in \Delta$ assigns each new-version field one at a time, but different functions may choose different right-hand-sides of the assignments. For example, consider the two User objects in Figure 5 that correspond to user 1. For this example, synthesis will infer δ_1 that assigns the *constants* john, yahoo.com, and poor to each of the new object's fields username, domain, and password, respectively, and δ_2 that *copies* the corresponding fields from its input object, i.e., n.username := o.username, n.domain := o.domain, etc. For fields of type String we may also assign the concatenation of other strings, e.g., those that are substrings of old-version fields.

For collections, we invoke synthesis recursively. We match two collections and then match the set of objects in the two collections. We then generate a transformation function between the collection pairs from a transformation function for the object pairs. For forwardedAddresses, we find that each String has the form "x@y" and can be mapped to an EmailAddress object whose username, domain, and _isEmpty fields are "x", "y", and **false**, respectively, where the first two fields are substrings of the input string. Once we have the element-wise function, we simply iterate over the old collection and map each element to one in the new collection. If there are more objects in the old collection than the new one, we search for a conditional function that identifies the missing objects and filters them out. If the collections have different sizes nondeterministically, we can take other steps, as described in Section 4.

Once we generate Δ for each pair of objects of type T, we intersect them to produce a $\hat{\Delta}$ that is consistent with all the examples. During this step we discard overly-specific functions, e.g., we would discover that δ_2 described above is consistent with all three of the matched pairs but δ_1 is not, and so will be discarded. If $|\hat{\Delta}| = 1$, then we choose $\delta \in \hat{\Delta}$ for the object transformer. If $|\hat{\Delta}| > 1$ we pick the *least* $\delta \in \Delta$ which is intuitively the *simplest* and *most general* function. For example, we mark functions that contain assignments from old fields to new fields as simpler than functions that assign constants. If $|\hat{\Delta}| = 0$ then no one function works for all examples. In this case, TOS picks the function that works for the most examples. Then, synthesis iteratively seeks a function that works for the remaining examples along with a conditional expression that will distinguishes between the two cases.

3. Matching

Now we present the TOS matching algorithm in detail; the next section describes the synthesis algorithm. The goal of matching is to produce pairs (o, o') of corresponding old and new objects taken from the heap snapshots. The synthesis phase takes these pairs as input and searches for a function δ such that $\delta(o) = o'$ for all the example pairs. The functions are type (class) based, thus all old objects o must have the same type τ and all new objects must have the same type τ', We first assume $\tau = \tau'$, and then consider $\tau \neq \tau'$, when matching recursively during synthesis.

Figure 6 gives the pseudocode for the matching algorithm. We write $\vec{X}$ to denote a list of Xs; $\vec{X} :: X$ to denote concatenating the element X to the end of the list $\vec{X}$; and $\vec{X}(i)$ to denote the i^{th} element of the list $\vec{X}$. We write $o.\vec{f}$ to denote the list $\vec{v}$ where $o.f_i = v_i$ for $0 < i \leq n$. We write $values_{\vec{f}}(\sigma)$ for the set of value tuples assigned to fields $\vec{f}$ by objects in σ, i.e., $values_{\vec{f}}(\sigma) = \{\vec{v} \mid o \in \sigma \wedge o.\vec{f} = \vec{v}\}$ Finally, we write $\vec{\sigma}\downarrow_{\vec{f}=\vec{v}}$ for $\{o \mid o \in \sigma \wedge o.\vec{f} = \vec{v}\}$.

The input to match is a pair of lists of object sets, $\vec{\sigma}_{old}$ and $\vec{\sigma}_{new}$. The object set $\sigma = \vec{\sigma}_{old}(i)$ contains objects collected from the i^{th} snapshot taken while running the old program, while $\sigma' = \vec{\sigma}_{new}(i)$ contains objects collected from the corresponding snapshot of a run of the new program. For example, if the class C changed between the old and new version, the input object sets would contain C objects from the corresponding snapshots, so that we can infer a δ from old to new C objects. When matching snapshots, each snapshot should contain the same number of objects (that is $|\vec{\sigma}_{old}| = |\vec{\sigma}_{new}|$). We then search for a bijection δ, so that $\vec{\sigma}_{new}(i) = \{o' \mid o \in \vec{\sigma}_{old}(i) \wedge o' = \delta(o)\}$.

Matching proceeds in two steps. The first step (line 2) finds *key fields* that partition the objects in each pair of snapshots into matching sets (the call to split_on_keyfields). Ideally, each of these sets is a singleton set, meaning that key fields were sufficient. If not, the next phase (the call to synthesis_match) uses the synthesis procedure itself to pair the remaining objects. Ultimately, matching returns a list of object pairs as input for synthesis.

