# Automatic Falsification of Java Assertions

Rafael Caballero Manuel Montenegro

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

Herbert Kuchen Vincent von Hof

University of Münster, Institute of Information Systems
Münster, Germany
email: {kuchen,vincent.von.hof}@wi.uni-muenster.de

***Abstract*—We present an approach for the static detection of possible assertion violations in Java. The main idea is to use an existing test-case generator in combination with a new program transformation. A possible assertion violation is indicated by a generated specific test case. In addition, this test case specifies the path in the program leading to the assertion violation. This heuristic approach is 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.**

## I. Introduction

Assertions are part of the Java language [1] and have become part of the routine employed by Java programmers to detect and correct bugs. They can be used e.g. for specifying pre- and postconditions of methods or invariants of loops. If an assertion is violated, this is detected at runtime and a corresponding exception is thrown. A drawback is that it may take a long time, until assertion violations occurring in rarely executed code show up. Possibly, this can happen when the code has already been deployed and assertions are turned off [2, chapter 6]. Thus, the error can be difficult to detect and its correction become very costly.

Callahan et al. [3] proposed the use of model checkers for the automated generation of test cases. This model-based testing approach has been a fruitful area of research in the last years [4], and encompasses the creation of an abstract model which is used to automatically create test cases. However, using model checkers often requires more effort and expertise than the simple introduction of assertions. Additionally, the process of finding the assertion violation can still become a hard, time-consuming task due to the typically huge search space.

Our idea is to find some compromise between the two mentioned approaches and use a test-case generator [5], [6] to obtain test-cases for the considered code. If such a tool generates a test-case aiming at producing an assertion violation, this indicates that such an assertion violation can actually happen and that there is some corresponding bug in the program. Test-case generators do not explore the complete space of all possible computations as done by a model-checker. Typically, they apply a heuristic based on a combination of random search and symbolic evaluation in order to generate a set of test cases which cover the control- and/or data-flow of a program systematically [5], [7], [8], [9], [10], [11]. This approach cannot guarantee to find all possible assertion violations. Nevertheless, it works quite well and it is helpful in practice.

As mentioned before, a violation of a Java assertion causes an exception to be thrown. Unfortunately, test-case generators often have difficulties to cover exception handling well. Thus, our approach does not just rely on an existing test-case generator. Before using it, we apply a program transformation, which replaces assertions and the corresponding exception handling by "ordinary control structures". As we will show, this improves the coverage rate of test-case generators significantly. In addition, it allows test-case generators such as *jPET* [6] to be used, which do not support assertions.

Roughly, the approach presented here introduces new boolean methods representing the paths leading to possible assertion violations. In the case of methods including directly assertions, the body of the new method is a copy of the method where the assertion occurs, but replacing the assertion `assert` *e* by `return` *e*. This converts assertion violations into first-class citizens from the point of view of automated test-case generators, which usually focus on methods and their results.

The new `return` statements often produce fragments of unreachable code in the body of the new methods. These fragments can be automatically removed, thus simplifying the task of the test-case generator, and achieving a simple form of static slicing, as computations which are not relevant for assertions are not taken into account. This transformation is simpler than the alternative approach presented in [12], where every method is replaced by another one delivering a pair of the original result and a value indicating whether an assertion violation occurred. Propagating such violation information is technically a bit clumsy and the mentioned slicing is not obtained.

The paper is structured as follows. In Section II, we explain our transformation based on a running example, while in Section III, we present our transformation in detail. Section IV contains some experimental results. Finally, in Section V we summarize and point out future work.

## II. Running Example

In order to get an overview of our transformation, let us consider the classes shown in Figure 1, which contain an implementation of the insertion-sort algorithm. An instance of `InsertionSort` contains a reference to the array to be sorted. This reference is initialised within the constructor, which previously checks, whether it is given a non-null reference. The `insert` method receives a number `n` and performs an ordered insertion of the element `x[n]` into the sub-array

```
public class InsertionSort {
   private int[] x;

   public InsertionSort(int[] x) {
      assert x != null;
      this.x = x; }

   public void insert(int n) {
      assert isSorted(n-1);
      assert n <= x.length;
      int i = n;
      while (i >= 1 && x[i-1] > x[i]) {
         int e = x[i-1];
         x[i-1] = x[i];
         x[i] = e;
      }
      assert isSorted(n); }

   public void insertSort() {
      for (int i = 1; i < x.length; i++)
         insert(i); }

   public boolean isSorted(int n) {
      for (int i = 1; i < n; i++)
         if (x[i-1] > x[i]) return false;
      return true; }
}

public class Check {
   public static void check(int []x) {
      InsertionSort is = new InsertionSort(x);
      is.insertSort(); }
}
```

