Continuous Test Generation on Guava

José Campos[1], Gordon Fraser[1], Andrea Arcuri[2], and Rui Abreu[3]

[1] Department of Computer Science, The University of Sheffield, UK
[2] Scienta, Norway, and University of Luxembourg, Luxembourg
[3] PARC, Palo Alto, USA, and University of Porto, Portugal

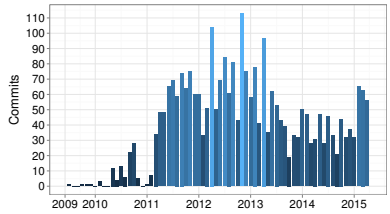
Abstract. Search-based testing can be applied to automatically generate unit tests that achieve high levels of code coverage on object-oriented classes. However, test generation takes time, in particular if projects consist of many classes, like in the case of the Guava library. To allow search-based test generation to scale up and to integrate it better into software development, *continuous test generation* applies test generation incrementally during continuous integration. In this paper, we report on the application of continuous test generation with EvoSuite at the SSBSE'15 challenge on the Guava library. Our results show that continuous test generation reduces the time spent on automated test generation by 96%, while increasing code coverage by 13.9% on average.

Keywords: Search-based testing, automated unit test generation, continuous integration, continuous test generation

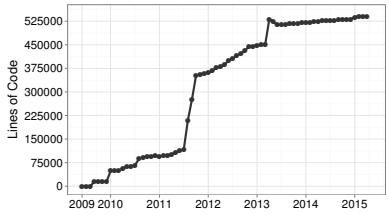
1 Introduction

To support software testers and developers, tests can be generated automatically using various techniques. Search-based testing is well suited for the task of generating unit tests for object-oriented classes, where a test typically consists of a sequence of method calls. Although search-based unit test generation tools have been successfully applied to large projects [4], performance is not one of the strengths of search-based test generation: Every fitness evaluation requires costly test execution. For example, we found that our EvoSuite [2] search-based unit test suite generator requires somewhere around 2 minutes of search time to achieve a decent level of coverage on most classes, and more time for the search to converge. While 2 minutes may not sound particularly time consuming, it is far from the instantaneous result developers might expect while writing code. Even worse, a typical software project has more than one class — for example, Guava version 18 has more than 300 classes, and consequently generating tests for 2 minutes per class would take more than 10 hours.

In practice, however, generating tests for an entire software project of this size may not be required frequently. Instead, software projects evolve and grow over size. For example, Figure 1a shows that the Guava library is actively developed, with many commits per month. However, not every class is changed every day, and the software takes time to grow (Figure 1b). An opportunity to exploit this incremental nature of software is offered by the regular build



(a) Number of commits per month.



(b) Lines of code over time.

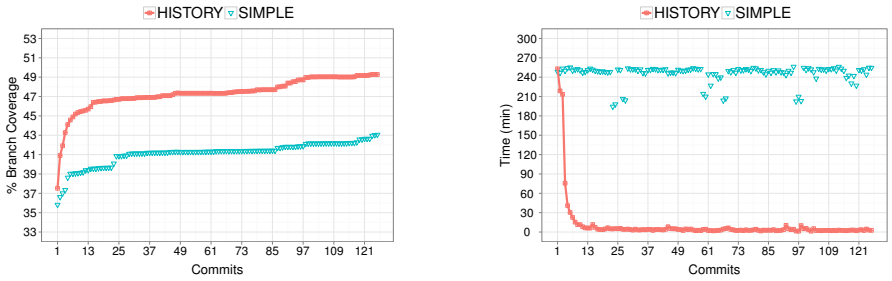
Fig. 1: Activity of the Guava library based on the Git log.

and test phases applied every day as part of continuous integration. *Continuous test generation* [1] is the synergy of automated test generation with continuous integration: Tests are generated during every nightly build, but resources are focused on the most important classes, and test suites are built incrementally over time. CTG supports the application of *test suite augmentation* [7, 8], but importantly addresses the time-budget allocation problem of individual classes, is not tied to an individual coverage criterion, and is applicable for incremental test generation, even if the system under test did not change. In this paper, as part of the SSBSE'15 challenge, we describe the application of continuous test generation on the Guava library.