3.1 Key fields

The get_keyfields function searches for key fields that partition the objects in corresponding snapshots into singleton lists such that synthesis can find a bijection between them. Given a list of fields kfs, and the old and new object set lists, the function split_on_keyfields returns a pair of set lists whose i^{th} elements correspond. In particular, each object $o \in \vec{\sigma}_{old}(i)$ and the corresponding object $o' \in \vec{\sigma}_{new}(i)$ have the same values for fields in kfs. Ideally, the size of $\vec{\sigma}_{old}(i)$ and $\vec{\sigma}_{new}(i)$ returned by split_on_keyfields will be 1 for all i. If so, then each object is uniquely identified by the fields kfs in every snapshot and there is a corresponding object in the old (respectively, new) snapshot with the same field values. The size of a set will be greater than 1 if there are multiple objects in a single snapshot that contain the same values for fields in kfs. The size of a set will be 0 if there are objects in an old (new) snapshot with field values not present in the new (old) snapshot.

The function get_keyfields iteratively adds new fields to the list kfs. When it reaches a fixed point, it returns the list. The first nested loop (line 11) considers each possible relevant field f for objects of type τ. The function assigns each field a *score* based on how well f and the current fields in kfs' distinguishes all the objects. The inner loop (line 14) considers each pair of snapshots and computes the set V, which contains the distinct lists of kfs' fields' values in old-version ojects. Line 16 then considers each of the values in set V. It must be the case that we have the same number of old and new objects with value $\vec{v}$ in field f. This preserves our ability to discover a bijection between the objects. The larger $|V|$ is, the finer the partition induced by splitting on kfs'. We want to prefer finer partitions, which indicate more effective key fields. Thus we add $|V|$ to score and then choose the field that maximizes score. If a field f leads to the condition on line 16 being violated, then we give f a score of 0 and proceed to the next field.

Once we score all the fields, we pick the field g that maximizes the score. If the best score does better at distinguishing objects than we previously did when using just fields in kfs, we add g to kfs and continue iterating.

If get_fields cannot find a bijection using one or more primitive fields, we extend it by changing the first nested loop (line 11) to consider *field paths*. A field path follows reference fields and tests if the values in objects to which o and o' point (e.g., $f \in o.m.f$ and $g \in o.n.g'$) are key fields. If this approach still does not produce a bijection, we apply synthesis based matching.

3.2 Synthesis-based matching

Line 4 get_fields splits $\vec{\sigma}_{old}$ and $\vec{\sigma}_{new}$ based on the discovered key fields. It there does not exist a bijection, one-to-one mapping of

```
1  match(σ⃗_old, σ⃗_new) =
2    kfs := get_keyfields (σ⃗_old, σ⃗_new)
3    if ∃i. |σ⃗'_old(i)| > 1 then
4      (σ⃗'_old, σ⃗'_new) := synthesis_match(split_on_keyfields(kfs,σ⃗_old,σ⃗_new))
5    return {(o,o') | ∃i. 0 < i ≤ length(σ⃗'_new) ∧ o ∈ σ⃗'_old(i) ∧ o' ∈ σ⃗'_new(i)}

6  get_keyfields (σ⃗_old, σ⃗_new) =
7    kfs := []
8    currscore := 0
9    repeat {
10     prevkfs := kfs;
11     for each field f {
12       kfs' := kfs :: f
13       score(f) = 0
14       for each i ∈ 1..length(σ⃗_new) {
15         V := values_kfs'(σ⃗_old(i))
16         if ∀v⃗ ∈V. |σ_old(i)|_kfs'=v⃗| = |σ_new(i)|_kfs'=v⃗| then
17           score(f) := score(f) + |V|
18         else {
19           score(f) = 0
20           break
21       } }
22     }
23     let g be the f that maximizes score(f)
24     if score(g) > currscore {
25       kfs := kfs :: g
26       currscore := score(g)
27     }
28   } until (prevkfs = kfs)
29   return kfs
30 }

31  split_on_keyfields (kfs, σ⃗_old, σ⃗_new) =
32    σ⃗'_old := []
33    σ⃗'_new := []
34    for each i ∈ 1..length(σ⃗_new) {
35      for each v⃗ ∈ values_kfs(σ⃗_old(i) ∪ σ⃗_new(i)) {
36        σ⃗'_old := σ⃗'_old :: σ_old(i)|_kfs=v⃗
37        σ⃗'_new := σ⃗'_new :: σ_new(i)|_kfs=v⃗
38      }
39    }
40    return (σ⃗_old, σ⃗_new)

41  synthesis_match(σ⃗_old, σ⃗_new) =
42    σ⃗'_old := []
43    σ⃗'_new := []
44    for each k ∈ 1..length(σ⃗_old) {
45     σ := σ⃗_old(k)
46     σ' := σ⃗_new(k)
47     if |σ| ≠ 1 ∨ |σ'| ≠ 1 then
48      while σ' ≠ ∅ {
49        choose o from σ
50        let Δ = ⋃_{o_i ∈ σ'} synth_non_branching(o,o_i)
51        let score(δ) = |σ' ∩ δ(σ)|
52        let δ̂ be the element of Δ that maximizes score
53        let Û = {(o,o') | (o,o') ∈ σ × σ' ∧ o' = δ̂(o)}
54        for each (o,o') ∈ Û {
55          σ⃗'_old := σ⃗'_old :: {o}
56          σ⃗'_new := σ⃗'_new :: {o'}
57          σ := σ − {o}
58          σ' := σ' − {o'}
59       }
60     }
61   }
62   return (σ⃗'_old, σ⃗'_new)
```

