Checking Java Assertions Using Automated Test-Case Generation

Rafael Caballero[1], Manuel Montenegro[1], Herbert Kuchen[2], Vincent von Hof[2]

[1] University Complutense de Madrid
[2] Institute of Information Systems, University of Münster

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. The program transformation ensures that the value obtained for these inputs represents a path of method calls that ends with a violation of some assertion. We show experiments with two different automatic test-case generators that demonstrate not just the applicability of our proposal but also that we can get a better coverage than the same test-case generators without our transformation.

Keywords: assertions, conditions,test-cases, Java, test-case generation.

1 Introduction

Using assertions is nowadays a common programming practice, and especially in the case of what is known as 'programming by contract' [13, 14], where they can be used e.g. to formulate pre- and postconditions of methods as well as invariants of loops. Assertions are generally used for finding errors in an implementation. Most of the proposals in this sense can be classified into two groups:

1. Check the validity of the assertions at run-time, as it is usually done for Java *assert* statements [15]. If the condition in an assert statement is evaluated to false during program execution an *AssertionException* is thrown.
2. Checking formally whether the assertion is fulfilled. Typically, a model checker such as Java Pathfinder [17] will be used for such a check.

The first approach is simple, and already available as part of the Java language but cannot find errors in advance, since it depends on the particular executions of the program. The second approach using a model checker can be very costly and even intractable for large software systems.

The proposal presented in this paper can be considered in the middle of the two groups. It relies on Java `assert` statements, looking for counterexamples

invalidating the conditions but, instead of doing this at run time, it aims at finding the counterexamples statically, using an automatic test-case generator. Thus, our goal is to look automatically for test-cases proving that some assertion condition is not verified. In contrast to model checking, this approach is not complete and it may miss possible assertion violations, but as our experiments will show it works quite well in practice and is helpful in situations where model checking cannot be applied. It is also possible to apply our approach during program development and use model checking as a final check once the software is finished. The overhead of our approach 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. A generator will stop if the considered code is sufficiently covered by the produced test cases. 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[3] to these methods (up to a fixed level of indirection), allowing checking the assertions in the context of the whole program.

In order to fulfil 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 that 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.

The idea of including the assertions as part of the code and use automated test-case generation to obtain inputs that falsify the conditions was already presented in [11] and has given rise to the so called *assertion-based software testing* technique. In particular this work can be included in what has been called *testability transformation* [8], a type of source-to-source transformation that aims to improve the ability of a given test generation method to generate test cases for the original program. An important difference of our proposal with respect to other works such as [3] is that instead of developing a specific test-case generator we propose a simple transformation that allows general purpose test-case generators to look for input data invalidating assertions.

The next section presents a running example and introduces some basic concepts. Section 3 presents the program transformation. Section 4 shows by means of experiments how two existing white-box, path-oriented test-case generators benefit from this transformation. Finally, Section 5 presents our conclusions.

[3] If in the body of method m_1 there is a call to m_2, then we we say that m_1 *calls* m_2 *directly*. If m_2 calls m_3 directly and m_1 calls m_2 directly or indirectly, then we say that m_1 *calls* m_3 *indirectly*.

2 Conditions, Assertions, and Automated Test-Case Generation

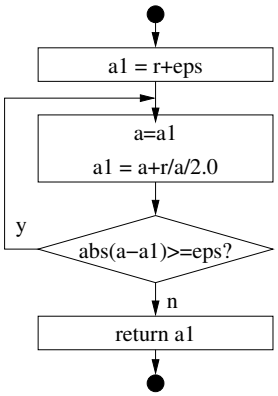
Java assertions allow to ensure at runtime (when executed with the right option) that the program state during a computation fulfills certain restrictions at a given program location. They can be used to formulate e.g. preconditions and postconditions of methods and invariants of loops. Fig. 1 presents two Java classes:
- `Sqrt` includes a method `sqrt` that computes the square root based on Newton's algorithm. The method uses an assertion which ensures that the computation makes progress. However the method contains an error: the statement `a1 = a+r/a/2.0;` should be `a1 = (a+r/a)/2.0;`. This error provokes a violation of the assertion for any input value different from 0.0.
- `Circle` represents a circle which has the area as its only attribute. Method `getRadius` obtains the radius employing method `Sqrt.sqrt` to compute the square root. The method includes an assertion checking whether the area is a non-negative number.

