An Application of Model Seeding to Search-based Unit Test Generation for Gson

Mitchell Olsthoorn[0000−0003−0551−6690], Pouria Derakhshanfar[0000−0003−3549−9019], and Xavier Devroey[0000−0002−0831−7606]

Delft University of Technology, Delft, The Netherlands
m.j.g.olsthoorn@tudelft.nl, p.derakhshanfar@tudelft.nl,
x.d.m.devroey@tudelft.nl

Abstract. Model seeding is a strategy for injecting additional information in a search-based test generation process in the form of models, representing usages of the classes of the software under test. These models are used during the search-process to generate logical sequences of calls whenever an instance of a specific class is required. Model seeding was originally proposed for search-based crash reproduction. We adapted it to unit test generation using EvoSuite and applied it to Gson, a Java library to convert Java objects from and to JSON. Although our study shows mixed results, it identifies potential future research directions.

Keywords: Model seeding · Search-based software testing · Case study

1 Introduction

Over the years, several techniques have been developed to generate unit tests by applying search-based algorithms to source code. Among the existing tools, EvoSuite is one of the references in the state-of-the-art for Java unit test generation [5]. It has been developed and maintained over the years and received several contributions to improve code coverage and mutation score of the generated tests or to generate tests for specific purposes. Despite the numerous improvements, one of the challenges still faced by EvoSuite is the generation of complex objects with logical (*i.e.,* not random) sequences of method calls. There exist several strategies to address this challenge partially. Among those, *seeding* [7] consists of the injection of additional information that will be used during the search. For instance, constant values collected from the source code, and the usage (as-is) of objects collected from an existing test suite.

In their recent study, Derakhshanfar *et al.* [3] propose to abstract the behavior of the different classes of the system under test (SUT) using a *transition system model*. For each class, one transition system describes the sequences of method calls previously observed on the instances of that class. They seeded those models to a search-based crash reproduction algorithm to generate complex objects and found that it improved the overall crash reproduction rate. Crash reproduction does not seek to cover all the elements of a class under test (CUT) but rather to generate a test exercising a specific behavior causing the software to crash.

For unit test generation, the coverage of different elements of a CUT also requires specific objects that might be difficult to generate randomly. In this paper, we applied model seeding for unit test generation using EvoSuite on a set of eight classes from the Gson library. We compare *model seeding* to the *default* configuration of EvoSuite *w.r.t.* the branch coverage and mutation scores achieved by the generated tests.

2 Evaluation setup

Classes under test. Gson is a *Java serialization* and *deserialization library to convert Java Objects into JSON and back.*[1] It is used as a dependency by more than 222 000 projects on GitHub. We used Gson v.2.8.7 (5924 LOC) and selected 8 classes with at least one method (with the exception of the `toString` and `equals` method) with a cyclomatic complexity above 3: `Gson`, `JsonReader`, `JsonTreeReader`, `JsonTreeWriter`, `JsonWriter`, `LinkedHashTreeMap`, `LinkedTreeMap`, and `TypeAdapters`. The overall branch coverage of the existing manually written tests is 79 %, and the overall mutation score is 75 %.

Learning the models. We followed Derakhshanfar *et al.*'s [3] approach and generated our models using the existing source code and tests of the Gson library. For each class used in the project, each time an object is created, we collected the sequence of methods called on this object. For that, we statically analyzed the source code of Gson and dynamically executed (a heavily instrumented version of the) existing test cases. The models are then *learned* from the collected call sequences using a 2-gram inference. Learning the models is a one-time operation. Models are then seeded to the different executions of EvoSuite. In total, we collected 328 models for 328 different classes. The average number of states is 7 and the average number of transitions is 15. We rely on the implementation of Derakhshanfar *et al.* [3] to collect call sequences and learn the different models, and on EvoSuite-RAMP,[2] a customized version of EvoSuite [5] for unit test generation.