Figure 6. Pseudo-code for the match function.

old to new objects, these lists will contain non-singleton sets. We would like to further decompose such sets, since whenever $\vec{\sigma}'_{old}(i)$ and $\vec{\sigma}'_{new}(i)$ contain more than one element each, it is not clear which pair (o_1, o_2) with $o_1 \in \vec{\sigma}'_{old}(i)$ and $o_2 \in \vec{\sigma}'_{new}(i)$ to use as an input-output example for the transformation function δ. In this case, match calls the function synthesis_match with the two current lists of corresponding sets to refine the non-singleton sets in the lists.

Synthesis-based matching tries to find a matching that is witnessed by a transformation function δ that maps objects in the old set to objects in the new set. We introduce some additional terminology to explain the algorithm. We say that δ is *consistent* with the object pair (o, o') iff $o' = \delta(o)$. We call a set of pairs U a *matching* iff for all pairs (o_1, o'_1) and (o_2, o'_2) in U, we have $o_1 \neq o_2 \Leftrightarrow o'_1 \neq o'_2$. That is, no old-version object is paired with multiple new-version objects, nor is any new-version object paired with multiple old-version objects. When U is a matching with elements in $\sigma \times \sigma'$, we write $\sigma \sim_U \sigma'$. Formally, δ is consistent with a matching U, written $U \subseteq \delta$, if and only if $\forall (o, o') \in U.\ o' = \delta(o)$.

The synthesis_match function iterates over each pair of snapshots, attempting to get a 1-to-1 matching between the old and new objects of each when the sets σ and σ' are not singletons. Consider the non-singleton pair σ and σ'. The function first chooses an old-version object o (line 49). Any object may be chosen since we have to eventually find transformation functions for each object in $\vec{\sigma}_{old}$. We then consider each pair (o, o_i), where o_i is a new-version object, with the aim of synthesizing Δ_i, a *set* of transformation functions consistent with the example (o, o_i) (line 50). The synthesis procedure used here is restricted to non-branching transformation functions. These are functions that do not perform case analysis on the old-version object. (Figure 9 gives pseudocode for this function, which is explained in the next section.) Without this restriction, the inferred functions can always creates a conditional function specific only to this one pair, but these functions are not general and do not help us create sets of examples for synthesis of general functions.

Next, the algorithm combines all these sets of functions into a single set of transformation functions Δ. Then it generates the set C containing pairs (δ, U) where U is a mapping consistent with the transformation function δ drawn from Δ. From this set we choose the pair $(\hat{\delta}, \hat{U})$ where $\hat{U}$ is the largest possible mapping between the two snapshots.[2] Thus, we take a greedy approach, choosing transformation functions according to how many example pairs they cover. Finally, the loop on line 54 adds the singleton sets comprising this mapping to $\vec{\sigma}'_{old}$ and $\vec{\sigma}'_{new}$ (which will be the ultimate output of this function) and we remove the mapped elements from the current snapshots σ and σ'. Once σ becomes empty, which is sure to happen because ultimately objects can be matched arbitrarily using constant functions, we exit the while loop and move on to the next snapshot, continuing until all snapshots have been considered.

4. Synthesis

The synthesis phase takes the example pairs produced by matching and synthesizes a function δ from them so that for the pair (o, o') we have $\delta(o) = o'$.

4.1 Transformation functions δ

The transformation function δ is defined according to the grammar in Figure 7. Transformation functions take an old-version object in o, allocate a new-version object in n, assign values to each of the fields of n, and then return n. Field assignments g are of the form $\mathrm{n}.f := c$ where c is a conditional and f is a *field path*—that is, it is a (possibly empty) list of field labels l. For n this path is almost always a single field; we sometimes allow it to be multiple fields as discussed in Section 4.5. Each conditional c specifies one or more cases distinguished by boolean expressions e_i over old-version state (i.e., they can only refer to objects via o, never n). Each case has an initializer expression d which is either a constant, a reference to an old field, a string produced by concatenating one or more (sub)strings, or a collection produced by map.

Updates	δ	::=	λo. new n; g_1; …; g_n;**ret** n
Field Updates	g	::=	$n.f := c$
Field Path	f	::=	$\varepsilon \mid f.l$
Conditional	c	::=	case $e_1 \Rightarrow d_1, \ldots, e_n \Rightarrow d_n$ end
Initializer	d	::=	$k \mid \mathrm{o}.f \mid \mathsf{concat}(se_1, \ldots, se_n)$
			$\mid \mathsf{map}(\delta, \mathrm{o}.f)$
Integer Constant	i, j	$\in$	$\mathbb{Z}$
Constant	k	::=	$i \mid$ `null` $\mid$ *delim*
Delimiter	*delim*	::=	`\` | `/` | `#` | `@` | `:`
String Expression	se	::=	*delim* $\mid \mathsf{substr}(\mathrm{o}.f, i, j)$
Boolean Expression	e	::=	$a \mid e_1 \wedge e_2 \mid e_1 \vee e_2$
Atomic Expression	a	::=	$\mathrm{o}.f_1 \diamond \mathrm{o}.f_2 \mid \mathrm{o}.f \diamond k$
Operator	$\diamond$	::=	$= \mid \neq \mid \ldots$

Figure 7. The language over which we perform synthesis.

