EvoSpex: A Search-Based Tool for Postcondition Inference

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

Pablo Ponzio
pponzio@dc.exa.unrc.edu.ar
University of Rio Cuarto and CONICET
Rio Cuarto, Argentina

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

Marcelo F. Frias
mfrias@itba.edu.ar
University of Texas at El Paso
El Paso, Texas, United States

ABSTRACT

Postconditions are predicates that specify the intended behavior of a program by capturing properties about the program state when the program finishes its execution. Although postconditions can help to improve many software reliability analyses, they are seldom found accompanying source code. Thus, tools that assist developers in specifying postconditions are useful. This tool demo paper presents EvoSpex, a tool based on evolutionary computation that automatically infers postconditions of Java methods. Given a target Java method and a test suite for it, our tool executes the test suite to obtain valid pre/post state pairs for the method under analysis. Then, these pairs are mutated to obtain (allegedly) invalid ones, and finally a postcondition assertion characterizing the current method behavior is produced, by using an evolutionary algorithm that searches for an assertion that is satisfied by the valid pre/post state pairs and leaves out the invalid ones. EvoSpex implements a classic genetic algorithm that explores the space of candidate postconditions over a JML-like specification language. The algorithm is guided by a fitness function that aims at precisely capturing the valid state pairs, rejecting the invalid ones, and that also favors more succinct assertions.

CCS CONCEPTS

• **Software and its engineering** → **Specification languages**; **Software verification and validation**; **Software testing and debugging**.

KEYWORDS

Oracle problem, specification inference, evolutionary computation.

ACM Reference Format:
Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo F. Frias. 2023. EvoSpex: A Search-Based Tool for Postcondition Inference. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3597926.3604928

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3604928

1 INTRODUCTION

The use of assertions as program specifications dates back to the works of Hoare [5] and Floyd [3], in the context of program verification and associated with the concept of program correctness. Technically, an assertion is a statement predicating on program states, that can be used to capture *assumed properties*, as in the case of preconditions, or *intended properties*, as in the case of postconditions. A program P with a precondition *pre* and postcondition *post* is said to be (partially) correct with respect to this specification, if every execution of P starting in a state that satisfies *pre*, if it terminates, it does so in a state that satisfies *post* [5].

Providing specifications that capture the expected software behavior is important for many reasons. They can facilitate modular software development, allowing components to be developed independently and swapped in and out as long as their respective specifications are maintained [9]. Also, when specifications are provided with the source code and expressed in a formal language, as in the case of postcondition assertions, they can enable or improve a number of program analysis tasks, including runtime assertion checking, bug finding [11], and verification [2]. Since formal specifications are seldom found in practice, and writing them can suppose a considerable effort from developers, the *specification inference problem* (a special case of the oracle problem [1]) is receiving increasing attention by the software engineering community.

To alleviate the oracle problem, we propose EvoSpex, a search-based tool that, given a Java method and a test suite for it, automatically infers a specification of the current behavior of the method, in the form of a postcondition assertion. EvoSpex is based on a classic genetic algorithm that explores a search space of candidate assertions. The search is guided by two sets of pre/post state pairs: a set S_v of *valid* pairs and a set S_i of *invalid* pairs. The valid pairs are obtained from executions of the target method by the provided test suite, thus capturing the current method behavior. The invalid pairs, on the other hand, are generated by mutating valid ones, and thus representing (allegedly) incorrect method behaviors. From these pairs, EvoSpex focuses then on searching for a postcondition assertion ϕ that: (i) characterizes the valid pairs while leaving out the invalid ones, and (ii) is as succinct as possible.

This paper extends our previous paper [10], which introduced the technique, by providing instructions on how to use EvoSpex, details of its implementation, and some improvements with respect to the original prototype. The improvements include the possibility of using any test suite as input (making the tool independent from

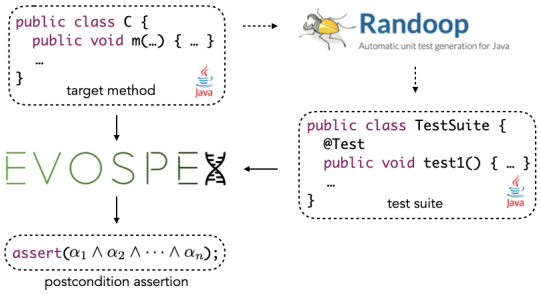

Figure 1: Usage of EvoSpex.

