Automatic Synthesis of Test Cases to Identify Software Redundancy

Matteo Brunetto

University of Milano-Bicocca

Abstract

Software system can include redundant implementation elements, such as, different methods that can produce indistinguishable results. This type of redundancy is called 'intrinsic' if it is already available in the software, although not intentionally planned. Redundancy can be a key element to increase the reliability of a system. Some fault tolerance and self-healing techniques exploit the redundancy to avoid failures at runtime. Unfortunately, inferring which operations are equivalent manually can be expensive and error prone.

A technique proposed in previous work allows to automatically synthesizes method sequences that are equivalent to a target method. However this technique needs an execution scenario to work. Currently, this execution scenario is generated manually that is expensive and makes the technique hard to use.

This paper proposes a technique to generate execution scenarios for a target method for which we are searching equivalent sequences. The experimental results obtained on the Java class `Stack` *show that the proposed approach correctly generates execution scenarios within reasonable execution time. Besides, the execution scenarios generated allow to maximize the effectiveness of the technique described above.*

1. INTRODUCTION

Modern software is redundant. For example, it can be redundant because redundant components are introduced in the system to obtain fault-tolerant systems. A new aspect is that software systems can be *intrinsically* redundant due to the presence, not deliberately planned, of equivalent code fragments, for example methods or method sequences. Two methods are *equivalent* if they can produce indistinguishable results. For example, method `pop()` of the Java class `Stack` is equivalent to the method sequence `remove(size()-1)`. Removing the element on the top of the stack leads to the same result as removing the element in the last position.

Equivalent method sequences find many useful applications, from the automatic generation of test oracle [1], to the design of self-healing system [2, 3]. In these applications, the equivalence must be identified manually. This is a non-trivial activity that may represent an obstacle to the practical applicability of these technique.

Gorla et al. proposed a search-based technique that, given a target method and a initial set of execution scenarios, automatically synthesizes method sequences that are equivalent to the target method. [4]. This technique has been implemented in a Java prototype tool called SBES that needs an execution scenario to work. The execution scenario is generated manually that is expensive and makes the technique of Gorla et al. hard to use

In this paper we propose a technique that can automate this activity. Given a target method and a target class, the technique automatically generates execution scenarios for the target method. The generation proceeds in two phases. In the first phase, the goal is to identify a list of methods of the target class that can allow to maximize the effectiveness of SBES. In the second phase, the goal is to generate test cases that contain only methods defined in the first phase and then convert them in execution scenarios.

The technique that we propose in this pa-

per is fully automatic and requires as input only the method for which we are searching equivalent sequences and its class. Our experiments indicate that the technique is effective in generating execution scenarios, and the execution time is reasonably efficient. On 15 methods of the Java class **Stack**, our approach always generates at least 1 execution scenario and an average of 26 execution scenarios for each method. Besides, on 6 methods of the Java class **Stack**, the effectiveness of SBES is similar both using execution scenarios generated automatically with the proposed approach and using execution scenarios generated manually by experts.

This paper is organized as follows. Section 2 introduces the software redundancy, describes the SBES approach, and discusses the aspects that affect the effectiveness of SBES. Section 3 presents the details of our technique. Section 4 discusses the experimental results obtained to validate the proposed approach. Section 5 summarizes the results presented in the paper and illustrates future developments.

2. IDENTIFYNG SOFTWARE REDUNDANCY

2.1. Software Redundancy

A software system is *redundant* if the execution of different methods or combinations of methods leads to indistinguishable result. Two executions lead to *indistinguishable* results if they produce the same output and state. For our purpose, we are considering a type of *observational* equivalence where the state produced by the executions may be internally different but not externally distinguishable by analysing the system through its public interface [5]. For example, the methods `get` and `elementAt` of the Java class `Stack` return the same object of the stack. We say that two methods are *equivalent* if they produce indistinguishable results for all possible inputs, as in the previous example.

