Synthesizing Replacement Classes

MALAVIKA SAMAK, Massachusetts Institute of Technology, USA
DEOKHWAN KIM, Massachusetts Institute of Technology, USA
MARTIN C. RINARD, Massachusetts Institute of Technology, USA

We present a new technique for automatically synthesizing replacement classes. The technique starts with an original class O and a potential replacement class R, then uses R to synthesize a new class that implements the same interface and provides the same functionality as O. Critically, our technique works with a synthesized inter-class equivalence predicate between the states of O and R. It uses this predicate to ensure that original and synthesized methods leave corresponding O and R objects in equivalent states. The predicate therefore enables the technique to synthesize individual replacement methods in isolation while still obtaining a replacement class that leaves the original and replacement objects in equivalent states after arbitrarily long method invocation sequences. We have implemented the technique as part of a tool, named Mask, and evaluated it using open-source Java classes. The results highlight the effectiveness of Mask in synthesizing replacement classes.

CCS Concepts: • **Software and its engineering → Programming by example**; *Object oriented frameworks.*

Additional Key Words and Phrases: Program Sketching, Class Replacement, Program Equivalence

ACM Reference Format:
Malavika Samak, Deokhwan Kim, and Martin C. Rinard. 2020. Synthesizing Replacement Classes. *Proc. ACM Program. Lang.* 4, POPL, Article 52 (January 2020), 33 pages. https://doi.org/10.1145/3371120

1 INTRODUCTION

Libraries play a central role in the software development process as they provide the necessary interfaces that can be used by client applications. Oftentimes, client applications are refactored to use an updated version of a library or a different library that provides similar functionality. There are many reasons for this refactoring – the original library is deprecated or is no longer supported [API Deprecation 2018; JDK Deprecation 2018], the need to switch library vendors to satisfy organizational requirements or intellectual property constraints [Guava Collections 2009; Oracle v/s Google 2019], libraries with improved performance and memory usage [Guava v/s Apache 2009; Hasan et al. 2016], updated library versions where bugs are fixed [Fraser and Arcuri 2011; JDK Bug fixes 2019], etc.

Manually updating the application to use a different library can be cumbersome and error-prone [Dig and Johnson 2006; Kapur et al. 2010]. For example, even when library versions are updated, backward compatibility is not always maintained.[1] Ensuring that the behavior of the application is unchanged can become non-trivial because an existing class in the library can differ

[1]Java incompatibility report https://abi-laboratory.pro/?view=timeline&lang=java&l=jre.

Authors' addresses: Malavika Samak, Massachusetts Institute of Technology, USA, malavika@csail.mit.edu; Deokhwan Kim, Massachusetts Institute of Technology, USA, dkim@csail.mit.edu; Martin C. Rinard, Massachusetts Institute of Technology, USA, rinard@csail.mit.edu.

2475-1421/2020/1-ART52
https://doi.org/10.1145/3371120

from the chosen replacement class in the new library across multiple dimensions – internal data representation, signatures of the provided interfaces, or the underlying functionality offered by these interfaces. This motivates the need for a technique that can synthesize an adapter class for a given replacement class so that the synthesized class is *equivalent* to the existing class.

1.1 Automatic Class Replacement

We present a new technique and a system that, given an original class O and desired replacement class R, automatically synthesizes methods that implement O's interface using only class R. To perform this replacement, our system constructs an inter-class equivalence predicate $\sigma_{O,R}$ that defines equivalent states in O and R objects. For every public method including constructors defined by O, our approach synthesizes a new replacement method that invokes only public methods defined by R. Each replacement method provides observably identical functionality as the corresponding original method, including updating O and R objects to equivalent states and driving any other manipulated objects to identical states. Similarly, each replacement constructor constructs an instance of class R equivalent to the object instance constructed by the corresponding constructor in O. The result is the automatic synthesis of a new adapter class G that provides a drop-in replacement for O.

Obtaining effective equivalence predicates $\sigma_{O,R}$ is critical for the success of our technique. Methods that update object states can affect the operation of subsequently invoked methods (which may observe the updated state). Identifying equivalent object states and requiring synthesized methods to leave objects in equivalent states enables the technique to synthesize individual methods in isolation while still guaranteeing that sequences of method invocations deliver equivalent behavior regardless of the length of the sequence.

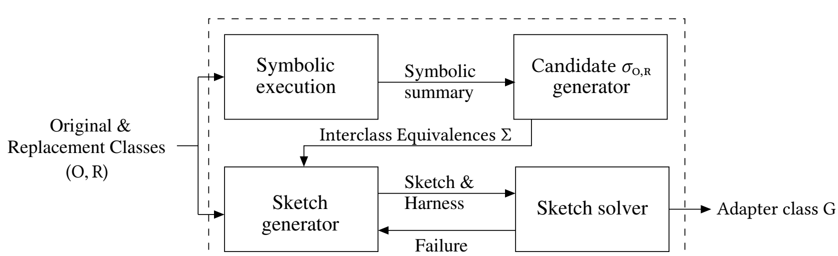

Fig. 1. Overall architecture of Mask.

1.2 Technique

Figure 1 presents the overall architecture of our tool, Mask, that implements our proposed solution. Broadly, Mask comprises two interleaved stages. The first stage generates a set of candidate equivalence predicates. The equivalence predicates are synthesized by symbolically executing [King 1976] *all* methods defined in classes O, R to identify a set of relevant symbolic expressions constructed by the classes and suitably equating these expressions to obtain the necessary predicates. The second stage starts with a candidate predicate and attempts to synthesize a replacement method for each method in the original class. Each synthesized method encapsulates a sequence of method invocations, where every method in the sequence is defined by class R.

Mask synthesizes replacement classes by generating and solving sketches that encode a search space of candidate method invocation sequences for each replacement method. For each replacement method, Mask generates an *adapter method sketch* (Figure 7), that consists of a sequence of conditionally invoked methods from the replacement class. The choice of whether to invoke a method or not is determined by choices that the Sketch solver [Solar-Lezama 2008] resolves when it solves a *synthesis correctness condition* implemented by a *Sketch method harness* (Figure 8). This method harness has the following structure:

- **Assumes:** Code that instructs the Sketch solver to assume the initial system states input to the original method and the adapter method sketch are equivalent, i.e., they satisfy the following, 1) all R objects, including the receiver R object, start in equivalent states as the corresponding O objects and 2) all other objects and primitive type parameters start in identical states.
- **Method Invocations:** Code that invokes the original method and the adapter method sketch on their respective equivalent input states.
- **Requires:** Code that requires the original method and adapter method sketch to produce equivalent final system states i.e, 1) leave all R objects in equivalent states as their corresponding O objects, 2) leave all other objects in identical states, and 3) return equivalent values to the caller.

Mask then invokes the Sketch solver on the method harness to resolve the choices in the adapter method sketch to satisfy the synthesis correctness condition encoded in the harness. The resolved choices identify a sequence of method invocations from the replacement class that together exhibit identical behavior as the original method. This sequence can then be used as a *correct* drop-in replacement implementation for the original method under *all contexts*.

If Mask succeeds in finding replacement methods for all original methods under the current candidate equivalence predicate, it terminates and returns the newly constructed replacement methods to the user. Otherwise, it repeats the process using the next candidate equivalence predicate.

Note that the soundness of the technique does *not* depend on any concept of the soundness of the candidate equivalence predicates. If the synthesis succeeds, all of the synthesized methods soundly preserve the current candidate equivalence predicate, which ensures the soundness of the technique. The only requirement is that, to avoid vacuously satisfying the assume/guarantee reasoning encoded in the method harness, each candidate equivalence predicate must be *satisfiable*. This desirable property enables Mask to use essentially *any* candidate equivalence predicate generation algorithm without sacrificing soundness (as long as the candidate equivalence predicates are filtered through a satisfiability checker, which Mask does).

1.3 Experimental Results

We have evaluated Mask using Java classes from open-source codebases (e.g., JDK [JDK 2019], apache [Apache Commons 2019]). We consider two scenarios: 1) replacing classes from one library with classes from another unrelated library and 2) replacing classes with upgraded versions from the same library. For the first scenario, our implementation is able to automatically and correctly generate replacements in a majority of the cases when replacements are feasible, including examples such as replacing `ArrayList` with `Vector`. For the second scenario, our implementation is able to synthesize the replacement classes except when defects in the original implementations were fixed in the updated version.

1.4 Contributions

This paper makes the following contributions:

- To the best of our knowledge, we are the first to address the problem of synthesizing an implementation to replace an original class O with a desired replacement class R.

- We propose a technique to identify equivalent object states to facilitate the synthesis of individual methods in isolation while still guaranteeing that any sequence of method invocations deliver equivalent behavior.
- We present a novel solution to synthesizing replacement methods by effectively integrating symbolic execution, constraint solving, and program synthesis.
- We describe the implementation of a tool, Mask, that incorporates our approach and evaluate its effectiveness by synthesizing adapter classes for closely related open source Java classes.

2 EXAMPLES

We illustrate using two examples how our system solves the problem of automatically replacing classes. The first example illustrates the need for a non-trivial equivalence predicate to enable synthesis of an adapter class. The second example demonstrates synthesis in the presence of subtle side effects to the system state.

2.1 Inter-class Equivalence

Figure 2 presents classes from the `eclipse` [Eclipse 2019] and `JMist`[2] code bases. Figure 2(a) presents the class `Box2` from `JMist` which is a representation for rectangles. It defines four private fields – `minX`, `maxX`, `minY`, and `maxY`. Each field captures the different edges of the rectangle. The class defines method `expand`, that constructs and returns a new instance of `Box2`, where each field of the new instance differs from the original instance, as specified by the input parameter `c`. This method does not modify the fields of the receiver.

Figure 2(b) presents the class `Rectangle` from the `eclipse` framework. This class implements the required functionality for a rectangle. Instead of representing the edges of the rectangle as in `Box2`, it declares four fields that together represent the corners of the rectangle — `x` and `y` represent the lower left corner of the rectangle; `width` and `height` represent the width and height of the rectangle.

To replace `Box2` with `Rectangle`, there must be a suitable replacement for `expand`. There is no implementation of `expand` in `Rectangle`. However, the implementation can be synthesized using the existing APIs in `Rectangle` as shown in Figure 2(c).

The synthesized implementation of `expand` initially creates a new instance of `Rectangle` that is identical to the current instance. Then, it invokes the `shrink` method on the created instance to expand its bounds by providing negative values to the `shrink` method. The modified instance of the rectangle is then returned to the client. This synthesized implementation is an effective replacement for the original `expand` under all contexts and leaves the overall system in an equivalent state. To verify the equivalence of the original implementation of `expand` in `Box2` and the synthesized implementation `expand`, the inter-class equivalence predicate needs to be available as given in Figure 2(d).

Our approach is able to automatically derive the inter-class equivalence predicate and synthesize the implementation of `expand` (and `Gen`). The class `Gen` is implemented using only class `Rectangle`. We show an example usage of `Box2` in a client and the refactored code obtained by replacing it with the synthesized drop-in replacement class `Gen` here.

```
Box2 b = new Box2(10,10,20,20);          Gen b = new Gen(10,10,20,20);
Box2 b2 = b.expand(10);                  Gen b2 = b.expand(10);
```

Similarly, our approach can also synthesize the implementations for other methods in `Box2`.

[2]JMIST: A research-oriented library for image synthesis https://github.com/bwkimmel/jmist

```
public final class Box2 {
  private int minX, maxX, minY, maxY;

  public Box2(int m1, int m2, int m3, int m4) {
    minX = m1; minY = m2; maxX = m3; maxY = m4;
  }
  private Box2 instance(int m1, int m2, int m3, int m4) {
    return new Box2(m1, m2, m3, m4);
  }
  ...
  public Box2 expand(int c) {
    return instance(minX - c, minY - c, maxX + c, maxY + c);
  }
}
```

(a) Simplified implementation of Box2.

```
public class Rectangle {
  public int x, y, width, height;
  public Rectangle(Rectangle r) {
    this(r.x, r.y, r.width, r.height);
  }
  public Rectangle(int v1, int v2, int w, int h) {
    x = v1; y = v2; width = w; height = h;
  }
  public int bottom() { return y + height; }
  public int right() { return x + width; }
  ...
  public Rectangle shrink(int h, int v) {
    x += h; width -= (h + h); y += v; height -= (v + v);
    return this;
  }
}
```

(b) Simplified implementation of Rectangle.

```
class Gen {
  Rectangle r;
  public Gen(Rectangle rect) {r = rect;}
  public Gen(int a, int b, int c, int d) {
    r = new Rectangle(a, b, c - a, d - b);
  }
  ...
  public Gen expand(int val) {
    Rectangle rect = new Rectangle(this.r);
    rect.shrink(-val, -val);
    return new Gen(rect);
  }
}
```

(c) Gen: Synthesized adapter class for Rectangle which is equivalent to Box2.

```
Box2.minX = Rectangle.x && Box2.maxX = Rectangle.x + Rectangle.width &&
Box2.minY = Rectangle.y && Box2.maxY = Rectangle.y + Rectangle.height
```

(d) Derived inter-class equivalence between Box2 and Rectangle.

Fig. 2. Motivating example: Box2 class from JMIST and Rectangle class from eclipse.

```
class ArrayList {
   int size; Objects [] elements;

   public  Object[] toArray(Object[] arr) {
     if (arr.length < size) return Arrays.copyOf(elements, size);
     System.arraycopy(elements, 0, arr, 0, size);
     if (arr.length > size) arr[size] = null;
     return arr;
   }
}
```

(a) Simplified implementation of ArrayList.

```
ArrayList.elements = FastArray.data && ArrayList.size = FastArray.size
```

(b) Inter-class equivalence of ArrayList and FastArray.

```
class FastArray {
  public int size; private Objects[] data;

  public FastArray(Object[] objects) { data = objects; size = objects.length;}

  public void set(int index, Object o) { data[index] = o;}
  public Object[] getArray() { return data; }

  public void addAll(Object[] array, int len) {
     if (len == 0) return;
     final int newSize = size + len;
     if (newSize > data.length) {
       Object nd[] = new Object [newSize];
       System.arraycopy(data, 0, nd, 0, size);
       data = nd;
     }
     System.arraycopy(array, 0, data, size, len);
     this.size = newSize;
  }
}
```

(c) Simplified implementation of FastArray.

```
class Gen {
  FastArray r;
  ...
  public Object[] toArray(Object[] arr) {
    FastArray fastarr = new FastArray(arr);
    fastarr.size = 0;
    try {
      fastarr.set(r.size(), null);
    } catch(Exception e) { }
    fastarr.addAll(r.getArray(), r.size());
    return fastarr.getArray();
  }
}
```

(d) Synthesized replacement class

Fig. 3. `ArrayList` class from JDK and `FastArray` class from Groovy.

