Synthesis of Equivalent Method Calls in Guava

Andrea Mattavelli[1], Alberto Goffi[1] and Alessandra Gorla[2]

[1] Università della Svizzera italiana (USI), Lugano, Switzerland
{andrea.mattavelli,alberto.goffi}@usi.ch
[2] IMDEA Software Institute, Madrid, Spain
alessandra.gorla@imdea.org

Abstract. We developed a search-based technique to automatically synthesize sequences of method calls that are functionally equivalent to a given target method. This paper presents challenges and results of applying our technique to Google Guava. Guava heavily uses Java generics, and the large number of classes, methods and parameter values required us to tune our technique to deal with a search space that is much larger than what we originally envisioned. We modified our technique to cope with such challenges. The evaluation of the improved version of our technique shows that we can synthesize 188 equivalent method calls for relevant components of Guava, outperforming by 86% the original version.

1 Introduction

Reusable software libraries frequently offer distinct functionally equivalent API methods in order to meet different client components' needs. This form of *intrinsic* redundancy in software [3] has been successfully exploited in the past for various purposes, such as to automatically produce test oracles [1] and to increase the reliability of software systems [2]. Even when completely automated in their internals, these techniques require developers to manually identify functionally equivalent sequences of method calls within the system. This activity can be tedious and error prone, and may thus be a showstopper for a widespread adoption of these techniques.

To support developers in this manual task, we developed a search-based technique that can automatically synthesize and validate sequences of method calls that are *test-equivalent* to a given target method [7]. This paper reports the results of using our prototype implementation SBES to automatically synthesize functionally equivalent method calls for the Google Guava library, and more precisely for its extensive set of collections. Guava collections heavily use Java generics, a language feature that was not supported in our original work. Moreover, the high number of classes, methods and parameter values made the search space large, and as a consequence more challenging for our search-based technique. We cope with such challenge by means of memetic algorithms.

We evaluated SBES on 220 methods belonging to 16 classes of the Google Guava collections library. Compared to the old version of our prototype, the support of Java generics and the use of memetic algorithms allow to find 86% more *true* functionally equivalent method sequences.

2 Synthesis of Equivalent Sequences of Method Calls

Our search-based technique aims to automatically synthesize a sequence of method invocations whose functional behavior is equivalent to a target method. For example, given the method put(key,value), which inserts a new key-value pair in a Guava Multimap instance, our technique may be able to synthesize Multimap m=new Multimap(); m.putAll(key, new List().add(value)) as a possible equivalence. Producing solutions that would be equivalent for all possibly infinite inputs and states, is a well-known undecidable problem. The problem becomes tractable, though, by reducing the number of potential executions to a finite set. Therefore, our technique deems as equivalent two sequences of method calls that produce identical results and lead to identical states on a given set of test inputs, which we refer to as *execution scenarios*. This definition of equivalence is based on the *testing equivalence* notion defined by De Nicola and Hennessy [4].

We implemented our technique in a tool for Java called SBES (Search-Based Equivalent Synthesis) that manipulates source code. SBES employs an iterative two-phase algorithm to generate sequences of method calls that are equivalent to a given input method m. In the first phase—the *Synthesis* phase—it generates a candidate sequence eq whose behavior is equivalent to m on the existing set of execution scenarios. In order to do that, SBES generates a stub class that extends the class declaring m, and encloses all execution scenarios and an artificial method method_under_test, which acts as the main driver for the synthesis:

```
1 public void method_under_test() {
2   if (distance(exp_s[0], act_s[0])==0 && distance(exp_s[1], act_s[1])==0 &&
3         distance(exp_r[0], act_r[0])==0 && distance(exp_r[1], act_r[1])==0)
4           ; // target
5 }
```