how this suite is obtained), and better support for maintainable/extensible assertion language definitions. The source code, evaluation subjects, and demo video, are available at:

https://github.com/facumolina/evospex

2 USAGE

The envisioned users of EvoSpex are researchers or Java practitioners who may be in need of obtaining postconditions for a given Java method, either with the aim of analyzing the method's behavior, or to improve some program analysis task that can be benefited from the postconditions. EvoSpex receives two inputs: a target method m of class C and a test suite $\mathcal{T}_c$ containing executions of m. As test suites may not be available, a typical usage of our tool may involve the execution of some automated test generation tool, such as Randoop[1], to generate the test suite $\mathcal{T}_c$ necessary to run EvoSpex, as is shown in Figure 1. From these tests, EvoSpex goes through a state generation phase, that creates the sets S_v and S_i of valid and invalid state pairs. These are in turn used to guide the postcondition inference phase, which essentially executes an evolutionary search process in order to infer a postcondition assertion capturing the method behavior.

To illustrate the use of EvoSpex, let us consider method `add(int, E)` of class `TreeList`, from Apache Commons Collections[2]. Figure 2 shows a fragment of the method. The class is providing an AVL-tree based implementation of lists, allowing one to perform operations such as insertion/deletion in $O(\log n)$. Particularly, method `add(int, E)` is an insertion routine, inserting a given object in a specific location. Notice how the precondition of the method is captured in the source code by the method `checkInterval(int, int, int)`, while the method's postcondition is not present.

To perform the state generation phase we need three inputs: the target classpath, the target test class, and the target method signature. We can generate the states for our example by running:

`./evospex.sh --genStates <cp> <test_suite> <method>`

where `<cp>` is the classpath for the target subject, `<test_suite>` is the test class (`TreeListTest` in our case), and `<method>` is the method signature (`TreeList.add(int,E)` in our case). The valid and invalid state pairs will be generated and placed within the `states` folder, setting the conditions to start the next phase.

EvoSpex's inference phase can be launched by running:

[1] https://randoop.github.io/randoop
[2] https://github.com/apache/commons-collections

```java
public class TreeList<E> extends AbstractList<E> {
  private AVLNode<E> root; private int size;

  // Adds a new element to the list.
  public void add(int index, E obj) {
    modCount++;
    checkInterval(index, 0, size()); // Checks whether the index is valid.
    if (root == null) {
      root = new AVLNode<>(index, obj, null, null);
    } else {
      root = root.insert(index, obj);
    }
    size++;
  }

  private static class AVLNode<E> {
    private E value;
    private int height, relativePosition;
    private boolean leftIsPrevious, rightIsNext;
    private AVLNode<E> left, right;

    // Inserts a node at the position index.
    public AVLNode<E> insert(int index, E obj) {
      int indexRelativeToMe = index - relativePosition;
      if (indexRelativeToMe <= 0) {
        return insertOnLeft(indexRelativeToMe, obj);
      }
      return insertOnRight(indexRelativeToMe, obj);
    }
    ...
  }
}
```

Figure 2: Target method `TreeList.add(int, E)`.

```
1- this.size = \old(this.size) + 1 && // size increased by 1
2- #(\old(this).root.*(left+right))=this.size-1 && // tree size is correct
3- index in this.root.*left.height && // index is a valid position
4- obj in this.root.*(left+right).value // obj inserted
```

Figure 3: Assertions inferred for `TreeList.add(int, E)`.

`./evospex.sh --infer <cp> <class> <method_states>`

where `<cp>` is the target classpath, `<class>` is the target class (e.g., `TreeList`), and `<method_states>` is the folder containing the previously computed states. The execution will report information of each generation of the evolutionary process (mutations performed, crossovers performed, best fitness value, etc). At the end, the found postcondition is reported as an assertion: `assert(`α_1 `&&` α_2 `&&` ... `&&` α_n`);`. The postcondition assertions inferred for our example are shown in Figure 3: 1 and 2 state that `size` was incremented and that is correct; 3 states that `index` is among the `height` values collected by traversing the tree through field `left`; and 4 states that the input `obj` has been indeed inserted in the tree.