2.2 Synthesizing Implementations

The utility class `ArrayList` from JDK [JDK 2019] offers a resizable array implementation, which can be used to store and retrieve objects. Other libraries also offer custom implementations of `ArrayList` that are fine-tuned for specific requirements. `FastArray` is an example of such an implementation offered by `Groovy` [Groovy 2019] that provides a subset of the APIs offered by the ArrayList class. Therefore, invocations to `ArrayList` can not be trivially replaced by invocations to `FastArray` using a one-to-one API replacement.

To explain the nuances of the problem, we examine the implementation of `toArray` in `ArrayList`. Figure 3(a) presents the implementation of `toArray`, which receives an array `arr` as a parameter. If the input array is greater than the `size` of the array in the receiver, the `toArray` method performs a shallow copy of the objects stored in the receiver into `arr` and returns `arr` to the client. Otherwise, the API allocates a new array of length stored in the field `size` and copies the objects into the newly allocated array before returning it to the client. We observe that after the array is returned to the client from the method, modifying the returned array has no impact on the array in the receiver instance and vice versa.

We now consider the problem of replacing `ArrayList` with `FastArray`. Even though `FastArray` is designed to store and retrieve objects, it does not implement a method that is equivalent to `toArray`. Figure 3(c) presents the implementation of `getArray` defined in `FastArray` which may appear to be a good candidate (based on naming) to replace `toArray`. Unfortunately, this replacement will be *incorrect* as this method simply returns a reference to field `data`, a private field maintained by the object. The client can modify the object state by updating the returned array leading to non-equivalent behavior.

The implementation of `toArray` can be synthesized using a carefully selected sequence of method invocations in `FastArray` as shown in Figure 3(d). Initially, a new `FastArray` instance `fastarr` is created by invoking the constructor, given in Figure 3(b), and passing `arr` as input parameter. The `data` field of the newly allocated instance `fastarr` now holds a reference to the input array `arr`. The constructor also initializes the `size` field to the length of `arr`. After creating `fastarr`, the `size` field of `fastarr` is reset to `zero` indicating there are not valid entries stored by `fastarr`.

Next, the `set` method is invoked to insert a `null` value. This is critical as it mimics the behavior of the second `if` branch in the original `toArray` method, which sets the last index of its corresponding array (`elements` field) to `null`. This invocation is surrounded by a `try-catch` block to catch the `ArrayIndexOutofBoundException` that may be thrown by this invocation. If an exception is thrown by this invocation, then the synthesized method will simulate the `else` branch from the original code thus ensuring equivalence. Next, `addAll` method is invoked to copy the data from `r.data` to `fastarr.data`. This operation will be successful if the array referenced by `fastarr.data` is large enough to store all the entries in `r.data`. Otherwise, the method will allocate a larger array to `fastarr.data` field and copy the data into it. This will leave the input array `arr` unmodified. Finally, the field `fastarr.data` is returned to the caller using `getArray` method.

By breaking the alias between input `arr` and `fastarr.data`, the `addAll` method invocation correctly captures the execution of the first `if` branch in the original method. Where, the original `toArray` implementation ignores the input array `arr` when it has insufficient space and instead allocates a new array to copy the data stored in the `ArrayList` instance. Thus the behavior of the synthesized implementation of `toArray` is equivalent to the original implementation of `toArray`, under all contexts. This also ensures that all relevant objects are driven to an identical state as the original `toArray` implementation in `ArrayList`. Our system is able to automatically synthesize this implementation.

3 PROBLEM FORMULATION

For a core object-oriented language, we formally describe the problem we address in this paper. The language has two kinds of *values*:

$$\begin{aligned} v &\in \text{Value} = \text{Int} \cup \text{Addr} \\ n &\in \ \text{Int} \\ a &\in \text{Addr} \end{aligned}$$

where Int is a set of integers and Addr is an address space of objects in a memory. An *environment* $\rho \in \text{Env}$ is a finite mapping from variables to values:

$$\rho \in \text{Env} = \text{Var} \xrightarrow{\text{fin}} \text{Value}$$

A *memory* μ receives an address of an object and its field name as its arguments, and returns a value stored in the object's field:

$$\begin{aligned} \mu &\in \text{Mem} = \text{Addr} \rightarrow \text{Field} \xrightarrow{\text{fin}} \text{Value} \\ f &\in \text{Field} \end{aligned}$$

We assume two functions class $\in (\text{Mem} \times \text{Addr}) \rightarrow \text{Class}$, which returns the class of an object at a given address, and fields $\in \text{Class} \rightarrow \mathcal{P}(\text{Field})$, which returns a set of fields of a class. A *system state* $s = \langle \rho, \mu \rangle \in \text{State}$ consists of the current environment ρ and memory μ:

$$s \in \text{State} = \text{Env} \times \text{Mem}$$

Value Equivalence. A value v_1 in a memory μ_1 is (deeply) equivalent to a value v_2 in a memory μ_2 if and only if

- v_1 and v_2 are the same integer value, or
- v_1 and v_2 are the addresses of two instances of the same class and the values of each of their fields are (deeply) equivalent:

$$\langle v_1, v_2 \rangle \in \mathsf{equiv}_{\mu_1,\mu_2} \equiv \begin{cases} v_1 = v_2 & \text{if } v_1, v_2 \in \text{Int} \\ \forall f \in \mathsf{fields}(C) : \langle \mu_1(v_1, f), \mu_2(v_2, f) \rangle \in \mathsf{equiv}_{\mu_1,\mu_2} & \\ & \text{if } \mathsf{class}(\mu_1, v_1) = \mathsf{class}(\mu_2, v_2) = C \end{cases}$$

State Equivalence. We now define equivalence between any two system states. We say two system states s_1, s_2 are equivalent, if the two states define the same set of variables and every variable in s_1 is equivalent to its mirror image in s_2. In addition, the two states maintain an identical aliasing relation between the variables and their field dereferences. Before we formally define equivalence, we define the alias function here:

$$\texttt{alias} \in \text{Mem} \times (\text{Var} \times \overrightarrow{\text{Field}}) \times (\text{Var} \times \overrightarrow{\text{Field}}) \rightarrow \text{Bool}$$

The `alias` function receives the system memory, two variables and their corresponding field dereferences as input. The function evaluates both the field dereferences under the memory and yields the corresponding memory addresses. If both the field dereferences yield the same address, then the function returns `true`; otherwise it returns `false`. We now formally define equivalence between system states.

Any two given states $s_1 = \langle \rho_1, \mu_1 \rangle$, $s_2 = \langle \rho_2, \mu_2 \rangle$ are equivalent (represented by $s_1 \equiv s_2$) if the states satisfy:

S.1 $\mathsf{Domain}(\rho_1) = \mathsf{Domain}(\rho_2) \land \forall r \in \mathsf{Domain}(\rho_1) : \langle \rho_1(r), \rho_2(r) \rangle \in \mathsf{equiv}_{\mu_1,\mu_2}$.

S.2 $\forall r_1, r_2 \in \mathsf{Domain}(\rho_1)\ \forall \overrightarrow{d_1}, \overrightarrow{d_2} \in \overrightarrow{\text{Field}} : (\texttt{alias}_{\mu_1}(r_1, \overrightarrow{d_1}, r_2, \overrightarrow{d_2}) \Leftrightarrow \texttt{alias}_{\mu_2}(r_1, \overrightarrow{d_1}, r_2, \overrightarrow{d_2}))$

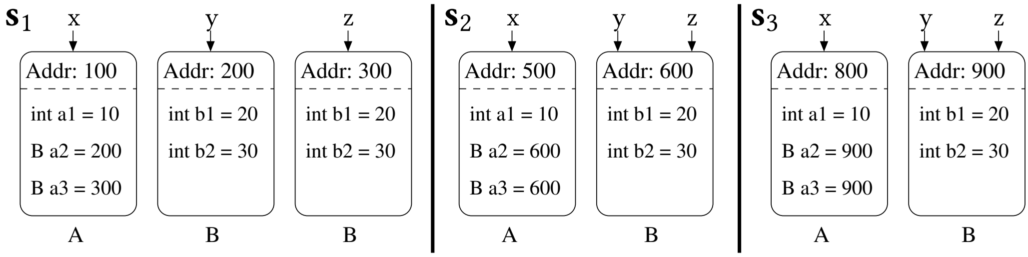

Fig. 4. System state equivalence: The states s_1 and s_2 are not equivalent ($s_1 \not\equiv s_2$) and the states s_2 and s_3 are equivalent ($s_2 \equiv s_3$)

We illustrate verifying equivalence between system states using Figure 4. The figure depicts three system states s_1, s_2 and s_3. All three states define the same set of variables `x`,`y` and `z`, i.e., $\text{Domain}(\rho_1) = \text{Domain}(\rho_2) = \text{Domain}(\rho_3)$. The variable `x` holds a reference to an object of type A and variables `y`, `z` hold references to objects of type B under all three states. Class A defines three fields: `a1`,`a2` and `a3`. Field `a1` stores an integer value and fields `a2`,`a3` hold reference to objects of type B. Class B defines two primitive fields: `b1` and `b2`.

The system states s_1 and s_2 satisfy check S.1, as the values stored by all three variables are equivalent under s_1 and s_2. For instance, for variable `x` this check translates to ensuring field dereferences `x.a1`, `x.a2.b1`, `x.a2.b2`, `x.a3.b1`, `x.a3.b2` all store the same integer value under both states. However, the states s_1 and s_2 do not satisfy check S.2, as `y` and `z` alias in s_2 but do not alias in s_1. Similarly, the pairs (`x.a2`,`z`) and (`x.a3`,`y`) alias only in s_2. Therefore the two states are not equivalent. On the other hand, the states s_2 and s_3 are equivalent as they satisfy both checks. In this case, all field dereferences of `x`,`y` and `z` yield the same value. In addition to this, the states maintain identical aliasing relations among all variables and their field dereferences.

Inter-Class Equivalence $\sigma_{O,R}$. Given two classes O and R, an *inter-class equivalence* $\sigma_{O,R} \in (\text{Addr} \times \text{Mem}) \times (\text{Addr} \times \text{Mem}) \rightarrow \text{Bool}$ is a function that determines whether an object of class O at an address can be considered to be equivalent to an object of class R at some other address, although they belong to different classes. If two values are equivalent up to an inter-class equivalence function $\sigma_{O,R}$, they are called *$\sigma_{O,R}$-equivalent.* More formally, a value v_1 in a memory μ_1 is *$\sigma_{O,R}$-equivalent* to a value v_2 in a memory μ_2 if and only if

- v_1 and v_2 are value equivalent, or
- the objects at the addresses v_1 and v_2 are of type O and R, respectively, and $\sigma_{O,R}(v_1, v_2)$ is true.

$$\langle v_1, v_2 \rangle \in \sigma_{O,R}\text{-equiv}_{\mu_1,\mu_2} \equiv$$
$$\langle v_1, v_2 \rangle \in \text{equiv}_{\mu_1,\mu_2} \vee (\text{class}(\mu_1, v_1) = \text{O} \wedge \text{class}(\mu_2, v_2) = \text{R} \wedge \sigma_{O,R}(v_1, \mu_1, v_2, \mu_2))$$

We also overload the definition of $\sigma_{O,R}$ as $\sigma_{O,R} \in \overrightarrow{\text{Field}} \rightarrow \mathcal{P}(\overrightarrow{\text{Field}})$. This function maps a valid field dereference of class O to set of valid field dereference of class R. For example, if $\sigma_{O,R}(\overrightarrow{d}) = \{\overrightarrow{d_1} \dots \overrightarrow{d_n}\}$, then the original $\sigma_{O,R}$ function has a check that asserts the field dereference $\overrightarrow{d}$ of class O instances is equated with the field dereferences $\overrightarrow{d_1} \dots \overrightarrow{d_n}$ of R instances.

State Equivalence under $\sigma_{O,R}$ [$s_O \equiv_{\sigma_{O,R}} s_R$]. Given the notion of inter-class equivalence $\sigma_{O,R}$, we can extend the definition of system state equivalence for system states s_O and s_R. We formally define equivalence for two states s_O, s_R, where s_O can contain instances of class O but cannot contain instances of R and s_R can contain instances of class R but not instances of class O.

Let $s_O = \langle \rho_O, \mu_O \rangle$, $s_R = \langle \rho_R, \mu_R \rangle$ be two system states where, $\forall v_o \in \text{Range}(\mu_O), \text{class}(\mu_O, v_o) \neq R$ and $\forall v_r \in \text{Range}(\mu_R), \text{class}(\mu_R, v_r) \neq O$. The two states s_O, s_R are equivalent under an inter-class equivalence predicate $\sigma_{O,R}$ (i.e., $s_O \equiv_{\sigma_{O,R}} s_R$), if they satisfy

IS.1 $\text{Domain}(\rho_O) = \text{Domain}(\rho_R) \wedge \forall r \in \text{Domain}(\rho_O) : \langle \rho_O(r), \rho_R(r) \rangle \in \sigma_{O,R}\text{-equiv}_{\mu_O,\mu_R}$.

