GRT: An Automated Test Generator using Orchestrated Program Analysis

Lei Ma*, Cyrille Artho†, Cheng Zhang§, Hiroyuki Sato*, Johannes Gmeiner‡ and Rudolf Ramler‡
* University of Tokyo, Japan {malei, schuko}@satolab.itc.u-tokyo.ac.jp
†AIST / ITRI, Japan c.artho@aist.go.jp
§ University of Waterloo, Canada c16zhang@uwaterloo.ca
‡Software Competence Center Hagenberg, Austria {johannes.gmeiner, rudolf.ramler}@scch.at

***Abstract*—While being highly automated and easy to use, existing techniques of random testing suffer from low code coverage and defect detection ability for practical software applications. Most tools use a pure black-box approach, which does not use knowledge specific to the software under test. Mining and leveraging the information of the software under test can be promising to guide random testing to overcome such limitations.**

Guided Random Testing (GRT) implements this idea. GRT performs static analysis on software under test to extract relevant knowledge and further combines the information extracted at run-time to guide the whole test generation procedure. GRT is highly configurable, with each of its six program analysis components implemented as a pluggable module whose parameters can be adjusted. Besides generating test cases, GRT also automatically creates a test coverage report. We show our experience in GRT tool development and demonstrate its practical usage using two concrete application scenarios.

***Keywords*—Automatic test generation, random testing, bug detection, static analysis, dynamic analysis**

I. Introduction

Unit testing is an important quality assurance technique in software development, but manually crafting unit test cases is labor-intensive. In general, unit testing consists of three steps: *creating test inputs*, *executing tests*, and *checking test outputs*. An easy-to-use test generation tool can be valuable, if it automates one or more of these steps and has reasonable test effectiveness (e. g., high code coverage, defect detection ability) to assist the development of practical software systems.

In unit testing of object-oriented programs, the methods in the software under test (SUT) are the basic entities. To exercise the target behavior of a method under test (MUT) m, test inputs type-compatible with the parameter types are needed to execute m. Starting with primitive values, which are the fundamental elements of object states, object-oriented testing techniques construct test input objects incrementally. The primitive values are used as parameters to invoke constructors and methods to create more complex object states. The sequence of method calls is further expanded to produce more diverse object states so that desirable inputs can be generated for the MUT. In the end, the generated test cases usually consist of three parts: *initial primitive (or constant) values*, *method sequences*, and *assertions*. As generating assertions amounts to the classic test oracle problem [36], [3], [29], which requires extra information of expected behavior, most automatic testing techniques focus on finding better initial values and creating useful method sequences.

However, a large number of software systems cannot be tested exhaustively due to the enormous state space of possible initial values and combinations of method calls. Random testing [14] automatically generates test sequences to execute different paths in an MUT. However, random testing techniques are known to suffer from low code coverage on real-world applications. To improve random testing, feedback-directed random testing (FRT) [25], [27], [26] incrementally builds more and longer test sequences by randomly selecting an MUT and reusing previously generated method sequences (that return objects) as input to execute the MUT until a time limit is hit. Unfortunately, FRT still suffers low code coverage in some cases [33], [18], [38].

In this paper, we present a fully automatic test generator, called *GRT*, which generates test cases with high code coverage [21]. GRT shares the common idea of FRT by leveraging the feedback of method execution, but it addresses several issues of random testing and FRT through an orchestration of static and dynamic program analyses. GRT only requires the SUT and the necessary dependency libraries to work: No extra information, such as program specifications, previous test cases, or code bases, is needed. As a tool for unit testing, GRT provides the following advantages:

1) GRT combines information from static and dynamic analysis to guide test generation. This enables GRT to generate useful unit test cases with high code coverage for real-world programs.
2) GRT is fully automatic and requires no extra information other than the SUT and its dependency libraries to generate test cases. A user just needs to start GRT and waits for the test generation process to complete.
3) GRT is well modularized in that all of its components are pluggable under the basic FRT framework. Each component can be easily improved or replaced. The modularity of GRT's architecture makes it highly flexible and adaptable. GRT can serve as platform to facilitate further research in software testing.

