SpecFuzzer: A Tool for Inferring Class Specifications via Grammar-based Fuzzing

Facundo Molina
IMDEA Software Institute
Madrid, Spain
facundo.molina@imdea.org

Marcelo d'Amorim
North Carolina State University
Raleigh, USA
mdamori@ncsu.edu

Nazareno Aguirre
University of Rio Cuarto and CONICET
Rio Cuarto, Argentina
naguirre@dc.exa.unrc.edu.ar

***Abstract*—In object-oriented design, class specifications are primarily used to express properties describing the intended behavior of the class methods and constraints on class' objects. Although the presence of these specifications is important for various software engineering tasks such as test generation, bug finding and automated debugging, developers rarely write them.**

In this tool demo we present the details of SPECFUZZER, a tool that aims at alleviating the problem of writing class specifications by using a combination of grammar-based fuzzing, dynamic invariant detection and mutation analysis to automatically *infer* specifications for Java classes. Given a class under analysis, SPECFUZZER uses *(i)* a generator of candidate assertions derived from a grammar automatically extracted from the class; *(ii)* a dynamic invariant detector –Daikon– in order to discard the assertions invalidated by a test suite; and *(iii)* a mutation-based mechanism to cluster and rank assertions, so that similar constraints are grouped together and the stronger assertions are prioritized. The tool is available on GitHub at https://github.com/facumolina/specfuzzer, and the demo video can be found on YouTube: https://youtu.be/IfakNCbzOUg.

***Index Terms*—Oracle Problem, Specification Inference, Grammar-based Fuzzing.**

I. Introduction

Software specifications are abstract descriptions of the software's intended behavior. In object-oriented (OO) design, where software is organized as a set of classes equipped with methods, a class specification describes the intended behavior of a class' methods and the constraints that the class' objects need to satisfy. Class specifications are typically described *informally*, through natural language documentation of class APIs. However, specifications become significantly more useful when expressed *formally*, through constraints known as *contracts* [8]. Contracts have been widely used for a variety of tasks, including test generation [1], [2], [6], automated debugging [7], [12], bug finding [5], [11], and verification [5]. Although the use of formal contracts has provided considerable benefits, developers rarely write them.

In this context, different techniques for inferring class specifications have been proposed [3], [10], [13], with the aim of providing developers with automated ways of equipping their implementations with contracts. Often, the expressiveness of the specification language used by these approaches is limited. Daikon [3], a mature powerful technique for contract inference based on dynamic analysis, supports a restricted set of templates, from which assertions are generated. It is limited to simple assertions (e.g., no direct support for quantification), or requires the developer to manually extend the assertion language. GAssert [13] and EvoSpex [10], two more recently proposed techniques for contract inference, try to address the expressiveness limitation of Daikon by supporting more expressive assertion languages, but their respective extensions focus on specific kinds of constraints: GAssert focuses on logical/arithmetic constraints (without quantification) and EvoSpex focuses on object navigation constraints (supporting simple logical and arithmetic operators). Moreover, as both techniques are based on evolutionary search, they are difficult to extend or adapt to support further expressions, as the evolutionary algorithms are targeted for the specific languages supported by the corresponding tools.

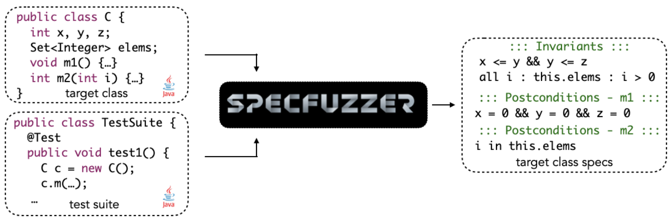

```
public class C {
  int x, y, z;
  Set<Integer> elems;
  void m1() {…}
  int m2(int i) {…}
}
```
```
public class TestSuite {
  @Test
  public void test1() {
    C c = new C();
    c.m(…);
  …
```
```
::: Invariants :::
x <= y && y <= z
all i : this.elems : i > 0
::: Postconditions - m1 :::
x = 0 && y = 0 && z = 0
::: Postconditions - m2 :::
i in this.elems
```

Fig. 1. Inferring class specifications with SPECFUZZER.