IS.2 $\forall r_1, r_2 \in \text{Domain}(\rho_O)$ one of the following is true:

(a) $\text{class}(\mu_O, \rho_O(r_1)) \neq O \wedge \text{class}(\mu_O, \rho_O(r_2)) \neq O \wedge \forall \overrightarrow{d_1}, \overrightarrow{d_2} \in \overrightarrow{\text{Field}} :$

$$\texttt{alias}_{\mu_O}(r_1, \overrightarrow{d_1}, r_2, \overrightarrow{d_2}) \Leftrightarrow \texttt{alias}_{\mu_R}(r_1, \overrightarrow{d_1}, r_2, \overrightarrow{d_2})$$

(b) $\text{class}(\mu_O, \rho_O(r_1)) = O \wedge \text{class}(\mu_O, \rho_O(r_2)) \neq O \wedge \forall \overrightarrow{d_1}, \overrightarrow{d_2} \in \overrightarrow{\text{Field}}. \forall \overrightarrow{d_i} \in \sigma_{O,R}(\overrightarrow{d_1}) :$

$$\texttt{alias}_{\mu_O}(r_1, \overrightarrow{d_1}, r_2, \overrightarrow{d_2}) \Leftrightarrow \texttt{alias}_{\mu_R}(r_1, \overrightarrow{d_i}, r_2, \overrightarrow{d_2})$$

(c) $\text{class}(\mu_O, \rho_O(r_1)) = O \wedge \text{class}(\mu_O, \rho_O(r_2)) = O \wedge \forall \overrightarrow{d_1}, \overrightarrow{d_2} \in \overrightarrow{\text{Field}}. \forall (\overrightarrow{d_i}, \overrightarrow{d_j}) \in \sigma_{O,R}(\overrightarrow{d_1}) \times \sigma_{O,R}(\overrightarrow{d_2}) :$

$$\texttt{alias}_{\mu_O}(r_1, \overrightarrow{d_1}, r_2, \overrightarrow{d_2}) \Leftrightarrow \texttt{alias}_{\mu_R}(r_1, \overrightarrow{d_i}, r_2, \overrightarrow{d_j})$$

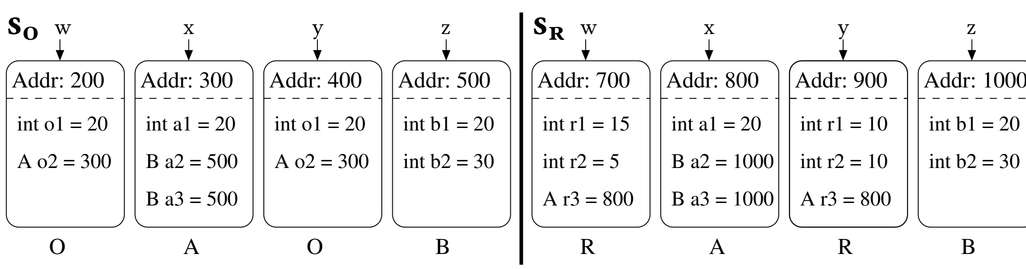

Fig. 5. System state equivalence under $\sigma_{O,R}$: If the equivalence predicate for the two classes O, R is such that, field o2 in class O is mapped to field r3 (i.e., $\sigma_{O,R}$(o2) = {r3}) and the predicate checks the following: $\sigma_{O,R}$(o1) = r1 +r2. Then under the predicate $\sigma_{O,R}$ the states s_O and s_R are equivalent (i.e., $s_O \equiv_{\sigma_{O,R}} s_R$).

We explain the working of the above checks using Figure 5. The figure presents two system states s_O and s_R. The state s_O contains instances of classes O, A, and B, whereas the state s_R contains instances of classes R, A, and B. The structure of classes A and B are the same as shown in Figure 4. The class O defines two fields: `o1` and `o2`. Field `o1` is primitive and it stores an integer value and field `o2` is a reference field of type A. The class R defines three fields: `r1`,`r2` and `r3`. Fields `r1`,`r2` are primitive and field `r3` holds a reference to an object of type A. Let $\sigma_{O,R}$ be defined as follows: $\sigma_{O,R}(v_1, \mu_1, v_2, \mu_2) \equiv \langle \mu_1(v_1, \texttt{o2}), \mu_2(v_2, \texttt{r3}) \rangle \in \text{equiv}_{\mu_1,\mu_2} \bigwedge \langle \mu_1(v_1, \texttt{o1}), \mu_2(v_2, \texttt{r1}) + \mu_2(v_2, \texttt{r2}) \rangle \in \text{equiv}_{\mu_1,\mu_2}$. The two states define the same set of variables: `w,x,y,z`.

We now elaborate on how all the variables `w,x,y,z` in states s_O and s_R are equivalent under $\sigma_{O,R}$ and satisfy the check IS.1. For variables `x` and `z`, the value equivalence is verified by checking $\langle \rho_O(\texttt{x}), \rho_R(\texttt{x}) \rangle \in \text{equiv}_{\mu_O,\mu_R}$ and $\langle \rho_O(\texttt{z}), \rho_R(\texttt{z}) \rangle \in \text{equiv}_{\mu_O,\mu_R}$ respectively. For variable `x`, this check boils down to verifying whether field dereferences `x.a1`, `x.a2.b1`, `x.a2.b2`, `x.a3.b1` and `x.a3.b2` yield the same integer value under both states. Similarly, for variable `z`, this ensures dereferences `z.b1` and `z.b2` yield the same integer values under both states. For variables `w` and `y`, the value equivalence is verified by checking $\sigma_{O,R}(\rho_O(\texttt{w}), \mu_O, \rho_R(\texttt{w}), \mu_R)$ and $\sigma_{O,R}(\rho_O(\texttt{y}), \mu_O, \rho_R(\texttt{y}), \mu_R)$ respectively. In other words, for variable `w`, this check ensures that value at `w.o1` in state s_O is equal to `w.r1` + `w.r2` in state s_R and that the object at `x.o2` in s_O is value equivalent to object at `x.r3` in s_R.

The two states are also equivalent under $\sigma_{O,R}$ as they satisfy check IS.2. We elaborate using three different pairs where each pair corresponds to a different check.

- *pair* (`x`,`z`) *under check IS.2(a)*: We consider all valid references reachable from `x` and `z` that may alias. For `x`, the references that are considered are {`x`, `x.a2`, `x.a3`}. For `z`, it is {`z`}. We check whether all reference pairs, (`x`,`z`), (`x.a2`, `z`) and (`x.a3`, `z`), either alias (or not) under both states. The pairs (`x.a2`, `z`) and (`x.a3`, zr_4) alias whereas the pair (`x`, `z`) do not alias under both the states. Hence, the pair satisfies the check.
- *pair* (`w`,`x`) *under check IS.2(b)*: Since there are many reference pairs that are feasible here, we explain using one reference pair (`w.o2`, `x`) under s_O. We identify the corresponding reference pair under s_R using $\sigma_{O,R}$. This yields the reference pair (`w.r3`,`x`). Since `w.o2` and `x` alias under s_O, and `x.r3` and `x` alias under s_R, the pair satisfies the check. Similarly, other references pairs that are derived for (`w`,`x`) satisfy the check.
- *pair* (`w`,`y`) *under check IS.2(c)*: We explain using one reference pair (`w.o2.a2`, `y.a3`) obtained under state s_O. The corresponding pair under s_R is (`w.r3.a2`, `y.r3.a3`) obtained using $\sigma_{O,R}$. The references `w.o2.a2` and `y.r3.a3` alias under s_O and references `w.r3.a2` and `y.r3.a3` alias under s_R, thereby, satisfying the check. Similarly, all other reference pairs that are derived from (`w`,`y`) satisfy the check.

Since the system states s_O and s_R satisfy the checks IS.1 and IS.2 under the given $\sigma_{O,R}$, the two states are equivalent under $\sigma_{O,R}$.

Goal. Given two different classes O and R, the goal of this work is to synthesize a new class G that contains a replacement method m_G for every method m_O of class O, such that every m_G provides an identical behavior as the corresponding method m_O, using the methods of class R only.

We realize this goal by synthesizing 1) a replacement class G that contains a reference to an instance of class R as its only field and 2) an inter-class equivalence predicate $\sigma_{O,G}$ (and, consequently, $\sigma_{O,R}$) that will be used to ensure the equivalence of the replacement class G with the original class O as follows:

> For all methods m_O in O, there exists a method m_G in G such that, if $s_O = \langle \rho_O, \mu_O \rangle$ and $s_G = \langle \rho_G, \mu_G \rangle$ are equivalent states under $\sigma_{O,G}$ (i.e., $s_O \equiv_{\sigma_{O,G}} s_G$) and if executing a statement $r = m_O(x_1 \ldots x_n)$ under s_O yields a state $s'_O = \langle \rho'_O, \mu'_O \rangle$ and executing $r = m_G(x_1 \ldots x_n)$ under s_G yields a state $s'_G = \langle \rho'_G, \mu'_G \rangle$, then
>
> - $\langle \rho'_O(r), \rho'_G(r) \rangle \in \sigma_{O,G}\text{-equiv}_{\mu'_O, \mu'_G}$. The values returned by the invocations (if any) are equivalent under $\sigma_{O,G}$.
> - $s'_O \equiv_{\sigma_{O,G}} s'_G$. The final states of the system are equivalent under $\sigma_{O,G}$.

In other words, if we execute a method m_O and m_G under two states s_O and s_G that are equivalent under $\sigma_{O,G}$, then the methods m_O and m_G return equivalent values to the user and drive the states s_O and s_G to equivalent states. Since this guarantee exists for every pair of m_O and m_G, replacing class O with class G is correct under every context.

4 DESIGN

We present the design of a tool, Mask, that achieves the goal described in the previous section. Algorithm 1 presents the overall working of Mask. The algorithm receives classes O, R and any known class invariants of O, R as input. If class invariants are not available, then Mask assumes a default `true` value for the class invariants. The algorithm symbolically analyzes classes O and R and constructs sets E_o, E_r populated with symbolic expressions built using field dereferences of classes O, R respectively. This is carried out by invoking the `symbolic-summary` procedure at line

Algorithm 1 Mask main algorithm

Require:
$O \leftarrow$ Original class; $R \leftarrow$ Replacement class
$\mathtt{I}_o \leftarrow$ Original class invariant; $\mathtt{I}_r \leftarrow$ Replacement class invariant ▷ Optional input
1: $\langle \mathsf{E}_o, \mathsf{E}_r \rangle \leftarrow$ `symbolic-summary`(O, R)
2: $\Sigma_{O,R} \leftarrow$ `candidate-generator`(E_o, E_r, $\mathtt{I}_o$, $\mathtt{I}_r$)
3: $G_s \leftarrow \emptyset$;
4: **for** every $m_o \in O$ **do** ▷ Generate sketch of class G
5: $\quad G_s \leftarrow G_s \oplus$ `generate-sketch`(m_o, R, k)
6: **end for**
7: **for** every $\sigma_{O,R} \in \Sigma_{O,R}$ **do**
8: $\quad$ `harness` $\leftarrow \emptyset$
9: $\quad$ **for** every $m_o \in O$ **do** ▷ Generate checks for all methods in G
10: $\quad\quad m_g \leftarrow$ `signature`(m_o, G) ▷ Signature of corresponding method in G
11: $\quad\quad$ `harness` $\leftarrow$ `harness` $\cup$ `generate-harness`(m_o, m_g, $\sigma_{o,r}$)
12: $\quad$ **end for**
13: $\quad G \leftarrow$ `sketch-solver`(G_s, `harness`) ▷ Resolve sketch using `class-harness`
14: $\quad$ **if** $G \neq \perp$ **then** return G;
15: $\quad$ **end if**
16: **end for**
17: return $\perp$

1. Next, the algorithm invokes the `candidate-generator` procedure, which receives the expressions in sets $\mathsf{E}o, \mathsf{E}_r$ and the class invariants $\mathtt{I}_o, \mathtt{I}_c$ as input. Leveraging the input expressions, the `candidate-generator` constructs a set of possible inter-class equivalence predicates $\Sigma_{O,R}$. Every predicate in set $\Sigma_{O,R}$ is satisfiable and relates every instance of class O to at least one instance of class R.

Once the candidate set of interclass equivalence definitions are constructed, the algorithm attempts to synthesize the required adapter class G. The algorithm begins the synthesis by constructing an overall sketch G_s of class G (lines 4-6). It adds a method sketch to G_s for every public method m_o in class O using procedure `generate-sketch`. The adapter method sketch is later resolved into a unique method invocation sequence to class R, which is at most k long, where k is specified by the user.

After creating the sketch G_s for class G, Mask creates the necessary harness to resolve it. The algorithm iterates over the set $\Sigma_{O,R}$ returned by `candidate-generator` procedure and generates a harness to resolve G_s in each iteration (line 7-16). The harness constructed in each iteration leverages a different interclass equivalence $\sigma_{O,R}$ in set $\Sigma_{O,R}$. The harness for every method m_g that must be synthesized for class G is constructed using `generate-harness` procedure and it encodes the correctness check for the specific method (line 11). After the harness is created for all methods in G_s, the Mask invokes the `sketch-solver` (line 13). If the `sketch-solver` is able to resolve the adapter class sketch G_s by satisfying the harness, it produces a completed adapter class G as output. The constructed class is then returned to the user at line 14. Otherwise, the algorithm attempts to instantiate the G_s using another harness in the next iteration. This is carried out until all the interclass equivalence predicates are considered for synthesis or the required class G is found. We now elaborate each of the above steps in detail.

```
public class Original {
  int x, y, count;
  public Original () {
    x = 0; y = 0; count = 0;
  }

  public void set(int v1, int v2) {
    x = v1; y = v2; count++;
  }

  public int sum() {return x+y;}
  public int diff() {return x–y;}

  public void moveX(int val) {
    x = x + val; count++;
  }

  public void moveY(int val) {
    y = y + val; count++;
  }

  public void scale(int val) {
   moveX(val); moveY(val);
   count++;
  }
}
```

(a)

```
public class Replacement {
  int a, b;
  public Replacement () {
    a = 0; b = 0;
  }

  public int getValue(boolean flag) {
    if (flag) return a;
    return b;
  }

  public void add(boolean flag, int val) {
    if (flag) a = a + val;
    else b = b + val;
  }

  public int subtract(boolean flag, int val) {
    if (flag) { a = a – val; return a;}
    else { b = b – val; return b; }
  }

  public void reset() {
    a = 0; b = 0;
  }

}
```

(b)

```
public class Generated {
  Replacement r;
  public Generated () {
    r = new Replacement();
  }

  public void set(int v1, int v2) {
     r.reset(); r.add(true, v1);
     r.add(true, v2); r.add(false, v1);
     r.subtract(false, v2);
  }

  public int sum() { return r.getValue(true);}
  public int diff() { return r.getValue(false);}

  public void moveX(int val) {
     r.add(true, val); r.add(false, val);
  }

  public void moveY(int val) {
     r.add(true, val); r.subtract(false, val);
  }

  public void scale(int val) {
     r.add(true, val); r.add(true, val);
  }
}
```

(c)

Fig. 6. Running example.

4.1 Illustrative Example

For clarity, we illustrate the working of our approach with the help of a simple running example, that will be used to explain the various aspects of our approach throughout this section.

