Falsification of Java Assertions Using Automatic Test-Case Generators

Rafael Caballero Manuel Montenegro
Universidad Complutense, Facultad de Informática
Madrid, Spain
email: {rafacr,mmontene}@ucm.es

Herbert Kuchen Vincent von Hof
University of Münster, ERCIS, Münster, Germany
email: {kuchen,vincent.von.hof}@wi.uni-muenster.de

***Abstract*—We present a technique for the static generation of test-cases falsifying Java assertions. Our framework receives as input a Java program including assertions and instruments the code in order to detect 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 transformation ensures that the value obtained for the test-case inputs represents a path of method calls that ends with a violation of some assertion. Our technique deals with Java features such as object encapsulation and inheritance, and can be seen has a compromise between the usual but too late detection of an assertion violation at runtime and an often too expensive complete analysis based on a model checker.**

Keywords–assertion; automatic test-case generation; program transformation; inheritance.

I. INTRODUCTION

The goal of this paper is to present a source-to-source program transformation useful for the static generation of test-cases falsifying Java assertions. In a previous paper [1], we addressed the same goal with a simpler approach which, however, could lead to a combinatorial explosion in the generated program. In this paper, we overcome this problem by introducing a data type containing the aforementioned path of method calls in case of assertion violation.

Using assertions is nowadays a common programming practice and especially in the case of what is known as 'programming by contract' [2], [3], where they can be used, e.g., to formulate pre- and postconditions of methods as well as invariants of loops. Assertions in Java [4] are used for finding errors in an implementation at run-time during the test-phase of the development phase. If the condition in an `assert` statement is evaluated to false during program execution, an *AssertionException* is thrown.

During the same phase, testers often use automated test-case generators to obtain test suites that help to find errors in the program. The goal of our work is to use these same automated test-case generators for detecting assertion violations. 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, which is the source of the assertion falsification can be an *indirect* call (if in the body of method m_1 there is a call to m_2, then we say that m_1 *calls* m_2 *directly*. When m_2 calls m_3 directly and m_1 calls m_2 directly or indirectly, we say that m_1 *calls* m_3 *indirectly*). Our technique considers indirect calls up to a fixed level of indirection, allowing checking the assertions in the context of the whole program.

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 [5] 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 next section discusses related approaches. Section III presents a running example and introduces some basic concepts. Section IV presents the program transformation, while Section V sketches a possible solution to the problem of inheritance. Section VI shows by means of experiments how two existing white-box, path-oriented test-case generators benefit from this transformation. Finally, Section VII presents our conclusions.

II. RELATED WORK

The most common technique for checking program assertions is model-checking [6]. It is worth observing that, in contrast to model checking, automated test-case generators are not complete and thus our proposal may miss possible assertion violations. However, our experiments show that the technique described in this paper performs quite well in practice and is helpful either in situations where model checking cannot be applied, or as a first approach during program development before using model checking [7]. 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.

The origins of our idea can be traced back to the work [8], which 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* [9], which 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 [10] 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.

In [1], we took another transformation-based approach to assertion falsification, in which methods containing assertions were transformed to return a boolean value indicating whether an assertion is violated. In the case of a method with several assertions, the transformation generates as many boolean methods as constraints exist in the corresponding method's body, so each method reports the violation of its corresponding assertion. If we want to catch assertion violations obtained through a given sequence of method calls, the transformation shown in [1] generates as many methods as sequences of method calls up to a maximum level of indirection given by the user. However, this could cause an exponential growth in the number of generated methods w.r.t. the indirection level. In this paper, we overcome this problem by defining a type that contains the path leading to an assertion violation, so the test case generator can report assertion violations through different paths by using a single transformed method. An extended abstract of this approach can be found in [11].

III. Conditions, Assertions, and Automated Test-Case Generation

Java assertions allow the programmer to ensure that the program, if executed with the right options, fulfils certain restrictions at runtime. They can be used to formulate, e.g., preconditions and postconditions of methods and invariants of loops. As an example, let us consider the code in Figs. 1 and 2, which introduces 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 with its radius as only attribute. The constructor specifies that the radius must be nonnegative. There is also a static method `Circle.ofArea` for building a `Circle` given its area. Besides checking whether the area is nonnegative, this method calls `Sqrt.sqrt` to compute a square root in order to obtain the radius.

