Search-Based Synthesis of Equivalent Method Sequences

Alberto Goffi[†], Alessandra Gorla[‡], Andrea Mattavelli[†], Mauro Pezzè[†*] and Paolo Tonella[§]

[†]University of Lugano
Switzerland
{goffia, mattavea, pezzem}@usi.ch

[‡]Saarland University
Germany
gorla@st.cs.uni-saarland.de

[§]Fondazione Bruno Kessler
Italy
tonella@fbk.eu

ABSTRACT

Software components are usually redundant, since their interface offers different operations that are *equivalent* in their functional behavior. Several reliability techniques exploit this redundancy to either detect or tolerate faults in software. Metamorphic testing, for instance, executes pairs of sequences of operations that are expected to produce equivalent results, and identifies faults in case of mismatching outcomes. Some popular fault tolerance and self-healing techniques execute redundant operations in an attempt to avoid failures at runtime. The common assumption of these techniques, though, is that such redundancy is known a priori. This means that the set of operations that are supposed to be equivalent in a given component should be available in the specifications. Unfortunately, inferring this information manually can be expensive and error prone.

This paper proposes a search-based technique to synthesize sequences of method invocations that are equivalent to a target method within a finite set of execution scenarios. The experimental results obtained on 47 methods from 7 classes show that the proposed approach correctly identifies equivalent method sequences in the majority of the cases where redundancy was known to exist, with very few false positives.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Measurement, Verification

Keywords

Redundancy, equivalent method sequences, search-based software engineering, specification mining

*Mauro Pezzè is also with the University of Milano-Bicocca.

FSE'14 , November 16–22, 2014, Hong Kong, China

1. INTRODUCTION

The presence of *equivalent* code fragments, for example methods or method sequences, make modern software systems *redundant*. Informally, two methods are equivalent if they produce indistinguishable results when called with proper parameters. This is the case, for instance, of methods put() and putAll() in the Google Guava class AbstractMultimap.[1] They produce indistinguishable results when putAll() is called with a collection containing the single value passed to put(). Beside interchangeable methods, as in the previous example, it is possible to obtain equivalent executions by combining several method invocations. For example, method pop() of class Stack in the Java standard library is equivalent to the method sequence remove(size()-1). Indeed, removing the element on top of the stack (pop()) leads to the same result as removing the element in the last position (remove(size()-1)).

This form of redundancy should not be confused with what are usually referred to as code clones. Code clones are typically the result of bad design and implementation practices, such as copy and paste, and indicate the need of code refactoring [16]. Instead, the redundancy described above is the result of good design practice, as it aims to offer a richer API to client components and to increase code reusability.

While in some cases redundancy exists only at the interface level, it often extends to the underlying code. For example, the code of methods pop and remove is substantially different, as shown in Figure 1. The difference in the implementation extends even to removeElementAt, which is invoked by pop, since it does not use the code of remove. We omit the code of removeElementAt for lack of space.

Recent studies indicate that redundancy is widely spread in software systems. Jiang and Su studied the Linux kernel and found more than 600,000 semantically equivalent code fragments [22], while Carzaniga et al. found more than 4,000 equivalent methods or method sequences in Java applications and libraries of non trivial size and complexity, including Apache Ant, Google Guava, Joda-Time and Eclipse SWT [5].

Equivalent method sequences find many useful applications, from the automatic generation of test inputs [8], to the design of self-healing systems [5, 6, 7], and the automatic generation of test oracles [4]. In all these applications, the equivalence is exploited automatically, but must be identified *manually*. The manual identification of equivalent method sequences is a non-trivial and error prone activity that may represent an obstacle to the practical applicability of these techniques.

[1]https://code.google.com/p/guava-libraries

```
1  public E pop() {
2      E obj;
3      int len = size();
4      obj = peek();
5      removeElementAt(len − 1);
6      return obj;
7  }
```

```
1  public E remove(int index) {
2      if (index >= elementCount)
3          throw new ArrayIndexOutOfBoundsException(index);
4
5      E oldValue = elementData(index);
6      int numMoved = elementCount − index − 1;
7      if (numMoved > 0)
8          System.arraycopy(elementData,
9              index+1, elementData, index, numMoved);
10     elementData[−−elementCount] = null;
11     return oldValue;
12 }
```

Figure 1: Methods pop and remove of class Stack from the Java Standard Library

In this paper we propose a search-based technique that can automate this activity. Given a target method and an initial set of execution scenarios, our technique automatically synthesizes method sequences that are *likely-equivalent* to the target method. The synthesized method sequences are equivalent with respect to the set of execution scenarios, and are expected but not guaranteed to be equivalent in the general case. The synthesis proceeds in two phases: In the first phase, the search goal is to synthesize a candidate method sequence to be likely-equivalent to the target method; In the second phase, the search goal is to synthesize a counterexample showing that the candidate method sequence is not equivalent to the target method on some previously unexplored scenarios. The two phases iterate, with the counterexamples added to the execution scenarios, until the second phase fails to find a new counterexample. At this point, the synthesized method sequence is deemed as likely-equivalent to the target method.

The technique is fully automatic and requires only as few as one test input (the initial execution scenario) that may be either provided by the developers or generated automatically. Our experiments indicate that the technique is effective in synthesizing equivalent method sequences, and at the same time is reasonably efficient. On 47 methods belonging to 7 different classes for which equivalent method sequences were known a priori, our approach synthesizes 87% of the known equivalences, finding one or more equivalent sequences for each target method, with few false positives and within reasonable execution time.

This paper is organized as follows: Section 2 introduces the concepts of software redundancy and equivalent sequences, and provides some basic terminology used throughout the paper. Section 3 overviews the essential characteristics of search-based engineering to make the paper self-contained. Section 4 presents our approach in detail. Section 5 discusses the validity of the proposed technique, referring to some experiments conducted on relevant case studies. Section 6 overviews the related work. Section 7 summarizes the results presented in the paper and illustrates our future research plans.

2. SOFTWARE REDUNDANCY

A software system is *redundant* if the execution of different methods or combinations of methods leads to indistinguishable results, albeit executing totally or partially different code. Two executions lead to *indistinguishable* results if they produce the same output and lead to states that cannot be differentiated by further interacting with the system. Intuitively, we are considering a type of *observational* equivalence where the states produced by the executions may be internally different but not externally distinguishable by probing the system through its public interface [21].

For example, both methods pop() and remove(size()-1) of the Java class Stack return the object on top of the stack and leave the stack without the top object. They produce indistinguishable results, but execute different code, as illustrated in Figure 1. We say that two methods or combinations of methods are *equivalent* if they produce indistinguishable results for all possible inputs, as in the previous example.

