Automatic Generation of Effective Unit Tests based on Code Behaviour

Andrea Fornaia, Alessandro Midolo, Giuseppe Pappalardo, Emiliano Tramontana
Dipartimento di Matematica e Informatica, University of Catania, Italy
Email: surname@dmi.unict.it

***Abstract*—A large amount of test cases is very useful to check the correctness of a software system while it is developed. Often a considerable time is dedicated by human programmers to designing effective test cases. This paper proposes an approach for automatically generating test cases tailored to the characteristics of the code under test. For this, the classes of a software system to be tested are characterised by a static code analysis aiming at summarising and representing their behaviour. As test cases check the behaviour of code, classes that exhibit a close behaviour may be checked using similar test cases. Therefore, in the approach proposed, for classes having a comparable behaviour, test cases are generated by taking as a template the test cases available for one of the classes among the similar ones. The approach has been assessed on a few open source projects and has proved to be viable for generating applicable and effective test cases for the classes.**

***Index Terms*—test case generation, static code analysis, test templating, verification**

I. Introduction

Producing test cases is an effective way to check the correctness of a software system, and to check that evolutionary changes aiming at improving functionalities are not introducing defects to previously correct code [11], [26]. However, developing tests is a time consuming activity. When designing tests, a developer has to take into account the behaviour of the component under test to determine the set of inputs and expected output, which are essential to implement a test case. Moreover, during implementation some boilerplate code needs to be added to give the needed context to the test case.

The existing literature on tests suggests several approaches for assisting the work of developers. Since one of the tasks of the developers is the selection of input values to be given to a method, combinatorial approaches for input values can be very effective and many tools have been implemented to find values among given validity ranges, as well as outside validity ranges [3], [25]. Another task developers have to perform is implementing methods calls that check the behaviour of a class, assistance for it has been given by tools that e.g. randomly generate a sequence of calls [7], [17], [23]. Moreover, for the task of finding expected output values to check against the resulting execution behaviour, often the solution is building a model of the system [4]. Other approaches for producing tests and make them robust include the generation of code variations to make sure that tests can find the erroneous behaviour [12]. Moreover, some approaches have been proposed to automatically generate code that passes all tests [10].

Most of the approaches aim at having and executing as many test cases as possible, which is worthy for checking a large amount of execution scenarios. However, given the large number of possible input data and execution paths inside the software system under test, execution could take an amount of time larger than the time frame available to have a timely feedback. This is mainly relevant for agile practices, which prescribe both as much tests as possible, as well as developing components, integrating and testing the overall system several times a day, to ensure minimal design and correct execution [1], [8]. For this, during development, test execution time is curbed, and test cases to be executed have to be selected among available ones [13], [15], [21].

This paper aims at automatically generating test cases tailored to the behaviour of the class under test. Our approach provides a class with a test by using static analysis to determine its behaviour, then such a behaviour is checked against the behaviour gathered for other classes, finally tests are generated starting from previously known tests for classes having a similar behaviour. Since generated tests are tailored to the code to be tested, they are effective for code coverage and for finding bugs. We have used our approach to generate tests for several software systems, whose source code is available. According to our experiments, the tests we have generated manage to extend the amount of code coverage significantly, both by executing new paths within classes that had some test cases, and by generating tests for classes that had not been previously tested.

The rest of the paper is organised as follows. Section II describes how we perform the analysis of classes to gather their behaviour. Section III reports our approach for generating tests according to the knowledge on the behaviour of classes and other tests used as samples. Section IV shows the results of the analysis of several software systems. Section V compares our approach with relevant related works. Finally, conclusions are expressed in Section VI.

II. Analysis of Software Systems

Generally, the developer creates a test case once he has gathered some knowledge on the expected behaviour of the class to be tested. Then, for each method of the class, he determines a set of inputs, among valid or invalid ranges for the needed parameters, and an expected output to be compared with the output provided by the method execution.

