Checking Java Assertions Using Automated Test-Case Generation

Rafael Caballero[1(✉)], Manuel Montenegro[1], Herbert Kuchen[2], and Vincent von Hof[2]

[1] University Complutense de Madrid, Madrid, Spain
rafacr@ucm.es
[2] Institute of Information Systems, University of Münster, Münster, Germany

Abstract. We present a technique for checking the validity of Java assertions using an arbitrary automated test-case generator. Our framework transforms the program by introducing code that detects whether the assertion conditions are met by every direct and indirect method call within a certain depth level. Then, any automated test-case generator can be used to look for input examples that falsify the conditions. We show by means of experimental results the effectiveness of our proposal.

Keywords: Assertions · Conditions · Test-cases · Java · Test-case generation

1 Introduction

Using assertions is a common programming practice, and especially in the case of what is known as 'programming by contract' [5], where they can be used e.g. to formulate pre- and postconditions of methods as well as invariants of loops. Assertions in Java [6] are used for finding errors in an implementation at runtime during the test phase of the development cycle. If the condition in an `assert` statement is evaluated to false during program execution, an *AssertionException* is thrown.

The goal of our work is to use automated test-case generators for detecting assertion violations. Observe that, in contrast to model checking, automated test-case generators are not complete and thus our proposal may miss possible assertion violations, but as our experiments show it works quite well in practice and is helpful as a first approach during program development before using model checking. The overhead of an automated test-case generator is smaller than for full model checking, since data and/or control coverage criteria known from testing are used as a heuristic to reduce the search space. However, finding an input for a method $m()$ that falsifies some assertion in the body of $m()$ is not enough. For instance, in the case of preconditions it is important to observe whether the methods calling $m()$ ensure that the call arguments satisfy the precondition.Thus, we extend the proposal to indirect calls of these methods (up to a fixed level of indirection), allowing checking the assertions in the context of the whole program.

M. Falaschi (Ed.): LOPSTR 2015, LNCS 9527, pp. 221–226, 2015.
DOI: 10.1007/978-3-319-27436-2_13

```
1  public class Sqrt {
2    static double eps = 0.00001;
3
4    public double sqrt(double r){
5      double a, a1 = r + eps;
6      do { a = a1;
7           a1 = a+r/a/2.0; //erroneous!
8           assert a==1.0 ||
9                  (a1>1.0 ? a1 < a
10                         : a1 > a);}
11     while (Math.abs(a - a1) >= eps);
12     return a1;
13   }}
14 public class Circle {
15   double area;
16
17   Circle(double area) {this.area = area; }
18
19   public double getRadius() {
20     assert area>=0;
21     return Sqrt.sqrt(area/Math.PI); }}
```

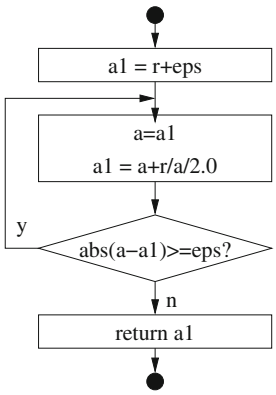

Fig. 1. Java method `sqrt`, corresponding control-flow graph, and class `Circle`.

In order to fulfill these goals we propose a technique based on a source-to-source transformation that converts the assertions into `if` statements and changes the return type of methods to represent the path of calls leading to an assertion violation as well as the normal results of the original program. Converting the assertions into a program control-flow statement is very useful for white-box, path-oriented test-case generators, which determine the program paths leading to some selected statement and then generate input data to traverse such a path (see [2] for a recent survey on the different types of test-case generators). Thus, our transformation allows this kind of generators to include the assertion conditions into the sets of paths to be covered.

2 Assertions and Automated Test-Case Generation

Java assertions ensure at runtime (if executed with the right option) that the program state fulfills certain restrictions. Figure 1 shows our running example. The radius of a circle is computed based on an erroneous implementation of the `sqrt` method (`a1 = a+r/a/2.0;` should be `a1 = (a+r/a)/2.0;`).

Our idea is to use a test-case generator to detect possible violations of the occurring assertions. A test-case generator is typically based on some heuristic which reduces its search space dramatically. Often it tries to achieve a high coverage of the control and/or data flow.