Configurations. Model seeding works either *online* or *offline*. In the offline mode, during the initialization of the search, for each model, EvoSuite-RAMP creates a fixed number of objects by selecting *abstract behaviors* from the model. For each selected abstract behavior, it instantiates the object, calls the corresponding methods, and adds the result to an *object pool.* Whenever an object is required, the search process *copies* (with a defined probability `p_object_pool`) one object and its method calls from this object pool. Additionally, during the initialization of the population, EvoSuite-RAMP can also copy (with a defined probability `seed_clone`) an instance of the CUT (as-is) from the object pool and use it as a plain (initial) test. In the online mode, objects are created during the search process using the same procedure. The main difference with the offline mode is that the objects are created *on demand*, slightly overloading the search process. For our evaluation, we used the *online* mode as it does

[1] https://github.com/google/gson
[2] https://github.com/STAMP-project/evosuite-ramp

not overload the initialization and generates only objects required by the search (and therefore leaves more budget for the search itself). We used probabilities `p_object_pool` $= 0.3$, and `seed_clone` $= 0.3$, following Derakhshanfar *et al.* [3].

To select abstract behaviors, EvoSuite-RAMP supports *random* selection, corresponding to random walks in the models, and *dissimilarity* selection, trying to increase diversity in the selected behaviors. For our evaluation, we used *random selection* as it gave slightly better results in our initial trial.

In our evaluation, we compare unit test generation with model seeding activated (*model s.*) to the default EvoSuite configuration (*default*). For both configurations, we used *DynaMOSA* [6] with the default set of objectives (*i.e.,* branch, line, weak mutation, input, output, method, and exception coverage) and a search budget of 180 seconds. Additionally, we deactivated model seeding after 90 seconds to increase exploration. We ran our evaluation (1600 runs) on a server with 12 CPU cores @ 3.50 GHz. The total execution time for unit test generation took around 40 minutes. Our replication package is available on Zenodo [4].

Data analysis. For each class under test, we compare the generated test suites *w.r.t.* their branch coverage (reported by EvoSuite-RAMP) and their mutation score, computed using Pit v1.4.3 [1] with `ALL` mutation operators activated. The total execution time for the mutation analysis of the 1600 generated test suites took around 2 days. We used the the non-parametric Wilcoxon Rank Sum test ($\alpha = 0.05$) for Type I error, and the Vargha-Delaney statistic $\hat{A}_{12}$ to evaluate the effect size between *model s.* and *default*.

3 Results

Figure 1 presents the branch coverage and mutation score of the test suites generated using the *default* and *model s.* configurations. On average, the highest branch coverage is achieved by the *default* configuration for the class `JsonTreeReader` with 92.04 %. The lowest branch coverage is, on average, also achieved by the *default* configuration for the class `TypeAdapters` with 50.04 %. For the mutation score, the highest average mutation score is achieved by the *model s.* configuration for the `TypeAdapters` class with a score of 93.75 %, and the lowest average mutation score is also achieved by *model s.* configuration on the `LinkedHashTreeMap` class with an average of 35.91 %.

For each class, we compared the coverage and mutation score of the generated test suites. The configuration *model s.* achieved significant better results ($\alpha \leq 0.05$) for three classes (with two small and one large $\hat{A}_{12}$ magnitudes). It performs worse compared to the *default* configuration for two classes (with one small and one medium $\hat{A}_{12}$ magnitudes). For the mutation score, the *model s.* configuration performed significantly better ($\alpha \leq 0.05$) on three classes (with two small and one large $\hat{A}_{12}$ magnitudes). The *default* configuration performed better for two classes (with one small and one medium $\hat{A}_{12}$ magnitudes).