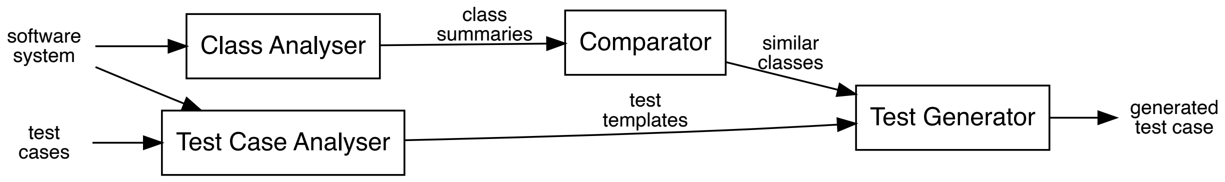

Fig. 1. Main components realised the proposed approach for generating test cases: analysing classes, analysing test cases, comparing classes, and generating test cases.

The proposed automatic analysis tool aims at revealing the behaviour of a class by extracting some characteristics of the static code. Moreover, existing test cases, implemented by developers, are analysed in order to be later used as templates of test cases for some other classes. Figure 1 shows the essential components implementing the stages of the proposed approach: (i) the analysis of classes and the extraction of their main characteristics; (ii) the computation of similarity scores through comparison; (iii) the analysis of test cases aiming at finding useful templates, mainly for untested classes; (iv) the generation of the code implementing new test cases for selected classes.

A. Revealing Class Behaviour

For a software system, all the classes are analysed to find their behaviour, this is summarised starting from all the used APIs (component *Class Analyser* in Figure 1). APIs are classes and methods provided to the application by the underlying platform. E.g. the standard library APIs of the Java platform provides applications with useful classes and methods giving many functionalities such as generating random numbers, accessing the file system, etc. The methods of a class under analysis will then be marked according to the APIs used. This is a way to recognise that classes and methods of the APIs are an indication of the intent of the calling code [5], [6], [24]. E.g. a method using a variable having type `List` and using methods `isEmpty()` and `add()` on such a variable exhibits, at runtime, a behaviour comprising the update of the elements on such a list (Java APIs include interface `List`, which declares methods `isEmpty()` and `add()`).

Therefore, for each analysed class, all declared variables, and method calls are searched, then a list is created that holds all the APIs that have been found. Once classes have been characterised by the list of used APIs, they will be checked to find out whether they have a comparable behaviour (component *Comparator* in Figure 1).

Figure 2 shows the listing of two classes found in the RepoDriller software system[1]. Both classes use Java APIs `List` and `String`, hence both have been characterised by the list of APIs {List, String}.

B. Comparing Classes

In order to reveal whether a pair of classes have a comparable behaviour, we measure similarity according to *Jaccard similarity coefficient*. The latter, when measuring the similarity of two sets is the ratio between the cardinality of the intersection and the cardinality of the union of the sets to be compared.

For a pair of classes A and B, we measure the cardinality of the intersection of the two sets of APIs found in their code, hence used inside the respective classes (given as A_{API} and B_{API}, respectively), and the cardinality of the union of the two API sets. Then, Jaccard similarity coefficient for the pair of classes is the ratio of the latter cardinalities.

$$J(A,B) = \frac{|A_{API} \bigcap B_{API}|}{|A_{API} \bigcup B_{API}|}$$

As such, our similarity measure finds whether a class pair has many or few APIs in common, where 1 is the upper bound and 0 is the lower bound for the measure.

Once a pair of classes has been found to have a high degree of similarity (i.e. Jaccard similarity coefficient near to 1), a test that could be available for one class will be used as a template for the other one.

For the pair of classes shown in Figure 2, the similarity measured is 1, as they both use the same Java APIs `List` and `String`.

C. Analysing Test Cases

The code of test cases is analysed to find the class that is the target of each test (component *Test Case Analyser* in Figure 1). Moreover, for each tested class, we reveal the list of the methods that are invoked by the test. Therefore, for a test case T we will have the list of the tested classes, e.g. $\{A, B, C\}$ and the methods invoked on each class. Then, when a test case uses several classes, in order to determine which class is under test, we distinguish classes whose instances are used as parameters of method calls, from classes whose methods are invoked. We associate the test with the class whose methods are called. Therefore, a test for class A is found when in the test code there is at least one method called on an instance of class A. As a result, the same test case could be associated with more than one class, when its code calls methods on instances of different classes.