EvoSuite [4] generates test cases also for code with `assert` conditions. However, its search-based approach does not always generate test cases exposing assertion violations. In particular, it has difficulties with indirect calls such as the assertion in `Sqrt.sqrt` after a call from `Circle.getRadius`. A reason is that EvoSuite does not model the call stack. Thus, the test cases generated by EvoSuite for `Circle.getRadius` only expose one of the two possible violations, namely the one related to a negative area.

```
public abstract class MayBe<T> {

  public static class Value<T> extends MayBe<T> { // ...
  public static class CondError<T> extends MayBe<T> { // ...

  // did the method return a normal value (no violation)?
  abstract public boolean isValue();

  // value returned by the method.
  abstract public T getValue();

  // No condition violation detected. Return the same value
  // as before the instrumentation.
  public static <K> MayBe<K> createValue(K value) {
    return new Value<K>(value);  }

  // an assert condition is not verified
  public static <T> MayBe<T> generateError(String method,
                                           int position) {
    return new CondError<T>(new Call(method, position));}

  // method calls another method whose precondition or
  // postcondition is not satisfied.
  public static <T,S> MayBe<T> propagateError(String method,
                                  int position, MayBe<S> error){
    return new CondError<T>(new Call(method, position),
                            (CondError<S>) error);}
}
```

Fig. 2. Class `MayBe<T>`: new result type for instrumented methods.

There are other test-data generators such as JPet [1] that do not consider `assert` statements and thus cannot generate test cases for them. In the sequel, we present a program transformation that allows both EvoSuite and JPet to detect both possible assertion violations.

3 Program Transformation

The idea of the program transformation is to instrument the code in order to obtain special output values that represent possible violations of assertion conditions. Then, an automatic test-case generator is employed to obtain the inputs that produce these special values. In our case the instrumented methods employ the class `MayBe<T>` of Fig. 2. The overall idea is that a method returning a value of type `T` in the original code returns a value of type `MayBe<T>` in the instrumented code. `MayBe<T>` is in fact an abstract class with two subclasses, `Value<T>` and `CondError<T>`. `Value<T>` represents a value with the same type as in the original code, and it is used via the method `MayBe.createValue` whenever no assertion violation has been found. If an assertion condition is not satisfied, a `CondError` value is returned. There are two possibilities:

– The assertion is in the same method. Suppose it is the i-th assertion in the body of the method following the textual order. In this case, the method returns `MayBe.generateError(name,i);` with `name` the method name. The purpose of the method `generateError` is to create a new `CondError` object.

```
public class Sqrt {

  static double eps = 0.000001;

  public static MayBe<Double> sqrtCopy(double r){
    double a, a1 = 1.0;
    a = a1;
    a1 = a+r/a/2.0;
    double aux = Math.abs(a-a1);
    while (aux >= eps);{
      a = a1;
      a1 = a+r/a/2.0;
      if (!(a==1.0 || (a1>1.0 ? a1<a : a1>a)))
         return MayBe.generateError("sqrt", 2);
      aux = Math.abs(a-a1); }
    return MayBe.createValue(a1);
  } }

public class Circle {
  double area;
  Circle(double area) {this.area = area;}

  public MayBe<Double> getRadius() {
    if (!(area>=0))
          return MayBe.generateError("getRadius", 1);
    MayBe<Double> r =  Sqrt.sqrtCopy(area/Math.PI);
    if (!r.isValue())
        return MayBe.propagateError("getRadius", 2, r);
    return r;
  } }
```

Fig. 3. Transformed running example.

Observe that the constructor of `CondError` receives as parameter a `Call` object. This object represents the point where a condition is not verified, and it is defined by the parameters already mentioned: the name of the method, and the position i.

– The method detects that an assertion violation has occurred indirectly through the i-th method call in its body. Then, the method needs to extend the path and propagate the error. This is done using a call `propagateError(name,i,error)`, where `error` is the value to propagate. The corresponding constructor of class `CondError` adds the new call to the path.

Figure 3 shows the transformed running example. The methods not related (in)directly to assertions, e.g. the constructor of `Circle`, remain unchanged. Due to the lack of space, we omit the treatment of inheritance here. It can be found in [3].

4 Experiments

We have evaluated a few examples with different test-case generators with and without our program transformation. We have also developed a prototype that performs this transformation automatically. It can be found at https://github.com/wwu-ucm/assert-transformer, whereas the aforementioned examples can be found at https://github.com/wwu-ucm/examples.