II. GRT: Guided Random Testing

During our design of GRT [21], we identified several common concerns that potentially influence the testing effectiveness of FRT:

Selection of initial values. Although random testing produces test inputs randomly in general, to compose useful

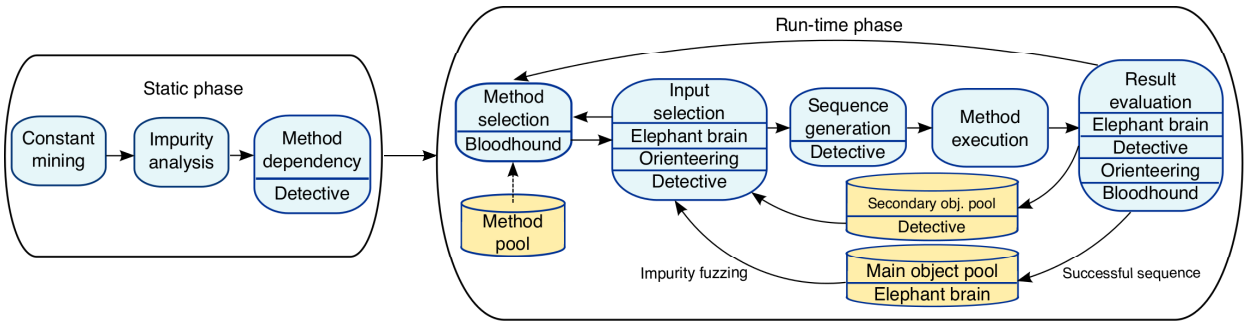

Fig. 1. The workflow overview of guided random testing (GRT), which combines information from static analysis with run-time guidance.

complex objects, suitable initial values, especially primitive values, are required as the basis for input object construction.

Selection of MUT. To achieve higher code coverage, MUTs with uncovered code are natural targets for generating new test cases. Run-time coverage information can be used to steer the process of test generation towards MUTs whose code is not well covered.

Selection of methods to generate input data. Object-oriented testing executes methods and usually requires objects as input arguments. The selection of the right methods to generate valid sequences (that return objects) is a central and difficult problem in random testing. We identify two main aspects of method selection:

1) **Properties of methods.** The main properties of a method are characterized by the method signature, including the return type and parameter types. In object-oriented programs, high-level super-types, instead of exact subtypes, are often used to declare parameters for the purpose of design flexibility (i. e., to allow run-time substitution of objects with compatible types). Considering both static and dynamic type information can capture properties of methods more accurately. Moreover, the purity of methods, whether they have side effects or not in changing program states, is an important method property relevant to test generation.
2) **Dependencies of methods.** It is usually infeasible to consider all applicable methods (MUTs and their dependent methods) in test generation, because this wastes testing effort on methods that are not the targets. However, if the pool of methods to choose from is limited (e. g., only the MUTs), necessary objects of certain types cannot be created by FRT, making all the MUTs depending on such objects unable to be tested. Thus, the pool of available methods needs to be chosen appropriately. Dependencies among methods can be explored to identify relevant methods.

Composition of method sequences. FRT generates new method sequences based on concatenation of existing sequences. Existing tools mostly compose sequences by selecting a sequence returning an object with compatible type at random, but a well designed strategy can optimize this task by considering several factors related to the cost-effectiveness of testing, such as the time required to execute a sequence and the likelihood to cover more unexecuted code.

To address these issues and also keep the technique open to further enhancement, we propose the architecture of GRT [21] as shown in Fig. 1: First, GRT performs static analysis on the MUTs to extract information. Second, tests are generated at run-time guided by dynamic analysis. The run-time phase is based on the original framework of feedback-directed random testing [27], with GRT components plugged into each step. Specifically, GRT has six collaborative components, which are orchestrated so that program information (either static or dynamic) is extracted by some components at specific steps and passed to others to facilicate their tasks. The rest of this section describes details of the design and implementation of each GRT's component.

Constant mining enriches the initial set of primitive values by extracting constants from the SUT through static analysis. The mined constant values are useful to create more diverse objects to cover more branches. We implement constant mining as an abstract interpreter based on the ASM interpreter framework [5]. It analyzes the Java bytecode of each class under test (CUT) and performs constant propagation and folding. Since constant mining analyzes Java bytecode, rather than source code, it is able to work even if the source code of the SUT is unavailable.