Here, $\mathsf{map}(\delta, \mathrm{o}.f)$ takes the collection at $\mathrm{o}.f$, and transforms these elements using δ. The initializer $\mathsf{substr}(\mathrm{o}.f, i)$ splits the string $\mathrm{o}.f$ at positions where a delimiter *delim* appears and then selects the i^{th} substring. For example, substr("foo@bar.com", 2) returns "bar.com" since '@' is a delimiter that splits the string into two substrings and "bar.com" is the second substring. Our language of string updates supports a concatenation of substrings and is sufficient for the examples we considered. If a more robust string transformation language were needed, the approach taken by Gulwani [5, 6] or any other example-based synthesis technique for strings would work without modification.

We have simplified the form of δ to keep the algorithm tractable. The most obvious restriction is that δ transforms a single object, rather than multiple objects at once. This restriction derives from the nature of the underlying DSU system we are using, Jvolve. Note that different objects of the same class will not necessarily be transformed in the same way—conditionals c may evaluate to different branches for different objects and thereby trigger different initializers. Another restriction is that each field of the new object is initialized independently, albeit with access to the full contents (i.e., multiple fields) of the old-version object o. Transformers that are ruled out by this setup include those that initialize field f according to the *updated* value for field f' as well as those that pass values across the object graph, e.g. using the value in $\mathrm{o}.f_1$ when updating field $\mathrm{n}.f_2.f_3$.

4.2 Synthesis algorithm

Figure 8 gives pseudo-code for the synthesis algorithm. The main function is synthesize. This takes a set of input-output examples U (pairs of old-version, new-verion objects) as input and produces an transformation function that is consistent with those examples.

Synthesis proceeds one field at a time. For each field, we synthesize a conditional update (c in the grammar in Figure 7) that is capable of producing all the values for that field seen in the example pairs in U. We do this by first generating the initializers d and then searching for conditions that tell us which d to apply.

[2] In the implementation we directly generate these maximal mappings, omitting from C the pairs (δ, U) where U can be extended with additional pairs (o, o') that are consistent with δ. This improves the efficiency by decreasing the size of the sets we have to consider.

```
1  synthesize(U) =
2    for each new-version field f_i {
3      not_covered := U
4      update_fns := []
5      while not_covered ≠ ∅ {
6        choose (o,o') ∈ not_covered
7        let D = synth_field(f_i,o,o')
8        let score(d) = |{(o,o') | (o,o') ∈ not_covered ∧ o'.f = d(o)}|
9        let d̂ be the element of D that maximizes score
10       let Û = {(o,o') | (o,o') ∈ not_covered ∧ o'.f = d̂(o)}
11       update_fns := update_fns :: (d̂, Û)
12       not_covered := not_covered − Û
13     }
14     cond := []
15     for i ∈ 1..length(update_fns) {
16       let (d̂, Û) = update_fns(i)
17       let σ_in = {o | ∃o'. (o,o') ∈ U ∧ (o,o') ∈ Û_i}
18       let σ_out = {o | ∃o'. (o,o') ∈ U ∧ (o,o') ∉ Û_i}
19       cond := cond :: (synth_cond(σ_in,σ_out))
20     }
21     c_{f_i} := ⟨case ĉ_1 ⇒ d̂_1, ..., ĉ_n ⇒ d̂_n end⟩
22       where ĉ_j = cond(j) and d̂_j = update_fns(j)(0)
23   }
24   return (λo. new n; n.f_1 := ⟨c_{f_1}⟩; ...; n.f_n := ⟨c_{f_n}⟩; ret n)
```

Figure 8. Main synthesis algorithm.

To find the initializers d, we maintain a set update_fns of initializers that have been discovered thus far, as well as a set not_covered that contains all example pairs that cannot be produced by an initializer in update_fns. During each iteration through the loop on line 5, we choose an element from not_covered and call synth_field, which returns the set D of all initializers d that are consistent with that field's values in the provided example pair. Of these, we choose $\hat{d}$, which covers the largest number of pairs in not_covered, and add it update_fns while removing the pairs that d covers from not_covered.

To find the conditions that indicate which d to apply to a given old-version object, we execute the loop on line 15. For each transformation function $\hat{d}$, the loop builds σ_{in}, containing the old-version objects from the example pairs consistent with $\hat{d}$, and σ_{out}, containing the remaining example pairs. It then calls synth_cond to find a condition that separates these two sets based on values of old-version fields, and adds it to the list cond.

Finally, we construct c_{f_i}, the conditional update containing all the logic we just synthesized for field f_i. The update for the class is then the sequence of all these field updates. We write synthesized code in angle brackets to distinguish it from the code of the synthesis algorithm.

