EvoSpex: An Evolutionary Algorithm for Learning Postconditions (artifact)

Facundo Molina[*][†], Pablo Ponzio[*][†], Nazareno Aguirre[*][†], Marcelo Frias[†][‡]
[*]Department of Computer Science, FCEFQyN, University of Río Cuarto, Argentina
[†]National Council for Scientific and Technical Research (CONICET), Argentina
[‡]Department of Software Engineering, Buenos Aires Institute of Technology, Argentina

***Abstract*—Having the expected behavior of software specified in a formal language can greatly improve the automation of software verification activities, since these need to contrast the intended behavior with the actual software implementation. Unfortunately, software many times lacks such specifications, and thus providing tools and techniques that can assist developers in the construction of software specifications are relevant in software engineering. As an aid in this context, we present EvoSpex, a tool that given a Java method, automatically produces a specification of the method's current behavior, in the form of postcondition assertions. EvoSpex is based on generating software runs from the implementation (valid runs), making modifications to the runs to build divergent behaviors (invalid runs), and executing a genetic algorithm that tries to evolve a specification to satisfy the valid runs, and leave out the invalid ones. Our tool supports a rich JML-like assertion language, that can capture complex specifications, including sophisticated object structural properties.**

Introduction

Software verification seeks to ensure that the software implementation meets its corresponding expected behavior. At the level of source code, the expected behavior can be specified in a number of ways, e.g., informally as comments that describe what the software is supposed to do, or formally, as statements that assert properties that the software must satisfy at certain program points. While formal program assertions would be preferred, since these can be exploited more directly for some automated verification activities, they are seldom found accompanying source code. Formal program specifications in the form of contract assertions, such as preconditions, postconditions and invariants, can be used for a variety of powerful verification tasks, such as runtime assertion checking, and property-based test generation. Then, the lack in general of these assertions calls for tools and techniques that can aid developers in constructing specifications.

```java
import java.util.AbstractList;

public final class AvlTreeList<E> extends AbstractList<E> {

  private Node<E> root;

  public void add(int index, E val) {
    if (index < 0 || index > size())
      throw new IndexOutOfBoundsException();
    if (size() == Integer.MAX_VALUE)
      throw new IllegalStateException("Max size reached");
    root = root.insertAt(index, val);
  }

  private static final class Node<E> {

    private E value;
    private int height;
    private int size;
    private Node<E> left;
    private Node<E> right;

    public Node<E> insertAt(int index, E obj) {
      assert 0 <= index && index <= size;
      if (this == EMPTY_LEAF)
        return new Node<>(obj);
      int leftSize = left.size;
      if (index <= leftSize)
        left = left.insertAt(index, obj);
      else
        right = right.insertAt(index-leftSize-1, obj);
      recalculate();
      return balance();
    }
  }
}
```

Fig. 1. Add method of class AvlTreeList

Motivated by the above observations, we developed EvoSpex, a tool that takes a Java method, and automatically produces a specification of the method's current behavior, in the form of postcondition assertions. While the assertions capture the actual software behavior rather than the intended one, the produced specifications have various applications. For instance, they can be examined as a summary of the method's behavior, and if correct, they may be used for regression analysis on subsequent, improved modifications of the method. EvoSpex first generates software runs from the implementation, which are considered *valid* runs, as well as divergent behaviors, i.e., *invalid* runs that do not correspond to the method's execution. It then executes a genetic algorithm that tries to evolve an assertion to satisfy the valid runs, and leave out the invalid ones. Our tool supports a rich JML-like assertion language, that can capture complex specifications involving object navigations, transitive closure and standard arithmetic and relational operators. It can capture expressive postconditions, including sophisticated object structural properties.