A method is trivially equivalent to itself and to an exact copy of itself. However, in this paper we are interested in methods or combinations of methods that are equivalent but execute at least partially different code—as in the example of pop() and remove(size()-1)—because of the interesting applications in the area of software testing and self-healing.

Redundancy can be explicitly added to a software system to increase reliability, or may be due to modern design practices that span from backward compatibility to the inclusion of overlapping libraries and design for reusability [7]. A library might maintain different versions of the same component to ensure compatibility with previous versions. For example, Java 7 contains at least 45 classes and 365 methods that are deprecated and that overlap with the functionality of newer classes and methods. Modern development practices naturally induce developers to use reusable components that already implement the needed functionality. It is common to find several components that provide similar or identical functionalities. For instance, the Trove4J library implements collections specialized for primitive types that overlap with the Java standard library. Yet another form of redundancy is due to performance optimization. For example, the GNU Standard C++ Library implements its basic stable sorting function using the insertion-sort algorithm for small sequences, and merge-sort for the general case.

Although all kinds of intrinsic redundancy that concretize in equivalent code fragments are interesting, in this work we focus on equivalent methods and combinations of methods in the same component.

In this paper, we present a technique that synthesizes equivalent methods or combinations of methods by exploiting search-based metaheuristics. We synthesize equivalences by examining the program behavior on a finite set of execution scenarios, and we refer to the synthesized methods or combinations of methods as *likely-equivalent*, to indicate that they may behave differently for inputs not considered in the synthesis process.

3. SEARCH-BASED ENGINEERING

Search-based software engineering is an emerging field that consists in applying search-based algorithms to software engineering problems [18]. In the last years, search-based engineering has produced interesting results especially in the area of automatic test case generation, where Genetic Algorithms (GAs) play a dominant role mostly because of their good performance [25].

GAs are inspired by the natural laws of evolution discovered by Charles Darwin, and in particular by the "survival of the fittest" principle. Informally, GAs look for approximate solutions to optimization problems whose exact solutions cannot be obtained at acceptable computational cost. In a nutshell, a GA aims to either minimize or maximize the value of a *fitness function* that quantifies the distance of the candidate solutions from optimality. Each candidate solution of the problem is encoded in what we refer to as a *chromosome*. A *population* is a set of chromosomes that iteratively evolves through *generations* by means of genetic operators. Genetic operators select and evolve candidate solutions to produce new "fitter" chromosomes. The genetic operators commonly employed in GAs have two objectives. First, they select chromosomes with the best fitness values, that is those candidate solutions to be preserved in the next generation. Second, they create new chromosomes by introducing variations in a candidate solution, for example through chromosome mutation and crossover. GAs terminate when they either find the desired solution or exhaust the time budget for the search, and return the best solution found during the evolution process.

To apply GAs to a problem, it is necessary to define *(i)* a representation of a candidate solution as chromosome, *(ii)* a fitness function, defined on the basis of the chosen candidate representation, and *(iii)* a set of genetic operators.

GAs have been successfully used to generate test cases for both procedural [27] and object-oriented software systems [15, 30]. GAs transform the problem of generating test cases into the problem of searching for inputs that maximize the coverage metrics associated with the chosen test adequacy criterion, for instance branch coverage.

When generating test cases for object-oriented systems, a chromosome is a combination of invocations of constructors and methods terminated with the invocation of the method under test. Primitive types are generated randomly, while the objects needed for the final call are generated by invoking their constructors. Intermediate method calls are introduced in a chromosome to change the internal state of an object.

Some *mutation* operators are general, for example the mutation of a primitive value, while others are designed specifically to manipulate method sequences by inserting, removing or replacing method calls. Differently from mutation operators, which are applied to single chromosomes, the *crossover* operator combines pairs of chromosomes, for instance by swapping their suffixes.

The fitness function commonly employed in the literature to maximize branch coverage is the sum of the *approach level* and the *branch distance*. The approach level rewards those executions that get close to the target branch, referring to the control flow graph, while the branch distance quantifies heuristically the distance of a condition from the opposite boolean value [25]. By evolving iteratively over generations, GAs produce test cases with increasing fitness values, until either all branches are covered, or a time limit is reached.

4. SYNTHESIS OF EQUIVALENT METHOD SEQUENCES

We exploit search-based algorithms, and in particular GAs, to synthesize a sequence of method invocations that is likely-equivalent to a target method m by means of a two-phase iterative process. We start with an initial non-empty set of execution scenarios that represent a sample of the input space of m. The initial execution scenarios may be as simple as a single test case. In the first phase, we use GAs to generate a likely-equivalent candidate *eq* for the given set of scenarios. In the second phase, we validate *eq* by using GAs to find a counterexample, which corresponds to an execution scenario for which *eq* and m behave differently. If we find a counterexample, we add it to the set of execution scenarios, and we iterate through the first phase looking for a new candidate *eq*. Otherwise, we have successfully synthesized a method sequence *eq* that is likely-equivalent to m. Method m may be equivalent to many different method combinations. Therefore, once an equivalent method sequence has been synthesized, we incrementally remove the methods used in the synthesized sequence from the search space, and we iterate looking for further equivalences.

The process for identifying a likely-equivalent method sequence for a target method m is detailed in Algorithm 1. The algorithm needs a non-empty set of execution scenarios for m. If the method comes with one or more test cases, the algorithm uses them, otherwise it generates an initial set of execution scenarios (line 1), and then iterates over the two phases (lines 2-13).

The first phase is detailed with function FIND-EQUIVALENT (lines 14-29). The search-based algorithm is employed to generate a sequence of method invocations that is likely-equivalent to the input method m for the current set of execution scenarios. The algorithm iteratively generates a candidate sequence of method invocations (line 16), and evaluates the equivalence of the synthesized sequence with m for all the executions e in the set of execution scenarios (lines 18-23). If the candidate is equivalent to m for all the execution scenarios, the phase terminates and returns the candidate (line 25). Otherwise, the algorithm discards the candidate and generates a new one, which will then be evaluated for all the execution scenarios. The function EQUIVALENT compares the object attributes and the return values obtained by executing the original method m and the candidate method sequence on a given execution scenario. If no candidate equivalent sequence is found within a given time bound, the first phase terminates (line 28), and the whole algorithm terminates as well (line 5).