To be scalable, constant mining manages the extracted constants in two levels, namely *global level* and *local level*. At the global level, constants are prioritized so that those used frequently by many classes are more likely to be used for test input generation. At the local level, when generating test cases for an MUT, constants mined from the MUT's declaring class (i. e., the CUT) are preferable, as they are often more relevant to the MUT and thus effective to cover specific branches.

Impurity boosts the effect of constant mining through input fuzzing by favoring methods that can change object states. Meanwhile, the input fuzzing can be enhanced with constants extracted by constant mining. For primitive numeric values, we adopt a Gaussian distribution, with the assumption that the extracted constants from constant mining are already close to satisfying some branch conditions. The deviation of the Gaussian distribution and the ratio to perform fuzzing are provided as parameters that can be adjusted according to SUT-specific features. String values are fuzzed by randomly inserting, replacing or removing a character, and taking a substring of a given string.

It is challenging to fuzz non-primitive objects, as it requires to identify methods that mutate the object states. To this end,

we perform purity analysis to determine whether an MUT has side effects and prefer methods with side effects. When a non-primitive object is selected as input and requires fuzzing, we use the corresponding impure methods to mutate the object before passing it to the MUT. The current GRT implements Impurity based on the ReIm & ReImInfer framework [15] because of its scalability and robustness. ReIm & ReImInfer analyzes the reference purity information of an SUT, and infers side effects of all methods. This framework does not require an expensive, complete analysis of the program. Its algorithm infers method purity using the type system.

Elephant Brain manages all the objects stored in the object pool by using exact (run-time) type. The type information is obtained by analyzing objects returned by the execution of the method sequences through reflection. We store a method sequence into the object pool by using the exact type of the returned object as key. When an MUT requires an input with declared static type T, we search in the object pool for sequences that return object with compatible type of T. We perform random selection to use exact type match or subtype match with a user-defined ratio (default=0.5). Elephant Brain can find objects that cannot be generated using static type information alone, so it equips other components, such as Impurity and Detective, with additional exact type information to improve their effectiveness.

Detective constructs objects that are needed to test MUTs but cannot be found in the main object pool. Detective performs static analysis on MUTs to find their input and output type dependencies, and constructs the missing input objects on demand at run-time. When an input object of type T is needed, Detective first randomly constructs an object with type T or subtypes of T (if T represents an interface or an abstract class). Then, it analyzes all methods of T that can return the corresponding objects, and recursively finds methods returning objects that are type compatible with parameters of the previous methods. After finding all relevant methods, Detective executes them in a topological order: the method that does not depend on return values of other methods is executed first. If an object of the required type T is successfully generated, it is stored in the main object pool for further use. Other generated objects are kept in a secondary object pool for further fast querying. Using the secondary object pool avoids polluting the main object pool with many irrelevant objects.

Orienteering estimates the execution cost of each method sequence by analyzing its execution time and length of the sequence (i. e., the number of method invocation statements). We calculate a weight for each sequence based on these two kinds of information. The weight is so defined that less costly method sequences have higher probability of being selected, while expensive method sequences that generate interesting new objects are also included. Orienteering accelerates the overall test generation process of GRT.

Bloodhound intelligently selects MUTs that are not well covered. By covering more code of an MUT, more program states can be reached, which potentially creates objects to further improve testing effectiveness. Bloodhound records coverage information of the target MUTs, calculates a weight for each of them, and randomly selects the next MUT based on the weight. In most cases, it is prohibitively expensive to monitor code coverage after the execution of each single method sequence. In addition, always selecting MUT with low code coverage is problematic, since they can contain branches that are too difficult to cover. To solve these issues, Bloodhound recalculates the coverage weight of each MUT periodically, and balances the selection of MUTs with low code coverage and those selected less often.