Figure 1. Running example "insert sort"

`x[0..n-1]`, which is assumed to be sorted. That is why we include an assertion that calls the method `isSorted`, which checks whether the array `x` is sorted up to the position given as parameter, disregarding the elements after that position. After the insertion, we check again (via another `assert`) that the resulting sub-array `x[0..n]` is sorted. Notice that there is a mistake in this method, as variable `i` should be decremented at the end of each iteration. Otherwise, the loop would always terminate either before the first iteration (if `x[n-1] <= x[n]`) or before the second one (when `x[n-1] > x[n]` holds before the first iteration, but not afterwards).

The `insertionSort` method calls `insert` as many times as the length of the array indicates, successively performing ordered insertions from the first element to the last one. Finally, class `Check` represents any arbitrary application class that employs an object of class `InsertionSort`. It is clear that some possible inputs of `Check.check` can trigger an assertion exception, exposing the existence of an error in the code. Our goal is to find such input values employing an automated test-case generator. In particular, it would be great, if the test-case generator could pay special attention to the assertions in the code, since any input data producing an assertion falsification reveals a code bug. This reduces the problem of checking the test-suite (known as the *oracle problem* [13]), as we can focus first on those test-cases producing assertion violations.

**Input:** A Java program $P$
**Output:** A Java program $P_0$ for the testing assertions

```
$P_0 = P$
for all method C.M $\in P$ containing assertions do
   Create a boolean copy C.M' of C.M in $P_0$
   Let $n$ be the number of assertions in C.M
   Let $a_j \equiv$ assert $e_j$ be these assertions, where $1 \leq j \leq n$, represents the textual order of occurrence of the assertion in C.M
   for $i = 1 \ldots n$ do
      Create in $P_0$ a new method C.M$^0_i$ as copy of C.M' except for:
      Assertion $a_i$ is replaced by return $e_i$
      Every $a_j \equiv$ assert $e_j$, $j < i$ is replaced by boolean $v_j = e_j$, with $v_j$ a new variable name
      Every assertion $a_j$ with $j > i$ is removed.
   end for
   Remove C.M'
end for
```

Figure 2. Algorithm 1: Level 0, methods including assertions

However, some test-case generators, such as *jPET* [6] or *Muggl* [11], do not support assertions. Others, like *EvoSuite* [5] support assertions but have some problems when the assertions are located in a different class. In our running example, the three automated test-case generators find a test-case corresponding to the case of null array input, but fail to generate any test-case exposing the error in the code of method `insert`. In the next section, we present the program transformation that will change this situation.

## III. Transformation

Given a Java program including assertions, we introduce new methods that return the value `false` whenever the assertion property does not hold. Each method represents a certain path to an assertion violation.

In the rest of the section, given a method C.M we use the expression *create a boolean copy C.M' of method C.M* to indicate the creation a new method M' in class C such that C.M' is a copy of C.M except for:

1) The return type of C.M', which is `boolean`,
2) Statements `return e;` in C.M, which are replaced by `return true;` in C.M'.
3) If the type of C.M was `void`, then `return true;` is added as last statement of C.M'.
4) If C.M is a constructor, then add the access modifier `static` to the declaration of C.M'.

First, we produce the methods that correspond directly to methods containing assertions.

### A. Methods including assertions

The first algorithm creates a transformed program $P_0$ as a copy of the initial $P$ with some additional methods (see Figure 2).

Thus, a method C.M containing $n$ assertions gives raise to $n$ new methods C.M$^0_1$, ..., C.M$^0_n$, all of them with return type `boolean` and each one checking a particular assertion.

The auxiliary method C.M' is only introduced to facilitate the generation of the new methods, and is removed at the end.