The second phase is detailed in function FIND-COUNTER-EXAMPLE (lines 30-38). During this phase the algorithm validates the candidate through the exploration of new scenarios in the attempt to violate the equivalence between m and the candidate equivalent sequence synthesized after the first phase. The search for a counterexample terminates when either a counterexample is found (line 34), or the search budget expires (line 37). If this process produces a counterexample, then the candidate is deemed as not equivalent to m and the second phase terminates.

The algorithm iterates from the first phase adding the counterexample to the execution scenarios. The main iteration (line 2-13) terminates when a timeout expires and the algorithm fails in synthesizing an equivalent sequence (line 5).

Algorithm 1 Synthesis of an equivalent method sequence.

```
INPUT: m
 1: execScenarios := LOAD-INITIAL-TS
 2: while time < overall-time-limit do
 3:     candidate := FIND-EQUIVALENT(m,execScenarios)
 4:     if candidate is NIL then
 5:         return NIL
 6:     end if
 7:     counterex := FIND-COUNTEREXAMPLE(m,candidate)
 8:     if counterex is NIL then
 9:         PRINT(candidate)
10:         REMOVE-CALLS(candidate)
11:     end if
12:     add counterex to execScenarios
13: end while

14: function FIND-EQUIVALENT(m,execScenarios)
15:     while time < time-limit do
16:         candidate := SYNTHESIZE-EQUIVALENT-CALLS
17:         candidateFound := true
18:         for each e in execScenarios do
19:             if ¬EQUIVALENT(m,candidate,e) then
20:                 candidateFound := false
21:                 break
22:             end if
23:         end for
24:         if candidateFound then
25:             return candidate
26:         end if
27:     end while
28:     return NIL
29: end function

30: function FIND-COUNTEREXAMPLE(m,candidate)
31:     while time < time-limit do
32:         counterex := SYNTHESIZE-COUNTEREXAMPLE
33:         if ¬EQUIVALENT(m,candidate,counterex) then
34:             return counterex
35:         end if
36:     end while
37:     return NIL
38: end function
```

If the algorithm cannot produce new counterexamples, it prints the likely-equivalent sequence (line 9), removes the method calls used in the synthesis of the candidate (line 10), and iterates to synthesize new equivalent sequences.

We implemented the algorithm illustrated above in a Java prototype tool called SBES (Search-Based Equivalent Synthesis). Figure 2 shows the main components of SBES, which exploits EvoSuite as search-based engine. The Execution Scenarios Generator generates a set of execution scenarios by invoking EvoSuite. The Stub Generator creates a modified version of the target class by removing the target method m to enable the synthesis of equivalent sequences. The Driver iteratively invokes EvoSuite to synthesize equivalent sequences and to search for counterexamples.

EvoSuite natively supports the generation of test cases with method calls, constructors, arrays of random length, and primitive values. We modified EvoSuite to better deal with arrays of given length and values. Currently EvoSuite does not generate some arithmetic operators, loops and conditional statements, and thus our current prototype implementation

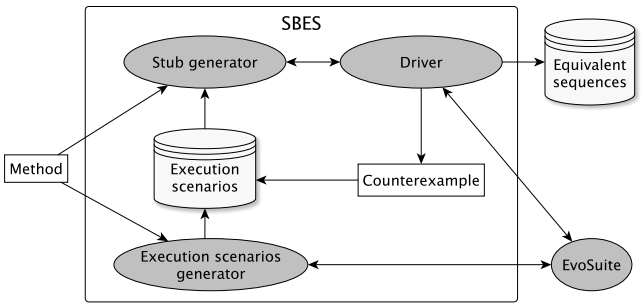

Figure 2: Main components of SBES

cannot synthesize equivalent sequences that contain these constructs. In the next sections we detail the key components of SBES, and describe the generation process.

4.1 Initialization: Execution Scenarios

Deciding whether two methods are equivalent for all the possibly infinite execution scenarios is an undecidable problem. However the problem becomes tractable by limiting the number of scenarios to a finite set. We thus synthesize a method sequence to be equivalent to a target method m by comparing the functional behavior of m and the candidate sequence on a finite set of execution scenarios. Execution scenarios can be either provided by developers, typically in the form of a test suite for the target method m, or can be generated automatically with tools such as Randoop and EvoSuite [14, 26].

In our experiments we used test suites when available, and we generated the execution scenarios with EvoSuite otherwise [14]. EvoSuite generates and evolves test suites in the attempt to cover a set of target branches through the invocation of any accessible method. Since the tool may generate method invocations that call the target method m indirectly, we modified EvoSuite forcing every generated execution to include an explicit call to m as its last statement.

In the context of Java programs, an execution scenario is a sequence of method invocations that generates objects by means of constructors, operates on such objects by means of public and protected methods, and terminates with an invocation of method m. The following method sequences are two examples of valid execution scenarios for the pop() method of the Stack class:

```
<Stack s=new Stack();s.push(1);int result=s.pop();>
<Stack s=new Stack();s.push(1);s.push(1);int result=s.pop();>
```

4.2 First Phase: Candidate Synthesis

The first phase synthesizes a sequence of method invocations that is equivalent to the target method m on a set of execution scenarios. For this task, the prototype relies on the Stub Generator and the Driver components. The Stub Generator creates a stub for the target class, namely the class that includes the declaration of the method m. The stub class encloses all the execution scenarios, and evaluates the equivalence between the target method and the synthesized candidate sequence. The Driver iteratively synthesizes method sequences by invoking EvoSuite, and uses the stub class to evaluate whether the generated sequence is equivalent to the target method m.

Given a class C that declares the target method m, the Stub Generator produces a new stub class C_Stub that contains the following core elements:

expected_states is an array containing one object of type C for each execution scenario. This data structure stores the state of these objects after the execution of each scenario.

expected_results is an array containing the return values of each execution scenario on the target method m.

actual_states is an array containing one object of type C for each execution scenario. This data structure stores the state of the objects after the execution of the synthesized method sequence on each scenario.

actual_results is an array containing the return values of the execution of the synthesized method sequence on each scenario.

custom_methods the stub class declares every method originally declared in class C and every method that C inherits from any of its superclasses. Each of these custom methods simply invokes the corresponding original method of C on every object in the actual_states array and returns the corresponding return values of such executions in the form of an array.

class_constructor the constructor of the stub class invokes each scenario on the objects stored in expected_states, and stores the results in expected_results. Similarly, it invokes all the methods of each scenario on the objects stored in actual_states. These latter invocations do not include calls to the target method m.

set_results_method is a utility method that stores the return values of the synthesized sequence in actual_results.

method_under_test is the target method for the search-based test case generator. It contains a single branch, whose condition asserts the equivalence of method m and the synthesized sequence with respect to all the execution scenarios. The equivalence considers both the object states, as stored in expected_states and actual_states, and the return values, as stored in expected_results and actual_results.