Redundancy can be explicitly added to a software system to increase reliability. There is another type of redundancy that already exists in the software. This type of redundancy is called *intrinsic redundancy* and it is present due to modern design practices like backward compatibility and design for reusability [3]. A library might contain different version of the same component to ensure compatibility with previous system. For example, the Java 7 standard library contains at least 365 methods that are deprecated and that overlap with functionality of newer methods. Modern development practices induce developers to use external libraries that already implement the needed functionality. It is possible to find several libraries that provide similar functionalities. For example, the Guava library implements collections that are similar to the Java standard library. In this work we focus on equivalent methods and combinations of methods in the same component.

This type of redundancy finds many interesting applications that span from fault tolerance [1] to self-healing [2, 3]. However these techniques rely on manual identification of the equivalence, and this limits their applicability.

Gorla et al. proposed a technique to automatically identify equivalent methods and combinations of methods by exploiting genetic algorithms [4].

2.2. Synthesis of Equivalent Methods Sequences

In this section we describe the technique of Gorla et al. [4] in details. This technique synthesizes a sequence of method invocations that is equivalent to a target method m on a finite set of *execution scenarios* by means of a two-phase iterative process.

It starts with a initial set of execution scenarios that represent a sample of the input space of m. The initial set of execution scenarios may be as simple as a single test case.

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 methods, and terminates with an invocation of method m. Test Case 1 is an example of execution scenario for the method `pop()` of

the class `Stack`.

$$< \text{Stack s=new Stack();s.push(1);int result=s.pop()} > \quad (1)$$

In the first phase, the technique uses genetic algorithms to generate an equivalent candidate *eq* for the given set of execution scenarios. In the second phase, the technique validates *eq* by using genetic algorithms to find a counterexample, which corresponds to an execution sequence for which *eq* and *m* are not equivalent. This phase is necessary because the candidate might be valid only on a specific scenario. If it finds a counterexample, it adds the counterexample to the set of execution scenarios, and it iterates through the first phase looking for a new candidate. Otherwise, if no counterexample is identified, it has successfully synthesizes a method sequence *eq* that is equivalent to *m*.

This technique has been implemented in a Java prototype tool called SBES.

The experiments indicate that the technique is effective in synthesizing equivalent method sequences within reasonable execution time. On 47 methods belonging in 7 different classes for which equivalent method sequences were know a priori, the technique synthesizes 87% of the equivalences and one or more equivalent sequences for each target method, with few false positive. [4]

SBES needs an execution scenario to work. Currently, the execution scenario is manually generated that is expensive in term of time and makes intrinsic redundancy hard to use.

2.3. Effectiveness of SBES

The effectiveness of SBES depends on the execution scenarios, especially on the size of the *object graphs* in the execution scenarios. The *size* of the object graph is the number of nodes of the object graph whose root is an instance of the target class for which we are searching equivalent sequences.

We experimented with 2 methods of the class `Stack` as reported in Table 1. We ran the experiments by feeding SBES with the class `Stack`, the target method and 12 execution scenarios with different number of element from 0 to 11. We repeated the experiments 30 times because of the random nature of genetic algorithms.

For each execution scenario, the table shows the following information: *(i)* the average amount of equivalent sequences identified in the 30 runs (column **Avg**), *(ii)* the precision (**Prec**) and *(iii)* the number of iterations needed to identify the equivalent sequence `removeAllElements()` for the method `clear()` and `add(Object)` for the method `push(Object)` (column **Iterations**).

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).

$$\frac{\text{true positives}}{\text{true positives} + \text{false positives}}$$

Table 1 indicates that we need a stack object that contains between 5 and 8 elements to maximize the effectiveness of SBES. Between 5 and 8 elements we obtain the maximum number of equivalent method sequences, with a precision of 100% and only 1 iteration required.

3. SYNTEHSIS OF EXECUTION SCENARIOS

We propose a technique that can automate the creation of initial execution scenarios by means of a two-phase process. We start with the method *m* for which we are searching equivalent sequences and its class *c*. In the first phase, we generate a list of methods of *c* that can maximizes the effectiveness of SBES (see section 2.3). In the second phase, we generate test cases that contain only methods defined before. Then, these test cases are converted in execution scenarios.

The process for generate execution scenarios for a target method *m* and a target class *c* is detailed in Algorithm 1.