D. Implementation Details

The JavaParser[2] library has been used to perform code inspection of classes and test cases, for generating tests [22]. JavaParser lets us navigate the source code as an Abstract

[1]https://github.com/mauricioaniche/repodriller

[2]https://javaparser.org/

```java
package org.repodriller.filter.diff;

import java.util.List;
import org.repodriller.util.RDFileUtils;

/**
 * Only process diffs on files with certain file extensions.
 *
 * @author Ayaan Kazerouni
 */
public class OnlyDiffsWithFileTypes implements DiffFilter {
 private List<String> fileExtensions;

 public OnlyDiffsWithFileTypes(List<String> fileExtensions){
  this.fileExtensions = fileExtensions;
 }

 @Override
 public boolean accept(String diffEntryPath) {
  return RDFileUtils.fileNameHasIsOfType(diffEntryPath,
              this.fileExtensions);
 }
}
```

```java
package org.repodriller.filter.diff;

import java.util.List;
import org.repodriller.util.RDFileUtils;

/**
 * Only process diffs on files without certain file extensions.
 *
 * @author Ayaan Kazerouni
 */
public class OnlyDiffsWithoutFileTypes implements DiffFilter {
 private List<String> fileExtensions;

 public OnlyDiffsWithoutFileTypes(List<String> fileExtensions){
  this.fileExtensions = fileExtensions;
 }

 @Override
 public boolean accept(String diffEntryPath) {
  return !RDFileUtils.fileNameHasIsOfType(diffEntryPath,
              this.fileExtensions);
 }
}
```

Fig. 2. Classes `OnlyDiffsWithFileTypes` and `OnlyDiffsWithoutFileTypes` implemented in RepoDriller software system, using the same Java APIs, and having the same signature for their method.

Syntax Tree (AST) having a root representing the entire file, to which all code elements are connected, in particular each class declarations. From this, in turn, multiple nodes can be reached, which represent the fields or methods of the class.

Code inspection has been performed by using JavaParser's class `VoidVisitorAdapter`, which lets us define a *Visitor* class to search for a specific property. In the *Visitor* class, a `visit()` method is implemented, which takes as parameters the type of object being searched (class declaration, method declaration, etc.) and the container where data are saved; the method body will implement the operations to be carried out each time the element specified in the parameter is found. The *Visitor* class works within a `CompilationUnit` (representing a Java file), therefore it is necessary to associate the Visitor with a `CompilationUnit`. This association was carried out for all the files of the source code.

Additionally, code inspection was performed for each `CompilationUnit` using method `findAll(Class<T> NodeType)`, which searches within a `CompilationUnit` all the elements that match the `NodeType` passed as parameter. We searched e.g. `MethodCallExpr` that represents method calls. For each expression found, some operations were performed in order to resolve the type found so as to trace its origin. This was useful to find all the APIs used within each class.

III. Test Generation

This section describes component *Test Generator* shown in Figure 1.

Suppose that a pair of classes A and B has a high similarity: the idea is to generate tests for class B by *replacing* all occurrences of class A in the tests found for class A, with occurrences of class B. Of course, a test for class A would call methods implemented in class A. Hence, when using the test for class A as a template for class B, all the methods called have to be checked, and possibly changed.

Four possible cases have been identified to guide method replacement on a template test. Firstly, classes A and B have the same method signatures, this is the case when e.g. they implement the same interface. Secondly, classes A and B have methods having the same name but a different signature, i.e. the type and number of parameters differ. Thirdly, classes A and B have methods having different names with some resemblance to each other. Method names exhibiting some resemblance are, e.g., like the following: `getNumberOfValues()` and `getNumberValues()`; though these names differ, a human reader can easily recognise that they could be used for the same goal. Fourthly, methods having the same (or almost the same) input parameters, as their expected types, have some resemblance to each other.

To begin with, classes are automatically inspected to find the list of all their methods, together with each associated signature.