Figure 6(a) presents a simple class `Original` which needs to be replaced with another class `Replacement` given in Figure 6(b). The classes differ from each other across multiple dimensions:

- `Original` contains three private fields `x`, `y`, and `count`, while `Replacement` declares two fields, `a` and `b`.
- `Original` provides 6 public APIs as compared to the 4 APIs present in `Replacement`.
- The APIs have distinct signatures with respect to the number of parameters, the types of the parameters, etc.

Given the two classes, Mask can generate a replacement class `Generated` as shown in Figure 6(c) where the interfaces of `Original` are implemented using the `Replacement` class. Each method in `Generated` performs a sequence of invocations to `Replacement`.

4.2 Symbolic Summary Generation

As a first step, Mask explores the two input classes using symbolic execution. The analysis builds the set of symbolic expressions returned by invoking various public methods defined by the class. This set will be later used to build the required $\sigma_{\mathrm{O,R}}$ checks for input classes O, R.

Algorithm 2 presents the generation of symbolic summaries where classes O and R form the input. The procedure iterates over all public methods, that have a non-void return type, defined in the class O (lines 3-7). For a given method, the procedure assigns fresh a symbolic variable to each parameter (line 4). If a parameter is of reference type, it assigns a unique symbol to every

Algorithm 2 The `symbolic-summary` procedure

```
1: procedure symbolic-summary(O, R)
2:     E_o ← ∅; E_r ← ∅
3:     for method m_i ∈ O do                        ▹ skip methods that have a void return type
4:         P ← parameters(m_i); S ← create-symbols(P)
5:         E_i ← symbolic-explore(m_i, S)
6:         E_o ← E_o ∪ filter(E_i, O)
7:     end for
8:     for method m_i ∈ R do                        ▹ skip methods that have a void return type
9:         P ← parameters(m_i); S ← create-symbols(P)
10:        E_i ← symbolic-explore(m_i, S)
11:        E_r ← E_r ∪ filter(E_i, R)
12:    end for
13:    E_o ← E_o ∪ field-dereferences(O)       ▹ Additional expression for stronger predicates
14:    E_r ← E_r ∪ field-dereferences(R)
15:    return ⟨ E_o, E_r ⟩
16: end procedure
```

field dereference of the parameter. This is carried out to precisely track the data flow from object fields to the return values.

The method is then executed symbolically using the generated symbolic parameters. The symbolic execution is inter-procedural and explores all paths in the method. If a path returns a value to the caller, the symbolic expression associated with the return value is added to the set E_i, at line 5. Next, the procedure filters the set of expressions contained in the set E_i at line 6. The `filter` function retains symbolic expressions in E_i that are purely constructed using symbolic variables initially assigned to field dereferences of objects of type O. This is to enable Mask to use these symbolic expressions to build candidate $\sigma_{\text{O,R}}$ checks. More specifically, we require the evaluation $\sigma_{\text{O,R}}(v_o, \mu_o, v_r, \mu_r)$ check to only depend on the state of object of type O at v_o and object of type R at v_r, and not contain additional free variables. The same process is repeated for class R to build the corresponding set E_r (lines 8-12).

The procedure next explores classes O and R to derive the set of valid field dereferences which are then added to the sets E_o and E_r respectively. This is carried out to widen the space of candidate $\sigma_{\text{O,R}}$ predicates considered by Mask. For some classes, predicates that only leverage symbolic return expressions may not be strong enough and in such cases these additional predicates may become essential for synthesizing the required G.

For the running example given in Figure 6, `Original` defines two public methods that return a value to its caller – `sum` and `diff`. These methods are symbolically explored by `symbolic-summary` procedure. Both methods are invoked on an instance of the class `Original` and receive no additional parameters. The procedure assigns symbolic variables to field dereference of the receiver objects and assigns the symbols x_s, y_s and count_s for the fields `x`, `y` and `count` respectively. The method `sum` returns x_s+y_s and `diff` returns x_s-y_s symbolic expressions. Since, the symbolic expressions are composed only of symbolic variables drawn from the fields in `Original`, they remain intact and are not removed by the `filter` method at line 5 resulting in $\mathsf{E}_o = \{ \mathsf{x}_s+\mathsf{y}_s, \mathsf{x}_s-\mathsf{y}_s \}$. Symbolic expressions are not generated for the remaining methods in `Original` as they do not return a value to the client.

Next, the procedure explores the two public methods that have a non-void return type, `getValue` and `subtract`, in the class `Replacement`. The procedure creates the symbols a_s, b_s, flag_s and val_s

corresponding to object fields `a`, `b` of receiver object and input parameters `flag` and `val` of method `subtract`. After symbolically executing `subtract`, two symbolic expressions that are returned are identified: $\mathtt{a}_s$-$\mathtt{val}_s$ and $\mathtt{b}_s$-$\mathtt{val}_s$. However, both the expressions are filtered out by `filter` method at line 11. This is because the expressions contain a free variable $\mathtt{val}_s$ which cannot be used in a candidate check. Then, the procedure symbolically analyses `getValue` method which results in $\mathtt{E}_r$ = { $\mathtt{a}_s$, $\mathtt{b}_s$}. Subsequently, the procedure adds field dereferences $\mathtt{x}_s$, $\mathtt{y}_s$ and $\mathtt{count}_s$ to $\mathtt{E}_o$. When the procedure concludes, $\mathtt{E}_o$ = {$\mathtt{x}_s$+$\mathtt{y}_s$, $\mathtt{x}_s$-$\mathtt{y}_s$, $\mathtt{x}_s$, $\mathtt{y}_s$, $\mathtt{count}_s$} and $\mathtt{E}_r$ = {$\mathtt{a}_s$, $\mathtt{b}_s$}.

4.3 Equivalence Candidate Generator

We now describe the *candidate generator* which builds the set of $\sigma_{\mathrm{O,R}}$ candidates.

4.3.1 Generating Possible Inter-class Equivalence Predicates. In the previous stage, Mask constructed sets $\mathtt{E}_o$ and $\mathtt{E}_r$ that contain expressions built with field dereferences of classes O and R. Using these expressions, the candidate generator synthesizes the required inter-class equivalence predicate by establishing a relation between the field dereferences of classes O and R. This is carried out by equating expressions in set $\mathtt{E}_o$ with expressions in set $\mathtt{E}_r$.

While building these predicates, higher importance is given to expressions in $\mathtt{E}_o$ that are added by the symbolic execution of public methods. As these expressions are returned to the client application, the synthesized adapter class G must be able to return equivalent expressions to the client. Therefore, we require that each such expression in $\mathtt{E}_o$ is equivalent to at-least one expression in set $\mathtt{E}_r$. We define function `return` to extract these expressions.

We attempt to construct a relation $\eta \subseteq \mathtt{E}_o \times \mathtt{E}_r$, where $\forall e_o \in \mathtt{return}(\mathtt{E}_o)\ .\exists e_r \in \mathtt{E}_r$ s.t. $(e_o, e_r) \in \eta$. Every such relation can be regarded as a potential $\sigma_{\mathrm{O,R}}$:

$$\sigma_{\mathrm{O,R}} \equiv \bigwedge_{(e_o, e_r) \in \eta} (e_o = e_r)$$

However, all possible relations need not be meaningful. For instance, a relation that pairs two expressions of different types will not produce a useful equivalence predicate $\sigma_{\mathrm{O,R}}$. Therefore, we choose only those mappings that are type compatible as potential candidates. Further, we must only consider those predicates that offer a valid mapping for every object of class O. We now discuss these criterion in detail.

4.3.2 Eliminating Invalid Predicates. Intuitively, we want every object instance of type O to be represented (and eventually replaced) by some object instance of class R. If such an object mapping is not possible, then O can not be replaced by R as it can not offer equivalent functionality under all contexts. Any $\sigma_{\mathrm{O,R}}$ predicate that we consider must ensure that this requirement is met. In this subsection, we elaborate on how Mask identifies such *valid* $\sigma_{\mathrm{O,R}}$ predicates.

We use class invariants to identify all possible object instances of a given class. We define a class invariant checker, I_c, that returns true, if and only if, an input value is an instance of type C and satisfies the class invariant.

$$\mathrm{I}_c \in \mathrm{Inv} = \mathrm{Value} \times \mathrm{Mem} \rightarrow \mathrm{Bool}$$

An inter-class equivalence predicate $\sigma_{\mathrm{O,R}}$ for any two classes O, R is considered valid, if it satisfies the following check given $\mathrm{I_O}$ and $\mathrm{I_R}$:

$$\forall \langle v_o, \mu_o \rangle \in \mathrm{Value} \times \mathrm{Mem}.\ \exists \langle v_r, \mu_r \rangle \in \mathrm{Value} \times \mathrm{Mem}.$$
$$\mathrm{I_O}(v_o, \mu_o) = \mathtt{false}\ \vee\ (\mathrm{I_R}(v_r, \mu_r) = \mathtt{true} \wedge \sigma_{\mathrm{O,R}}(v_o, \mu_o, v_r, \mu_r) = \mathtt{true})$$

In other words, if we can find an instance of class O which causes the $\sigma_{\mathrm{O,R}}$ check to be unsatisfiable, then $\sigma_{\mathrm{O,R}}$ is invalid. We define a function `CheckValid` that performs this validity check, given a candidate predicate and two class invariant checkers. If the candidate fails the check, the function returns false and returns true otherwise.

$$\texttt{CheckValid} = \Sigma \times \text{Inv} \times \text{Inv} \rightarrow \text{Bool}$$

Algorithm 3 presents the working of the CheckValid function which is used to eliminate the invalid $\sigma_{O,R}$ candidates. It takes as input a $\sigma_{O,R}$ predicate and class invariant checkers, I_o and I_r, in the form of constraints. The implementation of the procedure uses the minimal satisfiability theory [Dillig et al. 2012].

Algorithm 3 The `CheckValid` procedure

1: **procedure** `CheckValid`($\sigma_{O,R}$, I_o, I_r)
2: $\quad \mathtt{D}_o \leftarrow$ `field-dereferences(O)`; $\mathtt{D}_r \leftarrow$ `field-dereferences(R)`
3: $\quad$ `limit` $\leftarrow |\mathtt{D}_o| + 1$; `cost` $\leftarrow$ `empty-map`
4: $\quad$ **for** $e_o \in \mathtt{D}_o$ **do** `cost`$[e_o] \leftarrow 1$ **end for** ▷ Assign suitable cost for O and R dereferences
5: $\quad$ **for** $e_r \in \mathtt{D}_r$ **do** `cost`$[e_r] \leftarrow$ `limit` **end for**
6: $\quad$ `mincost` $\leftarrow$ `MSA-cost`($I_o \wedge \neg(\sigma_{O,R} \wedge I_r)$, `mincost`) ▷ Minimal cost for satisfying constraint.
7: $\quad$ **if** `mincost` < `limit` **then** return `false` ▷ Found an O instance without a valid mapping
8: $\quad$ **else** return `true`
9: $\quad$ **end if**
10: **end procedure**

A minimal satisfying assignment (MSA) is defined for a input constraint α under a cost function C, where C specifies the cost of every free variable in α. A satisfying assignment is a value mapping for a set of variables in α, such that α is satisfied. The cost of the assignment is the sum of the costs associated with every variable that is assigned a value. Then, the minimal satisfying assignment is a satisfying assignment that incurs the minimal cost.

The CheckValid procedure begins by building sets $\mathtt{D}_o$ and $\mathtt{D}_r$ that contain all field dereferences of classes O and R respectively (line 2). The idea is to only assign values to variables in set $\mathtt{D}_o$ such that $\mathtt{I}_o \wedge \neg(\sigma_{O,R} \wedge \mathtt{I}_r)$ is satisfied. To do this, we create a unique cost function for variables in $\mathtt{D}_o$ and $\mathtt{D}_r$. Each variable in $\mathtt{D}_o$ is assigned cost 1 and each variable in $\mathtt{D}_r$ is assigned cost $|\mathtt{D}_o| + 1$. Therefore, the cost of assigning a value to a variable in $\mathtt{D}_r$ is higher than the total cost of assigning values to all variables in $\mathtt{D}_o$. Subsequently, CheckValid computes the MSA cost of constraint $\mathtt{I}_o \wedge \neg(\sigma_{O,R} \wedge \mathtt{I}_r)$. If the cost is less than $|\mathtt{D}_o| + 1$, then only variables in $\mathtt{D}_o$ are assigned values. With this value assignment, we now have an instance of class O that has no mapping in $\sigma_{O,R}$, making it an invalid candidate. If the cost is higher, the solver had to assign values to variables in $\mathtt{D}_r$. This means that there is no object instance of O without a mapping under $\sigma_{O,R}$, making it a valid candidate.

4.3.3 Candidate Generator Procedure. We now present `candidate-geneartor` procedure that generates possible $\sigma_{O,R}$ candidates in Algorithm 4. The procedure receives the sets of symbolic expressions, $\mathtt{E}_o$ and $\mathtt{E}_r$, as input. It returns a valid set of $\sigma_{O,R}$ candidates that appropriately map expressions in $\mathtt{E}_o$ with those in $\mathtt{E}_r$. The procedure is recursive and it employs divide and conquer strategy to build the candidate predicates, where simpler predicates are built before building more complex ones.

The procedure begins by initializing an empty set of predicates at line 2. If $\mathtt{E}_o$ contains only a single expression e_o, then the procedure executes the base case between lines 3-9. The procedure iterates over every expression $e_r \in \mathtt{E}_r$ that is of the same type as e_o. For every such expression e_r, the procedure creates a simple constraint that maps expression e_o to e_r (i.e., $e_o = e_r$). The validity

Algorithm 4 The candidate-generator procedure

1: **procedure** candidate-generator(E_o, E_r)
2: $\Sigma \leftarrow \emptyset$ ▷ Initialize $\sigma_{O,R}$ candidate set
3: **if** $|E_o| = 1$ **then** $e_o \leftarrow$ element(E_o)
4: **for** every $e_r \in$ type(E_r, e_o) **do** ▷ e_r and e_o are of compatible types
5: **if** CheckValid($(e_o = e_r), I_o, I_r$) **then** $\Sigma \leftarrow \Sigma \cup \{(e_o = e_r)\}$
6: **end if**
7: **end for**
8: **return** Σ;
9: **end if**
10: $(E_i, E_j) \leftarrow$ divide-set(E_o) ▷ Split the expressions
11: $\Sigma_i \leftarrow$ candidate-generator(E_i, E_r); ▷ Generate partial $\sigma_{O,R}$
12: $\Sigma_j \leftarrow$ candidate-generator(E_j, E_r)
13: **for** every $(\sigma_i, \sigma_j) \in \Sigma_i \times \Sigma_j$ **do**
14: **if** CheckValid($\sigma_i \wedge \sigma_j, I_o, I_r$) **then** $\Sigma \leftarrow \Sigma \cup \{\sigma_i \wedge \sigma_j\}$ **end if**
15: **end for**
16: **if** return(E_o)$\cap$ E_i $= \emptyset$ **then** $\Sigma \leftarrow \Sigma \cup \Sigma_j$ **end if**
17: **if** return(E_o)$\cap$ E_j $= \emptyset$ **then** $\Sigma \leftarrow \Sigma \cup \Sigma_i$ **end if**
18: **return** Σ
19: **end procedure**

of the newly built predicate $(e_o = e_r)$ is checked by the CheckValid procedure at line 5. All valid predicates are added to Σ at line 5. The base case terminates by returning the populated Σ.