3 EVOSPEX

Let us now discuss the implementation details of EvoSpex. Figure 4 shows an overview of our tool, implemented in Java. Given a target method m of class C and a test suite $\mathcal{T}_c$, EvoSpex will infer a postcondition ϕ_m, through the two main phases described below.

3.1 State Generation Phase

The state generation phase is in charge of generating the valid and invalid state pairs S_v and S_i. First, it instruments the given test suite $\mathcal{T}_c$ with instructions to serialize the pre/post states of executions of

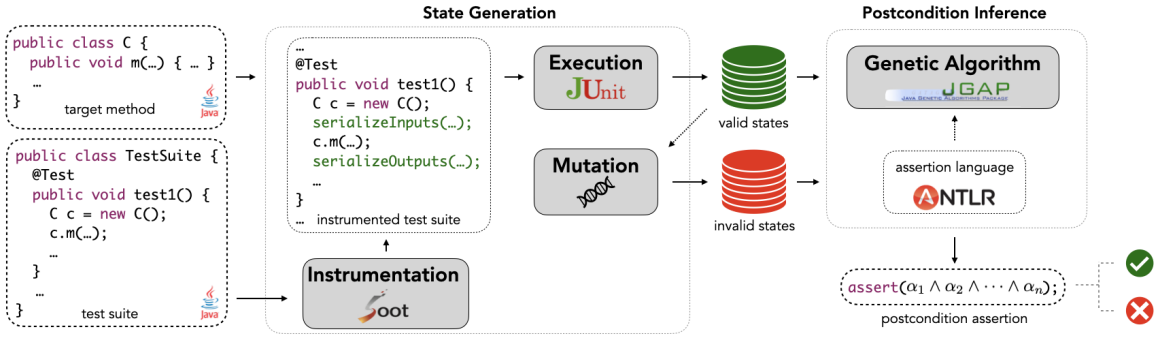

Figure 4: An overview of EvoSpex.

m. Then, it executes the instrumented test suite, and collects the serialized states pairs, obtaining the set S_v. Finally, it performs a state mutation process to generate the set S_i of invalid states pairs.

3.1.1 Instrumentation. This instrumentation sets the conditions to obtain the set of *valid* state pairs S_v, composed of state pairs of the form $\langle s_{pre}, s_{post}\rangle$, capturing the current behavior of a target method m. That is, if we execute m from the state s_{pre}, and the execution leads to the state s_{post}, then we record the valid pair $\langle s_{pre}, s_{post}\rangle$. Our instrumentation is performed using the Soot library[3] as follows. Let $C, C_1, \ldots, C_n$ be classes, and m a method in C with parameters of types $C_1, \ldots, C_n$. Each initial state s_{pre} for executing m will be a tuple $\langle o_C, o_{C_1}, \ldots, o_{C_n}\rangle$ of objects of types $C, C_1, \ldots, C_n$, respectively. To form these tuples, for every invocation of m within $\mathcal{T}_c$, we insert one instruction that serializes the object and arguments from which m is invoked, obtaining s_{pre}. Similarly, to generate s_{post}, we insert one instruction that serializes the state resulting from m's execution. These serializations are performed using XStream[4].

3.1.2 Execution. The instrumented $\mathcal{T}_c$ is executed in order to actually perform the serialization of the pre/post states, obtaining the set S_v. This step uses the `JUnitCore` runner, from JUnit[5].

3.1.3 Mutation. This step is performed to produce the set of *invalid* pre/post state pairs S_i. For each valid pair $\langle s_{pre}, s_{post}\rangle$, we create a new pair $\langle s_{pre}, s'_{post}\rangle$, where s'_{post} is a mutation of s_{post} in which the value of a randomly selected field in the receiving object or return value (the constituents of s_{post}), is replaced by a randomly generated value of the corresponding type. Before including it into S_i, we check that the resulting pair $\langle s_{pre}, s'_{post}\rangle$, is not in S_v.

3.2 Postcondition Inference Phase

The inference phase is the execution of EvoSpex's genetic algorithm, which searches for a postcondition for m. The goal is to produce a postcondition assertion ϕ such that every valid state pair in S_v satisfies ϕ, and every invalid state pair in S_i violates ϕ.