Figure 3 shows the automatically generated stub for the Stack class in the Java standard library. Given the two execution scenarios presented in Section 4.1, the stub class declares two arrays of length 2 for the expected and the actual object states, and two arrays of length 2 for the expected and the actual execution results. The constructor at line 7 executes both scenarios and stores states and results in the expected_states and expected_results arrays, respectively. The actual_states array contains the object states obtained by applying each execution scenario up to the invocation of the target method (for example, pop() in the running example).

Method push (line 25–30) is an example of how the stub generator redirects the invocations of the methods of the Stack class to the objects stored in actual_states. Such redirections occur for every method that was originally declared in the Stack class, with the exception of the target method pop.

The method method_under_test at line 32 is the main driver for the synthesis of a candidate equivalent sequence. By generating an execution that covers the TRUE branch of this method, we obtain a method sequence that is equivalent to the target method in all the considered scenarios. We

```
1  class Stack_Stub {
2    Stack exp_state[2] = new Stack[2];
3    int exp_result[2] = new int[2];
4    Stack act_state[2] = new Stack[2];
5    int act_result[2] = new int[2];

7    public Stack_Stub() {
8      // execution scenario 1
9      exp_state[0] = new Stack();
10     exp_state[0].push(1);
11     exp_result[0] = exp_state[0].pop();
12     act_state[0] = new Stack();
13     act_state[0].push(1);

15     // execution scenario 2
16     exp_state[1] = new Stack();
17     exp_state[1].push(1);
18     exp_state[1].push(1);
19     exp_result[1] = exp_state[1].pop();
20     act_state[1] = new Stack();
21     act_state[1].push(1);
22     act_state[1].push(1);
23   }

25   public int[] push(int item) {
26     int res[2];
27     for (int i = 0 ; i < 2 ; i++)
28       res[i] = act_state[i].push(item);
29     return res;
30   }
31   ...
32   public void method_under_test() {
33     if (distance(exp_state[0], act_state[0])==0 &&
34       distance(exp_state[1], act_state[1])==0 &&
35       distance(exp_result[0], act_result[0])==0 &&
36       distance(exp_result[1], act_result[1])==0)
37       ; // target
38   }

40   public void set_results(int res[]) {
41     for (int i = 0 ; i < 2 ; i++)
42       act_result[i] = res[i];
43   }
44 }
```

Figure 3: The stub automatically generated for the Stack class of the Java standard library

generate such sequence with EvoSuite [14], which has been modified so that its only goal is to cover the TRUE branch of method_under_test.

The generation of likely-equivalent method sequences is guided by the fitness function that quantifies the distance of each candidate sequence from satisfying the condition at lines 33-36. Since the condition is a conjunction of atomic clauses, the fitness function is the sum of the branch distances for each single clause, so that the overall distance is zero when all the clauses evaluate to TRUE. In turn, the branch distances for the atomic clauses are computed as numeric, object or string distances, depending on the involved types. When the distance involves objects, the search-based algorithm cannot guide the evolution, since comparing objects with the boolean method equals flattens the fitness landscape [19]. To overcome this problem, we resort to an object distance that quantifies the difference between two objects, similarly to what ARTOO [9] and RECORE [29] implement. Such notion of equivalence between objects is stronger and in fact *implies* the notion of behavioral equivalence.

The Driver component of our tool controls all the elements described so far, and drives the whole process towards the synthesis of a candidate equivalent sequence as follows: *(i)* it generates the initial set of scenarios by means of the Execution Scenarios Generator, or it loads the initial scenarios if these are available, *(ii)* it generates the stub class, and *(iii)* it invokes EvoSuite to generate a sequence of method invocations that tries to cover the target branch in method_under_test, after saving the results of the execution by calling set_results.

In an attempt to find an equivalent sequence for method pop of class Stack, the driver may generate the following sequence of method calls:

```
1 Stack_Stub x0 = new Stack_Stub();
2 int x1[] = x0.remove(0);
3 x0.set_results(x1);
4 x0.method_under_test();
```

which in turn can be automatically transformed into the candidate sequence:

$$\text{stack.pop()} \equiv \text{stack.remove(0)}$$

This candidate expresses the equivalence between pop(), which removes the object on top of the stack and returns such object, and remove(0), which removes the first element in the stack and returns it. This equivalence holds only because the first and the last elements in the two scenarios considered above are the same (two integer values equal to 1). This equivalence, though, does not hold in other scenarios. The next section describes how the second phase can invalidate such a spurious candidate.

4.3 Second Phase: Candidate Validation

The second phase validates the candidate equivalence synthesized in the first phase by considering other execution scenarios. More precisely, this phase aims to identify a scenario for which the equivalence does not hold.

Similarly to the first phase, the prototype automatically generates a method_under_test containing a single branch asserting the *non equivalence* between method m and the synthesized candidate sequence. The prototype then automatically includes such method in the declaration of class C. For instance, this is how our SBES prototype automatically transforms class Stack:

```
1  class Stack {
2    public int pop() {...}
3    public int push(int item) {...}
4    ...
5    public void method_under_test() {
6      Stack stack = deepClone(this);
7      int expect = this.pop();
8      int actual = stack.remove(0);
9      if (distance(this,clone)>0 || distance(expect,actual)>0)
10       ; // target
11   }
12 }
```

Similarly to the first phase, we exploit EvoSuite to automatically generate an execution covering the target branch (line 13), hence generating a counterexample for the equivalence. The original method pop is applied on object this, while the candidate sequence is applied to a clone of object this. We rely on a deep clone library to create exact copies of the current state of the object. This operation is crucial to avoid spurious results, since not all classes may contain a sound and complete implementation of the optional method clone().

For the Stack example, EvoSuite might produce the following method sequence in an attempt to cover the TRUE branch of method_under_test:

```
1 Stack x0 = new Stack();
2 x0.push(2);
3 x0.push(1);
4 x0.method_under_test();
```

which indeed provides a scenario that shows that the equivalence between pop() and remove(0) does not hold. In this scenario the first and the last elements in the Stack are different, and consequently the two operations have different effects on the Stack. As described in Algorithm 1, the process iterates, and the synthesis of a new candidate takes into account also the new execution scenario:

<Stack s=new Stack();s.push(2);s.push(1);int result=s.pop();>

In the second iteration of the first phase for this example, the stub considers three scenarios, and the size of all the arrays and the branch conditions to cover are updated accordingly.