4.3 Field synthesis

Figure 9 gives the pseudo-code for synth_field, which produces the initializers for a new-version field according to a given example pair. The figure also shows the code for synth_non_branching, which is the function used to perform synthesis-based matching (Section 3.2), and uses synth_field as a subroutine.

The synth_field function takes a field, an old-version object o, and a new-version object o', and returns a set of initializers (production d in the grammar in Figure 7), where each initializer can produce the value in $o'.f$ given the object o. The set is constructed by first checking whether the value in $o'.f$ is also present in a field in o. If so, the value can be copied over. Next, we check whether $o'.f$ is one of the constants in our language. If so, this production method is also added to the returned set. Finally, we have two type-based synthesis checks. If $o'.f$ is a string, we invoke string synthesis to produce a set that describes all possible methods for construction the string from substrings present in o. We do not give code for this function since it closely follows the approach from [5]. If $o'.f$ is a collection, then we recursively invoke synthesis in order to transform the elements of the collection.

```
1  synth_non_branching(o,o') =
2    g := empty field update
3    for each f in o' {
4      let c_set = synth_field (f,o,o')
5      for each c ∈ c_set do { g := g + ⟨ ; n.f := c ⟩ }
6    }
7    return λo. new n; ⟨g⟩;ret n
8
9  synth_field (f,o,o') =
10   ret_set := ∅
11   if o'.f = o.g for some field g
12     ret_set := ret_set ∪ {⟨o.g⟩}
13   if o'.f = k
14     ret_set := ret_set ∪ {⟨k⟩}
15   if typeof(o'.f) = String
16     ret_set := ret_set ∪ string_synth(o.f,o'.f)
17   if o'.f is a collection
18     let σ' = multi−set of objects in collection o'.f
19     for each collection−valued field o.f_2 {
20       let σ = multi−set of objects in collection o.f_2
21       let δ = collection_synth(σ,σ')
22       ret_set := ret_set ∪ {⟨map(δ,o.f_2)⟩}
23     }
24   return ret_set
25
26 collection_synth (σ,σ') =
27   let examples = match([σ],[σ'])
28   return synthesize(examples)
```

Figure 9. Non-branching and per-field synthesis subroutines.

4.4 Condition synthesis

Condition synthesis synthesizes a condition that distinguishes two sets of examples following the basic approach of Gulwani [5]. The pseudocode is given in Figure 10. This code uses notation $[\![e]\!]$ to denote a function from objects to truth values where $[\![e]\!]\ o = \text{true}$ if and only if condition e is true when o (an actual object) is substituted for o (the variable) in e. If $[\![e]\!]\ o = \text{true}$ we will say that *e includes o* and otherwise we will say that *e excludes o*. Given a set of objects σ_{in} and σ_{out}, the goal of condition synthesis is to return an expression e such that $\forall o \in \sigma_{\text{in}}.\ [\![e]\!]\ o = \text{true}$ and $\forall o \in \sigma_{\text{out}}.\ [\![e]\!]\ o = \text{false}$. If this is the case, we say that *e separates* σ_{in} and σ_{out}.

The construction of e proceeds in a greedy fashion. The loop at line 4 calls synth_conj to synthesize a conjunction $a_1 \wedge \ldots \wedge a_n$, where each a_i is an atomic expression, that excludes all the elements in σ_{out} while including as many elements of σ_{in} as possible. The synth_conj function builds up this conjunction iteratively using the loop at line 15. Given $e = a_1 \wedge \ldots \wedge a_j$, we first check to see if e already separates σ_{in} and σ_{out}. If so, we are done—in the code this happens when $\sigma'_{\text{out}} = \emptyset$. If not, let $\sigma'_{\text{in}} = \{o \mid o \in \sigma_{\text{in}} \wedge [\![e]\!]\ o = \text{true}\}$ and let $\sigma'_{\text{out}} = \{o \mid o \in \sigma_{\text{out}} \wedge [\![e]\!]\ o = \text{true}\}$. Thus σ'_{in} and σ'_{out} are the subsets of σ_{in} and σ_{out} that satisfy e. We then choose a_{j+1} to be the atomic expression that maximizes $\mathit{rank}(a_{j+1}, \sigma'_{\text{in}}, \sigma'_{\text{out}})$. We define the *rank* of a condition e as follows.

Definition 1. *Let* $\mathit{rank}(e, \sigma_{in}, \sigma_{out}) = m \times n$ *where* $m = |\{o \mid o \in \sigma_{in} \wedge [\![e]\!]\ o = \text{true}\}|$ *and* $n = |\{o \mid o \in \sigma_{out} \wedge [\![e]\!]\ o = \text{false}\}|$.

```
1   synth_cond(σin,σout) =
2     σ'in := ∅
3     e := ⟨false⟩
4     while σ'in ≠ σin {
5       let e' = synth_conj(σin − σ'in,σout)
6       e := e + ⟨ ∨ e'⟩
7       σ'in := σ'in ∪ {o | o ∈ σin ∧ [[e']] o = true}
8     }
9     return e
10
11  synth_conj(σin,σout) =
12    σ'in := σin
13    σ'out := σout
14    e := ⟨true⟩
15    while σ'out ≠ ∅ {
16      let a be the condition that maximizes rank(a, σ'in, σ'out)
17      e := e + ⟨ ∧ a⟩
18      σ'in := {o | o ∈ σ'in ∧ [[e]] o = true}
19      σ'out := {o | o ∈ σ'out ∧ [[e]] o = true}
20    }
21    return e
```

Figure 10. Synthesizing conditions.