Bloodhound adapts the robust coverage evaluation tool JaCoCo [17] to monitor code coverage. JaCoCo does not support online coverage profiling, so we modify JaCoCo so that each covered code fragment and branch of the executed method sequence are stored for further analysis. When test generation starts, Bloodhound loads and instruments the CUT with the probes to collect code coverage information. If the corresponding part of the code is executed, the probes in that executed code fragment will be marked as covered by the executed test case.

III. Usage Scenarios

As GRT is designed to be an automatic test generator, which does not require any user intervention after being launched, the current implemention is a command-line tool without graphical user interface. This section demonstrates two main usage scenarios of GRT: (1) detecting defects in the current version of a program and (2) generating regression tests for future program versions.

A. Defect Detection

The first usage scenario is using GRT to generate test cases to find defects in the current version of a program. There are only three steps to follow:

1) Prepare the program to be tested. Specifically, the user first builds the program and puts the built classes (*.class files) of the SUT into a directory `<sut_dir>`. Then the user needs to place all libraries necessary to build the program and the program source code (optional) into directories `<lib_dir>` and `<src_dir>`, respectively.
2) Run GRT to generate test cases using the command below:
```
java -jar GRT.jar <sut_dir>\
--mptest_SUTLibPaths=<lib_dir>\
--mptest_SUT_Src_Path=<src_dir>\
--timelimit=<time_budget>\
--output-tests=fail
```
GRT also provides a shell script template that allows the user to directly execute GRT with slight adjustments on the parameters. After GRT completes test generation, it outputs JUnit test cases that have failed.[1]
3) For the failed test cases, the user needs to perform manual analysis to check whether they detect real bugs in the SUT or they are false positives.

For non-trivial programs, GRT may generate many failed tests, making it difficult to manually analyze all of them. We recommend the following steps of manual analysis:

[1]By default, GRT uses exceptions and the predefined contracts of the base class `java.lang.Object` (e. g., the reflexivity property of `equals()`) as the test oracle [27].

1) Filter out test cases that are obviously caused by issues that are known to be less relevant to real defects. These issues include:
   - **Deprecated methods.** Methods may be marked as *deprecated* if they contain or expose design flaws. Such methods are usually removed from the program in the future. Tests involving such code often reveal previously known issues. The failed test cases may be considered as true positives, but as the code tends to be removed soon, the possible defects may not be fixed. Ignoring these issues allows the user to focus on unknown flaws.
   - **Internal packages.** Some packages are specific to the given reference implementation and not supposed to be used by others. Possibly unsafe API uses found by test cases are therefore irrelevant.
   - **Stack overflows in containers.** Container classes, such as those in the Java Collections, Apache Collections and Guava, usually allow recursive nesting of data. For example, a test case can insert a list l_1 into another list l_2, followed by an insertion of l_2 into l_1. As a result, operations such as list iteration or a call to method `toString()` will never terminate on the list objects.[2]
2) Compare the failure stack traces of the failed test cases, and classify similar failed tests (with similar stack traces) as one issue for further analysis.
3) For the remaining classified failed tests, check whether they reveal true defects by analyzing the program code and relevant documentation (if available).

B. Regression Test generation

The second usage scenario is using GRT to generate regression test cases to help developers capture defects introduced by program changes. This scenario consists of four steps:

1) Preparation the program to be tested is identical to that of the first scenario: After compilation, the source files, compiled files, and necessary libraries have to be placed in their respective directories.
2) Run GRT to generate test cases using the command below:
```
java -jar GRT.jar <sut_dir>\
--mptest_SUTLibPaths=<lib_dir>\
--mptest_SUT_Src_Path=<src_dir>\
--timelimit=<time_budget>\
--output-tests=pass
```
   Note that the command sets the option `--output-tests` to `pass`, instead of `fail`. It is critical, because GRT is used to output all passed test cases, rather than failed ones, as regression tests that capture the behavior of the current version of the SUT.
3) Although all the generated test cases have passed during test generation, some of them may still fail in future executions (when being used as regression tests) on the *same* program, due to non-deterministic behavior. To mitigate this problem, GRT provides a test cleaner that removes such flaky tests iteratively.
4) When program changes occur, the user can run the generated test cases as normal JUnit test cases to detect regression in the new program version.