For instance, in the case of the method `insert` of our running example, the C.M' method is obtained by replacing the return type by `boolean` and adding a new statement `return true;` at the end:

```
   public boolean insertPrime(int n) {
      // same body as insert
      ....
      return true; }
```

The method `insert` contains three assertions. Hence $n = 3$, and three new methods are included in the same class. The method associated to the first assertion is:

```
   public boolean insert1(int n) {
      return isSorted(n-1);// assertion
      int i = n;
      // code of the while loop in insert
      ....
      return true; }
```

Since the first statement is a `return`, any Java optimizer will prune the rest of the code, as it is unreachable, compiling instead:

```
   public boolean insert1(int n) {
      return isSorted(n-1); }
```

This is one of the main advantages of our approach: the new methods are often much smaller than the original ones and thus, the test-cases are obtained more easily. With respect to `insert_2`, we obtain:

```
public boolean insert_2(int n) {
   boolean _unused_1 = isSorted(n-1);
   return n <= x.length; }
```

Although we are interested only in the second assertion of `insert`, we still evaluate the condition of the first assertion, since it may involve side effects that may affect the result of the second one. The code of `insert_3`, the method associated to the last assertion, can be found in Figure 3.

It is worth observing that the constructors can be considered as any other method (except for the introduction of the `static` modifier), and no special treatment is needed. This is an important difference with respect to [12], where a more complicated treatment of constructors is necessary.

After this initial transformation, we can use our test-case generator to look for values $v_1, \ldots, v_k$ such that C.M$_i^0(v_1, \ldots, v_k)$ produces the value `false`, indicating that C.M$(v_1, \ldots, v_k)$ triggers assertion $a_i$. However, we would like to go one step beyond and consider if such a call can actually occur in our application. This is the purpose of the algorithm in the following subsection.

### B. *Indirect access to assertions*

We say that the level of indirection of a method C.M is zero, if the method contains an assertion (case considered in the previous section), and $l > 0$, when it contains a call to method C'.M', and C'.M' has a level of indirection $l - 1 \geq 0$. The idea behind this definition is that methods with levels greater than zero can end triggering an assertion and must be transformed as well. If a method does not contain an assertion and it does not contain method calls (possibly indirectly) leading to assertions, the level of indirection is undefined.

Notice that the same method can have different levels of indirection related to different method calls. For instance, method `Check.check` (Figure 1) has an indirection level of 1 with respect to the call of constructor `InsertionSort` that contains an assertion, but also level 2 with respect to the call `is.insertSort`, as method `InsertionSort.insertSort` has indirection level 1. A *maximal level of indirection* is assumed as an input parameter of the following algorithm. It can either be obtained previously by the tool analyzing the code of considered as an input parameter fixed by the user to limit the number of methods generated in case of large applications.

Our transformation creates new methods associated to each level of indirection. We assume that it is possible to distinguish the auxiliary methods created when processing a method at a certain level of indirection. For instance, all the auxiliary methods C.M$_i^0$ created by Algorithm 2 correspond to the transformation at level 0 of their source method C.M.

Now, we apply the transformation to the `insertionSort` method, which does not contain any explicit assertion, but it may indirectly trigger an assertion violation in `insert`. The call to `insert` occurs in a loop, and hence first the loop is unfolded. Let us suppose that the parameter *Unfold* takes value 2.

```
public void insertSortPrime() {
  int i=1;
  if (i<x.length) {
     insert(i);
     i++;
     if (i<x.length)
        insert(i); } }
```

In the algorithm, this means that $p = 2$ (two calls to `insert`) and $q = 3$ (three versions of `insert` have been already generated). Thus, we obtain a family of $2 \times 3 = 6$ methods `insertSort_`$i$`_`$j$, with $1 \leq i \leq 2$ and $1 \leq j \leq 3$. The method `insertSort_`$i$`_`$j$ checks whether the condition of the $j$-th assertion executed within the $i$-th call of `insert` is satisfied.

For instance, `insertSort_1_3` in Figure 3 corresponds to the possibility that the first `insert` in `insertSortPrime` falsifies the post-condition of `insert` (its third assertion).