Thus, $rank(e, \sigma_{\text{in}}, \sigma_{\text{out}})$ is the product of the number of objects from σ_{in} that are included by e and the number of objects in σ_{out} that are excluded. We can

The atomic expressions we consider are those involving equality or disequality between pairs of fields (of which there are a finite number) and equality or disequality with a constant appearing in the object (of which there are a finite number). Since there are a finite number of possible atomic expressions, we can simply iterate over them, although our implementation is able to avoid considering many expressions that are guaranteed to not satisfy the conditions required. Provided $rank(\vec{\sigma}'_{new}, \sigma'_{\text{in}}, \sigma'_{\text{out}})$ is non-zero, we know that conjoining a to e will cause some elements of σ'_{out} to be excluded, while still including some elements of σ_{in}. If no atomic expression produces a rank greater than zero, then this indicates a failure to find the necessary condition and we abort the synthesis process for this field. We continue adding atomic expressions until we have excluded all elements of σ_{out} and constructed conjunct e_1.

Returning to the code of synth_cond, we know that e' returned from synth_conj excludes all elements of σ_{out}. But there may be elements of σ_{in} that it does not yet include. To handle this we then let $\sigma'_{\text{in}} = \{o \mid o \in \sigma_{\text{in}} \wedge [\![e']\!]\ o = \text{true}\}$ add e' to the current list of disjuncts, and continue iterating until the expression includes all elements of σ_{in} and excludes all elements of σ_{out}.

4.5 Discussion

Ultimately, the synthesize algorithm tries to produce all transformation functions that are semantically distinct. For example, suppose class Foo contains an integer-valued field f and f is always 0 in the old version and new versions. Then we will generate the function (λo.new n; case true $\Rightarrow$ n.$f := 0$ end; **ret** n) and the function (λo.new n; case true $\Rightarrow$ n.$f :=$ o.f end; **ret** n) because there are objects for which these functions would produce different results, even if such objects do not appear in our examples. But our algorithm will not produce the function

$$\lambda\text{o.new n; case } (\text{o}.f = 0) \Rightarrow \text{o}.f := 0,\ (\text{o}.f \neq 0) \Rightarrow \text{o}.f := 0 \text{ end; } \textbf{ret } \text{n}$$

since this is semantically equivalent to the first function above.

The algorithm as described has assumed that synthesis takes place on a per-object basis, one field at a time. In fact, it is can also synthesize the contents of a new object's chidlren, e.g., assigning not just n.$f := d$ but n.$f_1.f_2 := d$ as well. Likewise it can read children of old objects in initializers d. To support this extension we simply consider field *paths* rather than fields, up to a specified depth, e.g., on line 2 in Figure 8 and on line 11 in Figure 9. Matching can be similarly extended, e.g., using paths on line 11 in Figure 6. We have found this flexibility useful in practice, as we discuss in Azureus example in the next section.

5. Experimental evaluation

We implemented TOS and applied it to synthesize object transformers for several program updates we observed in the wild. This section provides a few details about our implementation and experiences with it.

5.1 Implementation

We use the Oracle HotSpot JVM to collect heap snapshots using the `agentlib:hprof` command line option. This option invokes the heap profiler which sends the current snapshot over a network socket. We wrote a small server that coordinates snapshots with the application. It initiates snapshots at update points and saving them to disk. In particular, when the applicate reaches an update point, it calls into a helper class to trigger a snapshot.

TOS is implemented in Java and comprises roughly 4200 lines of code. About 1300 lines implement matching, 1600 implement synthesis, and the rest is common code. The implementation works basically as we have described in the last two sections but makes one optimization. We examine multiple heap snapshots while running the old version of the program and identify fields whose values remain unchanged over the lifetime of an object. We then perform matching only on the unchanged fields, which are more likely to be key fields, improving the overall matching time. Should we fail to generate an update for a field, we observe whether that field was deterministic. If it has different values at the same point during two runs in the same version, then we do not expect to derive useful synthesis examples for that field across versions.

5.2 Experiments

The rest of this section presents our experience using TOS to generate state transformers for updates to open-source Java applications that we found in the wild. We consider two updates to Azureus (a bittorrent client); two updates to jEdit (a text editor); one update to JavaEmailServer (an SMTP and POP mail server); and one update to CrossFTP (an FTP server). Table 1 summarizes information about the updates and the inferred transformers. We also include an update to jEdit that fixes a memory leak where TOS cannot infer the transformer because we were unable to force program execution down a path that causes the leak. This problem is an inherent limitation of our approach and is similar to the coverage problem in program testing. We can only generate transformers for properties of objects that we observe during execution.