[2]It is possible to make iteration robust against infinite recursion, but a fix entails keeping track of previously visited object instances during iteration. This requires an amount of memory that is linear to the size of the collection, which worsens the problem in most cases.

IV. Experience

The design and implementation of GRT have lasted for more than a year. During this period, we gained valuable experience in using GRT on real-world programs.

We applied GRT to over 30 open source projects. The results show its scalability and effectiveness in achieving high code coverage [21]. We also used GRT to detect real bugs by generating test cases for 10 well developed and maintained programs. The test cases successfully detected 23 previously unknown bugs that are confirmed by developers. For example, in the Apache Commons Codec project, GRT generates the simple test case as below:

```
1  public void test() throws Throwable {
     DoubleMetaphone var0 = new DoubleMetaphone();
     boolean var3 =
       var0.isDoubleMetaphoneEqual("", "", false);
5  }
```

After the test case failed, our manual analysis revealed that the failure was caused by the incorrect implementation of encoding empty strings (or other special strings). In the method `isDoubleMetaphoneEqual` in class `DoubleMetaphone`, a string is first encoded and then compared by invoking method `equals()`. However, some strings (e. g., the empty string) are encoded as `null`, causing a null pointer exception during the comparison of equality.

Another example is a test case generated by GRT on Apache Commons Compress. The test case reveals a bug of `ChangeSet` by adding the input data and invoking `deleteDir`, which leads to an exception. Randoop cannot detect the bug, because `ByteArrayInputStream` is not a class in the SUT and none of its methods is present in the method pool of Randoop, making it impossible to create an instance of the class (i. e., `var5`). In GRT, the Detective component starts with `ChangeSet` and finds missing objects transitively, so GRT can identify the need for an instance of `java.io.ByteArrayInputStream`.

```
1  public void test() throws Throwable {
     ChangeSet var0 = new ChangeSet();
     SevenZArchiveEntry var1 = new SevenZArchiveEntry();
     byte[] var3 = new byte[] {(byte)10, (byte)10};
5
     java.io.ByteArrayInputStream var5 =
       new java.io.ByteArrayInputStream(var3, 1, 1);
     TarArchiveInputStream var6 =
       new TarArchiveInputStream((java.io.InputStream)var5);
10
     var0.add((ArchiveEntry)var1,
        (java.io.InputStream)var6);
     var0.deleteDir(
         "0x7875␣Zip␣Extra␣Field:␣UID=1000␣GID=1002");
15 }
```

We used GRT to detect defects collected by the Defects4J framework [19]. The purpose of Defects4J is to facilitate controlled study in software testing. For each bug, Defects4J has a fixed version, a buggy version, and a patch. The fixed version is converted into the buggy version by applying the patch, which simulates the situation of introducing a bug during program changes. We used GRT to generate regression test cases for the fixed (correct) version of each pair of program versions, applied the patch, and ran the regression test cases to check whether they could reveal the introduced bug in

the buggy version. In this setting of regression testing, GRT detects 147 (out of 224) real bugs from four studied subject programs [21].

GRT also participated in the third round of the SBST Java unit testing tool contest, in which test cases were generated on a per class basis (instead of complete programs) [30], [22]. Among all the seven participating tools, GRT obtained the highest score, in terms of code coverage, mutation score, and execution time. While the detailed scores indicate that GRT still has room for improvement (the combined result of all participating tools is better than individual tools), the overall result demonstrates the effectiveness and efficiency of GRT.

V. Related Work

While a large body of work on automated testing exists [1], [24], [28], [9], this section only discusses publicly accessible tools that are closely related to GRT.

1) Random Test Generators: Most mainstream random test generators create input objects as method sequences, whose returning objects can be used as inputs for further test generation. JCrasher [6] adopts top-down input object construction by using a parameter graph to analyze method dependencies and to create test sequences. Eclat [25] and Randoop [27], [26] perform feedback-directed random test generation to incrementally construct complex objects from primitive types, in a bottom-up style. GRT adopts the basic framework of feedback-directed random testing and addresses several common issues to improve test effectiveness.