If the size of E_o is more than one, then the `candidate-generator` procedure builds the candidate predicates recursively using a divide and conquer strategy (lines 10-18). It divides the set E_o into E_i and E_j at line 10 and invokes the procedure on these divided sets (lines 11-12). The candidate predicates built by these recursive invocations are captured in Σ_i and Σ_j respectively.

Using the predicates in Σ_i and Σ_j, the procedure builds stronger predicates that map expressions in set E_o with those in E_r by considering all pairs of predicates (σ_i, σ_j), where $\sigma_i \in \Sigma_i$ and $\sigma_j \in \Sigma_j$. The validity of $\sigma_i \wedge \sigma_j$ is checked using CheckValid procedure. If it is valid, then $\sigma_i \wedge \sigma_j$ is added to Σ. After evaluating all pairs, the procedure checks if the predicates in Σ_i and Σ_j can be added to Σ.

Recall that we require every symbolic expression returned by a method in O be mapped to some expression in E_r. Therefore, the predicates in set Σ_j can be added to Σ provided the set E_i does not contain any symbolic expression returned by method of class O (line 16). Similarly, predicates in set Σ_i are added (line 17).

We now explain the `candidate-generator` procedure using the running example. Initially, the procedure is invoked with sets $E_o = \{x_s + y_s, x_s - y_s, x_s, y_s, count_s\}$ and $E_r = \{a_s, b_s\}$ as input. As there are no class invariants for this example, the constraints I_o and I_r are `true`. Since the set E_o contains more than one entry, the procedure builds the predicates recursively.

Table 1 presents the details for all recursive calls with E_o as input, where each row presents the data corresponding to an invocation. The details pertaining to the first invocation of the procedure is presented in the last row. The procedure makes a recursive invocations with $E_i = \{x_s{+}y_s, x_s{-}y_s\}$ and $E_j = \{x_s, y_s, count_s\}$ as inputs respectively. The invocations return predicate sets $\Sigma_6 = \{\sigma_1 \wedge \sigma_4, \sigma_2 \wedge \sigma_3\}$ and $\Sigma_8 = \{\sigma_5 \wedge \sigma_{10}, \sigma_6 \wedge \sigma_9, \ldots\}$. The definitions of the constructed predicates $\sigma_1 \ldots \sigma_{10}$ are shown in Table 1 and are self-explanatory.

Table 1. The first column presents the recursive depth of the candidate-generator instance processing the input E_o which is shown in the second column. The third column indicates the Σ_i and Σ_j sets constructed by the recursive calls. The fourth column presents the Σ built by this invocation and the fifth column presents the number of candidates that are found to be invalid by this invocation.

D	E_o	Σ_i, Σ_j	Constructed Σ	|Invalid|
4	$\{x_s+y_s\}$	$\varnothing$	**[2]** $\Sigma_1 = \{\sigma_1 : x_s+y_s=a_s, \sigma_2 : x_s+y_s=b_s\}$	0
4	$\{x_s-y_s\}$		**[2]** $\Sigma_2 = \{\sigma_3 : x_s-y_s=a_s, \sigma_4 : x_s-y_s=b_s\}$	0
4	$\{x_s\}$		**[2]** $\Sigma_3 = \{\sigma_5 : x_s=a_s, \sigma_6 : x_s=b_s\}$	0
4	$\{y_s\}$		**[2]** $\Sigma_4 = \{\sigma_7 : y_s=a_s, \sigma_8 : y_s=b_s\}$	0
3	$\{count_s\}$		**[2]** $\Sigma_5 = \{\sigma_9 : count_s=a_s, \sigma_{10} : count_s=b_s\}$	0
3	$\{x_s, y_s\}$	Σ_3, Σ_4	**[6]** $\Sigma_7 = \Sigma_3 \cup \Sigma_4 \cup \{\sigma_5 \wedge \sigma_8, \sigma_6 \wedge \sigma_7\}$	2
2	$\{x_s+y_s, x_s-y_s\}$	Σ_1, Σ_2	**[2]** $\Sigma_6 = \{\sigma_1 \wedge \sigma_4, \sigma_2 \wedge \sigma_3\}$	6
2	$\{x_s, y_s, count_s\}$	Σ_5, Σ_7	**[12]** $\Sigma_8 = \Sigma_7 \cup \Sigma_5 \cup \{\sigma_5 \wedge \sigma_{10}, \sigma_6 \wedge \sigma_9, \sigma_7 \wedge \sigma_{10}, \sigma_8 \wedge \sigma_9\}$	8
1	$\{x_s+y_s, x_s-y_s$ $x_s, y_s, count_s\}$	Σ_6, Σ_8	**[2]** $\Sigma_9 = \Sigma_6$	36

For every pair of predicates (σ_i, σ_j), where $\sigma_i \in \Sigma_6$ and $\sigma_j \in \Sigma_8$, the procedure checks whether $\sigma_i \wedge \sigma_j$ is valid. Let us consider one such case, where σ_i: $x_s+y_s=a_s \wedge x_s-y_s=b_s$ and σ_j: $x_s=a_s \wedge count_s=b_s$. The conjunction of these predicates is input to the CheckValid procedure to check its validity which reports the predicate as invalid. This is because when $x_s=5$ and $y_s=5$, $\sigma_i \wedge \sigma_j$ is unsatisfiable. Similarly, CheckValid evaluates every such $\sigma_i \wedge \sigma_j$ as invalid. Next the procedure checks if set `return(E_o)∩E_1` is empty. Since the expressions x_s+y_s, x_s-y_s in set E_1 are returned by methods in class `Original`, the resulting set is non empty. Hence, candidates in Σ_8 are discarded. However, the check to add elements in Σ_6 succeeds and therefore the procedure returns predicates $x_s+y_s=a_s \wedge x_s-y_s=b_s, x_s+y_s=b_s \wedge x_s-y_s=a_s\}$.

4.4 Sketch Generator

In this section, we describe the working of the *sketch generator*. It receives the two classes O and R, a maximum sequence length k and the set Σ returned by the `candidate-generator` as input. It constructs an overall *sketch* of the adapter class and a class harness to resolve it. The sketch for class G is a set of smaller sketches corresponding to its methods – { $sketch_1 \dots sketch_n$}. For every $sketch_m$, there exists a unique method harness in the class harness and the method sketch can be independently resolved by the `Sketch` solver [Solar-Lezama 2008] using its harness. We now explain the process of generating a method sketch and the corresponding method harness.

4.4.1 Generating a Sketch. A sketch $sketch_m$ is constructed to replace every method $m \in$ O using the `generate-sketch` procedure presented in Algorithm 5. The procedure receives the method m and the replacement class R as input. It iterates over every formal parameter defined by m and identifies the corresponding parameter type to be input to the sketched method (line 3-8). If method m receives a parameter of type O, then the sketched method will receive a parameter of type G. Otherwise, it receives a parameter of the same type. Using the established parameter types to the sketched method and the input method m, the procedure creates the method signature and adds it to $sketch_m$ at line 8.

Next, the procedure creates a sketch for the method body. It constructs a sketch that allows *at-most* k method invocations to class R (line 9-14). This is done iteratively and each iteration adds a choice to select at-most one method in R to invoke. Each iteration identifies the set of methods from

Algorithm 5 The `generate-sketch` procedure

1: **procedure** generate-sketch(m, R, k)
2:     parameters $\leftarrow \emptyset$; sketch$_m$ $\leftarrow \emptyset$
3:     **for** every $\langle t_i, p_i \rangle \in$ formal-parameters(m) **do**   ▷ t_i is the type of parameter p_i
4:         **if** t_i = O **then** parameters $\leftarrow$ parameters $\cup \{\langle G, p_i \rangle\}$
5:         **else** parameters $\leftarrow$ parameters $\cup \{\langle t_i, p_i \rangle\}$
6:         **end if**
7:     **end for**
8:     sketch$_m$ $\leftarrow$ signature(m, parameters)
9:     **while** $|$sketch$_m| < k$ **do**   ▷ k is set by the user
10:         M$_i$ $\leftarrow$ enabled-methods(R, parameters)
11:         sketch$_m$ $\leftarrow$ sketch$_m$ $\oplus$ create-choice(M$_i$, parameters)
12:         parameters $\leftarrow$ parameters $\cup$ new-values(M$_i$)
13:     **end while**
14:     sketch$_m$ $\leftarrow$ sketch$_m$ $\oplus$ ret-choice(parameters, m)
15:     **return** sketch$_m$
16: **end procedure**

R that can be invoked using only the input parameters to the sketched method and the values that maybe returned by the previous invocations using function `enabled-methods` (line 10). Using this set of methods and all the currently available values (stored in `parameters`), the procedure creates a new choice and adds it to sketch$_m$ at line 11. Subsequently, the set `parameters` is updated with any values that may be returned by the newly added choice. After adding k choices, the procedure adds a final choice to return a suitable value to the caller, provided the original method m also returns a value.

```
public void scale(int val) {
  int choice1=??; int choice2=??;
  int v1=??, ret1=0;
  if(choice1 == 1) r.getValue({|true,false|});
  else if(choice1 == 2) r.add({|true,false|}, {|val,v1|});
  else if(choice1 == 3) ret1 = r.subtract({|true,false|}, {|val,v1|});
  else if(choice1 == 4) r.reset();
  else do_nothing

  int v2 =??; int ret2 = 0;
  if(choice2 == 1) r.getValue({|true,false|});
  else if(choice2 == 2) r.add({|true,false|}, {|val,v2,r1|});
  else if(choice2 == 3) ret2 = r.subtract({|true,false|}, {|val,v2,ret1|});
  else if(choice2 == 4) r.reset();
  else do_nothing

  return;
}
```

Fig. 7. The generated sketch for method scale

We illustrate the construction of a sketched method for the running example given in Figure 6. Let us consider generating a sketch for method `scale` with k set to 2. The final sketched method is shown in Figure 7. Here, `choice1` and `choice2` are free variables that are to be resolved by the Sketch solver to concretize the methods that need to be invoked. The receiver for these invocations is field `r` of type `Replacement` defined by the adapter class. The generated sketch also provides a choice to select appropriate input parameters to the method invocation. For example, under `choice1`, the `add` method can accept `true` or `false` value for the first parameter. For the second parameter, it can choose between the input parameter `val` to the sketched method `scale`, or a constant value assigned to `v1` by the Sketch solver. Further, under `choice2`, the second parameter to `add` can also accept `ret1` which stores the value returned by `subtract` under `choice1`.

Algorithm 6 The method harness generator

procedure generate-harness(m_o, m_g, $\sigma_{O,R}$)
  $P_o \leftarrow$ formal-parameters(m_o) ▷ Build states s_O, s_G
  $(s_O = \langle \rho_O, \mu_O \rangle, s_G = \langle \rho_G, \mu_G \rangle) \leftarrow$ create-states(P_o, O, R)
  assume $\bigwedge_{i \in (1 \ldots n)} (\rho_O(p_i), \rho_G(q_i)) \in \sigma_{O,R}\text{-equiv}_{\mu_O, \mu_G}$ ▷ Assume state equivalence under $\sigma_{O,R}$
  for every $(i, j) \in [1 \ldots n] \times [1 \ldots n]$ **do**
    for every $(\overrightarrow{d_1}, \overrightarrow{d_2}) \in \text{deref}(p_i) \times \text{deref}(p_j)$ **do**
      $S_1 \leftarrow \{\overrightarrow{d_1}\}$; $S_2 \leftarrow \{\overrightarrow{d_2}\}$
      if type(p_i) = O **then** $S_1 \leftarrow \sigma_{O,R}(\overrightarrow{d_1})$ **end if**
      if type(p_j) = O **then** $S_2 \leftarrow \sigma_{O,R}(\overrightarrow{d_2})$ **end if**
      for every $(\overrightarrow{d_i}, \overrightarrow{d_j}) \in S_1 \times S_2$ **do**
        assume($\text{alias}_{\mu_O}(p_i, \overrightarrow{d_1}, p_j, \overrightarrow{d_2}) \Leftrightarrow \text{alias}_{\mu_G}(q_i, \overrightarrow{d_i}, q_j, \overrightarrow{d_j})$)
      end for
    end for
  end for
  execute($p_{n+1} = p_1.m_o(p_2 \ldots p_f)$) ▷ Execute m_o under s_O to yield state $s'_O = \langle \rho'_O, \mu'_O \rangle$
  execute($q_{n+1} = q_1.m_g(q_2 \ldots q_f)$) ▷ Execute m_g under s_G to yield state $s'_G = \langle \rho'_G, \mu'_G \rangle$
  assert $\bigwedge_{i \in (1 \ldots n+1)} (\rho'_O(p_i), \rho'_G(q_i)) \in \sigma_{O,R}\text{-equiv}_{\mu'_O, \mu'_G}$ ▷ Check state equivalence under $\sigma_{O,R}$
  for every $(i, j) \in [1 \ldots n+1] \times [1 \ldots n+1]$ **do**
    for every $(\overrightarrow{d_1}, \overrightarrow{d_2}) \in \text{deref}(p_i) \times \text{deref}(p_j)$ **do**
      $S_1 \leftarrow \{\overrightarrow{d_1}\}$; $S_2 \leftarrow \{\overrightarrow{d_2}\}$
      if type(p_i) = O **then** $S_1 \leftarrow \sigma_{O,R}(\overrightarrow{d_1})$ **end if**
      if type(p_j) = O **then** $S_2 \leftarrow \sigma_{O,R}(\overrightarrow{d_2})$ **end if**
      for every $(\overrightarrow{d_i}, \overrightarrow{d_j}) \in S_1 \times S_2$ **do**
        assume($\text{alias}_{\mu_O}(p_i, \overrightarrow{d_1}, p_j, \overrightarrow{d_2}) \Leftrightarrow \text{alias}_{\mu_G}(q_i, \overrightarrow{d_i}, q_j, \overrightarrow{d_j})$)
      end for
    end for
  end for
end procedure

4.4.2 Generating Harness. The next step is to formulate the correctness condition and invoke Sketch to instantiate, if possible, the choices in the sketched method to obtain the final generated method. The correctness condition is formulated as a harness that invokes the sketched method. Algorithm 6 presents `generate-harness` procedure, that generates a method the harness. Figure 8 presents the resulting generated Sketch program in our example.