The applications range from 2.3k to 250k source lines of code. The examples include constant updates, conditional updates, string transformations, and collection updates. All updates we considered here are examples that prior DSU systems cannot handle automatically. Many DSU systems support filling in new fields with a default value. Even in these cases, our approach would be useful, as it provides evidence that the choice of default value was correct.

For JavaEmailServer and CrossFTP, we used the generated update functions in Jvolve [14] and verified that the system continued running following the update. Other issues prevent Azureus and jEdit from executing in the Jvolve VM and prevented us from evaluating the updates that we inferred for those applications.

Application	SLOC	Version	Inferred	Type
Azureus	250,000	r2514	yes	conditional
		r120	no	-
JEdit	154,000	r14027	yes	constant
	90,000	r5178	yes	constant
	150,000	r13413	no	
JavaEmailServer	2,300	1.3.2	yes	collection
CrossFTP	14,000	1.07	yes	constant

Table 1. Summary of updates and inferred transformers

JavaEmailServer The JavaEmailServer example is the one discussed in Section 2 and presented in Figure 3. We generate a correct collection update function for it.

Azureus update SVN r2514 Azureus is a widely used BitTorrent server/client. Version r2514 contains about 250k lines of code. This update changes 2 classes and methods from its previous version. Figure 11 shows a portion of the update. The new version clears some resources used by a torrent's server when it stops the server, and then garbage collector can reclaim the unused memory.

By comparing objects of class `PEPeerControlImpl`, we inferred the following conditional fix.

```
if (_bContinue == false)
  _server.adapter = null;
else
  _server.adapter = _server_old.adapter;
```

The synthesis algorithm identifies that when _bContinue is false, the server is not running, and the adapter field should be set to null. Note that to generate this transformation function requires using field paths instead of single fields.

Azureus update SVN r120 Figure 12 shows update r120 to Azureus. This update modifies the condition under which Azureus releases the read buffer between a client and its peer. Since the the allocation of buffers is non-deterministic, TOS matching fails to find a mapping and produce an input for synthesis.

jEdit jEdit is a text editor for programmers that provides common features such as syntax highlighting, folding, automatic indentation, and advanced features such as a built-in macro language, macro recording, and plugin support. Here we consider two similar memory leaks fixed in jEdit versions r5178 and r14027. Figure 13 shows the source patch for the leak fixed in r14027. jEdit calls the function `markTokens` when it needs to split a string into tokens based on the type of the file being edited. Each file type (C, Java, Verilog, etc.) has special logic to split text in a line and embedded it in an object of type `TokenHandler`. The leaky jEdit version fails to set the field `TokenMarker.tokenHandler` to `null`.

The inferred object transformer is simple. It sets the leaky field to `null`. It is safe to execute the transformer as long as the `markTokens` function is not active on stack.

Figure 14 shows a change to jEdit in SVN revision r13413. The leak is in function `HistoryText.showPopupMenu()`. jEdit calls `showPopupMenu()` when the user performs certain actions, such as right clicking a text field with history. In the old version, some instances of `HistoryText` have the boolean field `popup.visible` set to true and others set it to false. In the new version, the field `popup` is not null only when an object also has its `popup.visible` field set to true. We want to infer this property and generate a transformer that nullifies the `popup` field of instances that have `popup.visible` set to false. While creating snapshots, we called the `showPopup()` function. However, we were unable to create a situation where `popup.isVisible()` was true during a snapshot,

```
class PEPeerControlImpl {
  void stopAll() { ...
    // 3. Stop the server
    _server.stopServer();
    _server.clearServerAdapter();
    ... }
}
class PESharedPortServerImpl {
  void clearServerAdapter() {
    adapter = null;
  }
}
```

Figure 11. Azureus r2514 update

```
public class PeerSocket ... { ...
    // 4. release the read Buffer
-   if (readBuffer != null && !readingLength)
+   if (readBuffer != null)
      ByteBufferPool.getInstance().freeBuffer(readBuffer);
 ... }
```

Figure 12. Azureus update r120

```
class TokenMarker {
  public LineContext markTokens(...) {
        ...
        tokenHandler.setLineContext(context);

        /* for GC. */
+       this.tokenHandler = null;
        this.line = null;
        return context;
    }
}
```

Figure 13. Update r14027 to jEdit

```
class HistoryText {
 showPopupMenu() {
   if (popup != null && popup.isVisible())
   {
       popup.setVisible(false);
+      popup = null;
       return;
   }

- popup = new JPopupMenu();
+ popup = new JPopupMenu() {
+      @Override
+      public void setVisible(boolean b) {
+          if (!b) {
+              popup = null;
+          }
+          super.setVisible(b);
+      }
+ };
   JMenuItem caption = new JMenuItem(jEdit.getProperty(
       "history.caption"));
   caption.addActionListener(new ActionListener()
 }
}
```

Figure 14. Update to jEdit: SVN revision r13413

```
class DataConnectionConfig {
   ...
+  boolean enableBonjour = true;
+  String listEncoding = "utf-8";
}
```

Figure 15. Update to CrossFTP in version 1.07

which would execute instructions controlled by the condition and exercise the leak. This example update is one which our synthesis framework could generate, but which we fail to synthesize because of test coverage problems.