3.2.1 Assertions Search Space. Each candidate postcondition ϕ in the search space is of the form $\phi = \alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_n$. Each assertion α_i belongs to an assertion language that is, from an expressiveness point of view, a JML-like [2] language. The language, designed following the Alloy notation [6], supports quantifiers, navigation and reachability expressions. Most operators have an intuitive reading (equality/inequality, boolean connectives, quantifiers); the dot operator (`.`) is relational composition and captures navigation; relational union and intersection are denoted by `+` and `&`, respectively, and can be used to combine fields in navigations; set/relational cardinality is denoted by `#`; finally, `*` and `^` are reflexive-transitive and transitive closures, respectively (used to express reachability). Also, assertions might use the `\old(e)` notation, to refer to the value of expression `e` at the precondition. In our tool, the language definition is implemented using the ANTLR parser generator[6].

3.2.2 Genetic Algorithm. EvoSpex's classic genetic algorithm is implemented using JGAP[7]. The algorithm first creates a population $\mathcal{P}$ of initial candidate postconditions, which is then evolved using genetic operators until a postcondition is found or a termination criterion is met (e.g., a timeout). At each generation, candidates in $\mathcal{P}$ are selected according to a fitness function that measures how good they are at distinguishing between the valid/invalid state pairs.

Initial population. We initialize $\mathcal{P}$ with two types of expressions: *single value expressions* and *set expressions*. Single value expressions are built from the AST of the target method class. For example, considering class `TreeList` in Figure 2, we can build expressions such as `this.root`, `this.root.left`, `this.size`, and so on. To build assertions, each expression `expr` is evaluated on a randomly selected subset of valid (resp. invalid) state pairs, as follows: if the result of evaluating `expr` in a valid (resp. invalid) state pair returns a value `v`, then we create the assertion `expr = v` (resp. `expr != v`). Also, assertions comparing random expressions of the same type (e.g., `this.root = this.root.right`) are included. Set expressions are built from recursive fields (e.g., `left` or `right`) in the AST or from fields whose type implements the `java.util.Collection` interface. From the `TreeList` class, expressions such as `this.root.*left` (set of nodes reachable from `this.root` via `left` traversals), and `this.root.*(left+right)`, can be created. Each set expression `expr` is used to build quantified assertions (`all n: expr : n = n.right`) and comparisons between integer expressions and set cardinalities (`this.root.height = #(expr)`). Finally, we create initial postconditions comparing

[3] https://soot-oss.github.io/soot/
[4] https://x-stream.github.io/
[5] https://junit.org/junit4/
[6] https://www.antlr.org/
[7] Java Genetic Algorithms Package - http://jgap.sourceforge.net/

(using random operators) result/argument expressions against random expressions of the same type (e.g., `index < this.size`).

Genetic Operators. The genetic operators allow us to explore the search space by producing candidate postconditions with new characteristics as well as combinations of existing ones. Our algorithm implements three well known operators. The *mutation* operator, applied to individual assertions of a candidate, can perform a variety of modifications: assertion deletion, assertion negation, numeric addition/subtraction (applied to numeric comparisons), expression replacement (replaces some part of an assertion with a randomly selected expression of the same type), expression extension (extends an expression with a new field, for example replacing `this.root` by `this.root.left`), and operator replacement. The *crossover* operator takes two candidates and creates a new one out of them. Given two candidate postconditions ϕ_1 and ϕ_2, our implementation simply produces the new candidate $\phi_1 \wedge \phi_2$. Finally, the *selection* operator selects postconditions to be included in the next generation. In our case, it selects a fraction of candidates with the highest fitness values, a fraction of the best unary (with only one assertion) non-valid postconditions (failing on some valid state), and a fraction of the unary valid ones (only failing on invalid states).