Algorithm 7 Symbolic state creation procedure

1: **procedure** `create-states(P,O,R)`
2: $\quad s_O = \langle \rho_O, \mu_O \rangle \leftarrow$`empty-state`; $s_G = \langle \rho_G, \mu_G \rangle \leftarrow$`empty-state`
3: $\quad$ **for** every $\langle$`t,p`$\rangle \in$ `P` **do** ▹ Create parameters for method invocation
4: $\quad\quad$ **if** `t` = O **then** $s_O \leftarrow$ `create`(O, s_O); $s_G \leftarrow$ `create`(G, s_G)
5: $\quad\quad$ **else** $s_O \leftarrow$ `create(t,` s_O`)`; $s_G \leftarrow$ `create(t,` s_G`)`
6: $\quad\quad$ **end if**
7: $\quad\quad$ **for** every $\vec{d} \in$ `deref(t)` **do** ▹ Additional objects for tracking side-effects
8: $\quad\quad\quad$ **if** `type(`$\vec{d}$`)` = O **then** $s_O \leftarrow$ `create(O,` s_O`)`; $s_G \leftarrow$ `create(G,` s_G`)`
9: $\quad\quad\quad$ **else if** `type(`$\vec{d}$`) = C` **then** $s_O \leftarrow$ `create(C,` s_O`)`; $s_G \leftarrow$ `create(C,` s_G`)`
10: $\quad\quad\quad$ **end if**
11: $\quad\quad$ **end for**
12: $\quad$ **end for**
13: $\quad$ return $\langle$`symbolize(`ρ_O, μ_O`),symbolize(`ρ_G, μ_G`)`$\rangle$ ▹ Symbolic primitive values and aliases
14: **end procedure**

The procedure takes as input parameters the original method m_o, the sketched method signature m_g, and a candidate equivalence predicate $\sigma_{O,R}$. It first invokes `create-states` procedure shown in algorithm 7 to create two symbolic states s_O and s_G. Here s_O is the symbolic state under which m_o will execute; s_G is the symbolic state under which m_g will execute. The symbolic state s_O has variables $p_1 \ldots p_f \ldots p_n$ while s_G has corresponding variables $q_1 \ldots q_f \ldots q_n$.

We explain the construction of the symbolic states here. The `create-states` procedure in algorithm 7 begins the construction of the required states by creating two empty states at line 2. It iterates over formal parameters of method m_o and creates a new value for each parameter in states s_O, s_G (lines 4-6). If the parameter is of type O, then the procedure creates an object of type O in state s_O and an object of type R wrapped by a G object in state s_O. Otherwise, the procedure adds an object of the same type `C` to both states. Next, the procedure creates additional variables under both states to track the side effect of executing method m_o, m_g under states s_O, s_G (line 7-11). The procedure considers all possible field dereferences of input parameters and iterates over them. If the dereference if of type O, procedure adds instance of type O to s_O and instance of type G to s_G, otherwise it adds an object of the same type `C` to both states. The procedure symbolizes both states by assigning symbolic variables to all primitive values in the state. It also creates aliases symbolically, by allowing a reference variable or its dereference to reference any type compatible object in the state. The states are then returned to the `generate-harness` procedure. For the running example the corresponding generated state construction and initialization code appears on lines 5-17 of Figure 8.

The `generate-harness` procedure enforces an **assume** construct at line 4, based on the equivalence predicate $\sigma_{O,R}$, and adds the value equivalence assumption into the generated harness. The equivalence assumption is implemented via Sketch `assume` statements. In the example the generated equivalence assumption appears on lines 19-22 of Figure 8. The loop on lines 5-14 enforces the state equivalence of the two states under $\sigma_{O,R}$. This corresponds to lines 24-26 in the example.

The **execute** construct on lines 15-16 of Algorithm 6 generates the invocations of m_o and m_g. The corresponding generated code appears on line 29 of Figure 8. Lines 17-27 of the procedure use the **assert** construct to generate the required correctness condition. This condition requires the equivalence predicate $\sigma_{O,R}$ to hold in the state after the execution of the two invoked methods.

```
 1.  /*
 2.   * All asserts must be satisfied for every input array s.
 3.   */
 4.  harness static void check(int s[14]) {
 5.    // Construct state  by initializing variables
 6.    Original p1 = new Original(); int p2; Original p3 = new Original();
 7.
 8.    // Symbolize the state using symbols s[1], s[2], ... s[7]
 9.    p1.x = s[1]; p1.y = s[2];  p1.count = s[3];  p2 = s[4];
10.    p3.x = s[5]; p3.y = s[6]; p3.count = s[7];
11.
12.    // Construct state by initializing variables
13.    Generated q1 = new Generated(); int q2; Generated q3 = new Generated();
14.
15.    // Symbolize the state  using symbols s[8], s[9], ... s[12]
16.    q1.r.a = s[8]; q1.r.b =s[9]; q2 = s[10];
17.    q3.r.a = s[11]; q1.r.b = s[12];
18.
19.    // Enforce value equivalence on states
20.    assume(p1.x+p1.y == q1.r.a && p1.x-p1.y == q1.r.b);
21.    assume(p2 == q2);
22.    assume(p3.x+p3.y == q3.r.a && p3.x-p3.y == q3.r.b);
23.
24.    // Either alias (p1, p3) and (q1.r, q3.r) or enforce that both do not alias
25.    if(s[13] > 0) {assume alias(p1, p3); assume alias(q1.r, q3.r);}
26.    else {assume !alias(p1, p3); assume !alias(q1.r, q3.r);}
27.
28.    // Execute methods with symbolic inputs
29.    p1.scale(p2);  q1.scale(q2);
30.
31.    // Value equivalence in final states
32.    assert(p1.x+p1.y == q1.r.a && p1.x-p1.y == q1.r.b);
33.    assert(p2 == q2);
34.    assert(p3.x+p3.y == q3.r.a && p3.x-p3.y == q3.r.b);
35.
36.    // Equivalent aliases in final states
37.    if(alias(p1, p3)) {assert alias(q1.r, q3.r);}
38.    else {assert !alias(q1.r, q3.r);}
39.  }
```

Fig. 8. The correctness check for method scale from the running example.

The equivalence condition is implemented via Sketch `assert` statements, which the Sketch implementation must verify to resolve the Sketch variables in m_g and produce a correct generated method that preserves the equivalence condition. In our example the corresponding generated `assert` statements appear on lines 32-38.

4.5 Correctness Argument

We next present a correctness argument for Algorithm 6. The generated Sketch program uses assume/guarantee reasoning to ensure that, if methods m_o and m_g start out in equivalent states under a given inter-class equivalence $\sigma_{O,R}$, then the two states remain equivalent under the same inter-class equivalence $\sigma_{O,R}$ after the methods execute. The assume part of the assume/guarantee reasoning is implemented with Sketch `assume` statements (lines `20-26` in the example in Figure 8). Critically, the assume/guarantee reasoning ensures that the two methods return equivalent values and drive all reachable objects in the program state to equivalent states. The soundness of this assume/guarantee reasoning relies on the symbolic bindings of variables $\mathsf{p}_1 \ldots \mathsf{p}_n$ and $\mathsf{q}_1 \ldots \mathsf{q}_n$ to ensure that these variables correctly reflect the externally visible effects of m_o and m_g. We ensure this property by creating a variable to represent every reference value the two methods access and generating appropriate equivalence conditions for these variables as required by the state equivalence condition defined in IS.1 and IS.2.

Note that Algorithm 6 can work with arbitrary satisfiable equivalence predicates $\sigma_{\mathrm{O,R}}$ — the only requirement is that Mask find *some* equivalence implementation that satisfies the harness. This fact enables Mask to work with essentially arbitrary equivalence predicate generation algorithms as long as the algorithm is able to generate an equivalence predicate that verifies. Algorithm 4 is one example of such an algorithm.

The Sketch algorithm chooses a correct implementation of m_g by resolving the holes in the sketch of method m_g so that the generated Sketch program verifiably satisfies the correctness condition as expressed in the Sketch `assume` and `assert` statements. Note that an unsatisfiable equivalence predicate $\sigma_{\mathrm{O,R}}$ would enable the Sketch solver to arbitrarily resolve the holes and create a potentially incorrect implementation of m_g. We avoid this situation by requiring, the $\sigma_{\mathrm{O,R}}$ is satisfiable and for every instance of class O, there exists some instance of class R that causes the equivalence predicate $\sigma_{\mathrm{O,R}}$ to hold as ensured by Algorithm 3. This property ensures the `sketch-solver` produces a correct m_g by resolving the `sketch`.

4.6 Sketch Solver

The *sketch solver* receives the generated sketch for class G and the correctness checks constructed in the previous step as input. It then invokes the Sketch [Solar-Lezama 2008] engine to complete the input sketch. If the $\sigma_{\mathrm{O,R}}$ associated with the check is *valid* and the sequence length of the input sketch can represent the implementation of O, then the solver succeeds and returns the completed class. If not, the *sketch solver* requests for a new set of checks encoding a different $\sigma_{\mathrm{O,R}}$ candidate. This is carried out until all $\sigma_{\mathrm{O,R}}$ candidates are exhausted or Mask synthesizes the required class.

5 IMPLEMENTATION

We incorporated our ideas as part of a tool, named Mask, for synthesizing adapter classes in Java. The implementation of Mask is in Java and leverages existing frameworks – the JavaPathFinder (`JPF`) [Khurshid et al. 2003] for symbolic summary generation, `Mistral` [Dillig et al. 2012] to compute the minimal satisfying assignment, and `JavaSketch` [Jeon et al. 2015], that internally invokes `Sketch` [Solar-Lezama 2008]. It takes as input two Java classes and any known class invariants, and outputs the synthesized adapter Java class. We now elaborate on a few implementation details.

Handling public fields. Java allows a client to directly modify public fields defined by a class. Therefore, we require Mask to synthesize suitable methods in G, that can be used to replace instructions that access public fields in class O. To do this, we refactor classes O, R by creating public `get` and `set` methods for all suitable public fields in classes O, R. This enables Mask to synthesize the required adapter methods in G.

Handling constructors. We require Mask to synthesize a constructor in class G, for every public constructor defined by class O. Constructors are handled as a special case by Mask and it will refrain from adding assume clauses for the object under creation in the generated harness (Algorithm 6, lines 4-14). However, the harness will contain assert clauses that requires the newly created objects to be in equivalent states (Algorithm 6, lines 17-27).

Handling arrays and field dereferences. The symbolic execution engine constructs unique symbols for every dereference by walking the type tree of the associated field. In a few cases (e.g., linked lists), there can be infinite symbols that are possible. We address this issue by setting a limit on the dereference sequence to a constant `c` in our implementation. Similarly, handling arrays is challenging due to the size of the array. We also set an upper bound on the number of elements in the array to `c`. Note that imposing such a bound means that the correctness guarantee holds only within the specified bound `c`.

Handling loops and recursion. The presence of loops or recursive method invocations can affect the scalability of the symbolic execution engine. If the loop condition is symbolic, the number of unique paths that need to be considered by the engine can be infinite. Therefore, we limit the unrolling of the loop to a constant `c` in symbolic summary generation. Similarly, we also bound the maximum depth of recursive calls to a constant `c`. Once again, note that imposing such a bound means that the correctness guarantee holds only within the specified bound `c`.

Handling generics. Handling generic object types can be challenging for the underlying symbolic execution engine and the sketch solver. This is because the set of paths that need to be explored by these systems become unconstrained, as every instantiation of a generic type variable can introduce more paths. For our implementation, we restrict the usage of generics and use pre-defined concrete types.

Handling exceptions. The sketch synthesized by Mask contains choices to catch the exceptions that may be thrown by the method invocations to R. Since, `JSketch` currently does not support handling exceptions, we refactored [Refaster 2019] the input classes to set a state variable when an exception is thrown and suitably handle the set exception.

Optimizing the equivalence predicate generation process. Because the generation of the equivalence predicates can be expensive, we perform additional optimizations to reduce the associated costs. We employ a ranking mechanism to rank the set of possible candidate predicates. Higher ranks are assigned to stronger predicates. We apply the analysis for top ranked predicates to synthesize the required class before analyzing lower ranked predicates.

6 EXPERIMENTAL EVALUATION

In this section, we describe our experimental setup and present the details of evaluating our system on Java classes from open-source codebases. We ran our experiments on Ubuntu-14.04 VMware running on a 2.9Ghz Intel Core i7 processor with 8GB RAM. For our experiments, we let our system explore the possibility of replacing as many methods as possible (i.e., allow classes to be replaced partially) from the original class and present those results.

6.1 Results

Table 2 presents details of the classes used for our experimentation. For each class used in our experiments, the table specifies the source library, the version of the library, the number of fields and methods in the class. We used `ArrayList` – a class for handling arrays in the JDK, `Vector` – a class for operating over a vector of elements in the JDK, `FastArray` – a class for processing arrays in `Groovy`, `FastVector` – a class for processing vectors in `weka`, `Box2` – a class that implements a two dimensional box, `Rectangle` – a class that implements a rectangle, `MutablePair`,

Table 2. Benchmark Information. |F| is the field count and |M| is the method count.

Class name	Benchmark	Version	|F|	|M|
ArrayList	JDK	1.7	3	24
Vector	JDK	1.7	4	38
FastArray	Groovy	2.4.4	2	13
FastVector	WEKA	3.6.12	4	21
Box2	JMist	0.1.1	4	11
Rectangle	eclipse	3.9.0	4	23
MutablePair	apache	3.4	2	9
ImmutableTriple	apache	3.4	3	4
MutableTriple	apache	3.4	3	8
Point3D	openimaj	1.3.1	2	15

`ImmutableTriple`, `MutableTriple` – classes in `apache` used to process pairs and triples, and `Point3D` – a class in `openimaj` that implements 3D points.