2 Continuous Test Generation (CTG)

Experimentation on automated test generation typically (e.g., [4]) consists of applying a tool to an entire software project, and to allocate the same amount of time to every artifact (e.g., class under test). In practice, even if one would restrict this test generation to code that has been changed since the last time of test generation, the computational effort (e.g., CPU time and memory used) to generate tests may exceed what developers are prepared to use their own computers for while they are working on them. However, integration and testing is often performed on remote continuous integration systems — these continuous integration systems are well suited to host continuous test generation.

Continuous integration is often invoked on every commit to a source code repository, or for nightly builds which test all the changes performed since the last nightly build. Applying test generation during continuous integration creates a scheduling problem: Which classes should be tested, and how much time should be spent on each class? For example, in order to distinguish trivial classes from more complex classes, EvoSuite uses the number of branches in the class to allocate a time budget proportional to the size of a class. The choice of which class to test can be based on change information: First, as new or modified code is more likely to be faulty [6], EvoSuite prioritises changed classes, and allocates more time to them. Second, EvoSuite monitors the coverage progress of each class: As long as invoking EvoSuite leads to increased coverage, it is invoked even if the class has not been changed. However, once EvoSuite can no longer increase coverage, we can assume that all feasible goals have been covered and stop the invocation of the tool on those classes.



(a) Test suite branch coverage. (b) Time spent on test generation.

Fig. 2: Branch coverage, and time spent on test generation for the Guava open source case study over the course of 126 commits.
For each commit we report the average branch coverage over all classes and over the five runs for "History", and "Simple" strategies.

As a further optimisation, we exploit the fact that a test generation tool may be repeatedly applied to the same classes, with or without changes. Instead of initialising the initial population of the genetic algorithm in EvoSuite completely randomly, we can include the previous version of the test suite as one individual of the initial population of the genetic algorithm.

To assess the effectiveness of CTG, we consider the following scenario: CTG is invoked after every commit to the source code repository as part of continuous integration. To simulate this scenario, we selected all compiled Git commits to the Guava project between version 17 and 18. In total 126 (out of 148) different versions of Guava have been selected. This represents 45,025 classes in total, and an average of 357 classes per version. We configured CTG with an amount of time proportional to the number of classes in each project, i.e., one minute per class under test. We compare two configurations of CTG: The baseline configuration (*Simple*) tests each class for one minute for every invocation. The advanced configuration (*History*) allocates budget based on change and coverage history, and reuses past results. We repeated each experiment five times (more repetitions were not possible due to the computational costs of the experiment).

In the first commit, the number of classes tested by both configuration is the same, and thus so is the time spent on test generation (Figure 2b) and the achieved coverage (Figure 2a). Between commits 1 and 11, only nine classes have been changed which allowed the History strategy to reduce the time spent on test generation from 253 minutes (at first commit) to 8 minutes (at commit number 11). During the same time the coverage increased from 0.37 to 0.46, while during the same period the Simple strategy just increased coverage from 0.35 to 0.39 on average, spending on average 250 minutes per commit. After 126 commits, History achieved 49% branch coverage on average, having spent a total of 1,333 minutes on test generation. On the other hand, Simple just achieved 43% coverage with 30,937 minutes spent on test generation. This means that History achieved a relative improvement of +13.9% in branch coverage within -95.7% of Simple's time.