Thus, `Circle.getRadius` will raise an assertion exception if the area is negative, but also if the area is positive due to an error in `Sqrt.sqrt`, which causes a violation of the assertion in this method.

Our idea is to use a test-case generator to detect possible violations of these 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. In the `sqrt` example in Fig. 1, the tool would try to find test cases covering all edges in the control-flow graph and all so-called def-use chains, i.e. pairs of program locations, where a value is defined and where this value is used. E.g. in method `sqrt` the def-use chains for variable `a1` are (ignoring the assertion) the following pairs of line numbers (5,6), (7,11), (7,6), and (7,12).

There are mainly two approaches to test-case generation [2]. One approach is to generate test inputs randomly (see [12] for an overview). Another approach is to symbolically execute the code (see e.g. [9, 7, 4]). Inputs are handled as logic variables and at each branching of the control flow, a constraint is added to some constraint store. A solution of the accumulated constraints corresponds to a test case leading to the considered path through the code. Backtracking is often applied in order to consider alternative paths through the code. Some test-case generators offer hybrid approaches combining search-based techniques and symbolic computation, e.g. EvoSuite [5], CUTE [16], and DART [6]. EvoSuite 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.

```
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 } }
```




```
1  public class Circle {
2    double area;
3
4    Circle(double area) {this.area = area; }
5
6    public double getRadius() {
7      assert area>=0;
8      return Sqrt.sqrt(area/Math.PI); }}
```

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

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 the program transformation that allows both EvoSuite and JPet to detect both possible assertion violations.

3 Program Transformation

We start defining the subset of Java considered in this work.

3.1 Java Syntax

In order to simplify this presentation we limit ourselves to the subset of Java defined in Table 1. This subset is inspired by the work of [10]. Symbols e, e_1, ..., indicate arbitrary expressions, b, b_1 ..., indicate blocks, and s, s_1, ..., indicate sentences. Observe that we assume that variable declarations are introduced at the beginning of blocks, although for simplicity we often omit the block delimiters '{' and '}'. A Java method is defined by its name, a sequence of arguments with their types, a result type, and a body defined by a block. The table also indicates if the construction is considered an expression and/or a statement.

The table shows that some expressions e can contain subexpressions e'. A position p in an expression e is represented by a sequence of natural numbers

Description	Syntax	Expr.	Stat.
creation of new objects	`new C(`e_1`, ...,`e_n`)`	yes	no
casting	`(C) e`	yes	no
literal values	k	yes	no
binary operation	e_1 `op` e_2	yes	no
variable access	`varName`	yes	no
attribute access	`e.x`	yes	no
method call	`e.M(`e_1`,...,`e_n`)`	yes$^{(*)}$	yes$^{(**)}$
variable assignment	`vaName = e`	no	yes
attribute assignment	`e.x = e`	no	yes
conditional statements	`if (e)` b_1 `else` b_2	no	yes
while loop	`while(`e_1`) b)`	no	yes
catching blocks	`try` b_1 `catch(C V)` b_2	no	yes
return statements	`return e`	no	yes
assertions	`assert e`	no	yes
block	{s_1`;...;` s_n`;}`	no	yes
block with local var. decl.	`{T V;` s_1`; ...;` s_n`}`	no	yes

(*) Method calls are expressions if the return type is different from `void`
(**) Method calls are statements when they are not contained in another expression

Table 1. Java subset

Statement in program P:

```
double radius = (new Circle(400.5)).getRadius();
```

Flattened program statement in P^F:

```
Circle aux;
aux = new Circle(400.5);
double radius;
radius = aux.getRadius();
```