In the first phase (lines 1-8) the algorithm extracts methods of *c* (line 1) and puts them

Table 1: *Effectiveness of SBES.*

Clear			
Num. Elements	**Equivalence Synthesized**		**Iterations**
	Avg	**Prec**	
0	0	-	-
1	2.3	0.71	2
2	2.63	0.72	2
3	2.67	0.74	1
4	2.96	0.90	1
5	3	1	1
6	3	1	1
7	3	1	1
8	3	1	1
9	2.60	1	1
10	2.30	1	1
11	2.13	1	1

Push			
Num. Elements	**Equivalence Synthesized**		**Iterations**
	Avg	**Prec**	
0	0	-	-
1	0.8	0.51	2
2	0.83	0.58	2
3	1.2	0.64	2
4	1.97	0.87	1
5	2	0.90	1
6	2	0.90	1
7	2	0.90	1
8	2	0.90	1
9	1.7	0.90	1
10	1.6	0.95	1
11	1.23	1	1

in a list called *methods*. Then, it searches pure methods (line 2) and methods that decrease the size of the object graph (lines 4-8) and removes them from *methods*. The function `REMOVE-METHOD` compares the method we want to remove from *methods* with *m* and if they are equal does not remove it.

Methods can be divided into two categories: *pure methods* and *impure methods*. A method is pure if it does not change the state of the object. Otherwise, a method is impure if it changes the state of the object [7]. Impure methods are divided into three categories defined by us: *methods that increase the size of the object graph*, *methods that decrease the size of the object graph* and *methods that change the nodes in the object graph*.

To maximise the effectiveness of SBES we must exclude pure methods and methods that decrease the size of the object graph from the generation of execution scenarios. Pure methods do not change the size of object graph while methods that decrease the size of the object graph can generate a stack object with less than 5 elements.

In the second phase (lines 9-21) the algorithm generates test cases that contain only the methods defined in *methods* (line 9). Then, these test cases are converted in execution scenarios (line 11). The function `NORMALIZE-TEST-CASE` reduces the test case by eliminating the unnecessary instructions.

The main iteration terminates when the algorithm generates a set of execution scenarios (line 17) or if there are not execution scenarios (line 19).

We implemented the algorithm illustrated above in a Java prototype tool called ESG (Execution Scenarios Generator). Figure 1 shows the main components of ESG. The `Methods Finder` component generates the list of methods by removing pure methods and methods that decrease the size of the object graph. The `Execution Scenarios Creator` component generates test cases by invoking Randoop and converts them into execution scenarios.

[1]https://code.google.com/p/type-inference/

Algorithm 1: *Synthesis of execution scenarios.*

```
   INPUT: c, m
1  methods := EXTRACT−METHODS(c)
2  pureMethods := FIND−PURE−METHODS(↩
       c)
3  REMOVE−METHOD(methods,pureMethods↩
       )
4  for each method in methods do
5    if DECREMENT−NODE(method) then
6      REMOVE−METHOD(methods,method)
7    end if
8  end for
9  testCases := SYNTHESIZE−TEST−CASES(↩
       methods)
10 for each test in testCases do
11   es := NORMALIZE−TEST−CASE(test)
12   if NUM−ELEMENTS(es) >= 5 and NUM↩
         −ELEMENTS(es) <=8 then
13     executionScenarios += es
14   end if
15 end for
16 if executionScenarios != 0 then
17   return executionScenarios
18 else
19   return NIL
20 end if
```

In the next sections we detail the key components of ESG, and describe the generation process.

3.1. First Phase: Generation of the List of Methods

The first phase generates a list of methods of the target class *c* that can maximizes the effectiveness of SBES. For this task, the prototype relies on the `Methods Finder` component. The `Methods Finder` finds pure methods of *c* by invoking Type-Inference[1] and removes them from the list of all methods of *c*. Then it finds methods that decrease the size of the object graph by pattern matching approach and removes them from the list of all methods of *c*. For this task, the `Methods Finder` uses a file manually generated called *blacklist*. This file contains the most common pattern of methods that decrease the size of the object graph. The `Methods Finder` looks for methods that con-