SPECFUZZER [9] is a tool for generating likely class specifications, in a way that makes it easier to adapt to different specification expressions, through the use of *fuzzing*. As illustrated in Figure 1, SPECFUZZER takes as input a target class and a test suite for the class, and produces a set of properties that are likely to hold in specific program points of the class, including class invariants, preconditions, and postconditions. SPECFUZZER uses grammar-based fuzzing to automatically generate constraints that can be used as candidate specifications by an invariant detection tool. Fuzzing [14], traditionally used to efficiently produce structured random data for testing, has two key advantages in this context: (1) it eliminates the need for developers to manually define candidate assertions, and (2) it enables developers to straightforwardly adapt the language of assertions by manipulating the fuzzing grammar, overcoming language limitations of existing approaches.

This paper extends our previous paper [9] by providing further implementation details of SPECFUZZER, as well as

```
public class SortedList {
  private int elem;
  private SortedList next;
  private static final int SENTINEL = Integer.MAX_VALUE;

  /* Constructors */
  public SortedList() { this(SENTINEL, null); }
  private SortedList(int elem, SortedList next) {
    this.elem = elem;
    this.next = next;
  }

  /* Insert an element in the list */
  void insert(int data) {
    if (data > elem) {
      next.insert(data);
    } else {
      next = new SortedList(elem, next);
      elem = data;
    }
  }

  /* Clear the list */
  void clear() {
    elem = SENTINEL;
    next = null;
  }
}
```

Fig. 2. Class SortedList implementing an ordered list of integers.

instructions on how to use the tool. The source code, data, evaluation subjects, and demo video are available at:

https://github.com/facumolina/specfuzzer

II. Usage

The envisioned users of SpecFuzzer are researchers and Java practitioners who may be in need of obtaining contracts for a given Java class, either with the aim of analyzing its behavior or to improve some program analysis task that can benefit from contracts. SpecFuzzer takes as input a Java class C, and a test suite $\mathcal{T}_c$ for C. As shown in Figure 4, if a test suite for the given class is absent, one can automatically build one using any automated test generation tool, such as Randoop [11]. From these two inputs, SpecFuzzer will use a combination of grammar-based fuzzing, dynamic invariant detection (e.g,. Daikon [3]), and mutation analysis (e.g., Major [4]) in order to produce class specifications for C. SpecFuzzer generates plausible assertions characterizing properties at different program points in C, including preconditions, postconditions and class invariants.

To better illustrate the use of SpecFuzzer, let us consider the class SortedList shown in Figure 2, implementing ordered lists of integers. The implementation has two instance fields, elem and next, representing the value of a linked list node and the reference to the next node, respectively. It also has a class field (SENTINEL) storing a special value – the maximum Java integer value– as a mark for the end of the list. The sentinel should be placed at the end of the list and should not be repeated. The default constructor creates a node marking the end of the list. The insert method takes the

```
// ::: Invariants :::
1- SENTINEL in this.*(next).x
2- all n : this.*(next) : n.x <= n.next.x
// ::: Postconditions - insert(int data) :::
3- some n : this.*(next) : n.x = data
// ::: Postconditions - clear() :::
4- elem = SENTINEL
5- next = null
```

Fig. 3. Sample invariant and postconditions inferred by SpecFuzzer for class SortedList.

integer data as parameter and inserts it in its correct sorted position in the linked list. As it is not possible for any integer value to be greater than the sentinel, the search is guaranteed to insert the element before the sentinel. Finally, the clear method simply resets the list to its initial state.

To infer class specifications, the use of SpecFuzzer involves two steps. The first step performs a setup to set the conditions for the inference, while the second step is the actual fuzzing-based inference process. The setup involves the following tasks: the extraction of a grammar G_c, expressing the language of candidate assertions for C (that will later on be used by our assertion fuzzer), the execution of the test suite $\mathcal{T}_c$ to collect the executions traces in the *dtrace* format[1] (used by Daikon [3], the dynamic detector we employ), and the generation of mutants of the target class C (to be used by our mutation-based assertion selector) with the Major tool[2]. Assuming that the test suite SortedListTest for the class SortedList is available, we can perform the setup step by running the following command:

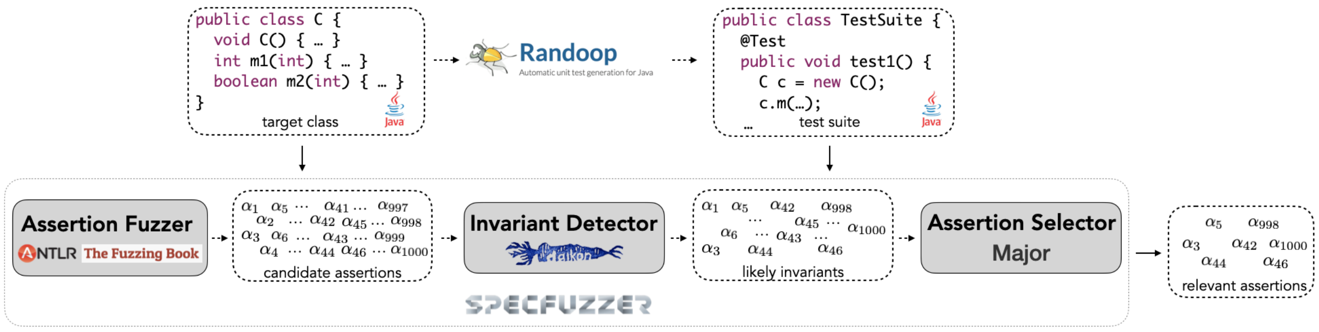
```
./specfuzzer.sh --setup <cp> SortedList SortedListTest
```

where <cp> is the target class classpath. All necessary files will be generated within the working directory, leaving it ready for the inference step. In this step, SpecFuzzer will first use the assertion fuzzer to generate candidate assertions to be fed to the dynamic detector. The dynamic detector will then be executed to determine which of the candidate assertions are allegedly valid, i.e., consistent with the behavior observed in the test suite. Finally, the assertion selector will eliminate irrelevant and equivalent assertions by using a mechanism based on mutation analysis and clustering. For our SortedList class we can simply execute the script with the infer option:

```
./specfuzzer.sh --infer <cp> SortedList SortedListTest
```

The execution will summarize the number of candidate assertions considered, the number of assertions identified by the dynamic detector, and the number of assertions after eliminating the irrelevant/redundant ones. All these sets of assertions will be saved in a corresponding file. The actual output of SpecFuzzer, the set of valid and stronger assertions, will be saved in a specific file. As an example, a sample of the inferred specifications for the SortedList class is shown in Figure 3. Assertions 1 and 2 are class invariants, stating that

[1]https://plse.cs.washington.edu/daikon/download/doc/daikon.html
[2]https://mutation-testing.org



```
public class C {
  void C() { … }
  int m1(int) { … }
  boolean m2(int) { … }
}
```

```
public class TestSuite {
  @Test
  public void test1() {
    C c = new C();
    c.m(…);
…
```

target class
Java
Randoop
Automatic unit test generation for Java
test suite
Java
Assertion Fuzzer
ANTLR
The Fuzzing Book
α_1 α_5 ⋯ α_{41} ⋯ α_{997} α_2 ⋯ α_{42} α_{45} ⋯ α_{998} α_3 α_6 ⋯ α_{43} ⋯ α_{999} α_4 ⋯ α_{44} α_{46} ⋯ α_{1000}
candidate assertions
Invariant Detector
Daikon
α_1 α_5 α_{42} α_{998} α_{45} α_{1000} α_6 α_{43} α_3 α_{44} α_{46}
likely invariants
Assertion Selector
Major
α_5 α_{998} α_3 α_{42} α_{1000} α_{44} α_{46}
relevant assertions
SPECFUZZER

Fig. 4. An overview of SPECFUZZER.

the sentinel value is always present in the list (1), and that the list is always sorted (2). Assertion 3 is a postcondition for method `insert`, and states that there is at least one element in the list that is equal to the value being inserted. Finally, assertions 4 and 5 are postconditions for method `clear`, and state that the list is reset to its initial state.

III. SPECFUZZER

Let us now dive into the implementation details of SPECFUZZER. Figure 4 shows an overview of SPECFUZZER's architecture, which involves three main components: the *Assertion Fuzzer*, the *Invariant Detector*, and the *Assertion Selector*.