Fig. 2. Flattening an expression

that identifies a subexpression of e. The notation $e_{|p}$ denotes the subexpression of e found at position p. For instance, given $e \equiv$ `(new C(4,5)).m(6,7)`, we have $e|_{1.2} =$ `(new C(4,5))`$_{|_2} = 5$, since e is a method call, the position 1 stands for its first subexpression $e' \equiv$ `new C(4,5)` and the second subexpression of e' is 5. Given two positions p, p' of the same expression, we say the $p < p'$ if p is a prefix of p' or if $p <_{\text{LEX}} p'$ with $<_{\text{LEX}}$ the lexicographic order. For instance $1 < 1.2 < 2 < 2.1$ (1 prefix of 1.2, $1.2 <_{\text{LEX}} 2$, and 2 prefix of 2.1).

In the rest of the paper, and for the sake of simplicity, we consider the application of a constructor (via the `new` operator) as a method call. A method call that does not include properly another method call as subexpression is called *innermost*. Let e be an expression and $e' = e_{|p}$ an innermost method call. Then, e' is called *leftmost* if every innermost method call $e'' = e_{|p'}$, with $p \neq p'$ verifies $p < p'$.

In the statement example in Fig. 2 the underlined expression is a leftmost innermost method call. The idea behind this concept is that a leftmost innermost expression can be evaluated in advance because it is not part of another method call and it does not depend on other method calls of the same expression due to the Java evaluation order.

The *minimal statement* of an expression e is a statement s that contains e and such that there is no statement s' such that s contains s' and s' contains e.

Observe that in Table 1 neither variable nor field assignments are allowed as part of expressions. This corresponds to the following assumption:

Assumption 1 *All the assignments in the program are statements.*

Using assignments as part of expressions is usually considered a very bad programming practice. Anyway, it is possible to eliminate these expressions by introducing auxiliary variables. We omit the corresponding transformation for the sake of simplicity.

3.2 Flattening

Before applying the transformation, the Java program needs to be *flattened*. The idea of this step is to extract each nested method call and assign its result to a new variable without affecting the Java evaluation order.

Algorithm 1 (Flattening expressions) *Let B be the body of a method and let $e \equiv o.M(es)$ be an expression in B such that:*

1. *e is a leftmost-innermost method call, and M is a user defined method*
2. *e is not the right-hand side of a variable assignment*
3. *e is not a statement*

Let T be the type of e. Finally, let s be the minimal statement associated with e and let V be a new variable name. Then, the following case distinction applies:

1. *s is a* `while` *statement, that is $s \equiv$* `while(`e_1`)` $\{e_2\}$.
   In this case e is a subexpression of e_1, and the flattening of e is obtained replacing s by:

   ```
   {   T V;
       V=e ;
       while (e1 [ e↦V] )  {
           e2 ;
           V=e ;      }   }
   ```

   where the notation $e_1[e \mapsto V]$ stands for the replacement of e by V in e_1.
2. *s is not a* `while` *statement.*
   Then, the flattening of s is defined as