In general, our results are mixed. The *model s.* configuration can lead to an improvement of the mutation score with, in general, a lower variability than the *default* configuration. The most interesting class of our evaluation is the

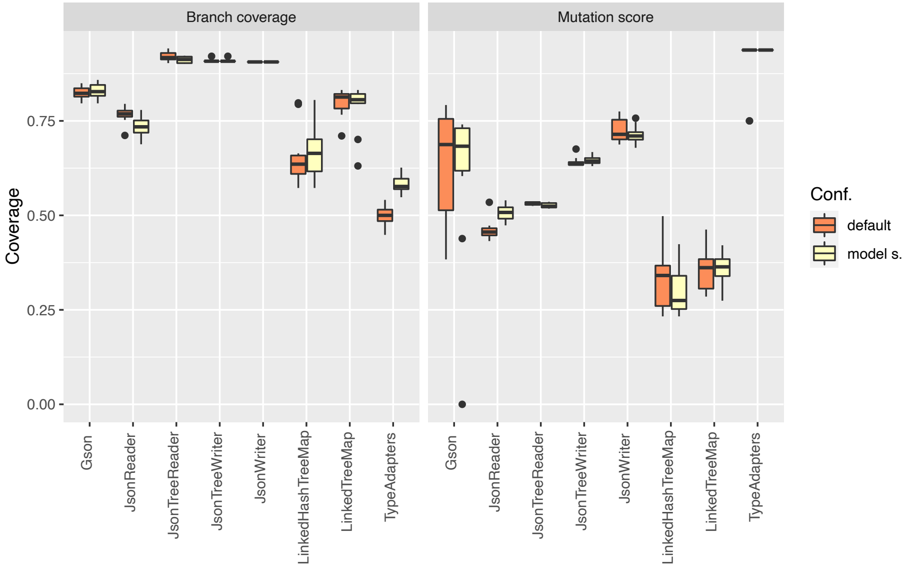

Fig. 1. Coverage of the tests generated using the *default* and *model s.* configurations.

`JsonReader` class for which *model s.* achieves a significantly worse branch coverage (p-value $= 4.98 \times 10^{-15}$) with a large magnitude ($\hat{A}_{12} = 0.180$) than *default*, but, in the same time, also achieves a significantly better (p-value $= 9.26 \times 10^{-25}$) mutation score, also with a large magnitude ($\hat{A}_{12} = 0.92$). We focus our discussion on the `JsonReader` class.

4 Discussion and future work

Code complexity and model generation. From analyzing the project using CodeMR,[3] we see that the `JsonReader` class is the most complex class of the project with a *very-high* complexity rate and a Weighted Method Count (WMC) of 359 for 891 lines of code (LOC). Complex code is a well-known challenge for search-based testing algorithms.

At the same time, the model generated from the collected call sequences for `JsonReader` is highly connected with an average degree (*i.e.,* the average number of incoming and outgoing transitions per state) of 9.0, 252 transitions for only 28 states, and a BFS height of 6 (*i.e.,* number of levels when navigating the model using a breadth-first search algorithm). This permissiveness of the model tends to indicate that the usages are not well captured and that the model can provide only limited guidance. Future research will investigate the usages of other learning approaches (including higher-values of n for the n-gram inference)

[3] `https://www.codemr.co.uk`

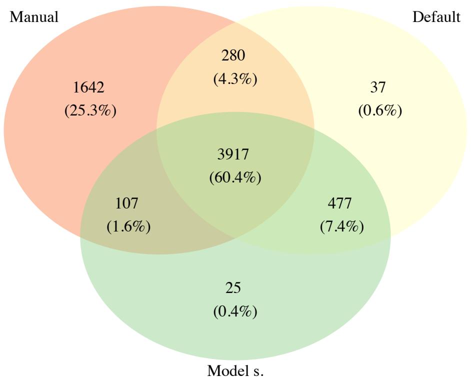

Fig. 2. Combined mutation analysis of the `JsonReader` class with the number and percentage of mutants killed by the *manually* written, *model s.*, and *default* test suites.

to better reflect the usages of the classes. Additionally, the models are created from the source code and the existing tests. We followed the procedure defined by Derakhshanfar *et al.* [3] with the same assumption that the existing tests are representative of valid usages of the classes (for crash reproduction). However, this assumption might not be right for unit testing. Therefore, future research will investigate other sources of call sequences, like projects using GSON, as well as including information about object and parameter values for those calls.