The transformation can also be applied to constructors, like `InsertionSort_1` prefix. Therefore, when transforming a method that involves the instance creations of `InsertSort`, we have to replace the `new InsertSort(...)` expressions by calls to this static method in order to check the validity of the assertions contained in the constructor. This is the case of `Check.check_1_1` in Figure 3.

The same figure includes a family of methods `check_2_`$i$`_`$j$ that reports the assertions being violated by the indirect through the `insertSort` method. Each

```
.... // original methods

public static boolean
   InsertionSort_1(int[] x) {
   return x != null; }

public boolean insert_1(int n) {
   return isSorted(n-1); }

public boolean insert_2(int n) {
   boolean _unused_1 = isSorted(n-1);
   return n <= x.length; }

public boolean insert_3(int n) {
  boolean _unused_1 = isSorted(n-1);
  boolean _unused_2 = n <= x.length;
  int i = n;
  while (i >= 1 && x[i-1] > x[i]) {
     int e = x[i-1];
     x[i-1] = x[i];
     x[i] = e;
  }
  return isSorted(n); }

 public boolean insertSort_1_1() {
   int i = 1;
   if (i < x.length)
      return insert_1(i);
   return true;}

 public boolean insertSort_1_2() {
   int i = 1;
   if (i < x.length)
      return insert_2(i);
   return true; }

 public boolean insertSort_1_3() {
   int i = 1;
   if (!(i < x.length))
      return insert_3(i);
   return true;}

  public boolean insertSort_2_1() {
    int i = 1;
     if (i < x.length) {
        insert(i);
        i++;
        if (i < x.length)
          return insert_1(i);}
    return true;}
```

```
public boolean insertSort_2_2() {
 int i = 1;
 if (i < x.length) {
   insert(i);
   i++;
   if (i < x.length)
     return insert_2(i);}
 return true; }

public boolean insertSort_2_3() {
 int i = 1;
 if (i < x.length) {
   insert(i);
   i++;
   if (i < x.length)
      return insert_3(i);}
 return true;}


public class Check {
 public static boolean check_1_1(int []x) {
  return InsertionSort.InsertionSort_1(x);}

 public static boolean check_2_1_1(int []x) {
  InsertionSort is = new InsertionSort(x);
  return is.insertSort_1_1(); }

 public static boolean check_2_1_2(int []x) {
  InsertionSort is = new InsertionSort(x);
  return is.insertSort_1_2();}

 public static boolean check_2_1_3(int []x) {
  InsertionSort is = new InsertionSort(x);
  return is.insertSort_1_3();}

 public static boolean check_2_2_1(int []x) {
  InsertionSort is = new InsertionSort(x);
  return is.insertSort_2_1();}

 public static boolean check_2_2_2(int []x) {
  InsertionSort is = new InsertionSort(x);
  return is.insertSort_2_2();}

 public static boolean check_2_2_3(int []x) {
  InsertionSort is = new InsertionSort(x);
  return is.insertSort_2_3();}}
```

Figure 3. Running example after the transformation

of these methods subsequently calls the corresponding `insertSort_`$i$`_`$j$ variant.

After the transformation and in order to check, whether an assertion may be violated at runtime, we just have to invoke a test-case generator on one of these generated methods and look for those cases that yield `false` as a result. For instance, when given the method `check_2_2_3`), *jPET* generates a test case (an instance of `InsertionSort` containing the array $[-8, -9, -10]$) which violates the third assertion executed by the second call to `insert`. Analogously, *EvoSuite* and *Muggl* also find an assertion violation associated with `check_2_2_3`. Observe that the name of the method specifies a very detailed scenario: it indicates that with the given input array, `check` causes an assertion falsification in its sec-

```
Input:
    • Program $P_0$: output of Algorithm 1
    • Level: maximum level of indirection allowed (greater than 0).
    • Unfold: A positive number indicating the number of iterations to unfold the loops in the body methods.
Output: $P^T$: A Java program ready to be used to obtain the test-cases falsifying the properties indicated in the assertions.

$P^T = P^0$
Mark $P^T$ methods containing assertions with level 0.
for l=1 ...level do
  for all method D.N in $P^T$ containing calls to methods marked as level $l-1$ do
    Mark D.N as method of level $l$.
    Create a boolean copy D.N' of D.N in $P^T$
    if any call to a $l-1$ method in D.N' occurs in a loop statement then
      Unfold the loop in the copy D.N' the number of times specified by the algorithm input parameter unfold.
    end if
    for all call T x = C.M(...); occurring in method D.N', with C.M marked as level $l-1$ do
      Let $p$ be the number of calls to method C.M in D.N'
      Let C.M$^{l-1}_{s_1}$, ..., C.M$^{l-1}_{s_q}$ be the $q$ auxiliary methods created at level $l-1$ for C.M
      for i=1 ...p, j = 1 ...q do
        Create a copy D.N$^{l}_{i,s_j}$ of D.N'.
        Replace in D.N$^{l}_{i,s_j}$ the statement T x = C.M; by return C.M$^{l-1}_{s_1}$;.
      end for
      Delete D.N'
    end for
  end for
end for
```