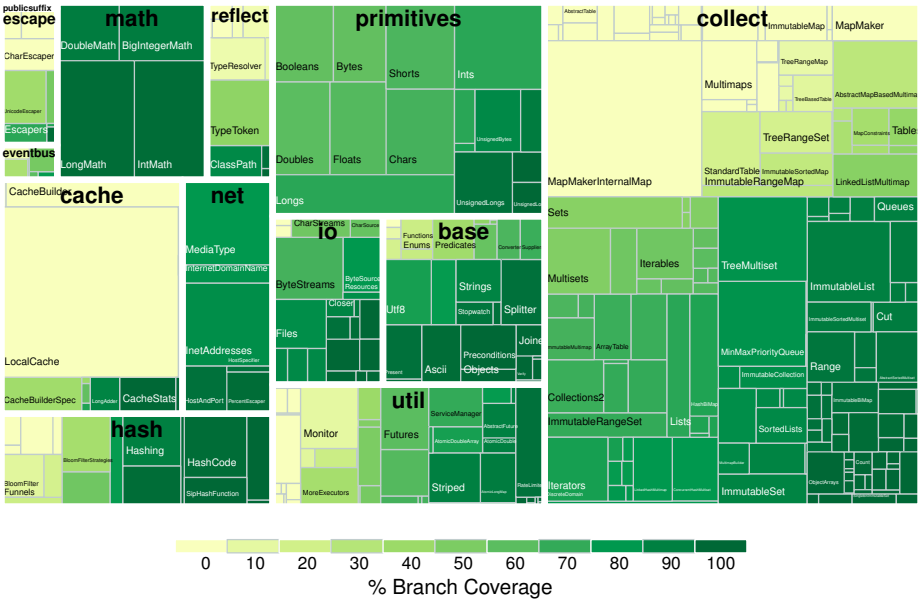


Fig. 3: Branch coverage for the Guava version 18.
The area of each box is proportional to the number of branches in each class, and the colour represents the coverage achieved in 10 minutes, averaged over 30 repetitions.

3 Guava's Challenges

Although the previous section showed that a CTG strategy based on *History* is able to achieve more coverage than a *Simple* strategy, even after 126 commits EvoSuite only covered 49% of the total number of branches. In this section we discuss the challenges of testing the Guava project. Figure 3 visualises the coverage achieved by EvoSuite after 10 minutes of test generation (averaged over 30 runs) in a tree-map, where the size of a box is proportional to the number of branch coverage goals in that class, and the colour represents the coverage achieved. EvoSuite identified 359 testable (i.e., top-level public classes) out of 468 source files. In total there are 1,692 classes including member classes, anonymous classes, but the latter are tested as part of their containing classes. Overall, there seems to be a large number of classes with high coverage, and some classes where EvoSuite failed to achieve any substantial coverage.

Figure 3 contains two noticeable problematic classes, `MapMakerInternalMap` and `LocalCache`, which both have around no coverage, but represent a large share of the area in the figure. `MapMakerInternalMap` consists of only six methods which all simply return a Boolean value, but also has a complex constructor that receives a `MapMaker` instance, and a total of 52 internal classes (including interfaces, abstract classes, and enum classes). As EvoSuite considers all methods of inner classes as target methods equal to the methods of the main class under test, most of the time is spent on trying to instantiate these inner classes. However, these classes all have several generic type parameters, and EvoSuite strug-

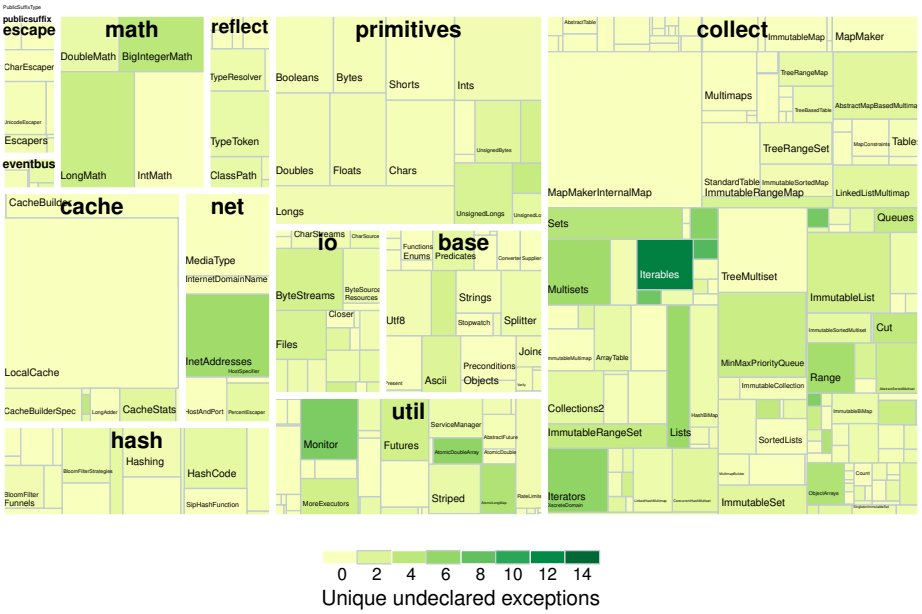


Fig. 4: Number of undeclared thrown exceptions for the Guava version 18. Results are based on 30 repetitions.