Mutation analysis. To further investigate the mutants killed by the generated tests, we performed a combined mutation analysis on the `JsonReader` class. For that, we used all the tests generated out of the 10 rounds of execution and the manually written tests from the `JsonReaderTest` and `JsonReaderPathTest` test suites. Figure 2 presents the number of mutants killed, grouped by the source of the test suite. Mutants killed by more than one test suite coming from different sources are placed at intersections in the diagram.

We see that between 7.8 % and 8 % of the mutants are killed only by a *generated test suite*. Additionally, we see that between 62 % and 64.7 % of the mutants are killed by both manually written and automatically generated tests, which tends to confirm that effort on testing can be reduced using automated approaches. Figure 2 also shows that the largest amount of mutants only killed by EVOSUITE (7.4 %) are killed both by the *default* and the *model s.* configurations. This tends to indicate that the randomness of the evolution process helps to explore new areas of the search space, compared to manually written tests. We also see that 25 mutants are killed only by the *model s.* configuration, and 37 mutants are killed only by the *default* configuration. Finally, from figs. 1 and 2, we see that the *default* configuration achieves a significantly lower mutation score, compared to *model s.*, but kills a larger diversity of mutants (at least once) when the 10 test suites are merged together.

Test case understandability. In this case study, we only consider the functional properties (*i.e.,* branch coverage and mutation score) of the generated tests. Recent studies have investigated other aspects of generated tests, like the readability and understandability by a developer [2]. We believe that, by generating objects with common usages, model seeding can contribute to improving test case readability and understandability. From the manual analysis of the test cases killing 25 mutants only killed by the *model s.* configuration, we could retrace the usages of the `JsonReader` class observed in the test case in the usage model of the class. The confirmation that having such usages in the test cases helps in reading and understanding them is left for future work.

5 Conclusion

In this case study, we applied model seeding for unit test generation using EvoSuite-RAMP on eight classes from the Gson library. We compared *model seeding* to the *default* configuration of EvoSuite. Overall, results are mixed. Using model seeding can lead to an improvement of branch coverage and mutation score in some cases. We also discussed several aspects of model seeding for unit test generation and identified potential future research directions regarding the collection of call sequences and generation of the models, and the usage of model seeding to improve the understandability of automatically generated tests.

Acknowledgement. This research was partially funded by the EU Horizon 2020 ICT-10-2016-RIA “STAMP” project (No.731529).

References

1. Coles, H., Laurent, T., Henard, C., Papadakis, M., Ventresque, A.: PIT: a practical mutation testing tool for Java. In: ISSTA ’16. pp. 449–452. ACM (2016). https://doi.org/10.1145/2931037.2948707
2. Daka, E., Campos, J., Fraser, G., Dorn, J., Weimer, W.: Modeling readability to improve unit tests. pp. 107–118 (2015). https://doi.org/10.1145/2786805.2786838
3. Derakhshanfar, P., Devroey, X., Perrouin, G., Zaidman, A., Deursen, A.: Search-based crash reproduction using behavioural model seeding. Softw. Test. Verif. Reliab. **30**(3), e1733 (may 2020). https://doi.org/10.1002/stvr.1733
4. Derakhshanfar, P., Olsthoorn, M., Devroey, X.: Replication package of An Application of Model Seeding to Search-based Unit Test Generation for Gson (Jul 2020). https://doi.org/10.5281/zenodo.3963956
5. Fraser, G., Arcuri, A.: EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In: ESEC/FSE ’11. p. 416. ACM (2011). https://doi.org/10.1145/2025113.2025179
6. Panichella, A., Kifetew, F.M., Tonella, P.: Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. IEEE Trans. on Softw. Eng. **44**(2), 122–158 (2018). https://doi.org/10.1109/TSE.2017.2663435
7. Rojas, J.M., Fraser, G., Arcuri, A.: Seeding strategies in search-based unit test generation. Softw. Test. Verif. Reliab. **26**(5), 366–401 (aug 2016). https://doi.org/10.1002/stvr.1601