Thus, `Circle.ofArea` will raise an assertion exception if the area is negative, but it may also raise an exception even when the area is nonnegative, due to the aforementioned error in `Sqrt.sqrt`.

```
public class Circle {
  private double radius;

  public Circle(double radius) {
    assert radius >= 0;
    this.radius = radius;
  }

  public double getRadius() {
    return radius;
  }

  public static Circle ofArea(double area) {
    assert area >= 0;
    return new Circle(
          Sqrt.sqrt(area / Math.PI)
        );
  }
}
```

Figure 1: Class `Circle`.

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. 2, 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: (5,8), (9,11), (9,8), and (9,13).

There are mainly two approaches to test-case generation [5]. One approach is to generate test inputs metaheuristically, i.e., search-based with hill climbing or genetic algorithms, which often involve randomizing components (see [12] for an overview). Another approach is to symbolically execute the code (see, e.g., [13], [14], [15]). 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 [16], CUTE [17], and DART [18].

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.ofArea`. A reason is that EvoSuite does not model the call stack. Thus, the test-cases generated by EvoSuite for `Circle.ofArea` only expose one of the two possible violations, namely the one related to a negative area.

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

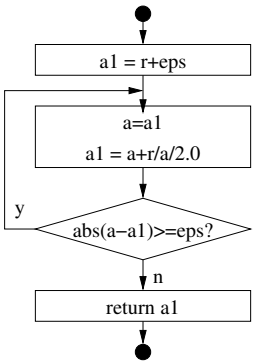

Figure 2: Java method `sqrt` and its corresponding control-flow graph.

There are other test-data generators such as JPet [19] 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.

IV. Program Transformation

We start defining the subset of Java considered in this work. Then, we shall define an auxiliary transformation step that *flattens* the input program, so it can be subsequently handled by the main transformation algorithm.

A. Java Syntax

In order to simplify this presentation we limit ourselves to the subset of Java defined in Table I. This subset is inspired by the work of [20]. Symbols e, e_1, ..., indicate arbitrary expressions, b, b_1 ..., indicate blocks, and s, s_1, ..., indicate statements. 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 whether the construction is considered an expression and/or a statement.

The table shows that some expression e can contain subexpression e'. A position p in an expression e is represented by a sequence of natural numbers 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 $e|_{1.2} = (\texttt{new C(4,5)})_{|_2} = 5$, 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).

Last statement in `ofArea` method, in which the leftmost innermost call is underlined:

```
return new
    Circle(Sqrt.sqrt(area / Math.PI));
```

After extracting the `Sqrt.sqrt` call:

```
double sqrtResult;
sqrtResult = Sqrt.sqrt(area / Math.PI);
return new Circle(sqrtResult);
```

After extracting the constructor call:

```
double sqrtResult;
sqrtResult = Sqrt.sqrt(area / Math.PI);
Circle circleResult;
circleResult = new Circle(sqrtResult);
return circleResult;
```

Figure 3: Flattening an expression.

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. 3 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.

TABLE I: Java subset.

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 variable declaration	{`T V;` s_1`; ...;` s_n}	no	yes

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

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

Observe that in Table I 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 bad programming practice, as the evaluation of those expressions introduces side effects, namely the modification the variables in scope. Anyway, it is possible to eliminate these expressions by introducing auxiliary variables. For instance, given the following program:

```
int sum = 0;
int x;
while ((x = next()) != -1) {
  sum += x;
}
```

The subexpression `x = next()` can be factored out as follows:

```
int sum = 0;
int x;
x = next();
while (x != -1) {
  sum += x;
  x = next();
}
```

We omit the corresponding transformation for the sake of simplicity.

B. 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];
}
```

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. 3 shows the flattening of the last statement of the function `Circle.ofArea` in our running example.

C. 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. 5. 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>` (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<T>` 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<T>` object. Observe that the constructor of `CondError<T>` 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 call 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<T>` 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.

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

Figure 4: Classes `Value<T>` and `CondError<T>`.

```
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();

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

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

  // calls another method whose assertion 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);
  }
}
```

Figure 5: Class `MayBe<T>`: New result type for instrumented methods.

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 propagation 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 $level \geq 0$.

Output: a transformed program P^T

1) Flatten P delivering P^F as explained in Section IV-B.
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, which correponds to the path contained within the `CondError` object.

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: flat Java program P^F verifying Assumption 1.

Output: transformed program P^{Copy} with copies of the methods.

1) Let $P^{Copy} := P^F$.
2) For each method (not constructor) $C.M$ in P^F with result type T:
   a) Include in class C of P^{Copy} 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 e;
```

in the body of $C.M$ by

```
return MayBe.createValue(e);
```

3) For each constructor $C.M$ in P^F:

a) Include in the definition of class C of program P^{Copy} 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 in order to ensure 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 is a wrapper that calls the constructor, which will have been transformed in order to assign a non-null value to M^A in case of an assertion violation (see Algorithm 4). Then, the method checks whether there has been an assertion violation in the constructor (that is whether M^A `!= null` holds) and returns the new value as output result. If, on the contrary, M^A is `null` then no assertion violation has taken place in the constructor and the newly built object is returned wrapped by a `MayBe` object.