   ```
   {   T V;
       V=e ;
       s [ e↦V] ;  }
   ```

```
public abstract class MayBe<T> {

  public static class Value<T> extends MayBe<T> {//→ Fig. 4}

  public static class CondError<T> extends MayBe<T> {//→ Fig. 4}

  // 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. 3. Class `MayBe<T>`: new result type for instrumented methods.

This process is repeated recursively, until no method call needs to be transformed. Then the program obtained is called the flattened version of P and is represented by P^F in the rest of the paper. The second part of Fig. 2 shows the flattening of a statement in our running example.

3.3 Program Transformation

The idea of the following program transformation is to instrument the code in order to obtain special output values that represent possible violations of assertion conditions.

In our case the instrumented methods employ the class `MayBe<T>` of Fig. 3. 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` (Fig. 4). `Value<T>` represents a value with the same type as in the original code, and it is

used via 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 method `generateError` is to create a new `CondError` object. 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. In Fig. 4 we can observe that the corresponding constructor of `CondError` adds the new call to the path, represented in our implementation by a list.

The transformation takes as parameters a program P and a parameter not discussed so far: the *level* of the transformation. This parameter is determined by the user and indicates the maximum *depth* of the instrumentation. If $level = 0$ then only the methods including assertions are instrumented. This means that the tests will be obtained independently of the method calls performed in the rest of the program. If $level = 1$, then all the methods that include a call to a method with assertions are also instrumented, checking if there is an indirect condition violation and thus a propagating of the error is required. Greater values for *level* enable more levels of indirection, and thus allow to find errors occurring in a more specific program context.

The algorithm can be summarized as follows:

Algorithm 2
Input: P, a Java Program verifying Assumption 1 (all the assignments in the program are statements), and an integer level ≥ 0.
Output: a transformed program P^T

1. Flatten P delivering P^F as explained in Subsection 3.2.
2. Make a copy of each of the methods to instrument by replacing the result type by `MayBe`, as described in Algorithm 3. Call the new program P^C.
3. Replace assertions in P^C by new code that generates an error if the assertion condition is not met, as explained in Algorithm 4. This produces a new program P_0 and a list of methods L_0.
4. For k=1 to *level*: apply Algorithm 5 to P, P_{k-1}, and L_{k-1}. Call the resulting program P_k and list L_k, respectively.
5. Apply your favourite automatic test-data generator to obtain test cases for the methods in L_{level} with respect to P_{level}. Look for the test cases that produce `CondError` values. Executing the test case with respect to the original program P produces an assertion violation and thus the associated exception displays the trace of method calls that lead to the error.

```
public static class Value<T>  extends MayBe<T> {
  T value;

  public Value(T value) { this.value = value; }

  @Override
  public boolean isValue() {return true; }

  @Override
  public T getValue() {return value;}
}

public static class CondError<T> extends MayBe<T> {
  private List<Call> callStack;

  public CondError(Call newElement) {
    this.callStack = new ArrayList<Call>();
    this.callStack.add(newElement);
  }

  public <S> CondError(Call newElement, CondError<S> other) {
    this.callStack = new ArrayList<Call>(other.callStack);
    this.callStack.add(newElement);
  }

  public List<Call> getCallStack() { return callStack;  }

  @Override
  public boolean isValue() { return false; }

  @Override
  public T getValue() { return null;  }
}
```

Fig. 4. Classes `Value<T>` and `CondError<T>`

Now we need to introduce algorithms 3, 4 and 5.

We assume as convention that it is possible to generate a new method name M' and a new attribute name M^A given a method name M. Moreover, we assume that the mapping between 'old' and 'new' names is one-to-one, which allows to extract name M both from M' and from M^A.

Algorithm 3
Input: a flat Java program P^F verifying Assumption 1.
Output: a transformed program P^C with copies of the methods.

1. $P^C = P^F$.
2. For each method (not constructor) $C.M$ in P^F with result type T:
   (a) Include in class C of P^C a new method $C.M'$ with the same body and arguments as $C.M$, but with return type `MayBe<T>`
   (b) Replace each statement `return exp;` by: `return MayBe.createValue(exp);`
3. For each constructor $C.M$ in P:
   (a) Include in the definition of class C of program P^C a new static attribute M^A:

```
static MayBe<C> M^A;
```

   (b) Create a new method $C.M'$ as a copy of $C.M$ with the same arguments `args` as the definition of $C.M$, but with return type `MayBe<C>` and body:

```
MayBe<C> result=null;
M^A = null;
C constResult = new C(args);