The new iteration of the first phase may generate the following method sequence that covers the new target branch:

```
1 Stack_Array x0 = new Stack_Array();
2 int x1[] = x0.size();
3 int x2[] = ArrayUtils.add(x1, −1);
4 int x3[] = x0.remove(x2);
5 x0.set_results(x3); x0.method_under_test();>
```

thus producing the following new candidate:

```
stack.pop()  ≡  int x0 = stack.size();
                int x1 = x0-1;
                result=stack.remove(x1)
```

In this new iteration the search for a counterexample times out, and the synthesis process outputs the likely-equivalent sequence. Since a method can be equivalent to a code fragment that combines more than one method, SBES incrementally removes the methods used in the currently synthesized sequence from the stub. At each iteration SBES repeats the search process for each newly generated stub to obtain further equivalent sequences, when more exist.

5. EXPERIMENTAL EVALUATION

The evaluation of our work aimed to answer the following research questions:

RQ1 recall: Can the proposed approach correctly identify equivalent method sequences?

RQ2 precision: How often does the proposed approach wrongly identify non-equivalent method sequences as equivalent?

RQ3 performance: How efficiently can the proposed approach identify equivalent method sequences and counterexamples?

RQ4 role of counterexamples: How often do counterexamples correctly discard method sequences that are not equivalent to the target one?

Research questions RQ1 and RQ2 deal with the effectiveness of the proposed approach by considering its ability to

retrieve known equivalences (recall) and to report them with few false positives (precision). RQ3 deals with the efficiency and the practical applicability of the approach. RQ4 validates the need for the second phase of the approach to generate counterexamples and eliminate sequences that were at first wrongly identified as equivalent.

To answer RQ1 and RQ2 we resorted to the standard recall and precision metrics:

$$Recall = \frac{true\ positives}{true\ positives + false\ negatives}$$

$$Precision = \frac{true\ positives}{true\ positives + false\ positives}$$

Recall is defined as the ratio between the number of equivalent sequences correctly synthesized with the approach (true positives) and the total number of equivalent sequences, which include both the ones correctly synthesized (true positives) and the ones that the approach fails to synthesize (false negatives).

Precision is defined as the ratio between the number of equivalent sequences correctly synthesized with the approach (true positives) and the total number of sequences deemed as equivalent, which include both the equivalent ones (true positives) and the non-equivalent ones erroneously identified as equivalent by the approach (false positives).

To answer RQ3, we measured performance as the *time required to synthesize an equivalent sequence* and the *time required to find a counterexample*, since these two measures directly influence the overall performance of our approach. We use these two values to compute the optimal timeouts. In fact, we acknowledge a synthesized sequence as likely-equivalent to the target method when no counterexamples are found within a given timeout. The maximum time required to find a counterexample indicates the optimal value for the counterexample timeout: a smaller value would lead to missing some counterexamples, a larger value would cause time waste. Similarly, the maximum time required to synthesize a sequence indicates the optimal synthesis timeout, that is, the value that avoids missing sequences without wasting time.

For RQ4, we measured the role of counterexamples as the number of method sequences identified as candidates for equivalence that the counterexamples discard as false positives. This number corresponds to the number of iterations between the second and the first phase, since discarding a sequence results in re-executing the first phase.

5.1 Experimental setup

The first target of our experiments is class Stack, taken as a representative for the various containers available in the Java standard library.[2] Class Stack is particularly challenging because it contains many generics and many non trivial equivalent method sequences. We also considered a set of classes from Graphstream, a library to model and analyze dynamic graphs.[3]

Our experiments cover 15 methods of class Stack and 32 methods belonging to 6 classes of Graphstream, as reported in Table 2. Stack and Graphstream represent different application domains, are developed and maintained by different third party subjects, and include all the language characteristics that we can currently handle with EvoSuite, which constrains our prototype implementation.

We ran the experiments by feeding the prototype with the class under analysis, the target method and an initial scenario. The target method is the method of the class under analysis for which we would like to synthesize equivalent method sequences. The initial scenario consists of one test case that was either extracted from the existing test suite, or generated automatically with EvoSuite, depending on the availability of the test suites. We gave a search budget of 180 seconds to both the first and the second phase.

To answer RQ1 and RQ2 we compared the sequences that we synthesized automatically against the set of sequences that we previously identified with manual inspection within the limits of the current prototype. In theory, the amount of equivalent sequences would be infinite, since we can easily combine simple equivalent sequences to obtain new ones. For example, method pop() is equivalent to remove(size()-1), but is also equivalent to push() pop() remove(size()-1). In our experiments we considered only *minimal* equivalences that we informally identify as those that cannot be derived by suitably combining simpler equivalences or adding method calls with a globally null effect, as the pair push() pop() in the previous example.

In our experiments, we synthesized equivalent method sequences for single methods only. Synthesizing equivalent sequences for method sequences does not change the problem, but simply augments the size of the experiment. Our automatic synthesis is limited by EvoSuite that can deal with method calls, constructors, primitive values and arrays, but not with all the arithmetic operators, loops and conditional statements. These limitations are inherited from EvoSuite itself, and do not belong to the approach that can synthesize equivalent sequences for general method sequences, potentially exploiting all language constructs.

We repeated the experiments 30 times because of the random nature of search-based algorithms, and we considered both the maximum, averaged and cumulative results over the 30 executions. The maximum result is the one obtained from the best execution, while the average result gives the result expected from a single execution. The cumulative result aggregates results from all 30 executions.

The execution environment provides a listener that logs detailed information about the timing of the events. Each iteration consists of creating a stub, compiling and executing it. The listener logs the compilation and execution time, recording the execution time of both the prototype and EvoSuite. These data allowed us to compute all the performance metrics discussed above.