As for the first case, when the methods of the classes in a pair of similar classes have the same signature, tests are generated by simply replacing the name of the class A with the name of the class B. This is also a necessary step for test generation when the following second case occurs.

For the second case, the methods in the pair of classes have the same name, however different parameters. Then, parameters will be replaced as follows. When the parameters used by the method call on the template test are a superset of the needed parameters for generating the new test, then the method call will be rewritten by reducing the number of parameters and selecting those needed, in the proper position. When new parameters are needed for the method to be called, i.e. they are not contained within the parameters of the original method call, then a method call is written that generates new parameters with default values, in order for the test code to be syntactically correct.

The third case involves a different analysis, it consists in having to call a method on the generated test that is not in the

```java
package org.repodriller.filter.diff;

import java.util.Arrays;
import org.junit.Assert;
import org.junit.Test;

public class OnlyDiffsWithFileTypesTest {

  @Test
  public void shouldAcceptIfFileHasExtensionWithDot() {
    Assert.assertTrue(new OnlyDiffsWithFileTypes(
      Arrays.asList("cpp", ".java")).accept("/dir/File.java"));
  }

  @Test
  public void shouldAcceptIfFileHasExtensionWithoutDot() {
    Assert.assertTrue(new OnlyDiffsWithFileTypes(
      Arrays.asList(".cpp", "java")).accept("/dir/File.java"));
  }

  @Test
  public void shouldRejectIfFileDoesNotMatchExtensions() {
    Assert.assertFalse(new OnlyDiffsWithFileTypes(
      Arrays.asList("cpp", ".java")).accept("/dir/File.css"));
  }
}
```

```java
package org.repodriller.filter.diff;

import java.util.Arrays;
import org.junit.Assert;
import org.junit.Test;

public class GenOnlyDiffsWithFileTypesTest {

  @Test
  public void shouldAcceptIfFileHasExtensionWithDot() {
    Assert.assertTrue(new OnlyDiffsWithoutFileTypes(
      Arrays.asList("cpp", ".java")).accept("/dir/File.java"));
  }

  @Test
  public void shouldAcceptIfFileHasExtensionWithoutDot() {
    Assert.assertTrue(new OnlyDiffsWithoutFileTypes(
      Arrays.asList(".cpp", "java")).accept("/dir/File.java"));
  }

  @Test
  public void shouldRejectIfFileDoesNotMatchExtensions() {
    Assert.assertFalse(new OnlyDiffsWithoutFileTypes(
      Arrays.asList("cpp", ".java")).accept("/dir/File.css"));
  }
}
```

Fig. 3. Test case `OnlyDiffsWithFileTypesTest` for class `OnlyDiffsWithFileTypes` and generated test case `GenOnlyDiffsWithFileTypesTest` for class `OnlyDiffsWithoutFileTypes`, for RepoDriller software system.

original template test. In order to find the method that will replace that occurring in the original method call, a method with a similar name will be selected. As a general rule, each method name will be decomposed into its constituting words, according to conventions on the use of upper cases, to form a *bag of words*. Hence, e.g. a method `getNumberOfValues()` will produce a bag of words consisting of *{get, number, of, values}*. Then, set comparisons will be made to find the possible substitution method, selected among the most similar ones, on the basis of the Jaccard similarity index for the bag of words of each method. When computing similarity between the above method and method `getNumberValues()`, the intersection set will have cardinality 3 and the union set will have cardinality 4, hence similarity among methods will be determined as $3/4 = 0.75$. In this example, the two methods are highly similar, and can be interchanged for the sake of test generation.

The fourth case consists in finding which pair of methods from two different classes have a similar set of input parameters according to their type. Here, we list for each method of the pair the types of its parameters and find the Jaccard similarity index among the two sets of types.

For the purpose of generating a new test starting from a template test, the replacements of class and method names were performed by means of the JavaParser `LexicalPreservingPrinter` class, which yields Java code that is formatted correctly and in a standard style.