// if no assertion has been falsified
if (M^A !=null)
    result = M^A;
else
    result = MayBe.createValue(constResult);
return result;
```

Algorithm 3 copies the class methods, generating new methods M' for checking the assertions. This is done because we prefer to modify a copy of the method, ensuring that the change does not affect the rest of the program. Method M' returns the same value as M wrapped by a `MayBe` object.

Observe that in the case of constructors we cannot modify the output type because it is implicit. Instead, we include a new attribute M^A, used by the constructor, to communicate any violation of an assertion. The new method calls the constructor and checks if there is an assertion violation (M^A `!=null`), returning the new value as output result. If on the contrary M^A is `null` then no assertion violation has been detected and the constructed object is returned wrapped by a `MayBe` object.

The next step or the transformation handles `assert` violations in the body of methods:

Algorithm 4
Input: P^C *obtained from the previous algorithm.*
Output: – P_0, *a transformed program*
– L_0, *a list of methods in the transformed program*

1. $P_0 = P$, $L_0 = [\,]$
2. For each method $C.M$ including an assertion:
   (a) $L_0 = [C.M'|L_0]$, being M' the new method name obtained from M
   (b) If $C.M$ is a method with return type T, not a constructor, replace in $C.M'$ each statement `assert exp;` by:

   ```
   if (!exp) return MayBe.generateError("C.M",i);
   ```

   with i the ordinal of the assertion counting the assertions in the method body in textual order.
   (c) If $C.M$ is a constructor, replace in $C.M$ each statement `assert exp;` by:

   ```
   if (!exp)     M^A = MayBe.generateError("C.M",i);
   ```

   with i as in the case of a non-constructor.

In our running example L_0 = [`Sqrt.srqtCopy`, `Circle.getRadiusCopy`], which are the new names introduced by our transformation for the methods with assertions. Finally, the last transformation focuses on indirect calls. The input list L contains the names of all the new methods already included in the program. If L contains a method call $C.M'$, then the algorithm looks for methods $D.L$ that include calls of the form $C.M(args)$. The call is replaced by a call to $C.M'$ and the new value is returned. A technical detail is that in the new iteration we keep the input methods that have no more calls, although they do not reach the level of indirection required. The level must be understood as a maximum.

Algorithm 5
Input: – P, *a Java flat Program verifying Assumption 1*
– P_{k-1}, *the program obtained in the previous phase*
– *A list* L_{k-1} *of method names in* P_{k-1}
Output: – P_k, *a transformed program*
– L_k, *a list of methods in the* P_k

1. Let $P_k = P_{k-1}$, $L_k = L_{k-1}$
2. For each method $D.L$ in P including a call $x = C.M$ with $C.M$ such that $C.M'$ is in L_{k-1}:
   (a) Let i be the ordinal of the method call in the method body and `y` a new variable name'
   (b) If $C.M'$ is in L_k, then remove it from L_k.
   (c) $L_k = [D.L'|L_k]$
   (d) If $D.L$ is a method of type T, not a constructor then replace in $D.M'$ the selected call to $x = C.M$ by:

```
MayBe<T> y = C.M';
if (!y.isValue())
    return MayBe.propagateError("D.L",i,y);
x = y.getValue();
```

(e) If $D.L$ is a constructor, then let x' be a new variable name. Replace in the constructor $D.L$ the selected call to $x = C.M$ by:

```
MayBe<T> y = C.M';
if (!y.isValue())
     M^A = MayBe.propagateError("D.L",i,y);
x = y.getValue();
```

where M^A is the static variable associated to the constructor and introduced in Algorithm 3.

In our example we have $L_1 = L_0$ since the only indirect call is by means of `Circle.getRadiusCopy`, but this method is already in the list. In fact, $L_k = L_0$ for every $k > 0$.

The transformation of our running example can be found in Fig. 5. It can be observed that in practice the methods not related directly nor indirectly to assertion do not need to be modified. This is the case of the constructor of `Circle`.

4 Experiments

We observed the effects of the transformation by means of experiments. In addition to the running example shown above, we have investigated several other examples as well, ranging from the implementation of the binary tree data structure, Kruskal's algorithm, to the computation of the mergesort method. Furthermore we constructed an example with nested if-statements called *Numeric*. Finally we used two examples representing a blood donation scenario *BloodDonor* and a book lending system *Library*. In the next step, we have evaluated the examples with different test-case generators with and without our `level=1` program transformation. We have 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`.

We have used a test-case generator for exposing possible assertion violations. First of all, we can note that this approach works. In our experiments, all but one possible assertion violation could be detected. Moreover, we can note that additionally our program transformation typically improves the detection rate, as can be seen in Table 2. In this table, column *Total* displays for each example the number of possible assertion violations that can be raised for the method. 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. For instance