5.2 Results

In this section we discuss the experimental results.[4] We ran our prototype on 47 methods of 7 classes taken from the Stack Java Standard Library and the Graphstream library. We automatically synthesized 123 equivalent method sequences, which represent more than 87% of the 141 sequences that we manually identified ahead by inspecting the classes documentation. We considered only the *minimal* equivalences,

[2]http://docs.oracle.com/javase/7/docs/api/java/util/Stack.html

[3]http://graphstream-project.org/

[4]A replication package, containing both the subjects and the data collected during our experiments, is available at http://star.inf.usi.ch/sb-synthesis

Table 1: Sample sequences synthesized with SBES

Original sequence	Synthesized sequence
java.util.Stack	
addElement(Object e)	add(e)
	push(e)
	add(e,size())
	Collection c=new Collection(); c.add(e); addAll(c);
clear()	removeAllElements()
	setSize(0)
	Collection c=new Collection(); retainAll(c);
e = pop()	e=peek(); index=size()-1; removeElementAt(index);
e = set(int i, Object o)	e=remove(i); insertElementAt(o,i)
org.graphstream.graph.implementations.Edge	
getSourceNode()	temp=getTargetNode(); getOpposite(temp)
getTargetNode()	temp=getSourceNode(); getOpposite(temp)
org.graphstream.graph.implementations.SingleNode	
getAttribute(String s)	getAttribute(s,Object.class)
org.graphstream.ui.geom.Vector2	
fill(double d)	Vector2 v=new Vector2(); v.set(d,d); copy(v);
	Point2 p=new Point2(d,d); set(p.x, p.y);
	scalarAdd(d)
org.graphstream.ui.geom.Vector3	
copy(Vector3 v)	Point3 p=new Point3(); p.move(v); set(p.x,p.y,p.z)

and we excluded those that could not be found due to the limitations of our prototype.

Table 1 presents a sample of the equivalent sequences that we synthesized automatically. SBES can synthesize both simple equivalences, e.g. methods that can be replaced interchangeably, and complex equivalences that include non trivial combinations of method calls, as in the case of Collection c=new Collection(); c.add(e); addAll(c); that is equivalent to addElement(e).

Table 2 summarizes the experiment results. For each of the analyzed methods, the table shows the following information: *(i)* the number of *minimal* equivalent sequences identified with manual inspection (column Tot), which we use as baseline, *(ii)* the amount of equivalent sequences automatically synthesized in at least one run (column Max_t), *(iii)* the maximum amount of equivalent sequences synthesized with a single run (column Max_r), *(iv)* the average amount of equivalent sequences identified in the 30 runs (column Avg), *(v)* the precision (Prec) and the recall (Rec) computed over the 30 runs.

Table 2 shows that in all those cases where a target method has multiple equivalent sequences our approach can synthesize a substantial fraction—if not all—of the equivalences, even within a single run. This is a very interesting result, since all of the practical applications of redundancy typically benefit from a high level of redundancy [4, 5, 6, 7]. In the Stack case study, when SBES was not able to synthesize all the manually identified equivalences, we observed that sometimes the correct equivalent sequence was indeed synthesized during the evolution of the individuals. However, the objects holding the correct results were not used as parameters of the set_results method, and therefore the search did not stop.

Table 2: RQ1, RQ2: Effectiveness of the approach

Case Study	Tot	Synthesized			Prec	Rec
		Max_t	Max_r	Avg		
java.util.Stack						
add(int,Object)	2	2	2	0.43	1.00	1.00
add(Object)	6	3	3	2.10	1.00	0.50
addElement	6	4	3	2.17	1.00	0.70
clear	3	3	3	2.77	0.99	1.00
elementAt	1	1	1	0.90	0.79	1.00
firstElement	2	2	2	1.57	0.89	1.00
get	1	1	1	0.80	0.80	1.00
indexOf	2	2	2	1.70	0.93	1.00
lastElement	4	2	2	1.17	0.97	0.50
peek	2	2	2	1.23	0.97	1.00
pop	2	2	2	0.60	1.00	1.00
push	6	2	2	2.00	1.00	0.33
remove(Object)	4	2	1	0.80	0.92	0.50
remove(int)	2	2	2	0.70	1.00	1.00
set	2	2	2	0.50	0.65	1.00
org.graphstream.graph.Path						
getEdgeCount	2	2	2	1.80	0.73	1.00
getNodeCount	3	3	3	2.77	0.73	1.00
org.graphstream.graph.implementations.Edge						
getNode0	2	2	2	2.00	1.00	1.00
getSourceNode	2	2	2	2.00	0.85	1.00
getNode1	2	2	2	1.97	1.00	1.00
getTargetNode	2	2	2	2.00	0.80	1.00
changeAttribute	2	2	2	2.00	1.00	1.00
setAttribute	2	2	2	2.00	1.00	1.00
addAttribute	2	2	2	2.00	1.00	1.00
getAttribute	3	3	3	1.57	1.00	1.00
getFirstAttribute	3	3	3	1.80	1.00	1.00
org.graphstream.graph.implementations.SingleNode						
changeAttribute	2	2	2	2.00	1.00	1.00
setAttribute	2	2	2	2.00	1.00	1.00
addAttribute	2	2	2	2.00	1.00	1.00
getAttribute	3	3	3	0.63	1.00	1.00
getFirstAttribute	3	3	3	1.50	1.00	1.00
org.graphstream.graph.implementations.MultiNode						
changeAttribute	2	2	2	2.00	1.00	1.00
setAttribute	2	2	2	2.00	1.00	1.00
addAttribute	2	2	2	2.00	1.00	1.00
getAttribute	3	3	3	1.20	1.00	1.00
getFirstAttribute	3	3	3	1.80	1.00	1.00
org.graphstream.ui.geom.Vector2						
x	1	1	1	0.23	0.50	1.00
y	1	1	1	0.27	1.00	1.00
set	5	5	5	1.37	1.00	1.00
fill	10	10	7	3.87	0.97	1.00
copy	4	4	3	1.23	0.93	1.00
org.graphstream.ui.geom.Vector3						
x	1	1	1	0.10	1.00	1.00
y	1	1	1	0.27	0.89	1.00
z	1	1	1	0.20	0.86	1.00
set	5	5	3	0.40	1.00	1.00
fill	10	10	6	0.14	0.10	1.00
copy	4	4	4	2.27	0.99	1.00

Table 3: RQ3: Efficiency of the approach

Case Study	Synthesis	Counter example	Minimum Timeout
java.util.Stack	18.0s	11.0s	76s
graphstream.Path	20.0s	15.0s	60s
graphstream.Edge	16.0s	6.0s	7s
graphstream.Node	16.0s	6.0s	7s
graphstream.MultiNode	20.0s	8.0s	9s
graphstream.Vector2	15.0s	7.0s	36s
graphstream.Vector3	18.0s	6.0s	29s

We are currently working on improving the evolution process to make use of any object available in the current method sequence, instead of arbitrarily choosing one.

In summary, precision and recall are high, almost always close to or equal to one, indicating that the proposed approach can retrieve most of the known equivalent sequences with a low number of false positives. Therefore, we can answer positively to both RQ1 and RQ2:

> **RQ1, RQ2**: The proposed approach can correctly identify one and often more than one equivalent method sequences, with recall and precision which are close or equal to one in most of the cases.