CrossFTP CrossFTP is an FTP server. The class `DataConnectionConfig` maintains configuration information about the connection between the client and the server. Its fields control what response the server sends to various commands a client issues. Version 1.07 of CrossFTP adds two new fields to this class. The boolean field `enableBonjour` controls whether the server should support the Bonjour protocol and the String field `listEncoding` specifies what encoding the server should use when responding to the `LIST` command. There is always only one instance of this object in the heap. From the heap snapshots, we identify the value of these fields in the new version and generate the transformation function that sets these fields accordingly.

6. Related work

This paper contributes novel matching and synthesis algorithms. The matching algorithm analyzes unstructured heap snapshots from different program version executions. While some recent prior work analyzes a single heap to discover leaked objects and other inefficiencies [2, 3, 9, 11, 12, 15], none aligns heap objects from different program versions or considers how to generate transformation functions for the heap.

A lot of related work considers synthesizing code from specifications, but only recently have researchers considered the problem of synthesizing data transformation functions. The closest related work is by Gulwani et al. on synthesizing string and Excel spreadsheet data transformations [5, 6]. These approaches require users to specify the input/output examples. The string functions include concatonation, subsequence, and finding special symbols. We can synthesize functions that subsume those synthesized by these systems. Harris and Gulwani search structured spreadsheet data to find correlation between the input and output rows and columns. Their numeric transformations and filters are similar to ours. They exploit the structure of the spreadsheets, whereas TOS finds correlations in unstructured data by exploiting the Java static type system. TOS is the only approach that iterates synthesis and matching to produce input/output examples that result in better functions.

Many prior dynamic updating systems, including Ginseng [13], DLpop [8], POLUS [4], and Jvolve [14], provide primitive support for generating state transformation code. For changes that extend classes or structs with new fields, these systems simply copy the old fields and initialize the new ones with default values, e.g., **null** for object references, or 0 for **int** s. Systems that provide no direct support for state transformation, e.g., LiveRebel [16], effectively take this approach. In all of these cases, synthesis is based entirely on comparing the definitions of changed types/classes. None of them consider semantics by analyzing the code or the heap. By contrast, TOS obtains semantic information from program execution to derive data transformations. In short, while prior systems remove some of the tedium of writing transformation functions, they fail to handle any interesting program changes, which are exactly the cases which are harder for programmers to write correctly.

7. Conclusions

This paper has presented *Targeted Object Synthesis* (TOS), a novel technique for synthesizing *object transformer methods*, used by dynamic updating systems to convert existing objects an code version to a new one during a dynamic update. TOS is distinguished by its generality: whereas prior techniques for synthesizing object transformers follow simple syntactic rules, TOS produces functions based on observations of actual program executions of the old and new program versions. In particular, TOS works by taking periodic heap snapshots at corresponding points during executions of the old and new program when run on the same inputs. It then *matches* corresponding objects between these snapshots, and uses these as examples to *synthesize* object transformation functions. We have shown TOS to be efficacious in synthesizing transformation functions for actual changes to classes in various Java servers.

References

[1] J. Arnold and F. Kaashoek. Ksplice: Automatic rebootless kernel updates. In *EuroSys*, 2009.

[2] M. D. Bond and K. S. McKinley. Bell: Bit-Encoding Online Memory Leak Detection. In *ASPLOS*, 2006.

[3] M. D. Bond and K. S. McKinley. Leak pruning. In *ASPLOS*, 2009.

[4] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew. POLUS: A POwerful Live Updating System. In *ICSE*, 2007.

[5] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011.

[6] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *PLDI*, 2011.

[7] C. M. Hayden, E. K. Smith, E. A. Hardisty, M. Hicks, and J. S. Foster. Evaluating dynamic software update safety using efficient systematic testing. *IEEE Transactions on Software Engineering*, 99(PrePrints), Sept. 2011.

[8] M. Hicks and S. M. Nettles. Dynamic Software Updating. *Transactions on Programming Languages and Systems*, 27(6):1049–1096, November 2005.

[9] M. Jump and K. S. McKinley. Cork: Dynamic Memory Leak Detection for Java. In *POPL*, 2007.

[10] K. Makris and R. Bazzi. Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction. In *USENIX ATC*, 2009.

[11] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. In *OOPSLA*, 2007.

[12] N. Mitchell and G. Sevitzky. LeakBot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications. In *ECOOP*, 2003.

[13] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical Dynamic Software Updating for C. In *PLDI*, 2006.

[14] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic Software Updates: A VM-centric Approach. In *PLDI*, 2009.

[15] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. In *PLDI*, 2010.

[16] ZeroTurnaround. LiveRebel. `http://www.zeroturnaround.com/liverebel`.