Some tools combine external knowledge and random testing to improve code coverage and bug detection ability. MSeqGen [33] mines client code bases to extract frequently used sequence patterns to assist test generation. OCAT [18] captures input object states by running developer-written test cases, and directly using these objects as input for random test generation. Similar to OCAT, Palus [38] trains a method sequence model from existing test cases, and uses the derived model to guide test generation at run-time. GRT also performs program analysis and uses the analysis results to guide test generation at run-time. GRT differentiates from previous work in that GRT is fully automated and requires no additional sources of information (e. g., existing test cases and code bases) other than the SUT itself.

2) Systematic Test Generators: Symbolic execution represents input as symbolic values and uses abstract semantics for execution to collect the path constraints. The constraints are solved by constraint solvers to derive input object states that satisfy corresponding branch conditions. Symbolic PathFinder [20] is a representative test generator based on symbolic execution. There are also tools combining random testing and symbolic execution, including DART [13], Cute and JCute [32], [31], Pex [34], and Dsc [16].

Bounded exhaustive testing is an alternative systematic approach. It exhaustively generates method sequences up to a small bound of sequence length, and is implemented in tools such as TestEra [23], Korat [4] and Symstra [37]. The technique implemented by GRT is orthogonal to systematic approaches, and the combination of both kinds of techniques is an important direction of our future work.

3) Evolutionary Test Generators: Evolutionary test generators (e. g., eToc [35], Testful [2]) start with randomly generated test sequences and use evolutionary algorithms to evolve and search for test sequences that optimize certain fitness functions (e. g., branch coverage, mutation score). EvoSuite represents the current state-of-the-art evolutionary test generator [10], [7], [12], which goes beyond traditional techniques by adopting a hybrid approach to generate and optimize the whole test suites. EvoSuite is shown to be effective in achieving high coverage on real-world software [9], [11]. EvoSuite is also used to study the impact of constant seeding strategies in search-based software testing [8]. Compared with EvoSuite, GRT currently does not include some optimizations that could further improve efficacy of test generation. Using a basic framework with lower complexity (guided random testing vs. evolutionary testing), GRT usually requires less time for test generation, which is desirable when testing resources are limited. We believe that some components of GRT could also be adapted to evolutionary test generators for further enhancement.

VI. Discussion

The original motivation of developing GRT is to improve feedback directed random testing by addressing a number of key issues that limit the test code coverage. As the basic framework of FRT is quite general, GRT is highly flexible. Constant mining can be extended to mine constant values from other sources of information, such as existing test cases, execution traces, and even documentation. Impurity can be replaced with other side-effect analyses, for example, to know which variables may be changed by a method, in addition to whether or not the method is pure. Similarly, other components can be improved with more advanced program analysis techniques. The overall result can be used to demonstrate the usefulness of new techniques or combinations of existing techniques in the context of test generation.

As an automatic test generator, GRT can be directly used to generate test cases for different purposes. First, GRT itself can be considered as a representative random testing tool (similar to Randoop) for software testing research. For example, it can be the baseline in evaluation. Moreover, test cases generated by GRT can be useful to drive dynamic analysis or produce dynamic information (such as code coverage and runtime states) to facilitate related software engineering research, including model inference, fault localization, program repair, etc.

More information on GRT's usage, including the detailed descriptions of parameters and the complete set of experimental results, its video demonstration and reproduction package are available at:

http://www.sites.google.com/site/grtprojectut/download

VII. Acknowledgments

This work was supported by the *SEUT* project from the University of Tokyo, *kaken-hi* grants 23240003 and 26280019, the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria in the frame of the *COMET* center *SCCH*.

REFERENCES

[1] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. Mcminn. An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Softw.*, 86(8):1978–2001, Aug. 2013.

[2] L. Baresi and M. Miraz. Testful: Automatic unit-test generation for java classes. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE'10, pages 281–284, Cape Town, South Africa, 2010.

[3] L. Baresi and M. Young. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A., 2001.

[4] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA'02, pages 123–133, Roma, Italy, 2002.

[5] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[6] C. Csallner and Y. Smaragdakis. Jcrasher: an automatic robustness tester for java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

[7] G. Fraser and A. Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE'11, pages 416–419, Szeged, Hungary, 2011.