Table 3 reports the efficiency metrics. Column *Synthesis* shows the time required to synthesize an equivalent sequence, while column *Counterexample* reports the time for the counterexample generation. Column *Minimum Timeout* shows the minimum timeout that can be set to the counterexample phase without altering the effectiveness of the approach, that is the precision and recall values reported in Table 2. Columns *Synthesis* and *Counterexample* report the median of the values computed over the 30 runs across the target methods of each class (we aggregate performance data by class to save space; the interested reader can find the detailed data in our replication package). Column *Minimum Timeout* reports the worst computation time experienced in the experiments during the counterexample phase. The table reports only the counterexample timeout because the synthesis timeout is always lower than the counterexample one, and thus the counterexample timeout represents an upper bound for the performance of the approach. The execution time is acceptable and compatible with the typical usage scenarios in which redundancy is needed. In fact, even when equivalent sequences are used at runtime, for example in self-healing applications, the synthesis of equivalent sequences can be carried out in advance, offline. Hence, we can answer positively to research question RQ3:

> **RQ3**: The proposed approach requires a total execution time that is compatible with the typical application scenarios, where redundancy can be identified offline.

Table 4 reports the data about the effectiveness of the counterexamples: column *False Positives* indicates the amount of sequences that were erroneously identified as equivalent and were not automatically discarded with a counterexample. Column *Discarded* indicates the amount of overfitted candidate solutions that are identified as non-equivalent by a counterexample, and column *Efficiency* indicates the percentage of sequences automatically discarded with counterexamples.

Table 4: RQ4: Effectiveness of counterexamples

Class	False Positives	Discarded	Efficiency
java.util.Stack	36	201	84.81%
graphstream.Path	50	22	30.55%
graphstream.Edge	26	87	76.99%
graphstream.Node	0	0	-
graphstream.MultiNode	0	0	-
graphstream.Vector2	13	34	72.34%
graphstream.Vector3	40	36	47.36%

The table indicates that counterexamples are extremely effective in identifying and removing many method sequences erroneously proposed as equivalent, from 30% in the worst case to over 80% in the best case. *False positives* include both sequences for which EvoSuite fails to find a counterexample and sequences for which EvoSuite crashes silently due to a NullPointerException before completing the search. This last category corresponds to about 30% of the false positives, and we expect to significantly reduce them by solving the cause of the exceptions in EvoSuite. We can thus positively answer research question RQ4:

> **RQ4**: Counterexamples can discard a relevant amount of method sequences erroneously identified as equivalent to the target one, and can thus reduce significantly the number of reported false positives.

5.3 Threats to Validity

The main threats that affect the validity of the empirical study described above are the authors' bias and the external validity threats.

Authors' bias: The authors have manually identified the reference set of equivalent sequences used to assess the effectiveness of the approach. Such task was carried out before running SBESto avoid any influence from SBESoutput. Moreover, one author has cross-checked the equivalent sequences identified by another author to verify that no equivalent sequence was missed and that the identified equivalent sequences were correct.

External validity: We have validated our approach on 7 classes and 47 methods, taken from two real-world subjects. Different results could be obtained for different systems. We have chosen two subjects, java.util and Graphstream, that were known to contain some degree of redundancy in their implementation. By construction, our approach will not produce any valuable result on systems that do not include any redundancy at all.

The selection of the two subjects used in the experiments was driven exclusively by prior knowledge about the presence of redundancy. Hence, we expect our approach to behave similarly on other systems having a comparable degree and kind of redundancy. On the other hand, specific implementation details might affect the performance of search-based generators in finding candidate method sequences or counterexamples. For instance, the use of generic types in class Stack represented a technological obstacle that required some tool adaptation. We have tried to choose the subjects used in the reported experiment so as to maximize their diversity. The only way to further reduce the external validity threat consists of replicating our study on more subjects. For this reason we make our experimental package publicly available to other researchers.

6. RELATED WORK

The technique proposed in this paper uses a search-based approach to automatically synthesize method sequences that are equivalent to a target method. Relevant related work can be found in the areas of automatic inference of specifications and search-based techniques to synthesize redundancy.

6.1 Specification Inference

Specification mining finds its roots in the pioneer work of Ernst et al. who proposed Daikon to infer likely program invariants from a finite set of executions [12]. Dysy improves the quality of the invariants by exploiting dynamic symbolic execution [10]. Similarly, the feedback loop framework proposed by Xie and Notkin refines the likely invariants inferred with Daikon by feeding them to a test generator, and by using the newly generated executions to refine the invariants [31]. Our work shares the idea of using a finite set of executions to infer some information about the program, but these techniques infer program invariants, while we infer equivalence among operations.

Other specification mining techniques infer finite state machine models of software components. Mariani et al. infer models representing the protocol of components [24]. Pradel et al. can generate similar models, but target mainly multi-object protocols [28]. Ghezzi et al. build finite state machines that model the partial behavior of components, and then make such models more general via graph transformation rules [17]. Dallmeier et al., instead, infer finite state machines representing the objects behavior, and exploit test case generation to explore unobserved behavior [11]. More recently, Beschastnikh et al. proposed a framework to specify model inference algorithms declaratively [3]. Although finite state machines can express equivalence among different operations, they typically abstract from the concrete events observed in the program execution. Consequently, the event sequence equivalences that can be obtained from the inferred models hold for the abstract state, but not necessarily for the complete, concrete one.

The work of Henkel and Diwan is the most closely related to ours. They use reflection to get the list of methods in a Java class, and they generate executions to infer algebraic specifications for such class [20]. The axioms that they generate to describe the behavior of the class include information that can be used to infer the equivalence of method sequences. However, inferring the equivalence of method sequences is not their primary goal. Differently from them, we focus on the synthesis of equivalent method sequences, trying to produce as many different equivalences as possible. This affects the sequence generation approach: They generate method invocations randomly, while we employ a search-based technique to generate only those sequences that are relevant for synthesizing equivalence. Moreover, our two-phase approach reduces the number of invalid equivalences, while they have less chances to invalidate incorrect axioms.

6.2 Search-based Techniques for Redundancy

Search-based techniques have been employed in multiple domains to solve different problems. Test case generation is one of such domains, and search-based techniques have shown their potential in producing test suites that achieve high coverage according to specific criteria [1, 15, 30].