Table 1. Detecting assertion violations.

		EvoSuite		JPet	
Method	Total	P	P^T	P	P^T
Circle.getRadius	2	1	2	0	2
BloodDonor.canGiveBlood	2	0	2	0	2
TestTree.insertAndFind	2	0	2	0	2
Kruskal	1	1	1	0	1
Numeric.foo	2	1	2	0	2
TestLibrary.test*	5	0	5	0	5
MergeSort.TestMergeSort	2	0	1	0	1
java.util.logging.*	5	0	2	-	-

Table 2. Control and data-flow coverage in percent.

	Binary tree		Blood donor		Kruskal		Library		MergeSort		Numeric		StdDev		Circle	
	P	P^T	P	P^T	P	P^T	P	P^T	P	P^T	P	P^T	P	P^T	P	P^T
EvoSuite	90	95	83	91	95	100	63	92	82	82	76	82	71	71	80	100
JPet	–	89	–	99	–	49	–	20	–	87	–	82	–	74	–	100

We have used two test-case generators, JPet and EvoSuite, for exposing possible assertion violations. As can be seen in Table 1, almost all possible assertion violations could be detected. Moreover, our program transformation typically improves the detection rate, since it makes the control flow more explicit than the usual assertion-violation exceptions. Column *Total* displays the number of possible assertion violations. Column P shows the number of detected assertion violations using the test-case generator and the original program, while column P^T displays the number of detected assertion violations after applying the transformation. Notice that JPet cannot find any assertion violation without our transformation, since it does not support assertions. For large examples such as the JDK logging package (6500 LOC), the configuration of JPet is tedious. As a consequence, this example has not been processed by this tool.

Our program transformation typically requires only a few seconds and even for larger programs such as the JDK 6 logging package the transformation finishes in 18.2 s. The runtime of our analysis depends on the employed test-case generator and the considered example. It can range from a few seconds to several minutes (Table 2).

5 Conclusions

We have presented an approach to use test-case generators for exposing possible assertion violations in Java programs. Our approach is a compromise between the usual detection of assertion violations at runtime and the use of a full model

checker. Since test-case generators are guided by heuristics such as control- and data-flow coverage, they have to consider a much smaller search space than a model checker and can hence deliver results much more quickly.

Additionally, we have developed a program transformation which replaces assertions by computations which explicitly propagate violation information through an ordinary computation involving nested method calls. In case of a violation, our transformation makes the control flow more explicit than the usual assertion-violation exceptions. This helps the test-case generators to reach a higher coverage of the code and enables more assertion violations to be exposed and detected. Additionally, the transformation allows to use test-case generators such as JPet which do not support assertions.

We have presented some experimental results demonstrating that our approach helps indeed to expose assertion violations and that our program transformation improves the detection rate.

Acknowledgements. This work has been supported by the German Academic Exchange Service (DAAD, 2014 Competitive call Ref. 57049954), the Spanish MINECO project CAVI-ART (TIN2013-44742-C4-3-R), Madrid regional project N-GREENS Software-CM (S2013/ICE-2731) and UCM grant GR3/14-910502.

References

1. Albert, E., Cabanas, I., Flores-Montoya, A., Gómez-Zamalloa, M., Gutierrez, S.: jPET: An automatic test-case generator for Java. In: 18th Working Conference on Reverse Engineering, WCRE 2011, Limerick, Ireland, October 17–20, 2011, pp. 441–442. (2011)
2. Anand, S., Burke, E., Chen, T.Y., Clark, J., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., McMinn, P.: An orchestrated survey on automated software test case generation. J. Syst. Softw. **86**(8), 1978–2001 (2013)
3. Caballero, R., Montenegro, M., Kuchen, H., von Hof, V.: A program transformation for converting java assertions into control-flow statements. Technical report 24, ERCIS (2015)
4. Galeotti, J.P., Fraser, G., Arcuri, A.: Improving search-based test suite generation with dynamic symbolic execution. In: IEEE International Symposium on Software Reliability Engineering (ISSRE), pp. 360–369. IEEE (2013)
5. Meyer, B.: Object-oriented Software Construction, 2nd edn. Prentice-Hall Inc., Upper Saddle River (1997)
6. Oracle. Programming with Assertions (2014). https://docs.oracle.com/javase/jp/8/technotes/guides/language/assert.html