In our running example, assume we want to instrument the methods `Sqrt.sqrt`, `Circle.ofArea`, and the constructor of `Circle`. A copy of each of these methods would be generated by Algorithm 3. The new fields and methods are shown in Figure 6. Notice that the condition `circleM != null` in `CircleCopy` will never hold, since the current `Circle` constructor does not alter the `circleM` variable. We will change the code of the `Circle` constructor in the following algorithms.

Inside `Sqrt` class:

```
public MayBe<Double> sqrtCopy(double value) {
    ... // same body as sqrt except the last
    ... // return statement
    return Maybe.createValue(a1);
}
```

Inside `Circle` class:

```
private static MayBe<Circle> circleM;

public MayBe<Circle> CircleCopy(double radius)
{
  MayBe<Circle> result = null;
  circleM = null;
  Circle constResult = new Circle(radius);

  if (circleM != null) {
    result = circleM;
  } else {
    result = Maybe.createValue(constResult);
  }
  return result;
}

public static MayBe<Circle> ofAreaCopy(double
    area) {
  assert area >= 0;
  double sqrtResult;
  sqrtResult = Sqrt.sqrt(area / Math.PI);
  Circle circleResult;
  circleResult = new Circle(sqrtResult);
  return MayBe.createValue(circleResult);
}
```

Figure 6: New methods and fields generated after duplication.

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

Algorithm 4:

Input: P^{Copy} obtained from the previous algorithm.

Output: – P_0, a transformed program

– L_0, a list of methods in the transformed program

1) Let $P_0 := P^{Copy}$, $L_0 := [\,]$
2) For each method $C.M$ containing an assertion:
   a) Let $L_0 := [C.M'|L_0]$, 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, we get $L_0 = [$`Sqrt.sqrtCopy, Circle.CircleCopy, Circle.ofAreaCopy`$]$, which are the new names introduced by our transformation for the methods with assertions. In `Sqrt.sqrtCopy` the `assert` statement would be replaced by the following,

```
...
if (!(a == 1.0 ||
      (a1 > 1.0 ? a1 < a : a1 > a)))
   return MayBe.generateError("sqrt", 1);
...
```

whereas in `Circle` the transformation would affect the constructor and the `ofAreaCopy` method:

```
public Circle(double radius) {
  if (!(radius >= 0)) {
    circleM = MayBe.generateError("Circle", 1);
    return;
  }
  this.radius = radius;
}

public static MayBe<Circle> ofAreaCopy(double
    area) {
  if (!(area >= 0))
    return MayBe.generateError("ofArea", 1);
  ...
}
```

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'

```
public class Sqrt {
  public static final double eps = 0.000001;

  public static double sqrt(double r) { ... }

  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", 1);
      aux = Math.abs(a - a1);
    }
    return MayBe.createValue(a1);
  }
}
```

Figure 7: `Sqrt` class after transformation.

   b) If $C.M'$ is in L_k, then remove it from L_k.
   c) Let $L_k := [D.L'|L_k]$
   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 to a method in L_0 is by means of `Circle.ofAreaCopy`, but the latter 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 Figs. 7 and 8. It can be observed that in practice the methods not related directly nor indirectly to an assertion do not need to be modified. This is the case of the `getRadius` method.

```
public class Circle {
  private double radius;
  private static MayBe<Circle> circleM;

  public Circle(double radius) {
    if (!(radius >= 0)) {
      circleM =
        MayBe.generateError("Circle", 1);
      return;
    }
    this.radius = radius;
  }

  public MayBe<Circle> CircleCopy(double
      radius)
  {
    MayBe<Circle> result = null;
    circleM = null;
    Circle constResult = new Circle(radius);

    if (circleM != null) {
      result = circleM;
    } else {
      result = Maybe.createValue(constResult);
    }
    return result;
  }

  public double getRadius() {
    return radius;
  }

  public static Circle ofArea(double area) {
    ...
  }

  public static MayBe<Circle>
      ofAreaCopy(double area) {
    if (!(area >= 0))
      return MayBe.generateError("ofArea", 1);

    MayBe<Double> sqrtResultM;
    sqrtResultM = Sqrt.sqrtCopy(area /
        Math.PI);
    if (!sqrtResultM.isValue())
      return MayBe.propagateError("ofArea", 2,
          sqrtResultM);
    double sqrtResult = sqrtResultM.getValue();

    MayBe<Circle> circleResultM;
    circleResultM = CircleCopy(sqrtResult);
    if (!circleResultM.isValue()) {
      return MayBe.propagateError("ofArea", 3,
          circleResultM);
    }
    Circle circleResult =
        circleResultM.getValue();
    return MayBe.createValue(circleResult);
  }
}
```