gles to efficiently resolve these[4]. A further problem for `MapMakerInternalMap` lies in the constructor, as the top right hand corner of Figure 3 shows that EvoSuite struggles to cover `MapMaker` for similar reasons, and so it rarely manages to instantiate a valid instance to pass into the `MapMakerInternalMap` constructor; even if it does, this needs to satisfy many constraints before the entire constructor executes without exception. The `LocalCache` class similarly has a complex constructor that requires complex dependencies with generic types, and many inner classes with further generic type parameters. Clearly, adding a more sophisticated generic type system or treating inner classes separately would lead to higher coverage.

Beside achieving a certain degree of code coverage, EvoSuite can also find faults in the system under test [3]. Typical examples are methods crashing by throwing unexpected exceptions. Even in the case of Guava, many potential faults were found, as shown in Figure 4. For example, there are 14 distinct exceptions in class `Iterables` alone. However, a closer look at some of these exceptions show that those were expected, although not declared in the signature (e.g., with the `throws` keyword) of those methods. Java does not enforce to declare expected unchecked exceptions (e.g., input validation) in the signatures, although those are very important for a library to understand its behaviour in case of wrong inputs. To complicate the matter even further, a developer

[4]Note that Java would allow ignoring these type parameters, but we argue that this would severely degrade the usefulness of the generated tests to developers, and might miss important coverage scenarios [5].

might mention the expected exceptions (e.g., a `NullPointerException` if an input parameter is null) only in the JavaDocs, using the tag `@Throws`. However, JavaDocs do not become part of the compiled bytecode. Therefore, if a developer fails to write proper method declarations, an automated testing tool cannot distinguish between real, critical faults and expected failing input validations.

4 Conclusions

In this paper, we have presented the results of applying EvoSuite on the *Guava* library. Using a continuous test generation approach reduces the amount of time spent on test generation dramatically, while leading to overall higher coverage. On the majority of classes, EvoSuite achieves a substantial degree of coverage, but a closer look revealed problems with generic types and inner classes, which pose new research and engineering challenges to be faced in the future. To learn more about EvoSuite, visit our Web site at: http://www.evosuite.org/.

Acknowledgments. This work is supported by the EPSRC project "EXOGEN" (EP/K030353/1) and by the National Research Fund, Luxembourg (FNR/P10/03).

References

1. Campos, J., Arcuri, A., Fraser, G., Abreu, R.: Continuous Test Generation: Enhancing Continuous Integration with Automated Test Generation. In: IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 55–66. ASE '14, ACM, New York, NY, USA (2014)
2. Fraser, G., Arcuri, A.: EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In: ACM SIGSOFT European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE). pp. 416–419. ESEC/FSE '11, ACM, New York, NY, USA (2011)
3. Fraser, G., Arcuri, A.: 1600 faults in 100 projects: Automatically finding faults while achieving high coverage with EvoSuite. Empirical Software Engineering (EMSE) pp. 1–29 (2013)
4. Fraser, G., Arcuri, A.: A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. ACM Transactions on Software Engineering and Methodology (TOSEM) 24(2), 8:1–8:42 (Dec 2014)
5. Fraser, G., Arcuri, A.: Automated test generation for java generics. In: Software Quality. Model-Based Approaches for Advanced Software and Systems Engineering, pp. 185–198. Springer (2014)
6. Graves, T.L., Karr, A.F., Marron, J.S., Siy, H.: Predicting Fault Incidence Using Software Change History. IEEE Transactions on Software Engineering (TSE) 26(7), 653–661 (Jul 2000)
7. Santelices, R., Chittimalli, P.K., Apiwattanapong, T., Orso, A., Harrold, M.J.: Test-Suite Augmentation for Evolving Software. In: IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 218–227. ASE '08, IEEE Computer Society, Washington, DC, USA (2008)
8. Xu, Z., Kim, Y., Kim, M., Rothermel, G., Cohen, M.B.: Directed Test Suite Augmentation: Techniques and Tradeoffs. In: ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE). pp. 257–266. FSE '10, ACM, New York, NY, USA (2010)