When writing code, some elements not meaningful for the compiler are however important to humans, such as. e.g., indentation and comments, and should be preserved. The above class provides method `setup(Node node)`, which prepares the node to be printed. The node that is passed represents the `CompilationUnit`, it is therefore possible to make changes to the nodes (e.g. change the name of the class, change the name of the methods, insert comments, etc.), hence transforming the original code. Method `setup()` indicates the point from which all the changes will be made, which will then result in an output using method `print(Node node)`. Printing generates a new Java file containing the transformed code. Code writing was used for the generation of tests according to the substitutions described above (see Section III).

IV. Experiments and Results

For the analysed software system RepoDriller, and for the classes shown in Figure 2, unit test `GenOnlyDiffsWithFileTypesTest` was generated by taking as a template the existing unit test `OnlyDiffsWithFileTypesTest`. Figure 3 shows the code of existing and generated unit tests. Following the above considerations on the similarity of classes and methods, generated test case `GenOnlyDiffsWithFileTypesTest` has been produced by substituting occurrences of class `OnlyDiffsWithFileTypes` into occurrences of class `OnlyDiffsWithoutFileTypes`.

The proposed approach has been employed for analysing several Java software systems found on repositories. Since we exploit existing tests as templates for the generation of new tests, we have taken software systems from Maven Repository[3], which lets us quickly check the existence of tests.

Table I shows a summary of the metrics related to our test generation solution and produced for the software systems under analysis. For each analysed system, column *classes* shows the number of classes; column *tests* gives the number of existing tests; column *tested* gives the number of classes for which at least a test has been found in the repository; column

[3]https://mvnrepository.com

TABLE I

Metrics on the Analysed Software Systems, including the number of generated tests (column *gen*), and the number of classes previously not tested and covered by generated tests (column *cover*).

system	classes	tests	tested	single	gen	incr	cover	max	min	$t > 0.5$
argparse4j	60	29	31	3	21	72%	7	1	0.05	23
jnr-unixsocket	17	15	9	0	2	13%	0	0.8	0.06	2
junit4	180	228	94	9	61	27%	19	1	0.03	106
mybatis3	279	572	129	9	402	70%	63	1	0.03	495
plexus-io	50	14	18	2	6	43%	6	1	0.06	20
repodriller	57	33	32	3	26	79%	6	1	0.05	29
vertx-mail-client	38	54	18	8	10	19%	0	1	0.05	11

single gives the number of tests found in the repository that execute a single class; column *gen* gives the number of newly generated tests, thanks to our approach; column *incr* gives the percentage increment of available tests accruing from our test generation; column *cover* gives the number of classes for which a test has been generated that were not previously tested, hence increasing code coverage. Moreover, columns *max* and *min* give the maximum and minimum values of similarity found among a pair of classes on the project, respectively. Finally, column $t > 0.5$ gives the number of class pairs found to have a similarity greater than the threshold set as 0.5.

For the experiments, the minimum similarity threshold among classes has been set at 0.5, in order to select pairs of classes comparable enough to each other, and make the test generation effective. Among all software systems, similarity between pairs of classes was between 0.03 and 1. By setting the similarity threshold to 0.5, we have found a relatively low number of pairs (typically a bit less than 1%, $0,9\%$ for RepoDriller) compared to the total number of possible pairs. Still, for the analysed systems many classes have a comparable behaviour, i.e. between 2 and 495 pairs for the smallest and largest software system, respectively (see column $t{>}0.5$ in Table I).

For each pair of classes whose similarity is greater than the predetermined threshold, test generation was performed according to the four previously defined cases. The number of test cases generated ranged from 2 to 402 (see column *gen* in Table I). The percentage increment for tests was between 13% and 72%. Moreover, column *cover* shows that between 6 and 63 additional classes were tested. In our approach, classes that have no test cases in the repository are selected as candidates for the following analysis finding class similarity and applicable test cases to be used as templates. Hence, when it is possible to generate tests, code coverage is also improved.

For test generation, the number of class pairs that matched our third method substitutions strategy (i.e. method name similarity) were greater than the other cases (up to 245 for the largest software system). However, sometimes it was not possible to employ the third method substitution strategy or the fourth (i.e. input parameter similarity) either. Several methods (up to 80) were substituted by means of the first and the second cases. For such matchings a number of new tests were generated.