[8] G. Fraser and A. Arcuri. The seed is strong: Seeding strategies in search-based software testing. In *Proceedings of the 5th International Conference on Software Testing, Verification and Validation*, ICST'12, pages 121–130, Los Alamitos, CA, USA, 2012.

[9] G. Fraser and A. Arcuri. Sound empirical evidence in software testing. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 178–188, Piscataway, NJ, USA, 2012. IEEE Press.

[10] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Trans. Softw. Eng.*, 39(2):276–291, Feb. 2013.

[11] G. Fraser and A. Arcuri. A large-scale evaluation of automated unit test generation using evosuite. *ACM Trans. Softw. Eng. Methodol.*, 24(2):8:1–8:42, Dec. 2014.

[12] J. P. Galeotti, G. Fraser, and A. Arcuri. Extending a search-based test generator with adaptive dynamic symbolic execution. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA'14, pages 421–424, San Jose, CA, USA, 2014.

[13] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.

[14] R. Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[15] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst. Reim & ReImInfer: Checking and inference of reference immutability and method purity. *SIGPLAN Not.*, 47(10):879–896, Oct. 2012.

[16] M. Islam and C. Csallner. Dsc+mock: A test case + mock class generator in support of coding against interfaces. In *Proceedings of the 8th International Workshop on Dynamic Analysis*, WODA'10, pages 26–31, Trento, Italy, 2010.

[17] JaCoCo v0.6.4. http://www.eclemma.org/jacoco/.

[18] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang. Ocat: Object capture-based automated testing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA'10, pages 159–170, Trento, Italy, 2010.

[19] R. Just, D. Jalali, and M. D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA'14, pages 437–440, San Jose, CA, USA, 2014.

[20] K. S. Luckow and C. S. Păsăreanu. Symbolic pathfinder v7. *SIGSOFT Softw. Eng. Notes*, 39(1):1–5, Feb. 2014.

[21] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Rudolf. GRT: Program-analysis-guided random testing. In *IEEE/ACM Int. Conference on Automated Software Engineering*, ASE'15, Lincoln, Nebraska, USA, 2015. To appear.

[22] L. Ma, C. Artho, C. Zhang, H. Sato, M. Hagiya, Y. Tanabe, and M. Yamamoto. GRT at the SBST 2015 tool competition. In *The 8th Int. Workshop on Search-Based Softare Testing*, SBST'15, pages 48–51, Florence, Italy, 2015.

[23] D. Marinov and S. Khurshid. Testera: A novel framework for automated testing of java programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ASE'01, pages 22–31, San Diego, CA, USA, 2001.

[24] P. McMinn. Search-based software test data generation: A survey: Research articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156, June 2004.

[25] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 504–527, Glasgow, UK, 2005.

[26] C. Pacheco, S. K. Lahiri, and T. Ball. Finding errors in .net with feedback-directed random testing. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA'08, pages 87–96, Seattle, WA, USA, 2008.

[27] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE'07, pages 75–84, Minneapolis, MN, USA, 2007.

[28] C. S. Pasareanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Softw. Tools Technol. Transf.*, 11(4):339–353, Oct. 2009.

[29] M. Pezze and C. Zhang. Automated test oracles: A survey. *Advances in Computers*, 95:1–48, 2014.

[30] U. Rueda, T. Vos, and I. Prasetya. Unit testing tool competitions - round three. In *The 8th Int. Workshop on Search-Based Softare Testing*, SBST'15, pages 19 – 24, Florence, Italy, 2015.

[31] K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 419–423, Seattle, WA, USA, 2006.

[32] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. *SIGSOFT Softw. Eng. Notes*, 30(5):263–272, Sept. 2005.

[33] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Mseqgen: Object-oriented unit-test generation via mining source code. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE'09, pages 193–202, Amsterdam, The Netherlands, 2009.

[34] N. Tillmann and J. De Halleux. Pex: White box test generation for .net. In *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Prato, Italy, 2008.

[35] P. Tonella. Evolutionary testing of classes. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA'04, pages 119–128, Boston, USA, 2004.

[36] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[37] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 365–381, Edinburgh, UK, 2005.

[38] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined static and dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA'11, pages 353–363, Toronto, Ontario, Canada, 2011.