Figure 8: `Circle` class after transformation.

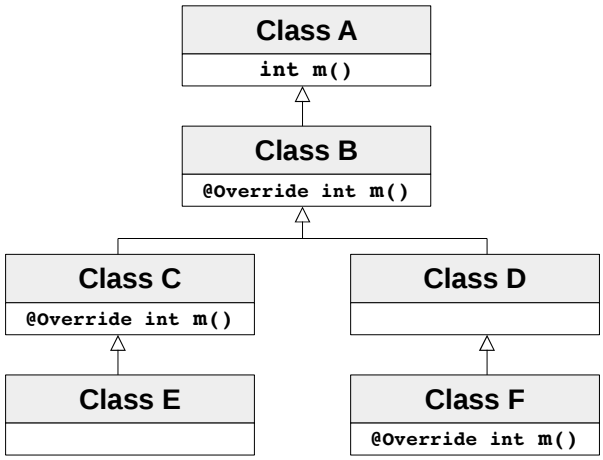

Figure 9: Inheritance example.

V. Inheritance

Inheritance poses a new interesting challenge to our proposal. Consider the hierarchy shown in Fig. 9, in which we assume that the implementation of `m()` in `B` contains an assertion, and hence, it is transformed according to Algorithm 4. If there are neither assertions nor calls to `B.m()` in the remaining classes of the hierarchy, it seems that there is no further transformations to apply. However, assume the following method:

```
public int foo(A a) {
  return a.m();
}
```

If we have the call `foo(new B(..))` then it becomes apparent that `foo()` can raise an assertion due to dynamic dispatching, because the call `a.m()` corresponds in this context to a call to `B.m()`. Thus, in order to detect this possible assertion violation, `foo()` needs to be transformed by introducing a `fooCopy()` method containing a call to `a.mCopy()` in its body. In turn, this implies that class `A` must contain a method `mCopy()` as well. Therefore, we create a method `mCopy()` in `A` with the following implementation:

```
public MayBe<Integer> mCopy() {
  return MayBe.createValue(m());
}
```

which wraps the result of `m()` into a `MayBe` value. This wrapper implementation must be replicated in classes `C` and `F` as well, since they also override `m()`.

In general, whenever we create a copy of a method $C.M$, we have to create a copy method with the wrapper implementation in the class where M is defined for the first time in the class hierarchy, and in each descendant C' of C overriding M unless there is another class between C and C' in the hierarchy which

TABLE II: Detecting assertion violations.

		EvoSuite		JPet	
Method	Total	P	P^T	P	P^T
Circle.ofArea	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	-	-

also overrides M, or C' already has a copy M' of the method M (e.g., because $C'.M$ contains another assertion). In the example of Fig. 9, this means that we need to create additional `mCopy()` methods in classes `A`, `C`, and `F`.

An obvious limitation is when we introduce an assertion in methods defined in a library class such as `Object` (for instance, when overriding method `toString`), since we cannot introduce new methods in these classes. Fortunately, introducing assertions when overriding library methods is quite unusual. A possible improvement, still under development, is to look in advance for polymorphic calls. For instance, maybe method `foo()` is never called with arguments of type `C` in the program and there is no need of transforming this class.