A. Assertion Fuzzer

The assertion fuzzer is in charge of generating candidate assertions for the target class C. Before producing the candidates, it first extracts a grammar G_c, expressing the language of candidate assertions. To create such grammar, the process starts with a base grammar B, captured with the ANTLR parser generator[3]. Essentially, B defines an assertion language supporting numerical comparisons, logical expressions, membership expressions, and quantified expressions. By inspecting the structure of C (fields declared in C as well as fields declared in classes reachable from C), the grammar G_c is obtained by adding or removing symbols and production rules from B. Intuitively, for every field and navigation expression, a terminal symbol of the corresponding type is defined.

Once G_c has been produced, the fuzzer uses it to generate a set $\mathcal{A}_c$ of candidate assertions for C. Our fuzzer is implemented in Java, reproducing a general grammar-based fuzzer written in Python, taken from The Fuzzing Book [14]. Assertions are produced by generating derivations of G_c –i.e., strings in $\mathcal{L}(G_c)$– to obtain the set $\mathcal{A}_c$. Intuitively, the process begins with the start symbol and keeps expanding non-terminal symbols until no more non-terminals are present. Through this derivation mechanism, our fuzzer is able to produce candidate predicates very efficiently. In our tool, it is configured to generate up to 2,000 different candidates, but the number of assertions to generate can be specified. It is worth noting that, as the grammar G_c has been specifically extracted for the input class C, all the assertions generated by the fuzzer are guaranteed to express properties over C.

B. Invariant Detector

The goal of the invariant detector is to determine the candidate assertions that are allegedly valid, in the sense that they are consistent with the behavior of C observed according to the test suite $\mathcal{T}_c$. Our invariant detector is built as an extension of Daikon[4], a state-of-the-art tool for likely invariant detection [3]. It incorporates into Daikon the ability of interpreting and evaluating the candidate assertions produced by the fuzzer. This is achieved by using a mechanism provided by Daikon to incorporate new constraints.

The output of the invariant detector is a set $\mathcal{I}_c \subseteq \mathcal{A}_c$ of likely invariants, composed of those assertions that were not falsified by the test suite $\mathcal{T}_c$. Also, each likely invariant is associated with its corresponding program point (precondition, postcondition or class invariant).

C. Assertion Selector

The third component of SPECFUZZER is responsible of obtaining the most relevant assertions from the set $\mathcal{I}_c$ of likely invariants. Starting from $\mathcal{I}_c$, the *Invariant Selector* produces a subset $\mathcal{R}_c \subseteq \mathcal{I}_c$ by discarding weak assertions and then grouping together similar assertions, so as to take a single representative from each partition. To achieve this, the selector takes as input the set $\mathcal{I}_c$ of likely invariants, and the set of mutants of the input class C, generated with Major. Then, to get the relevant assertions, two main steps are performed:

1) Weak Assertions Detection: The detection of weak assertions is based on the following idea: if a likely invariant cannot be falsified by any mutant of C, then it is considered *weak*, as it not only holds on C, but also on all synthetic buggy versions of C. Thus, we check the number of mutants that each assertion kills, and those that are not falsified by any mutant are discarded as being *irrelevant*, due to their weakness.

2) Equivalent Assertions Detection: To identify equivalent assertions, SPECFUZZER uses a notion of *equivalence* based on mutation analysis. Two assertions are considered equivalent if they kill the same set of mutants. By analyzing the set

[3]https://www.antlr.org/

[4]http://plse.cs.washington.edu/daikon/

of mutants killed by each assertion, SPECFUZZER is able to partition the set of likely assertions according to the mutants they kill. From each partition, SPECFUZZER chooses a representative assertion, using the following heuristic: the assertions in each partition are ranked by the number of times they fail when running the test suite on the mutants (while they all kill the same mutants, some assertions may fail a greater number of times, i.e., for more tests in the test suite). The rationale is that assertions that fail the most represent stronger properties, and thus they may subsume other assertions in the partition. The assertions representing each partition are the ones conforming the set $\mathcal{R}_c$ of relevant assertions.

The assertion selector, implemented as a Python script, works from the output files produced by the invariant detector (Daikon) and the results of the mutation analysis (performed with Major). Its output is the final output of SPECFUZZER, which consists of a file containing the assertions in $\mathcal{R}_c$, classified according to the program points in which they hold.