Figure 4. Algorithm 2: Level of indirection greater than 0

ond call `is.insertSort();` (first number 2). Moreover, it also shows that `insertSort` causes the assertion falsification in the second iteration of the loop (this is represented by the second number 2 in the name), and the falsification occurs in the last assertion of `insert` (final number 3).

By including the decrement instruction `i--;` at the end of the loop within `insert` (as explained above), no assertion violations are found.

## IV. EXPERIMENTAL RESULTS

To observe the effects of the transformation, we have utilized experimentation. In addition to the running example shown above, we have investigated several additional examples [14], ranging from the implementation of the binary tree data structure, Kruskal's algorithm, to Mergesort. Finally, we used two examples representing a blood donation scenario *BloodDonor* and a larger application, namely a library system, where users can lend and return books. In the next step, we have evaluated the examples with different test-case generators with and without our program transformation.

TABLE I. DETECTING ASSERTION VIOLATIONS

| | | EvoSuite | | jPet | |
|---|---|---|---|---|---|
| Example | Total | $P$ | $P^T$ | $P$ | $P^T$ |
| InsertionSort | 4 | 3 | 4 | 0 | 4 |
| CircleRadius | 2 | 2 | 2 | 0 | 2 |
| BloodDonor | 2 | 1 | 2 | 0 | 2 |
| InsertTree | 2 | 1 | 2 | 0 | 2 |
| Kruskal | 1 | 1 | 1 | 0 | 1 |
| Library | 5 | 0 | 5 | 0 | 5 |
| MergeSort | 2 | 1 | 1 | 0 | 1 |
| Numeric | 2 | 2 | 2 | 0 | 2 |

We have used two test-case generators for exposing possible assertion violations. First of all, we can note that this approach works. Moreover, we can note that our program transformation typically improves the detection rate, as can be seen in Table I. 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 (0 in the case of JPet because it does not handle assertions) and the original program and column $P^T$ displays them after applying the transformation. For instance in our running example four assertion violations can be raised. Without the transformation, three assertion violations are found by EvoSuite. With the transformation, EvoSuite correctly detects all four assertion violations. An improvement in the assertion violation detection rate is observed for all examples. *jPET* does not consider assertions in its current state, but can detect them after our program transformation.

Thus both tools that do and do not support assertions benefit from our program transformation. The runtime of our analysis can range from a few seconds to several minutes.

## V. CONCLUSION

Assertions constitute a useful, widely-used feature of the Java language. They are widely used for detecting bugs in the testing-phase. However, only those assertion violations actually occurring at runtime can be detected.

Automated test-case generators can be situated somehow in the middle of the very light-weight technique of run-time checking Java assertions and the formal methods such as model checking. They do not require the definition of abstract models, but aim to cover as many executions as possible of the program, yielding test-case suites that can be used to look for possible errors. The main difficulty is to check the generated test-suites looking for test-cases producing erroneous results. This is known as the *oracle problem* [13]. In order to solve this problem, [15] proposes including the assertions as part of the code and use automated test-case generation to obtain inputs that falsify the conditions. This approach was already presented in [11] and has given rise to the so called assertion-based software-testing technique.

In this paper, we have presented a proposal for transforming a Java program including new boolean methods that help to check the program assertions. Each of these methods returns false, whenever its input parameters lead -directly or indirectly- to a falsification of some assertion property. Moreover, the name of the method contains a path to the assertion.