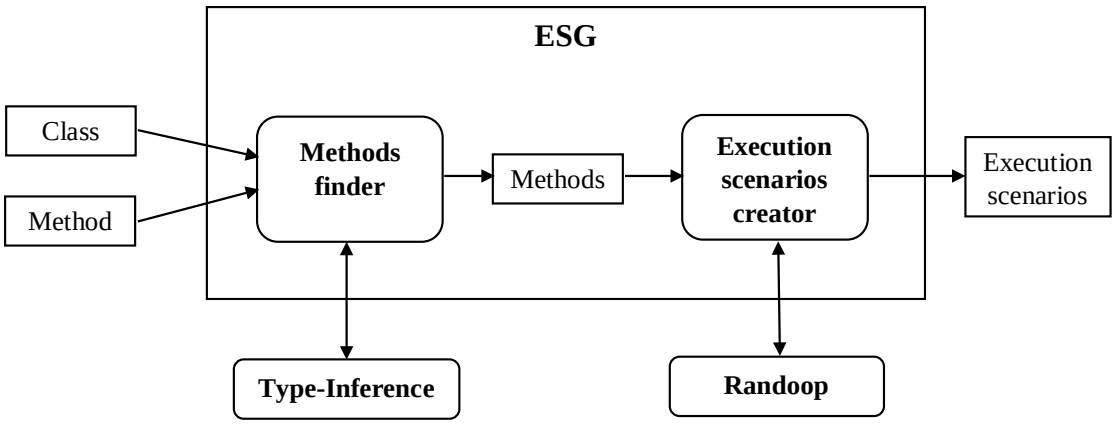

Figure 1: *Main components of ESG.*

tain in their name the pattern defined in the *blacklist*.

The following list is an example of pure methods identified for the class `Stack` and its superclass `Vector`:

```
Stack.empty()
Stack.peek()
Stack.search(Object)
Vector.capacity()
Vector.clone()
Vector.contains(Object)
...
```

The following list is an example of the common pattern for the class Stack contained in the *blacklist*:

```
remove
clear
retain
pop
setSize
```

The *Methods Finder* uses this list to remove the following methods that decrease the size of the object graph:

```
Vector.clear()
Vector.remove(int)
Vector.remove(Object)
Vector.removeAll(Collection)
Vector.removeAllElements()
Vector.removeElement(Object)
Vector.removeElementAt(int)
Vector.retainAll(Collection)
Vector.setSize(int)
Stack.pop()
```

Given the method `pop`, the `Methods Finder` uses the previous lists to generate the list of methods that can maximizes the effectiveness of SBES. The resulting list is reported below:

```
Stack.push(Object)
Stack.pop()
Vector.add(int,Object)
Vector.add(Object)
Vector.addAll(int,Collection)
Vector.addAll(Collection)
Vector.addElement(Object)
Vector.set(int,Object)
Vector.insertElementAt(Object,int)
Vector.setElementAt(Object,int)
```

The reader should notice that method `pop` is a method that decreases the size of the object graph. However we need this method because it is the method for which we are searching equivalent sequences.

3.2. Second Phase: Generation of Execution Scenarios

The second phase generates test cases that contain only the methods defined previously in

the first phase and converts them in execution scenarios. For this task, the prototype relies on the `Execution Scenarios Creator` component. The `Execution Scenarios Creator` creates test cases from c by invoking Randoop.[2] Randoop is an automatic test case generator for Java. Randoop accepts as input a list of the methods to be used to seed the test generation process [6].

Given the list generated in the first phase, the `Execution Scenarios Creator` creates test cases (by invoking Randoop) that belong to one of the following categories:

1. Test case that does not include the target method m.

Given m =`pop()` we can see in Test Case 2 an example of a test case from the first category.

Test Case 2: *Test case form the first category.*

```
1  Stack stack0 = new Stack();
2  stack0.addElement((Object)10);
3  Object obj0 = stack0.push((Object)1);
4  Stack stack1 = new Stack();
5  boolean b0 = stack0.addAll((Collection)↩
       stack1);
6  stack0.add(0, (Object)(−1));
7  try {
8    Object obj1 = stack0.set(10, (Object)10);
9    fail(''Expected exception of type java.lang.↩
         ArrayIndexOutOfBoundsException'');
10 } catch (java.lang.↩
       ArrayIndexOutOfBoundsException e) {
11   // Expected exception.
12 }
13 assertTrue(b0 == false);
```

2. Test case that includes the target method m within a try-catch clause.

We can see in Test Case 3 an example of a test case from the second category.