SBES aims to generate a sequence of method calls eq that covers the TRUE branch of this artificial method. The condition evaluates whether the *return value* and the *state reached* by executing eq in each execution scenarios are test-equivalent to executing m. Arrays act_r and exp_r store the return values of eq and m respectively. Similarly, act_s and exp_s store the corresponding reached states. The synthesis phase may lead to spurious results, since it considers only a finite set of execution scenarios. Therefore, in the second phase of the algorithm—the *Validation* phase—SBES aims to remove spurious results by looking for counterexamples (that is, previously unknown scenarios for which eq and m are not test-equivalent). In this phase, SBES automatically generates a slightly different stub class with the following artificial method:

```
1 public void method_under_test(Integer key, String value) {
2   ArrayListMultimap clone = deepClone(this);
3   boolean expect = this.put(key, value);
4   boolean actual = clone.putAll(key, new ArrayList().put(value));
5   if (distance(this,clone)>0 || distance(expect,actual)>0)
6           ; // target
7 }
```

SBES aims once again to generate a sequence of method calls that can cover the TRUE branch. In this case, though, the condition asserts the *non* equivalence between m and eq. The code shows the stub based on the example of the Guava

Multimap class, the target method put(key,value), and the candidate equivalence putAll(key, new List().add(value)). If this phase produces a counterexample, the algorithm iterates, adding the counterexample to the initial set of scenarios. Otherwise, SBES returns *eq* as the final result.

In both phases, SBES exploits a custom version of EvoSuite as a search-based engine [5]. We modified EvoSuite such that it has the TRUE branch of method_under_test as the sole goal to cover. Since the condition of the artificial method branch is a conjunction of atomic clauses, the fitness function evaluates the branch distance of each clause separately, aiming to generate an individual whose all clauses evaluate to TRUE. The branch distance of each atomic clause is computed using numeric, object or string distance functions depending on the involved type. EvoSuite does not have a proper notion of *object* distance, and as a consequence it is unable to effectively guide the evolution when a branch condition evaluates an object. Using method equals to compare objects would yet fail at providing any guidance, since this method returns a boolean value, and thus flattens the fitness landscape [8]. To overcome this issue, we implemented a notion of distance that *quantifies* the difference between two objects. To calculate such distance, we compute the distance of all the object's fields. For non-primitive fields, we recursively call the object distance function on them. As a result, the distance between two objects amounts to the sum of the distance of all the inspected fields. Two objects are deemed as identical if their distance is zero. We refer the interested reader to our previous paper for further details on SBES [7].

3 Extending SBES to Deal with Google Guava

In our previous work we demonstrated the effectiveness of SBES on few, selected Java classes such as Stack and a set of classes from the GraphStream library. Using SBES on Google Guava was challenging for at least two reasons. First, Guava contains more than 335 classes and 5,400 methods. The combinatorial explosion of classes, parameters, and method calls only considering the library itself is enormous. Second, most of the classes in the library are implemented using generic types. Generic types allow developers to abstract algorithms and data structures, but their presence increases the complexity of the synthesis process. Ignoring generic types exacerbates the combinatorial explosion mentioned before, since type erasure substitutes generics with the base class java.lang.Object. Yet, by considering generic types we must concatenate method calls that both satisfy and adhere to the generic types specified at class instantiation time, increasing the complexity of the generation process.

To cope with Guava, we extended SBES along two lines. First, we added generic-to-concrete type replacement to our prototype. In those cases where the execution scenarios declare and use concrete classes rather than generic types, we exploit such information. For example, suppose to synthesize equivalences for method Multimap<K,V>.put(key,value), with the following initial execution scenario: Multimap<Integer,String> m=new Multimap();m.put(15, "String"). Since

in the execution scenario the generic types K and V are replaced with Integer and String respectively, we can safely replace all the occurrences of the generic types with the concrete classes in the stub class. By resolving generic types, EvoSuite obtains more information to guide the search towards better individuals, without wasting time to find syntactically valid concrete classes. The second extension tries to mitigate the combinatorial explosion of method calls and parameters. In our previous evaluation we observed that in order to find valid solutions, it is necessary to invoke methods either in a specific order or with specific parameter values. To efficiently synthesize such sequences of method calls, we exploit *memetic algorithms*. Memetic algorithms combine both global and local searches to generate better individuals, thus accelerating the evolution towards a global optimum. EvoSuite already supports memetic algorithms [6], in Section 4 we briefly discuss how we found the optimal configuration of memetic search.