We applied our system to analyze pairs of classes that are closely related. For the classes `ArrayList`, `FastArray`, `Vector` and `FastVector`, we considered the possibility of replacing each class in the set with the other class. Similarly, we considered the possibility of replacing `Box2` with `Rectangle`, `MutablePair` with `MutableTriple`, and vice-versa. We also highlight the results associated with replacing arbitrary classes (e.g., `Point3D` with `MutableTriple`, `MutablePair` with `ImmutableTriple`) to study the behavior of our approach in synthesizing replacements for seemingly unrelated classes. We constrain the maximum field dereference, array and loop unrolling lengths to 5.

Table 3 presents the results of applying our system on different pairs of classes. The first column represents the ID assigned to each pair ($E_1 \ldots E_{10}$). The second and third columns represent two classes C_1 and C_2. For each row, we initially use C_1 as the original class O and C_2 as the replacement class R, and *vice versa.* We present details pertaining to the ratio of the overall number of methods for which an implementation could be synthesized ($|S|/|M|$ in column 4), the number of inter-class equivalence predicates ($|\Sigma|$ in column 5), the maximum length of method invocations in the replacement class (ℓ in column 6), and the overall time taken by the entire approach in seconds (in column 7). The time taken includes a summation of the time taken to symbolically analyze the class implementations, the time consumed by the candidate generator and the time required by `Sketch` to synthesize the final implementation. Columns 8 – 11 provide the results when C_2 is used as O and C_1 is used as R.

Ability to synthesize replacements. Table 3 shows the ability of Mask to synthesize adapter classes. For example, 33 methods out of 38 methods in the `Vector` class can be replaced using the implementations of `ArrayList`. Therefore, for an application that uses a subset of these 33 methods, our approach can be used seamlessly without any manual intervention. Even for applications that use the remaining five methods, it is sufficient to focus on identifying effective ways of replacing the corresponding invocations in the application while using the results of our implementation for the remaining 33 methods. In general, we observe that a significant number of methods in each class can be synthesized for a majority of the pairs considered by our analysis.

Also, the replacement process is not symmetric. In other words, the number of methods in C_1 that can be implemented using C_2 is not always equal to the number of methods in C_2 that can be implemented using C_1. This is due to the differing functionalities in the two classes. We observe that 24 out of 24 methods in `ArrayList` can be replaced using the implementation of `Vector` even

Table 3. Experimental validation: Each row considers a pair of classes (C_1 and C_2) and gives details on replacing C_1 with C_2 and *vice-versa*. The table specifies the number of methods for which a replacement could be automatically synthesized by our system (|S|), the number of σ candidates considered for synthesis (|Σ|), the maximum number of method invocations from the replacement class required in the result to implement the method in the original class (ℓ). The overall time (in seconds) required for generating the replacement is also provided. The time column is split into three components – time for symbolic analysis, time for candidate $\sigma_{O,R}$ generation, and the time consumed by the Sketch solver to synthesize a replacement.

ID	C_1	C_2	$O \leftarrow C_1 \wedge R \leftarrow C_2$				$O \leftarrow C_2 \wedge R \leftarrow C_1$			
			|S|/|M|	|Σ|	ℓ	time	|S|/|M|	|Σ|	ℓ	time
E1	Vector	ArrayList	33/38	1	1	6+1+1050	24/24	1	1	6+1+1674
E2	Vector	FastArray	10/38	1	5	5+2+1967	5/13	1	1	13+2+28
E3	Vector	FastVector	11/38	1	1	4+1+7	9/21	1	1	5+2+42
E4	ArrayList	FastArray	7/24	1	5	3+2+2377	8/13	1	1	3+2+15
E5	ArrayList	FastVector	8/24	1	1	4+1+6	7/21	1	1	4+1+10
E6	FastVector	FastArray	12/21	1	2	4+2+14	8/13	1	1	2+2+8
E7	Box2	Rectangle	9/11	8	2	2+39+100	8/23	8	1	1+38+4
E8	MutablePair	ImmutableTriple	6/9	6	1	1+1+4	0/4	0	0	2+1+0
E9	MutablePair	MutableTriple	9/9	6	2	2+2+4	0/8	0	0	2+1+0
E10	Point3D	MutableTriple	11/15	6	6	1+2+58	8/8	6	1	1+2+4

though not all methods in `Vector` can be replaced using `ArrayList` (33 out of 38). This contrast in behavior can be observed clearly (in `E6`) for the pair `MutablePair` and `ImmutableTriple`. While the methods of `MutablePair` can be implemented using `ImmutableTriple`, none of the methods in `ImmutableTriple` can be implemented using `MutablePair`. This is because `ImmutableTriple` defines three independent fields that are returned to the client, and this cannot be captured by the two fields defined by `MutablePair`. Therefore, our technique eliminates all $\sigma_{O,R}$ candidates as invalid.

Synthesized methods invoke method sequences. The methods synthesized involve invoking multiple methods in the replacement class to implement the original functionality. This behavior can be observed across multiple pairs of classes (`E4, E6, E7, E9, E10`). The maximum sequence length of invocations is six for implementing the method `copyFrom(Point3D)` in `Point3D` using the methods in the class `MutableTriple`.

Derived multiple inter-class equivalence predicates. Our approach derived multiple candidate inter-class equivalence predicates for various pairs of classes. The maximum number of predicates derived is eight for the pair `Box2` and `Rectangle`. These predicates provide the equivalence of states across the two classes. Even though each of the two classes contain four fields with a potential for 256 predicates (4 * 4 * 4 * 4), our system is able to prune many infeasible equivalence predicates. Further, while the semantics of the underlying fields are different in the various class implementations, Mask generates relevant constraints and is able to synthesize suitable replacements. Further, in the cases where the equivalence predicate cannot be derived (`E8`, `E9`), we manually verified that the predicate is indeed outside the scope of our technique.

Analysis times for various components. The time consumed by the overall process ranges from 3 secs for `E8` and `E9` to 40 minutes for `E4`. A significant portion of the time is consumed by `JavaSketch` to synthesize the class from the input sketch. Since the synthesizer has to ensure that the correctness condition is satisfied for any given input, the solver can consume more time than the other two components of our system. Further, the time taken for synthesis is dependent on multiple parameters including the length of the sequence for an input sketch and the complexity of the method implementations in the replacement class.

The implementation of methods in `FastArray` is more complex than the implementation in `MutableTriple` which explains the differing times for sketch (`E4` *vs* `E10`) even though longer method invocation sequences are considered by sketch in both cases. Also, the maximum sequence length is different for the two cases under `E4` (5 *vs* 1) explaining the contrast in the time consumed by the sketch solver. *Effectiveness of our approach.* We also studied the effectiveness of our approach

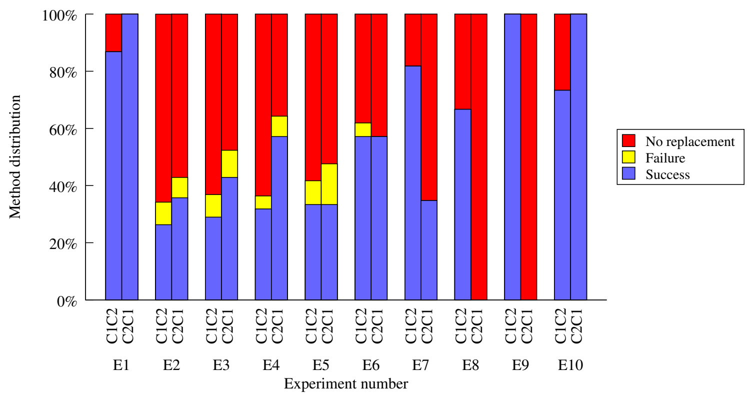

Fig. 9. Effectiveness of our approach.

by performing a manual analysis. More specifically, we wanted to understand the limitations of our strategy in synthesizing class replacements. Figure 9 presents the results of our study. We classify the methods into three categories – (a) approach correctly synthesized replacement implementations, (b) there is no feasible replacement for the method using the given replacement class, and (c) there is a feasible replacement for the method but our approach is unable to synthesize the replacement.

For a majority of the methods, we observe that the absence of synthesis can be mainly attributed to the lack of any feasible replacement. In the few cases where our approach failed to synthesize a replacement, we found that this was due to the presence of non-linear constraints in the system, or the underlying state invariant could not be captured as part of the symbolic expression.

6.2 Case Studies

We studied the application of our approach under two scenarios – (a) usefulness of the approach when classes are modified, and (b) effectiveness of the synthesized replacement classes by incorporating it in an application.

6.2.1 Analyzing Modified Classes. The proposed approach can be used while updating a client application to use the latest version of a class. The new version can differ from the older version in terms of the underlying data structures, the signatures of public methods, the method functionality and/or cosmetic changes. Therefore, blindly updating the application without considering these changes can lead to unforeseen changes to the application logic. In such cases, Mask can be used to synthesize an adapter class that invokes the public methods defined by the latest version of the class, but is still equivalent to the older version. If successful, the adapter class can be used as drop-in replacement for the older version of the class.

We studied the effectiveness of Mask for this use case by analyzing two different versions of a class. Table 4 presents the results of our study. It presents classes from various popular (> 450 stars

Table 4. Analyzing modified classes.

Class name	Benchmark	ID	Synthesis?
Image	Structurizr[Structurizr 2019]	a1975c2	Success
IntInterval	Eclipse Collection[Eclipse Foundation 2019]	4c069a8	Failed(Bug 451)
PnConfiguration	Pubnub Java[PubNub 2019]	ca45925	Success
GCPAuthenticator	Kubernetes[Kubernetes 2019]	080c384	Success
SpscArrayQueue	RxJava[RxJava 2019]	7aa0b34	Success
ParamValidatorUnwrapper	Dropwizard[Dropwizard 2019]	dbc1c5a	Failed(Bug 1405)

on GitHub) code bases, where developers have modified the specified classes. Table 4 presents the class name, the codebase, the commit identifier associated with the modification, and the result of applying our approach. The changes introduced to the classes in the new version includes: removing/introducing/updating fields defined in a class, changing the method signature in the class, modifying the implementation of the methods, and refactoring code for improving readability.

Our approach is able to synthesize a replacement class for four out of the six classes. For these four classes, we were able to synthesize an adapter for the new versions despite multiple changes to the internal data representation and method implementations. Also, our approach is unable to generate replacements for two classes because the underlying modifications were made to fix existing bugs.

6.2.2 Applying Replacements to a Client. We also performed a study to validate the effectiveness of our technique by applying the generated class replacement in a third-party client. For this purpose, we considered two classes `QueryExecutorImpl` and `V2Query` in `postgresql-jdbc-8.0-325`. There are multiple uses of `Vector` objects in these classes. We used the drop-in replacement that is synthesized for replacing `Vector` with `ArrayList` and applied it on these classes. We verified that these replacements are correct modifications to the client code. More interestingly, our modifications are also validated by the refactoring performed by the developers of `postgresql-jdbc` where they have modified the code to use `ArrayList` instead of `Vector` in `postgresql-jdbc-9.3-1104`. The author of this change discusses the absence of synchronization and the consequent speedup as one of the reasons for undertaking this change [PostgreSQL-JDBC-9.3 2019].

6.3 Limitations

To the best of our knowledge, this is the first attempt in synthesizing an adapter class for a given replacement class. We have proposed the design of Mask that handles many real classes. We now enumerate the limitations of Mask.

- Our approach is constrained by the strengths of the underlying approaches and inherits the following limitations from them:
  - The approach builds on sketch solver which performs bounded verification. Therefore, the synthesized replacements are only guaranteed to be correct as long as the replacement methods/constructors have bounded number of paths. This assumption will not be satisfied by methods that have unbounded loops or recursive calls.
  - The `create-states` method creates symbolic states under the assumption that, the possible field dereferences for every input parameter to method $m_o \in \mathrm{O}$ are bounded. This assumption will not hold if the input objects can contain arrays fields or uses recursive data structures.
  - The SMT based solvers used by the symbolic execution engines currently can only reason about linear computations. Therefore, the approach will fail to synthesize replacement for methods that contain non-linear computations.

- We do not handle generation of non-sequential structures (e.g., branches and loops).
- The synthesized replacements are not guaranteed to exhibit identical behavior under concurrency.
- We do not handle class hierarchies.
- The program may not always return the result of an operation but can store it in an external environment (e.g., I/O, network, etc). We have not modeled and handled such scenarios in the current implementation.

7 RELATED WORK

Contextual Equivalence. The equivalence of two expressions under all contexts can be proved using contextual equivalence [Koutavas and Wand 2006a,b; Lahiri et al. 2012; Sangiorgi et al. 2011; Sumii and Pierce 2004, 2005; Wand et al. 2018; Wang et al. 2017b; Wood et al. 2017]. A set of techniques [Koutavas and Wand 2006a,b; Sumii and Pierce 2004, 2005] establish a bisimulation invariant to prove the equivalence of two lambda calculus programs. Sangiorgi *et al* [Sangiorgi et al. 2011] extend this work by proposing techniques that can establish bisimulation invariants for higher order programs. Wang *et al* [Wang et al. 2017b] propose techniques for verifying the equivalence of database applications. Wood *et al* [Wood et al. 2017] propose an approach for verifying the equivalence of methods that may contain memory allocations, cyclic data structures and recursion. In contrast to these approaches where equivalence of two programs is verified, we synthesize a class that is equivalent to an original class by establishing inter-class equivalence predicates. Further, the approach addresses challenges pertaining to aliases, side-effects, etc.

Wang *et al* propose a technique [Wang et al. 2019] for synthesizing equivalent database queries for an application that has undergone schema migration. They establish equivalence predicates across two versions of the schema by equating the columns and then employ sketch based synthesis for generating equivalent queries. Although there is some conceptual similarity, this technique is not suitable for the class migration problem addressed by Mask. Firstly, their approach targets query migration for database applications, whereas our approach targets class migration for object oriented languages. Secondly, our approach synthesizes equivalence predicates by performing symbolic execution which can identify non-trivial equivalence predicates between the two classes, as illustrated in Figure 6. Finally, our approach has to reason about the aliases and side-effects which is critical for the correctness of the synthesized solution.