Summing up, obviously, the tests that can be generated for a software system depend both on the amount of classes that are found to have a similar behaviour and the existing tests that can be taken as a template.

V. Related Work

Randoop [17] produces test cases by generating random sequence of method calls after inspecting the class to be tested, unaware of the resulting code coverage. Our proposed approach aims at generating tests that execute class code as if tests were tailored to the code of classes.

EvoSuite [7] uses symbolic execution of the code to be tested to generate test cases containing sequences of method calls. Sequences of possible method calls are selected using an evolutionary approach to take the most fit over code coverage. Hence, it needs executing the software system under test several times, in order to check the fitness of each sequence of method calls. In our approach, the static analysis of given tests let us recognise whether some classes have no corresponding unit tests, and such classes will be considered for test generation.

With DynaMOSA [20], which is an extension of previous work MOSA [19], the authors model the coverage maximisation for test case generation as a *many-objective* optimisation problem. In this type of search problems, more than one *fitness* function has to be optimised; in this case, one for each target (e.g. statement) to be covered in the class under test. Traditional many-objective optimisation algorithms can deal with a relatively small number of objectives (up to 50 [14]), which is too low when compared with the number of coverage targets to be considered, even for a small class. The authors propose two selection strategies to overcome this limitation, making the multi-objective algorithm suitable for test case generation. Such strategies have been implemented by extending EvoSuite, implying, in this case too, the need to execute the system under test several times to check the fitness of each test sequence. In contrast, we only rely on static analysis to generate tests for new classes.

COFFEe [2] is a comprehensive framework for *Combinatorial Interaction Testing (CIT)* and *Fault Characterisation (FC)*. The authors provide a tool based on JUnit5 integrating several stages: input parameter modelling, test generation, test execution, and fault characterisation. The tester will still have to provide both the model for generating the combinations of input parameters and a unit test describing the test steps to be performed with every given parameter combination. Our approach can be used together with this framework, as it automatically generates unit tests for previously untested classes and avoids the manual preparation of such an artifact needed as input for COFFEe.

In [16], test cases are generated in order to cover as many paths as possible. Moreover, the number of test cases are reduced by finding paths covered by more than one test case. Their approach is computationally expensive, compared to ours, since it requires the execution of each test case. Moreover, it could be advantageously complemented by ours, in order to reduce computation time: along the lines we introduced, one could select pairs of classes with similar behaviour, pass only one class of the pair to the tool of [16] to generate tests cases, and then enlarge these.

Other approaches have been proposed to reduce the number of test cases, e.g. once they have been generated automatically, in order to reduce execution time, or when considering that code coverage has not been increased [9], [12], [18]. In our approach, by analysing the test code and the application code, we can reveal which classes have not been tested and generate proper test cases selectively.

VI. Conclusions

This paper has proposed a novel and effective approach capable of generating test cases which takes advantage of the knowledge gathered from the static code of classes to be tested and from the static code of existing test cases. As such, computation time for generating test cases is limited (depending on the amount of code to be analysed, and not on its execution time).

We have shown that the devised strategies for comparing classes and finding the pairs with similar behaviour, in order to reveal which test templates are applicable, are effective, since many analysed software systems have exhibited numerous enough pair of classes with similarities higher than 0.5 (the mid value was chosen as a threshold). Moreover, generated test cases were a significant number for the analysed software systems, showing that the proposed substitutions of class and method names performed on existing test cases (taken as templates) are a viable solution. Test cases generation managed to include also classes that had not been previously tested, hence increasing code coverage and enhancing the effectiveness of tests.

References

[1] K. Beck and E. Gamma. *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2000.

[2] J. Bonn, K. Foegen, and H. Lichter. A framework for automated combinatorial test generation, execution, and fault characterization. In *Proceedings of IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 224–233, 2019.

[3] A. Calvagna, A. Fornaia, and E. Tramontana. Random versus combinatorial effectiveness in software conformance testing: A case study. In *Proceedings of ACM Symposium on Applied Computing*, pages 1797–1802, 2015.