4 Experimental Evaluation

The purpose of evaluating SBES on the Google Guava library was twofold. First, we wanted to show that many methods of the Guava API have equivalent sequences of method calls. Second, we wanted to demonstrate that SBES can effectively synthesize such equivalent sequences. In particular, we wanted to assess whether the improvements that we brought to SBES with respect to our previous work could identify substantially more correct solutions.

Experimental Setup We limited our evaluation to the classes declared in package collect, and in particular we selected a random set of concrete classes for which we identified a list of equivalences in previous studies [1–3]. As a result, we selected 16 subject classes with a total of 220 methods under analysis. These classes represented a challenge for SBES since they declare a high number of methods, which strains the search process. Moreover, these classes make an extensive use of generic types. For each target method we first evaluated the effectiveness—measured in terms of *true* synthesized solutions—of the original version of SBES. We then evaluated the effectiveness of SBES with generic-to-concrete type replacement, which we refer to as $SBES^{G}$. Finally, we evaluated the effectiveness of combining the generic-to-concrete support and memetic search. We refer to this version as $SBES^{G,M}$. We ran the experiments by feeding the prototype with the class under analysis, the target method, and an initial execution scenario, which consists of one test case that was either extracted from the existing test suite, or generated automatically with EvoSuite. For each target method, we iterated the entire synthesis process 20 times—regardless of the success of the first phase—with a search budget of 180 seconds for both the first and second phase. The search budgets were validated in our previous evaluation [7].

Results Table 1 summarizes the results of our experiments.[1] For the selected target methods of Google Guava, SBES, $SBES^{G}$, and $SBES^{G,M}$ could success-

[1] A replication package is available at http://star.inf.usi.ch/sbes-challenge

Class	Methods	SBES		SBES^{G}		SBES^{G,M}	
		TP	FP	TP	FP	TP	FP
ArrayListMultimap	15	7	1	13	1	12	3
ConcurrentHashMultiset	16	5	0	9	1	6	2
HashBasedTable	16	3	6	3	8	2	8
HashMultimap	15	7	0	9	2	13	1
HashMultiset	16	6	0	15	3	19	5
ImmutableListMultimap	11	1	1	2	1	2	0
ImmutableMultiset	8	3	0	1	0	3	0
LinkedHashMultimap	15	6	1	9	1	12	3
LinkedHashMultiset	16	5	1	19	2	19	6
LinkedListMultimap	15	6	2	10	1	11	0
Lists	8	18	0	17	3	15	1
Maps	9	6	0	5	0	8	0
Sets	10	12	2	15	0	21	0
TreeBasedTable	15	0	8	4	8	3	10
TreeMultimap	14	4	1	9	3	8	2
TreeMultiset	20	12	5	32	13	34	10
Total	**220**	**101**	**28**	**172**	**47**	**188**	**50**

Table 1. Guava classes considered with their number of methods under evaluation and equivalences synthesized by the three prototype versions *SBES*, *SBES^{G}*, and *SBES^{G,M}*