Test Case 3: *Test case from the second category.*

```
1  Stack stack0 = new Stack();
2  try {
3    Object obj0 = stack0.pop();
4    fail(''Expected exception of type java.util.↩
         EmptyStackException'');
5    } catch (java.util.EmptyStackException e) ↩
         {
6      // Expected exception.
7  }
```

3. Test case that includes the target method m outside any try-catch clause.

We can see in Test Case 4 an example of a test case from the third category.

Test Case 4: *Test case from the third category.*

```
1  Stack stack0 = new Stack();
2  stack0.addElement((Object)0);
3  Stack stack1 = new Stack();
4  stack1.addElement((Object)10);
5  Object obj0 = stack1.push((Object)1);
6  Stack stack2 = new Stack();
7  stack2.addElement((Object)1);
8  boolean b0 = stack1.addAll((Collection)↩
       stack2);
9  stack1.add(0, (Object)(−1));
10 Object obj1 = stack1.pop();
11 Object obj2 = stack1.push(100);
12 try {
13   Object obj1 = stack0.pop();
14   fail(''Expected exception of type java.util.↩
         EmptyStackException'');
15   } catch (java.util.EmptyStackException e) ↩
         {
16     // Expected exception.
17 }

18 assertTrue(b0 == false);
19 assertNull(obj0);
```

For our purpose we are interested in the test cases of the third category. The first category is discarded because does not contain m while the second category is discarded because this type of test cases cannot be used due to the current limitations of SBES. The `Execution Scenarios Creator` discards test cases from the first two categories and converts the remaining test cases in execution scenarios. To convert test cases in execution scenarios, we perform the following operations:

1. **Elimination of instructions after m.**

   Given the execution scenarios structure, we need to keep m as the last instruction

[2] https://code.google.com/p/randoop/

of the execution scenarios. This operation allows us to obtain execution scenarios with m as last instruction.

2. **Reduction to a single object of c.**

   A test case can contain multiple object of the class c with a few method invocations. Moreover, as we have already discussed, if we have an object with less than 5 elements, this can lead to bad results. This operation allows us to obtain a single object of c with the right number of element.

3. **Specification of the generic objects as integer type and elimination of unnecessary object casting.**

   SBES works better when the classes that rely on generics are instantiated on integers. This operation allows us to create execution scenarios that only work on integers.

4. **Elimination of execution scenarios that contain object of type c with less than 5 or more than 8 elements.**

   The results in section 2.3 show that the maximum effectiveness of SBES is obtained with a number of elements between 5 and 8. This operation removes the execution scenarios that do not respect this constrain.

5. **Elimination of execution scenarios that are syntactically equivalent.**

   This operation removes the execution scenarios that are syntactically different.

Execution Scenario 1: *Execution scenarios obtained from test case.*

```
1 Stack<Integer> stack0 = new Stack<Integer↩
      >();
2 stack0.addElement(0);
3 stack0.addElement(10);
4 Integer obj0 = stack0.push(1);
5 stack0.addElement(1);
6 stack0.add(0,−1);
7 Integer obj1 = stack0.pop();
```

For example, given the Test Case 4, the operations described above allow us to obtain the Execution Scenario 1.

4. EVALUATION

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

RQ1 How effectively can the proposed approach generate execution scenarios?

RQ2 How efficiently can the proposed approach generate execution scenarios?

RQ3 How effective is the technique compared to manual generation of execution scenarios?

The research question RQ1 deals with the effectiveness of the approach. The research question RQ2 deals with the efficiency of the approach. The research question RQ3 deals with the effectiveness of SBES using execution scenarios generated automatically with the proposed approach.

To answer RQ1 we calculated the number of execution scenarios generated. To answer RQ2 we measured performance as the *time required to generate the list of methods* that can maximizes the effectiveness of SBES (first phase) and the *time required to generate execution scenarios* (second phase), since this two measures directly affect the overall performance of our approach.