Some automated test-case generation tools do not consider assertions. The presented transformation allows the user to employ even such test-case generators to generate test-cases exposing assertion violations. Moreover, we have seen that it can also contribute to increase the completeness of the test-cases obtained in some tools such as *EvoSuite* [5] that already consider assertions. We also think that this proposal can be useful during the development of new test-case generators in order to include readily the possibility of dealing with assertions. The advantage of our technique is that assertions are replaced by standard code that can be analyzed using the usual techniques.

It is worth observing that using our transformation, the test-cases corresponding to assertions are easy to distinguish, since they correspond to new auxiliary methods returning `false`. Thus, it is possible to implement readily an automatic tool that extracts from the test-suite the test-cases falsifying assertions.

The main limitations of the proposal are:

1) The necessity of unfolding the loop statements where assertions are included. Since the unfolding is done a fixed number of times, this can reduce the effective covering of the test-cases.
2) The combinatorial explosion in the number of methods. We have seen in the description of the transformation that if a method contains $n$ assertions and is called $m$ times by other methods, we need to generate $n$ auxiliary methods for the first one and $n \times m$ auxiliary methods for the second one.

The unfolding (or 'unrolling') of the loops containing methods using assertions is not a very severe restriction in practice, because most automated test-case generators do the same internally. Moreover, we have found that most errors show up after just two iterations, like in the running example of this paper. Anyway they can add more incompleteness to the results.

The positive part of unfolding the loops is that errors found are very precise. In our running example we can check that all the methods leading to assertion violations require two iterations of the loop. This points out the updating of variables at the end of the loop as a possible cause of the bug, which is indeed the case. To the best of our knowledge, no test-case generator can provide such detailed information.

The combinatorial explosion in the number of auxiliary methods can become an issue for large programs with many assertions. We have found that processing each assertion separately instead of all the assertions at the same time results in a considerable speed-up. Anyway, it is worth observing that the process is automatic and requires no user-interaction once it has been started.

As future work, we plan to finish a prototype that automatizes both the transformation and its connection with different test-case generators. An important part of the prototype is the decodification of the auxiliary method names once an assertion falsification has been found, in order to show to the user a detailed information about the source of the bug. We also plan to extend the framework to the case of inheritance and polymorphism. Our preliminary results in this sense indicate that the same technique can be applied in the presence of polymorphism with the creation of 'dummy' auxiliary methods in the ancestor classes of the class hierarchy to ensure that the method exists and can be used also in polymorphic contexts.

## 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] Oracle, "Programming With Assertions," http://docs.oracle.com/javase/6/docs/technotes/guides/language/assert.html, retrieved: August, 2015.

[2] G. Travis, JDK 1.4 Tutorial. Manning Publications, 2002.

[3] J. Callahan, F. Schneider, and S. Easterbrook, Eds., Automated software testing using model-checking, 1996, proceedings 2nd SPIN workshop.

[4] M. Shafique and Y. Labiche, "A systematic review of state-based test tools," Int. J. Softw. Tools Technol. Transf., vol. 17, no. 1, Feb. 2015, pp. 59–76. [Online]. Available: http://dx.doi.org/10.1007/s10009-013-0291-0

[5] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 416–419. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025179

[6] 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.

[7] 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.

[8] 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

[9] 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

[10] 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, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 263–272. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081750

[11] 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, 2012, pp. 271–274. [Online]. Available: http://dx.doi.org/10.1109/TASE.2012.39

[12] R. Caballero, M. Montenegro, H. Kuchen, and V. von Hof, "Checking java assertions using automated test-case generation," in 25th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2015), 2015, retrieved: August, 2015. [Online]. Available: https://gpd.sip.ucm.es/rafa/papers/lopstr15.pdf

[13] E. Barr, M. Harman, P. McMinn, M. Shabaz, and S. Yoo, "The oracle problem in software testing: A survey," IEEE Transactions on Software Engineering, vol. PP, no. 99, 2014, pp. 1–1.

[14] R. Caballero, M. Montenegro, H. Kuchen, and V. von Hof, "Examples used," https://github.com/wwu-ucm/valid-15-examples, retrieved: August, 2015.

[15] 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.