The work closest to ours is related to search-based techniques to automatically synthesize redundancy. Feldt uses genetic programming to automatically generate program variants for fault tolerance techniques like N-version programming and recovery blocks [13]. Such variants adhere to the specification of the original program, but are different enough to tolerate faults. Similarly, Benoit et al. generate "sosies", which are variants generated by adding, removing and replacing statements in the original program [2]. Sosies provide the same expected functionality as the original program, while exhibiting different executions. Langdon and Harman produce variants of an original program with different non-functional requirements [23].

These techniques *generate* redundancy in the sense that they synthesize programs that are slightly different from the original one, either in their functional or non functional behavior. Our technique, instead, aims to *identify* redundancy that already exists in the considered components.

7. CONCLUSIONS

Software redundancy that derives from the presence of equivalent method sequences finds many interesting applications that span from testing to fault tolerance and self-healing. The different approaches that exploit equivalent method sequences rely on manual identification of the equivalence, and this limits their applicability and scalability.

In this paper, we propose a novel technique that exploits search-based algorithms to infer equivalent method sequences. The approach is fully automatic and applies to any method sequence. In this paper we report the experimental results obtained with a prototype that implements the approach to infer method sequences equivalent to single methods. The results obtained for 47 methods belonging to two different libraries are extremely positive. We can automatically synthesize more than 87% of the known equivalences that include many non trivial combinations of method calls with a negligible number of false positives.

Our prototype implementation uses EvoSuite and inherits from it some limitations. We are currently working on relaxing the limitations imposed by EvoSuite to both widen the experimental scope and consolidate the validation results. We are also working on extending the automatic synthesis of equivalent method sequences beyond single methods to be able to identify additional equivalent elements in software systems. Finally, we are studying different applications of intrinsic redundancy beyond the results obtained so far in the automatic generation of self-healing systems [5] and test oracles [4].

8. ACKNOWLEDGMENTS

The authors would like to thank Gordon Fraser for providing the source code of EvoSuite; Fitsum Meshesha Kifetew for the useful advice on the implementation of the approach; Jens Krinke for discussing the idea of this paper during the Dagstuhl seminar #13061. This work was partially supported by the Swiss National Science Foundation projects *SHADE* (grant n. 200021-138006) and *ReSpec* (grant n. 200021-146607), and by the European Research Council Advanced Grant *SPECMATE* (n. 290914).

9. REFERENCES

[1] L. Baresi, P. L. Lanzi, and M. Miraz. Testful: An evolutionary test approach for java. In *IEEE International Conference on Software Testing,*

Verification and Validation (ICST), pages 185–194, 2010.

[2] B. Baudry, S. Allier, and M. Monperrus. Tailored source code transformations to synthesize computationally diverse program variants. In *ACM International Symposium on Software Testing and Analysis (ISSTA)*, pages 149–159, 2014.

[3] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy. Unifying FSM-inference algorithms through declarative specification. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2013.

[4] A. Carzaniga, A. Goffi, A. Gorla, A. Mattavelli, and M. Pezzè. Cross-Checking Oracles From Intrinsic Software Redundancy. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2014.

[5] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezzè. Automatic recovery from runtime failures. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 782–791, 2013.

[6] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè. Automatic workarounds for Web applications. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, pages 237–246, 2010.

[7] A. Carzaniga, A. Gorla, and M. Pezzè. Handling software faults with redundancy. In R. de Lemos, J.-C. Fabre, C. Gacek, F. Gadducci, and M. H. ter Beek, editors, *Architecting Dependable Systems VI*, LNCS, pages 148–171. Springer, 2009.

[8] T. Y. Chen, F.-C. Kuo, T. H. Tse, and Z. Q. Zhou. Metamorphic testing and beyond. In *International Workshop on Software Technology and Engineering Practice (STEP)*, pages 94–100, 2003.

[9] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. ARTOO: adaptive random testing for object-oriented software. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 71–80, 2008.

[10] C. Csallner, N. Tillmann, and Y. Smaragdakis. Dysy: Dynamic symbolic execution for invariant inference. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 281–290, 2008.

[11] V. Dallmeier, N. Knopp, C. Mallon, G. Fraser, S. Hack, and A. Zeller. Automatically generating test cases for specification mining. *IEEE Transactions on Software Engineering (TSE)*, 38(2):243–257, 2012.

[12] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering (TSE)*, 27(2):99–123, 2001.

[13] R. Feldt. Generating diverse software versions with genetic programming: an experimental study. *IEE Software Engineering*, 145(6):228–236, 1998.

[14] G. Fraser and A. Arcuri. EvoSuite: automatic test suite generation for object-oriented software. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, pages 416–419, 2011.

[15] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering (TSE)*, 39(2), 2013.

[16] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 321–330, 2008.

[17] C. Ghezzi, A. Mocci, and M. Monga. Synthesizing intensional behavior models by graph transformation. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 430–440. IEEE Computer Society, 2009.

[18] M. Harman. The current state and future of search based software engineering. In *Future of Software Engineering (FOSE)*, pages 342–357, 2007.

[19] M. Harman, L. Hu, R. M. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.

[20] J. Henkel, C. Reichenbach, and A. Diwan. Developing and debugging algebraic specifications for Java classes. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(3):14:1–14:37, 2008.

[21] M. Hennessy and R. Milner. On observing nondeterminism and concurrency. In *Proceedings of the 7th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 299–309, 1980.

[22] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *ACM International Symposium on Software Testing and Analysis (ISSTA)*, pages 81–92. ACM, 2009.

[23] W. Langdon and M. Harman. Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation (TEC)*, 2014.

[24] L. Mariani, F. Pastore, and M. Pezzè. Dynamic analysis for diagnosing integration faults. *IEEE Transactions on Software Engineering*, 37(4):486–508, 2011.

[25] P. McMinn. Search-based software test data generation: a survey. *Journal of Software Testing, Verification and Reliability (STVR)*, 14:105–156, 2004.

[26] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for Java. In *Conference on Object-Oriented Programming Systems and Applications (OOPSLA)*, pages 815–816, 2007.

[27] R. Pargas, M. J. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Journal of Software Testing, Verification and Reliability (STVR)*, 9:263–282, 1999.

[28] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross. Statically checking API protocol conformance with mined multi-object specifications. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 925–935, 2012.

[29] J. Rößler, A. Zeller, G. Fraser, C. Zamfir, and G. Candea. Reconstructing core dumps. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 114–123, 2013.

[30] P. Tonella. Evolutionary testing of classes. In *ACM International Symposium on Software Testing and Analysis (ISSTA)*, pages 119–128, 2004.

[31] T. Xie and D. Notkin. Mutually enhancing test generation and specification inference. In *International Workshop on Formal Approaches to Testing of Software (FATES)*, pages 60–69, 2003.