For RQ3, we calculated the effectiveness of SBES using execution scenarios generated automatically with the approach. Then we compared the results obtained against the results obtained using execution scenarios generated manually.

4.1. Experimental setup

We experimented with the class `Stack` taken as a representative for the various containers available in the Java standard library.

For RQ1 and RQ2 we experimented with 15 methods of the class `Stack` as reported in Table 2. We ran the experiments by feeding the prototype with the class `Stack`, the target method

and the *blacklist*. We repeated the experiments 5 times because of the random nature of Randoop.

For RQ3 we experimented with 6 methods of the class `Stack` as reported in Table 4. We ran the experiments by feeding SBES with the class `Stack`, the target method and 5 execution scenarios, randomly chosen, generated by the prototype. We repeated the experiments 30 times because of the random nature of genetic algorithms. Then we compared the results obtain against the results obtained from the original article about SBES [4].

4.2. Results

In this section we discuss the experimental results. Table 2 summarizes the results of the experiment for RQ1. For each of the analysed methods, the table shows the following information: *(i)* the minimal number of execution scenarios generated with a single run (column **Min**), *(ii)* the maximum number of execution scenarios generated with a single run (column **Max**) and *(iii)* the average amount of execution scenarios generated in the 5 runs (column **Avg**).

Table 2 shows a very interesting results because the approach always generates at least 1 execution scenario and on average it generates a large number of execution scenarios, about 26 per method.

In summary, we can answer positively to research question RQ1:

> ***RQ1**: The proposed approach can correctly generate one or more execution scenarios per method.*

Table 3 summarizes the results of the experiment for RQ2. For each method we calculated the median of the results obtained. The results obtained were similar between them so we show in Table 3 the results obtained for the method `pop`.

For this method, the table shows the following information: *(i)* the time required to identify pure methods (column **Pure**), *(ii)* the time required to create the list of methods (column **Methods**), *(iii)* the time required to generate test cases (column **Test**), *(iv)* the time required to convert test cases in execution scenarios (column **Convert**) and *(v)* the total time required by the approach to generate execute scenarios (column **Tot**).

Table 2: *Effectiveness of the approach.*

Method	Min	Max	Avg
add(int,Object)	8	24	16.4
add(Object)	6	37	24
addElement(Object)	4	30	16.8
clear()	1	45	32.6
elementAt(int)	14	24	20
firstElement()	20	41	28.8
get(int)	20	27	22.6
indexOf(Object)	12	57	33.8
lastElement()	14	41	24.2
peek()	23	50	36.4
pop()	25	56	40
push(Object)	11	43	28.4
remove(Object)	9	51	31.6
remove(int)	18	25	22
set(int,Object)	16	25	19.2

Table 3: *Efficiency of the approach.*

I Phase		II Phase		Tot
Pure	Methods	Test	Convert	
10.5s	0.06s	11s	10s	31.56s

The execution time is acceptable and less than the time required to manually generate 26 execution scenarios. Besides the results show that the second phase takes 67% of the total time. Hence, we can answer positively to research question RQ2:

> ***RQ2**: The proposed approach requires a total execution time that is acceptable.*

Table 4 reports the data about the effectiveness of SBES using execution scenarios generated automatically with the approach. Table 5 reports the data about the effectiveness of SBES using execution scenarios generated man-

ually by experts. For each 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 [4].

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).

$$\frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$$

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).

$$\frac{\text{true positives}}{\text{true positives} + \text{false positives}}$$

The results show that for the values **Max$_t$**, **Max$_r$** and **Rec** our approach is slightly lower than the manual approach. Besides, for the values **Prec** and **Avg** our approach is lower than the manual approach of 9% for the **Prec** values and 11% for the **Avg** values.

We have examined execution scenarios from the two approaches to discover the cause of the discordant results. We have discovered another aspect that increments the effectiveness of SBES. This aspect is the *heterogeneity* of the values of the stack. The more the values are heterogeneous, the greater the effectiveness of SBES.

We have repeated the previous experiments by introducing heterogeneous values in the stack. Table 6 shows the results obtained.