How to use EvoSpex

To describe how EvoSpex is used, let us consider as illustrating example a Java implementation of lists over balanced trees, `AvlTreeList`, and more specifically, an insertion routine (`add`) in this implementation. The implementation is shown in

```
// root
this.root != null &&
this.root.left != null &&
// height
all n : this.root.*(left+right) : (
  n.left != null => n.height > n.left.height &&
  n.right != null => n.height > n.right.height
) &&
// size
old_this.root.size < this.root.size &&
this.root.size == #(this.root.*(left+right - null)) - 1 &&
all n : this.root.*(left+right) : (
  n.left != null => n.size > n.left.size &&
  n.right != null => n.size > n.right.size
) &&
// arguments
index != this.root.size &&
val in this.root.*(left+right).value &&
// structural
all n : this.root.*(left+right) : n !in n.^(left + right)
```

Fig. 2. Postcondition generated by our tool for AvlTreeList.add(int, E)

Figure 1. As it can be seen, both the data representation and the method are relatively complex. When applied to method `add`, EvoSpex produces a set of postcondition assertions, that declaratively, and formally, capture the behavior of the method. In particular for this case, EvoSpex produces the specification shown in Figure 2. The assertion language is similar to JML [1], from an expressiveness point of view, as it features transitive and reflexive-transitive closures. Notice how the generated postconditions can refer to the receiver's attributes (direct and indirect) and method parameters, both at the "exit" point of the method and the "entry" point (the latter prepended with `old_`). Notice how the postcondition captures the fact that the `size` is increased, the relationship between `size` and the number of nodes in the tree (using reflexive-transitive closure `*`), that the inserted element belongs to the tree after the method is executed, and that the tree structure is indeed acyclic. We refer the reader to our technical paper [3] for further details regarding the assertion language.

The EvoSpex tool and the package for reproducing the experiments in [3] are publicly archived[1], and accessible from a repository[2]. The tool is entirely developed in Java, and we provide scripts for facilitating the execution of each step of our technique, on a variety of case studies. The tool can be installed and run natively, or via a provided Docker container, with the experimental data and the configured tool.

Running EvoSpex

EvoSpex receives as input a Java method. The first step in running the tool is the generation of sets of *valid* and *invalid* method executions, which our tool summarizes as corresponding sets of pre/post state pairs. As described in our research paper [3] the valid pre/post state pairs are generated by a bounded-exhaustive test generation approach (a bound is provided for the maximum number of allocated objects by each test during the generation); the invalid pre/post state pairs are produced by a mutation mechanism, that alters the valid pairs. For our `AvlTreeList` example, the valid/invalid state pairs can be generated, from the `$EVOSPEXOG/generate-objects-datastr` folder, as follows:

```
$ ./generate_objects.sh casestudies.motivation.AvlTreeList 4
```

Notice how the call to the script receives a bound for bounded-exhaustive generation (4 in this case), and is done for the whole class containing the method of interest, `add` in our case. That is, the above command will produce the valid and invalid pre/post state pairs required to enable the execution of the evolutionary stage of EvoSpex for every `AvlTreeList` method.

With the valid and invalid pre/post state pairs ready, the EvoSpex genetic algorithm can be executed, to generate the postcondition assertions. For our example, this can be done from folder `$EVOSPEX`, with the following command:

```
$ ./evospex.sh casestudies.motivation.AvlTreeList folder_name
```

The script receives a `folder_name`, the folder containing the valid/invalid state pairs produced in the previous step (details are in the repository). The output of the algorithm is shown in standard output, and can be easily redirected to a file.

Reproducing the Experiments

All the experiments in [3] can be reproduced following the instructions available with our artifact. For convenience, valid and invalid pre/post state pairs have already been computed, and are available with the experimental data.

Our evaluation involved measuring the precision of the generated postconditions, using the OASIs tool [2]. This precision measuring step is rather manual. The obtained postconditions have to be manually inserted as assertions at the end of the corresponding methods (we provide a runtime assertion checker for our language, that checks for the satisfaction of our produced assertions in a program point), to run OASIs. Also, OASIs typically involves a human-in-the-loop process, where first the false positives have to be identified and removed through multiple iterations, and then the false negatives, as we indicate in our artifact's instructions.

[1]http://doi.org/10.5281/zenodo.4458256
[2]https://github.com/facumolina/evospex-ae

References

[1] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and esc/java2. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 342–363. Springer, 2005.

[2] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. Test oracle assessment and improvement. In Andreas Zeller and Abhik Roychoudhury, editors, *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 247–258. ACM, 2016.

[3] Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo Frias. EvoSpex: An evolutionary algorithm for learning postconditions. In *Proceedings of the 43rd ACM/IEEE International Conference on Software Engineering ICSE 2021, Virtual (originally Madrid, Spain), 23-29 May 2021*, 2021.