Program specification inference. A number of techniques have been proposed to infer program specifications [Albarghouthi et al. 2016; Ammons et al. 2002; Bastani et al. 2015, 2018; Flanagan and Leino 2001; Livshits et al. 2009; Logozzo 2004; Nimmer and Ernst 2002; Pradel and Gross 2009; Ramanathan et al. 2007; Sharma and Aiken 2014; Yorsh et al. 2008]. Albarghouthi *et al* [Albarghouthi et al. 2016] propose a technique for extracting the weakest specification of an open program that meets a user specified post-condition. Bastani *et al* [Bastani et al. 2018] automatically infer the behavior of libraries by synthesizing points-to specifications. Logozzo [Logozzo 2004] proposes an approach to perform modular and automatic inference of class invariants. These techniques can potentially be used to provide useful hints to our approach to reduce the search space of equivalence predicates.

Program sketching. Solar-Lezama *et al* [Solar-Lezama 2008] proposed sketch based program synthesis that takes as input a partial program containing *holes* from the user. This partial program is also known as a sketch. The program obtained by resolving the holes is validated using input correctness conditions. There has been significant progress in this area subsequently [Solar-Lezama 2009; Solar-Lezama et al. 2008, 2006]. Our approach builds upon this to synthesize classes instead of closed programs. Our system generates the sketch for the method implementations in a class, constructs the correctness conditions and provides them as input to Sketch [Solar-Lezama 2008] to synthesize the replacement classes.

Component based synthesis. Many researchers have investigated the problem of synthesizing a method using available components [Feng et al. 2017a,b; Jha et al. 2010; Mandelin et al. 2005; Yessenov et al. 2017]. For instance, SyPet [Feng et al. 2017b] takes as input the components, a method signature and test cases and builds a petri-net using the APIs defined by the input components to prune the space of possible sketches. SyPet is used to synthesize one method at a time with the help of test cases. In contrast, our approach is designed to synthesize a class where the synthesized implementations of multiple methods need to work correctly, even though the synthesis process is performed in isolation. Further, unlike their approach, our approach does not take any test cases as input. To provide test cases to SyPet to synthesize a class, all possible contexts need to be provided as input as we are synthesizing a class. `Morpheus` [Feng et al. 2017a] is designed to synthesize a table transformation that is used for data management whereas we synthesize classes.

Applications of synthesis to various domains. Programming by example uses input-output examples to synthesize programs that produce the behavior specified by the examples. This strategy has been applied to synthesize programs in various domains [Barowy et al. 2015; Cheung et al. 2013; Drachsler-Cohen et al. 2017; Peleg et al. 2018; Polikarpova et al. 2016; Schlaipfer et al. 2017; Wang et al. 2017a]. Cheung *et al* [Wang et al. 2017a] use this technique to synthesize expressive SQL queries for databases. Schlaipfer *et al* [Schlaipfer et al. 2017] analyze a given query to identify sub queries that can be optimized effectively. Barowy *et al* [Barowy et al. 2015] employ this technique to extract structured relational data from spread sheets. These techniques cannot be used to synthesize class replacements.

Source code refactoring. There are a number of existing tools that are able to refactor the source code based on specified rules, including Refaster [Refaster 2019] and Rewrite [ReWrite 2019]. Our approach can be integrated with these tools to specify the appropriate rules for refactoring the application to use the synthesized adapter classes.

8 CONCLUSION

In this paper, we addressed the problem of automatic class migration. Our approach takes an original class O and a replacement class R, and synthesizes an adapter class G that implements the same interface as O using the implementation of R. The synthesized methods in class G are equivalent to those defined by O. We build state equivalence predicates to enable synthesis of G's methods in isolation while ensuring that arbitrarily long method invocations on the original and synthesized classes exhibit equivalent behavior. We design our solution by integrating symbolic execution, constraint solving and program synthesis to synthesize the required class. Our experimental results on opens Java classes demonstrate the efficacy of our approach.

ACKNOWLEDGMENTS

We thank our shepherd, Shuvendu Lahiri and all the anonymous reviewers for their constructive and useful feedback on this work. This research was supported by the DARPA transparent computing program (contract FA8650-15-C-7564).

REFERENCES

Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. 2016. Maximal Specification Synthesis. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 789–801. https://doi.org/10.1145/2837614.2837628

Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining Specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*. ACM, New York, NY, USA, 4–16. https://doi.org/10.1145/503272.503275

Apache Commons 2019. Apache Commons: An Apache project focused on all aspects of reusable Java components. (2019). https://commons.apache.org/

API Deprecation 2018. How and When To Deprecate APIs. (2018). https://docs.oracle.com/javase/7/docs/technotes/guides/javadoc/deprecation/deprecation.html

Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin Zorn. 2015. FlashRelate: Extracting Relational Data from Semi-structured Spreadsheets Using Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 218–228. https://doi.org/10.1145/2737924.2737952

Osbert Bastani, Saswat Anand, and Alex Aiken. 2015. Specification Inference Using Context-Free Language Reachability. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 553–566. https://doi.org/10.1145/2676726.2676977

Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2018. Active Learning of Points-to Specifications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 678–692. https://doi.org/10.1145/3192366.3192383

Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing Database-backed Applications with Query Synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/2491956.2462180

Danny Dig and Ralph Johnson. 2006. How Do APIs Evolve&Quest; A Story of Refactoring: Research Articles. *J. Softw. Maint. Evol.* 18, 2 (March 2006), 83–107. https://doi.org/10.1002/smr.v18:2

Isil Dillig, Thomas Dillig, Kenneth L. McMillan, and Alex Aiken. 2012. Minimum Satisfying Assignments for SMT. In *Computer Aided Verification*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 394–409.

Dana Drachsler-Cohen, Sharon Shoham, and Eran Yahav. 2017. Synthesis with Abstract Examples. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 254–278.

Dropwizard 2019. `Dropwizard`. https://github.com/dropwizard/dropwizard

Eclipse 2019. *Eclipse: The Platform for Open Innovation and Collaboration*. https://www.eclipse.org/

Inc. Eclipse Foundation. 2019. *Eclipse Collection: A comprehensive collections library for Java*. https://github.com/eclipse/eclipse-collections

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017a. Component-based Synthesis of Table Consolidation and Transformation Tasks from Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 422–436. https://doi.org/10.1145/3062341.3062351

Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017b. Component-based Synthesis for Complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 599–612. https://doi.org/10.1145/3009837.3009851

Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *FME 2001: Formal Methods for Increasing Software Productivity*, José Nuno Oliveira and Pamela Zave (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 500–517.

Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 416–419. https://doi.org/10.1145/2025113.2025179

Groovy 2019. *Groovy language: A multi-faceted language for the Java platform*. http://groovy-lang.org/

Guava Collections 2009. Why did Google build Guava collections. (2009). https://code.google.com/archive/p/google-collections/wikis/Faq.wiki

Guava v/s Apache 2009. Google Guava vs Apache commons. (2009). https://stackoverflow.com/questions/1444437/google-guava-vs-apache-commons

Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. 2016. Energy Profiles of Java Collections Classes. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 225–236. https://doi.org/10.1145/2884781.2884869

JDK 2019. Oracle technology network for java developers, oracle technology network, oracle 2019. (2019). https://www.oracle.com/technetwork/java/index.html

JDK Bug fixes 2019. List of bug fixes in JDK 8u221 release. (2019). https://www.oracle.com/technetwork/java/javase/2col/8u221-bugfixes-5480117.html

JDK Deprecation 2018. Deprecated JDK elements. (2018). https://docs.oracle.com/javase/7/docs/api/deprecated-list.html

Jinseong Jeon, Xiaokang Qiu, Jeffrey S. Foster, and Armando Solar-Lezama. 2015. JSketch: Sketching for Java. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 934–937. https://doi.org/10.1145/2786805.2803189

Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided Component-based Program Synthesis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 215–224. https://doi.org/10.1145/1806799.1806833

Puneet Kapur, Brad Cossette, and Robert J. Walker. 2010. Refactoring References for Library Migration. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 726–738. https://doi.org/10.1145/1869459.1869518

Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. 2003. Generalized Symbolic Execution for Model Checking and Testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*. Springer-Verlag, Berlin, Heidelberg, 553–568. http://dl.acm.org/citation.cfm?id=1765871.1765924

James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

Vasileios Koutavas and Mitchell Wand. 2006a. Bisimulations for Untyped Imperative Objects. In *Proceedings of the 15th European Conference on Programming Languages and Systems (ESOP'06)*. Springer-Verlag, Berlin, Heidelberg, 146–161. https://doi.org/10.1007/11693024_11

Vasileios Koutavas and Mitchell Wand. 2006b. Small Bisimulations for Reasoning About Higher-order Imperative Programs. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. ACM, New York, NY, USA, 141–152. https://doi.org/10.1145/1111037.1111050

Kubernetes 2019. *Kubernetes Client: Java client for the kubernetes API*. https://github.com/kubernetes-client/java

Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In *Computer Aided Verification*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 712–717.

Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. 2009. Merlin: Specification Inference for Explicit Information Flow Problems. *SIGPLAN Not.* 44, 6, 75–86. https://doi.org/10.1145/1543135.1542485

Francesco Logozzo. 2004. Automatic Inference of Class Invariants. In *Verification, Model Checking, and Abstract Interpretation*, Bernhard Steffen and Giorgio Levi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 211–222.

David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid Mining: Helping to Navigate the API Jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 48–61. https://doi.org/10.1145/1065010.1065018

Jeremy W. Nimmer and Michael D. Ernst. 2002. Automatic Generation of Program Specifications. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '02)*. ACM, New York, NY, USA, 229–239. https://doi.org/10.1145/566172.566213

Oracle v/s Google 2019. Oracle America, Inc. v. Google, Inc. (2019). https://en.wikipedia.org/wiki/Oracle_America,_Inc._v._Google,_Inc.

Hila Peleg, Sharon Shoham, and Eran Yahav. 2018. Programming Not Only by Example. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 1114–1124. https://doi.org/10.1145/3180155.3180189

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 522–538. https://doi.org/10.1145/2908080.2908093

PostgreSQL-JDBC-9.3 2019. Email thread on changes in PostgreSQL. (2019). https://www.postgresql.org/message-id/CA%2BnXnG8b7wTkb9SM5dX-X9NrEBfHYk_-DCkyoFk_ssnKm2sS9Q@mail.gmail.com

Michael Pradel and Thomas R. Gross. 2009. Automatic Generation of Object Usage Specifications from Large Method Traces. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. IEEE Computer Society, Washington, DC, USA, 371–382. https://doi.org/10.1109/ASE.2009.60

PubNub 2019. *Pubnub: Java-based SDKs for Java / Android*. https://github.com/pubnub/java

Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. 2007. Static Specification Inference Using Predicate Mining. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 123–134. https://doi.org/10.1145/1250734.1250749

Refaster 2019. Error Prone: A static analysis tool for Java that catches common programming mistakes at compiletime. (2019). https://errorprone.info/

ReWrite 2019. *ReWrite - Distributed Java Source Refactoring*. https://github.com/Netflix-Skunkworks/rewrite

RxJava 2019. *RxJava:Reactive Extensions for the JVM.* https://github.com/ReactiveX/RxJava

Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. 2011. Environmental Bisimulations for Higher-order Languages. *ACM Trans. Program. Lang. Syst.* 33, 1, Article 5 (Jan. 2011), 69 pages. https://doi.org/10.1145/1889997.1890002

Matthias Schlaipfer, Kaushik Rajan, Akash Lal, and Malavika Samak. 2017. Optimizing Big-Data Queries Using Program Synthesis. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17).* ACM, New York, NY, USA, 631–646. https://doi.org/10.1145/3132747.3132773

Rahul Sharma and Alex Aiken. 2014. From Invariant Checking to Invariant Inference Using Randomized Search. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559.* Springer-Verlag, Berlin, Heidelberg, 88–105. https://doi.org/10.1007/978-3-319-08867-9_6

Armando Solar-Lezama. 2008. *Program Synthesis by Sketching.* Ph.D. Dissertation. Berkeley, CA, USA. Advisor(s) Bodik, Rastislav. AAI3353225.

Armando Solar-Lezama. 2009. The Sketching Approach to Program Synthesis. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems (APLAS '09).* Springer-Verlag, Berlin, Heidelberg, 4–13. https://doi.org/10.1007/978-3-642-10672-9_3

Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. 2008. Sketching Concurrent Data Structures. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08).* ACM, New York, NY, USA, 136–148. https://doi.org/10.1145/1375581.1375599

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII).* ACM, New York, NY, USA, 404–415. https://doi.org/10.1145/1168857.1168907

Structurizr 2019. *Structurizr: Structurizr for Java.* https://github.com/structurizr/java

Eijiro Sumii and Benjamin C. Pierce. 2004. A Bisimulation for Dynamic Sealing. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04).* ACM, New York, NY, USA, 161–172. https://doi.org/10.1145/964001.964015

Eijiro Sumii and Benjamin C. Pierce. 2005. A Bisimulation for Type Abstraction and Recursion. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05).* ACM, New York, NY, USA, 63–74. https://doi.org/10.1145/1040305.1040311

Mitchell Wand, Ryan Culpepper, Theophilos Giannakopoulos, and Andrew Cobb. 2018. Contextual Equivalence for a Probabilistic Language with Continuous Random Variables and Recursion. *Proc. ACM Program. Lang.* 2, ICFP, Article 87 (July 2018), 30 pages. https://doi.org/10.1145/3236782

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017a. Synthesizing Highly Expressive SQL Queries from Input-output Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017).* ACM, New York, NY, USA, 452–466. https://doi.org/10.1145/3062341.3062365

Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. 2017b. Verifying Equivalence of Database-driven Applications. *Proc. ACM Program. Lang.* 2, POPL, Article 56 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158144

Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. 2019. Synthesizing Database Programs for Schema Refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019).* ACM, New York, NY, USA, 286–300. https://doi.org/10.1145/3314221.3314588

Tim Wood, Sophia Drossopolou, Shuvendu K. Lahiri, and Susan Eisenbach. 2017. Modular Verification of Procedure Equivalence in the Presence of Memory Allocation. In *Programming Languages and Systems*, Hongseok Yang (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 937–963.

Kuat Yessenov, Ivan Kuraj, and Armando Solar-Lezama. 2017. DemoMatch: API Discovery from Demonstrations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017).* ACM, New York, NY, USA, 64–78. https://doi.org/10.1145/3062341.3062386

Greta Yorsh, Eran Yahav, and Satish Chandra. 2008. Generating Precise and Concise Procedure Summaries. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08).* ACM, New York, NY, USA, 221–234. https://doi.org/10.1145/1328438.1328467