[4] A. Calvagna and E. Tramontana. Automated conformance testing of java virtual machines. In *Proc. of IEEE Conference on Complex, Intelligent, and Software Intensive Systems (CISIS)*, Taichung, Taiwan, 2013.

[5] A. Fornaia and E. Tramontana. Deduct: a data dependence based concern tagger for modularity analysis. In *Proceedings of IEEE Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 463–468, 2017.

[6] A. Fornaia and E. Tramontana. Is my code easy to port? using taint analysis to evaluate and assist code portability. In *Proceedings of IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 269–274, 2017.

[7] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proc. of ACM SIGSOFT Symposium and European conference on Foundations of software engineering*, 2011.

[8] D. Fucci, H. Erdogmus, B. Turhan, M. Oivo, and N. Juristo. A dissection of the test-driven development process: does it really matter to test-first or to test-last? *IEEE Transactions on Software Engineering*, 43(7):597–614, 2016.

[9] A. Gotlieb and D. Marijan. Flower: optimal test suite reduction as a network maximum flow. In *Proc. of ACM International Symposium on Software Testing and Analysis*, 2014.

[10] Y. Higo, S. Matsumoto, R. Arima, A. Tanikado, K. Naitou, J. Matsumoto, Y. Tomida, and S. Kusumoto. kgenprog: A high-performance, high-extensibility and high-portability apr system. In *Proceedings of IEEE Asia-Pacific Software Engineering Conference (APSEC)*, pages 697–698, 2018.

[11] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 426–437. IEEE, 2016.

[12] N. Jatana, B. Suri, P. Kumar, and B. Wadhwa. Test suite reduction by mutation testing mapped to set cover problem. In *Proc. of ACM International Conference on Information and Communication Technology for Competitive Strategies*, 2016.

[13] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov. An extensive study of static regression test selection in modern software evolution. In *Proc. of ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 583–594, 2016.

[14] B. Li, J. Li, K. Tang, and X. Yao. Many-objective evolutionary algorithms: A survey. *ACM Computing Surveys (CSUR)*, 48(1):1–35, 2015.

[15] M. Mongiovì, A. Fornaia, and E. Tramontana. A network-based approach for reducing test suites while maintaining code coverage. In *Proceedings of International Conference on Complex Networks and Their Applications*, pages 164–176. Springer, 2019.

[16] C. Murphy, Z. Zoomkawalla, and K. Narita. Automatic test case generation and test suite reduction for closed-loop controller software. Technical report, University of Pennsylvania, Department of Computer and Information Science, 2013.

[17] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for java. In *Proc. of OOPSLA Companion*, 2007.

[18] S. Panda and D. P. Mohapatra. Regression test suite minimization using integer linear programming model. *Software: Practice and Experience*, 47(11):1539–1560, 2017.

[19] A. Panichella, F. M. Kifetew, and P. Tonella. Reformulating branch coverage as a many-objective optimization problem. In *Proceedings of IEEE international conference on software testing, verification and validation (ICST)*, pages 1–10, 2015.

[20] A. Panichella, F. M. Kifetew, and P. Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, 2017.

[21] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on software engineering*, 27(10):929–948, 2001.

[22] N. Smith, D. van Bruggen, and F. Tomassetti. *JavaParser: Visited–Analyse, transform and generate your Java Code Base*. Leanpub, 2018.

[23] S. Thummalapenta, T. Xie, N. Tillmann, J. De Halleux, and Z. Su. Synthesizing method sequences for high-coverage testing. *ACM SIGPLAN Notices*, 46(10):189–206, 2011.

[24] E. Tramontana. Automatically characterising components with concerns and reducing tangling. In *Proceedings of IEEE Computer Software and Applications Conference (COMPSAC)*, pages 499–504, 2013.

[25] S. Xu, H. Miao, and H. Gao. Test suite reduction using weighted set covering techniques. In *Proc. of IEEE International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, 2012.

[26] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability*, 22(2):67–120, 2012.