The results show that now for the values **Max$_t$**, **Max$_r$** and **Rec** our approach is equal than the manual approach and for the method `push` we get even better results. Besides, for the values **Prec** and **Avg** our approach is lower than the manual approach but only of 1,33% for the **Prec** values and 2% for the **Avg** values.

In summary, we can answer positively to research question RQ3:

> ***RQ3**: The effectiveness of SBES is similar both using execution scenarios generated automatically with the approach and using execution scenarios generated manually by experts.*

5. CONCLUSIONS

Software redundancy that already exists in the software, called intrinsic redundancy, finds many interesting applications that span from fault tolerance to self-healing. However these techniques rely on manual identification of the equivalence, and this limits their applicability.

Gorla et al. proposed a technique that automatically identifies equivalent methods and combinations of methods by exploiting genetic algorithms, but needs an execution scenario to work. Currently, the execution scenario is manually generated that is expensive and makes the technique hard to use.

In this paper, we presented a novel technique that generates execution scenarios. The technique is fully automatic and applies to any methods. We reported the experimental results obtained with a prototype that implements the approach. The results obtained for the Java class `Stack` are encouraging. We can automatically generate at least 1 execution scenario and an average of 26 execution scenarios for each method in a total execution time that is acceptable and less than the time required to manually generate 26 execution scenarios. Besides, the effectiveness of SBES is similar both using execution scenarios generated automatically with the approach and using execution

Table 4: *Effectiveness of the automatic approach.*

Method	Tot	Max_t	Max_r	Avg	Prec	Rec
addElement	6	4	2.8	2.04	0.99	0.70
clear	3	3	3	2.73	0.91	1
firstElement	2	2	2	1.40	0.77	1
peek	2	2	2	1	0.77	1
push	6	2	2	2	0.94	0.33
remove(Object)	4	1.8	1	0.68	0.86	0.45

Table 5: *Effectiveness of the manual approach.*

Method	Tot	Max_t	Max_r	Avg	Prec	Rec
addElement	6	4	3	2.17	1	0.70
clear	3	3	3	2.77	0.99	1
firstElement	2	2	2	1.57	0.89	1
peek	2	2	2	1.23	0.97	1
push	6	2	2	2	1	0.33
remove(Object)	4	2	1	0.80	0.92	0.50

Table 6: *Effectiveness of the automatic approach with the new aspect.*

Method	Tot	Max_t	Max_r	Avg	Prec	Rec
addElement	6	4	3	2.15	0.99	0.70
clear	3	3	3	2.73	0.99	1
firstElement	2	2	2	1.54	0.85	1
peek	2	2	2	1.20	0.97	1
push	6	3	2.4	2	0.97	0.50
remove(Object)	4	2	1	0.80	0.92	0.50

scenarios generated manually by experts.

We are currently working on increasing automation with respect to the definition of the *blacklist* file. We are also working on evaluating the approach with new case studies to obtain a more general estimate of the effectiveness and efficiency of the approach.

References

[1] A. Carzaniga, A. Goffi, A. Gorla, A. Mattavelli, and M. Pezzè. Cross-checking oracles from intrinsic software redundancy. In *ICSE'14:Proceedings of the International Conference on Software Engineering*, pages 931–942. ACM, 2014.

[2] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezzè. Automatic recovery from runtime failures. In *ICSE'13:Proceedings of the International Conference on Software Engineering*, pages 782–791. ACM, 2013.

[3] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè. Automatic workarounds for web applications. In *FSE'10:Proceedings of the International Symposium on the Foundations of Software Engineering*, pages 237–246. ACM, 2010.

[4] A. Gorla, A. Goffi, A. Mattavelli, M. Pezzè, and P. Tonella. Search-based synthesis of equivalent method sequences. In *FSE'14:Proceedings of the International Symposium on the Foundations of Software Engineering*, pages –. ACM, 2014.

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

[6] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE'07:Proceedings of the International Conference on Software Engineering*, pages 75–84. IEEE Computer Society, 2007.

[7] A. Sălcianu and M. Rinard. Purity and side effect analysis for java programs. In *VMCAI'05:Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 199–215. Springer, 2005.