```
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. 5. Transformation of the running Example

in our running example `Circle.getRadius` can raise the two assertion violations explained in Section 2. Without the transformation, only one assertion violation is found by EvoSuite. Notice that JPet cannot find any assertion violation without our transformation, since it does not support assertions. Thus, our transformation is essential for tools that do not support assertions, such as JPet. With the transformation, EvoSuite correctly detects both assertion violations. An improvement in the assertion violation detection rate is observed for all examples.

Also tools that support assertions benefit from our program transformation, since it makes the control flow more explicit than the usual assertion-violation exceptions. This helps the test-case generators to reach a higher coverage as can be seen in Table 3. The dashes in the JPet row indicate that JPet does not support assertions and hence cannot be used to detect assertion violations in the untransformed program.

		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

Table 2. Detecting assertion violations.

Our program transformation only requires a few split seconds, even for large programs. 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.

	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

Table 3. Control and data-flow coverage in percent.

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. If the coverage is high, the analysis is nevertheless quite accurate and useful in practice; in particular in situations, where a model checker would require too much time.

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. The result of a computation is encapsulated in an object. The type of this object indicates whether the computation was successful or whether it caused an assertion violation. 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.

Although our approach accounts for the call path that leads to an assertion violation, this path is represented as a chain of object references, so some test case generators might not be able to recreate it in their generated tests. We are studying an alternative transformation that represents the call path in terms of basic Java data types. Another subject of future work is to use the information provided by a dependency graph of method calls in order to determine the maximum call depth level in which the transformation can be applied.

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. E. Albert, I. Cabanas, A. Flores-Montoya, M. Gómez-Zamalloa, and S. Gutierrez. jPET: An automatic test-case generator for Java. In *18th Working Conference on Reverse Engineering, WCRE 2011, Limerick, Ireland, October 17-20, 2011*, pages 441–442, 2011.
2. S. Anand, E. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn. An orchestrated survey on automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, August 2013.
3. C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. *SIGSOFT Softw. Eng. Notes*, 27(4):123–133, July 2002.
4. M. Ernsting, T. A. Majchrzak, and H. Kuchen. Dynamic solution of linear constraints for test case generation. In *Sixth International Symposium on Theoretical Aspects of Software Engineering, TASE 2012, 4-6 July 2012, Beijing, China*, pages 271–274, 2012.
5. J. P. Galeotti, G. Fraser, and A. Arcuri. Improving search-based test suite generation with dynamic symbolic execution. In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 360–369. IEEE, 2013.
6. P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223, 2005.
7. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Test case generation for object-oriented imperative languages in CLP. *TPLP*, 10(4-6):659–674, 2010.
8. M. Harman, A. Baresel, D. Binkley, R. Hierons, L. Hu, B. Korel, P. McMinn, and M. Roper. Testability transformation program transformation to improve testability. In R. Hierons, J. Bowen, and M. Harman, editors, *Formal Methods*

and Testing, volume 4949 of *Lecture Notes in Computer Science*, pages 320–344. Springer Berlin Heidelberg, 2008.
9. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
10. G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Trans. Prog. Lang. Syst.*, 28(4):619–695, 2006.
11. B. Korel and A. M. Al-Yami. Assertion-oriented automated test data generation. In *Proceedings of the 18th International Conference on Software Engineering*, ICSE '96, pages 71–80, Washington, DC, USA, 1996. IEEE Computer Society.
12. P. McQuinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.
13. B. Meyer. *Object-oriented Software Construction (2Nd Ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition, 1997.
14. R. Mitchell, J. McKim, and B. Meyer. *Design by Contract, by Example*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2002.
15. Oracle. Programming With Assertions. `https://docs.oracle.com/javase/jp/8/technotes/guides/language/assert.html`, 2014.
16. K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.
17. W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.