IV. EVALUATION

Our previous evaluation of SPECFUZZER [9] is focused on a direct computation of the precision and recall metrics, using a dataset of 65 ground truth assertions corresponding to 43 Java methods. From this dataset, our evaluation analyzes three aspects: the effectiveness of the assertion fuzzer to generate relevant assertions, the ability of the assertion selector to discard redundant/irrelevant assertions, and the performance of SPECFUZZER in comparison with alternative techniques.

Regarding the first two aspects, the assertion fuzzer correctly generated 40 assertions, representing a 97.5% of the assertions when considering the subset of the ground truth that was supported by our tool, and a 61.5% when considering the entire ground truth. In the case of the assertion selector, it was able to reduce the number of reported assertions by $\sim$95%, and only 6 out of the 40 correctly fuzzed assertions were discarded.

Finally, we performed a comparison with the state-of-the-art tools GAssert [13] and EvoSpex [10], based on standard performance metrics: precision, recall and f1-score. While in terms of precision GAssert and EvoSpex perform better (mainly due to the incorporation of mechanisms to actively reduce the number of false positives), SPECFUZZER outperforms both tools in terms of recall, being able to discover 52.3% of the whole ground truth assertions (compared to the 27.6% and 38.4% that GAssert and EvoSpex discover, respectively). This recall improvement makes SPECFUZZER more effective overall, as indicated by a better f1-score over the other tools.

V. CONCLUSION

SPECFUZZER is a tool that automatically infers likely class specifications of Java classes. It takes as input a Java class and a test suite for it. Specifications are inferred in the form of assertions by using a combination of grammar-based fuzzing, dynamic invariant detection, and mutation analysis, and can express properties at different program points, including preconditions, postconditions and class invariants. Our previous evaluation shows that SPECFUZZER has better performance in comparison with the state-of-the-art techniques. Furthermore, the use of grammar-based fuzzing enables SPECFUZZER to be easily adapted to different assertion languages.

REFERENCES

[1] Pablo Abad, Nazareno Aguirre, Valeria S. Bengolea, Daniel Alfredo Ciolek, Marcelo F. Frias, Juan P. Galeotti, Tom Maibaum, Mariano M. Moscato, Nicolás Rosner, and Ignacio Vissani. Improving test generation under rich contracts by tight bounds and incremental SAT solving. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*, pages 21–30. IEEE Computer Society, 2013.

[2] Marcelo d'Amorim, Carlos Pacheco, Tao Xie, Darko Marinov, and Michael D. Ernst. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan*, pages 59–68. IEEE Computer Society, 2006.

[3] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[4] René Just, Franz Schweiggert, and Gregory M. Kapfhammer. MAJOR: an efficient and extensible tool for mutation analysis in a java compiler. In Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors, *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, pages 612–615. IEEE Computer Society, 2011.

[5] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Sci. Comput. Program.*, 55(1-3):185–208, 2005.

[6] Lisa (Ling) Liu, Bertrand Meyer, and Bernd Schoeller. Using contracts and boolean queries to improve the quality of automatic test generation. In Yuri Gurevich and Bertrand Meyer, editors, *Tests and Proofs - 1st International Conference, TAP 2007, Zurich, Switzerland, February 12-13, 2007. Revised Papers*, volume 4454 of *Lecture Notes in Computer Science*, pages 114–130. Springer, 2007.

[7] Francesco Logozzo and Thomas Ball. Modular and verified automatic program repair. In Gary T. Leavens and Matthew B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 133–146. ACM, 2012.

[8] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.

[9] Facundo Molina, Marcelo d'Amorim, and Nazareno Aguirre. Fuzzing class specifications. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1008–1020. ACM, 2022.

[10] Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo F. Frias. Evospex: An evolutionary algorithm for learning postconditions. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 1223–1235. IEEE, 2021.

[11] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 75–84. IEEE Computer Society, 2007.

[12] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. *IEEE Trans. Software Eng.*, 40(5):427–449, 2014.

[13] Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè. Evolutionary improvement of assertion oracles. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1178–1189, New York, NY, USA, 2020. Association for Computing Machinery.

[14] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. The fuzzing book. In *The Fuzzing Book*. Saarland University, 2019. Retrieved 2019-09-09 16:42:54+02:00.