Fitness Function. Our fitness function f assesses how good a candidate postcondition ϕ is at distinguishing between the sets S_v and S_i of state pairs. To do so, we first compute the sets P_ϕ and N_ϕ of positive and negative counterexamples:

$$P_\phi = \{v \in S_v | \neg\phi(v)\} \quad N_\phi = \{i \in S_i | \phi(v)\}$$

Basically, the sets P_ϕ and N_ϕ contain those states for which ϕ does not behave correctly. Then, $f(\phi)$ is computed as follows:

$$\#P_\phi > 0 \rightarrow f(\phi) = (\text{MAX} - \#P_\phi - \#S_i) + \left(\frac{1}{l_\phi + comp_\phi}\right) + \frac{mca_\phi}{l_\phi}$$
$$\#P_\phi = 0 \rightarrow f(\phi) = (\text{MAX} - \#N_\phi) + \left(\frac{1}{l_\phi + comp_\phi}\right) + \frac{mca_\phi}{l_\phi}$$

This definition has three terms. The first term reflects the most important goal: to minimize the number of counterexamples. To prioritize the (more reliable) positive counterexamples information, when ϕ is falsified by a valid state, the whole set S_i of invalid state pairs are considered counterexamples too. The second term acts as a penalty regarding two aspects: the candidate length l_ϕ (number of conjuncts) and its "complexity" $comp_\phi$ (sum of each conjunct complexity, where, intuitively, the complexity of an equality between two integer fields is lower than the complexity of an equality between an integer field with a set cardinality, and both of these are lower than the complexity of a quantified formula, and so on). The last term acts as a reward favoring candidates with a greater number of "method component assertions" mca_ϕ, i.e., with a higher number of conjuncts stating properties regarding the parameters, the result, or a relation between initial and final object states.

4 EVALUATION

In our previous evaluation of EvoSpex [10], we focused on two aspects. First, we assessed whether the inferred postconditions presented oracle deficiencies, using the OASIs [7] tool. This analysis evaluated the quality of postconditions, in terms of its associated number of false positives/negatives, for 200 methods from 16 open source Java projects of the SF110[8] benchmark. Our results showed that EvoSpex is able to produce more accurate postconditions, with a 6.70% of false positives, compared to the 17.49% of false positives of related techniques. Second, we studied the ability of EvoSpex to infer manually written contracts, by analyzing methods equipped with manually written rich postconditions used for verification [4], and methods automatically synthesized from specifications [8]. In this last experiment, EvoSpex was able to reproduce 50% of the manually written contracts, and for 74% of the analyzed methods, EvoSpex reproduced at least one complex postcondition property. A comparison with related techniques and discussions regarding related work can be found in our previous paper [10].

[8] https://www.evosuite.org/experimental-data/sf110/

5 CONCLUSION

EvoSpex is a search-based tool for inferring postcondition assertions of Java methods. To infer a postcondition, it takes as inputs the target method, and a test suite containing executions of the method. Postconditions inferred by EvoSpex are *regression oracles*, involving assertions capturing rich constraints that concern method parameters, return values, internal object states, and the relationship between pre and post method execution states. Automatically inferring these specifications from source code is a relevant topic, as it enables a number of applications, including software evolution and maintenance, bug finding, and specification improvement.

REFERENCES

[1] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Trans. Software Eng.* 41, 5 (2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[2] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. 2005. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*. 342–363. https://doi.org/10.1007/11804192_16

[3] Robert W. Floyd. 1967. Assigning meanings to programs. In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19*, J. T. Schwartz (Ed.). American Mathematical Society, Providence, 19–32.

[4] Carlo A. Furia, Martin Nordio, Nadia Polikarpova, and Julian Tschannen. 2017. AutoProof: auto-active functional verification of object-oriented programs. *Int. J. Softw. Tools Technol. Transf.* 19, 6 (2017), 697–716. https://doi.org/10.1007/s10009-016-0419-0

[5] Charles A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

[6] Daniel Jackson. 2006. *Software Abstractions - Logic, Language, and Analysis*. MIT Press. http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=10928

[7] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. 2016. Test oracle assessment and improvement. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 247–258. https://doi.org/10.1145/2931037.2931062

[8] Calvin Loncaric, Michael D. Ernst, and Emina Torlak. 2018. Generalized Data Structure Synthesis. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 958–968. https://doi.org/10.1145/3180155.3180211

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

[10] Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo F. Frias. 2021. EvoSpex: An Evolutionary Algorithm for Learning Postconditions. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1223–1235. https://doi.org/10.1109/ICSE43902.2021.00112

[11] Todd W. Schiller and Michael D. Ernst. 2012. Reducing the barriers to writing verified specifications. In *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*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 95–112. https://doi.org/10.1145/2384616.2384624

Received 2023-05-18; accepted 2023-06-08