fully synthesize 101, 172 and 188 equivalent sequences of method calls respectively. SBES^{G} finds 70% more equivalences than the base version. Such result confirms that generic-to-concrete type replacement can indeed reduce the search space without reducing potential behaviors of the class, ultimately improving the synthesis process. Similarly, memetic algorithms successfully improve the synthesis process: SBES^{G,M} can generate 9% more solutions than SBES^{G}. However, the effectiveness of memetic algorithms largely depends on the frequency at which EvoSuite performs the local search [6]. If the local search occurs too often, it steals search budget from the global search. On the other hand, if the local search occurs infrequently, it does not bring much benefits. To find the optimal configuration, we ran SBES^{G,M} such that the local search was done once every 10, 50, 75, 85, and 100 generations. As expected, a frequent local search degrades the effectiveness of the approach. With 10 generations we obtained the worst result (we synthesized 30 equivalences less than SBES^{G}, i.e., -17%), and obtained consistently better results for the other configurations up to the optimal rate of once every 75 generations. After this threshold, local search seems not to be frequent enough, since the effectiveness decreased again (-7% w.r.t. the optimal configuration).

For all runs we manually validated the solutions. While on the one hand SBES^{G} and SBES^{G,M} identify more truly equivalent solutions (reported as TP in Table 1) than SBES, they also produce more false positives (FP in Table 1). In some cases, as for HashBasedTable, TreeBasedTable, and TreeMultiset, this is due to the inability of EvoSuite to generate a syntactically valid test case as a

counterexample. The validation phase, thus, fails in invalidating even the most trivial spurious candidate. In the reminder of the cases, instead, false positives are due to a major limitation of the technique: the behavior of the target branch in the artificial method during the validation phase is comparable to a flag variable. In fact, the object distance during the validation phase is zero for all generated solutions, except for those corner cases in which the behavior of the candidate is not equivalent. As a consequence, the evolution in the second phase lacks any guidance. This is a limitation of our approach, and we are actively working to overcome such issue.

5 Conclusion

This paper introduces significant improvements over our previous work on the automatic synthesis of functionally equivalent sequences of method calls [7]. The experiments on 220 methods belonging to 16 classes of the Google Guava library show that generic-to-concrete type replacement and memetic algorithms allowed the new prototype to outperform the previous version by 86% in terms of true equivalences synthesized.

Acknowledgment This work was supported in part by the Swiss National Science Foundation with projects SHADE (grant n. 200021-138006) and ReSpec (grant n. 200021-146607). The authors would like to thank Mauro Pezzè and Paolo Tonella for their contributions to the previous version of the technique.

References

1. Carzaniga, A., Goffi, A., Gorla, A., Mattavelli, A., Pezzè, M.: Cross-Checking Oracles From Intrinsic Software Redundancy. In: International Conference on Software Engineering (ICSE). pp. 931–942. ACM (2014)
2. Carzaniga, A., Gorla, A., Mattavelli, A., Perino, N., Pezzè, M.: Automatic recovery from runtime failures. In: International Conference on Software Engineering (ICSE). pp. 782–791. IEEE (2013)
3. Carzaniga, A., Mattavelli, A., Pezzè, M.: Measuring software redundancy. In: International Conference on Software Engineering (ICSE). pp. 156–166. IEEE (2015)
4. De Nicola, R., Hennessy, M.: Testing equivalences for processes. Theoretical Computer Science 34(1-2), 83–133 (1984)
5. Fraser, G., Arcuri, A.: EvoSuite: automatic test suite generation for object-oriented software. In: Symposium on the Foundations of Software Engineering (FSE). pp. 416–419. ACM (2011)
6. Fraser, G., Arcuri, A., McMinn, P.: A memetic algorithm for whole test suite generation. Journal of Systems and Software 103(0), 311–327 (2015)
7. Goffi, A., Gorla, A., Mattavelli, A., Pezzè, M., Tonella, P.: Search-based synthesis of equivalent method sequences. In: Symposium on the Foundations of Software Engineering (FSE). pp. 366–376. ACM (2014)
8. Harman, M., Hu, L., Hierons, R.M., Wegener, J., Sthamer, H., Baresel, A., Roper, M.: Testability transformation. IEEE Transactions on Software Engineering (TSE) 30(1), 3–16 (2004)