VI. Experiments

We observed the effects of the transformation by means of experiments, including the running example shown above, the implementation of the binary tree data structure, Kruskal's algorithm, the computation of the mergesort method, a constructed example with nested if-statements called *Numeric*, an example representing a blood donation scenario *BloodDonor* and two bigger examples, namely a self devised *Library* system, which allows customers to lend and return books and the 6500 lines of code of the package java.util.logging of the Java Development Kit 6 (JDK). In all the cases, the transformation has been applied with level *infinite*, i.e., application of the transformation until a fixed point is reached. 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 two test-case generators, JPet and EvoSuite, for exposing possible assertion violations. First of all, we can note that our approach works. In our experiments, all but one possible assertion violation could be detected. Moreover, we can note that our program transformation typically improves the detection rate, as can be seen in Table II. 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, in our running example, `Circle.getRadius` can raise the two assertion violations explained in Section III. Without the transformation, only one assertion violation is found by EvoSuite. With the transformation, EvoSuite correctly detects both assertion violations. For JPet no test cases are created for `java.util.logging`, since JPet does not support library method calls. 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. An improvement in the assertion violation detection rate is observed for all examples.

Additionally, tools that already support assertions to some degree 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 III. 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. Our program transformation often only requires a few seconds and even for larger programs such as the JDK 6 logging package the transformation finishes in 18.2 seconds. 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.

VII. 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. We tried to use the model checker Java Pathfinder [21] to our examples, but we had to give up, since this tool was too time consuming or stopped because of a lack of memory.

Additionally, we have developed a program transformation that 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

TABLE III: 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

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 where the transformation can be applied.

ACKNOWLEDGMENT

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] R. Caballero, M. Montenegro, H. Kuchen, and V. von Hof, "Automatic falsification of Java assertions," in Proceedings of the 7th International Conference in Advances in System Testing and Validation Lifecycle (VALID 2015), T. Kanstren and B. Gersbeck-Schierholz, Eds. IARIA, 2015, pp. 36–41.

[2] B. Meyer, Object-oriented Software Construction (2Nd Ed.), 2nd ed. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997.

[3] R. Mitchell, J. McKim, and B. Meyer, Design by Contract, by Example. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2002.

[4] Oracle, "Programming With Assertions," http://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html, Retrieved: 8 January 2017.

[5] 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, vol. 86, no. 8, August 2013, pp. 1978–2001.

[6] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," Automated Software Engineering, vol. 10, no. 2, 2003, pp. 203–232.

[7] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," Softw. Test. Verif. Reliab., vol. 22, no. 5, Aug. 2012, pp. 297–312. [Online]. Available: http://dx.doi.org/10.1002/stvr.456

[8] B. Korel and A. M. Al-Yami, "Assertion-oriented automated test data generation," in Proceedings of the 18th International Conference on Software Engineering, ser. ICSE '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 71–80.

[9] 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 Formal Methods and Testing, ser. Lecture Notes in Computer Science, R. Hierons, J. Bowen, and M. Harman, Eds. Springer Berlin Heidelberg, 2008, vol. 4949, pp. 320–344.

[10] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on Java predicates," SIGSOFT Softw. Eng. Notes, vol. 27, no. 4, Jul. 2002, pp. 123–133.

[11] R. Caballero, M. Montenegro, H. Kuchen, and V. von Hof, "Checking java assertions using automated test-case generation," in Proceedings of the 25th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR), ser. Lecture Notes in Computer Science, M. Falaschi, Ed., vol. 9527. Springer International Publishing, 2015, pp. 221–226.

[12] P. McQuinn, "Search-based software test data generation: A survey," Software Testing, Verification and Reliability, vol. 14, no. 2, 2004, pp. 105–156.

[13] J. C. King, "Symbolic execution and program testing," Commun. ACM, vol. 19, no. 7, 1976, pp. 385–394. [Online]. Available: http://doi.acm.org/10.1145/360248.360252

[14] M. Gómez-Zamalloa, E. Albert, and G. Puebla, "Test case generation for object-oriented imperative languages in CLP," TPLP, vol. 10, no. 4-6, 2010, pp. 659–674. [Online]. Available: http://dx.doi.org/10.1017/S1471068410000347

[15] 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, Beijing, China, 2012, pp. 271–274. [Online]. Available: http://dx.doi.org/10.1109/TASE.2012.39

[16] 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). IEEE, 2013, pp. 360–369.

[17] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in Proceedings of the 10th European Software Engineering Conference, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 263–272. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081750

[18] 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, 2005, pp. 213–223. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065036

[19] 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, 2011, pp. 441–442.

[20] G. Klein and T. Nipkow, "A machine-checked model for a Java-like language, virtual machine and compiler," vol. 28, no. 4, 2006, pp. 619–695.

[21] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda, "Model checking programs," Autom. Softw. Eng., vol. 10, no. 2, 2003, pp. 203–232. [Online]. Available: http://dx.doi.org/10.1023/A:1022920129859