# A 5-step hunt for faults in Java implementations of algebraic specifications

Isabel Nunes and Filipe Luís
Faculty of Sciences, University of Lisbon
Lisboa, Portugal

*Abstract*—**Executing thorough test suites allows programmers to strengthen the confidence on their software systems. However, given some failed test cases, finding the faults' locations can be a difficult task, thereby techniques that make it easier for the programmer to locate the faulty components are much desirable. In this paper we focus on finding faults in object-oriented, more precisely Java, implementations of data types that are described by algebraic specifications. We capitalize on models for the specification under study and JUnit test suites that cover all axioms of the specification, and present a collection of techniques and underlying methodology, that give the programmer, in a transparent way, a means to locate the fault that causes the implementation to violate the specification. We also present a comparative experiment that was carried out to evaluate this approach where very encouraging results were obtained.**

## I. Introduction

The development and verification of software programs against specifications of desired properties is growing weight among software engineering methods and tools for promoting software reliability. In particular, algebraic specifications have the virtue of being stateless, which allows to describe a given component operations in a way that is independent of internal representation and manipulation.

In this paper we capitalize on and enrich an integrated approach to specification-guided software development, which comprises the ConGu approach ([11], [12]) to the development of generic Java implementations of parameterized algebraic specifications, and the GenT tool ([2], [3]) that generates thorough test cases for those implementations. We forward the reader to the referred papers to a justification of the chosen languages and methodologies, and to the comparison with other proposals with similar objectives.

Generating test cases for generic Java implementations that are known to be thourough, i.e. that cover all axioms/properties of the specification, as GenT does, is a very important activity but, in order for these tests to be of effective use, we must be able to use their results to localize the faulty entities – simply executing the JUnit tests generated by GenT fail to give the programmer clear hints about the faulty method because all methods used in failed tests become equally suspect. The ideal result of a test suite execution is the exact localization of the fault(s).

Several methods have been proposed in the literature, in varied settings, to the diagnosis of test failures, that can be quite different in the way they approach the problem – in the related work section of this paper we will focus our discussion on some specific aspects that are relevant to the work here presented.

We present a 5-step approach that integrates a collection of techniques, and underlying methodology, that allows to interpret and manipulate the results of failed tests in order to give the programmer a good hint about the Java method, among the ones that implement the specification, where the fault lies.

The presented approach, unlike several existing approaches to fault-location, does not primarily inspect the executed code; instead, it exploits the specification and corresponding models in order to be able to interpret some failures and discover their origin.

Departing from failed JUnit tests, an initial universe of suspects is identified which is then augmented/reduced by the stepwise application of several techniques, eventually obtaining a unique suspect. Some of these techniques imply the automatic generation of new model-based tests from the initial ones, that give new and useful information towards the goal of reducing the number of suspects.

The adopted integration testing strategy is an incremental one in the sense that the Java types implementing the specification are not tested all together; instead, each one is tested conditionally, presuming all others from which it depends are correctly implemented. This incremental integration is possible since the overall specification is given as a structured collection of individual specifications (a ConGu module), whose structure is matched by the structure of the software system that implements it.

The remainder of this paper is organized as follows: section II gives an overview of the 5-step integrated approach – it presents an example that will be used throughout the paper and gives the reader some background information about GenT test generation since the techniques proposed in this paper use several of GenT's artifacts to draw conclusions and generate additional information; four probing techniques are presented in section III whose main objective is to collect and reason about suspect methods and decide the guilty one; section IV compares the results of the 5-step approach presented in this paper with the ones obtained using two fault-location tools [15], [16] and some specific issues about related work are discussed in section V. Finally, section VI concludes.

Fig. 1. The inputs and outputs of the 5-step integrated approach.

## II. Approach Overview

Before going into detail of each step of our integrated approach, we shall first give an overview of the approach and some preliminaries, in order to be possible for readers to fully understand the techniques presented ahead. Figure 1 shows the several types of inputs the 5-step approach works with: (A) a ConGu specification module (a collection of specifications together with the structure underlying it); (B) a set of Java classes implementing the module; and (C) a mapping telling how the specified ConGu types and operations refine into the Java implementing types and methods. The output is the location of some existing fault (a ranked list of methods suspect of containing the fault).

The 5-step approach comprises the generation and execution of a thorough set of tests and, if some failure happens (indicating a non-conformance of the implementation against the specification), the application of a series of techniques aiming at finding the guilty method.

We pick a classical yet rich parameterized data type – the SortedSet – and one of its possible implementations – a `TreeSet` Java class. The parameterized SortedSet data type is specified by the ConGu module containing the SortedSet *core* specification and the TotalOrder *parameter* specification, which are both defined using the ConGu specification language.

*Constructors*, *observers* and *other* operations can be specified with the ConGu specification language, where constructors compose the minimal set of operations that allow to build all instances of the type, observers can be used to analyse those instances, and the other operations are usually comparison operations or operations derived from the others; depending on whether they have an argument of the type under specification (self argument) or not, constructors are classified as *transformers* or *creators* (transformers are also referred to as *non-creator constructors*). All operations that are not constructors must have a self argument. Functions can be partial (denoted by $\rightarrow ?$), in which case a *domains* section specifies the conditions that restrict their domain. Axioms define every value of the type through the application of observers to constructors. Mappings between specification types and Java types, and between specification operations and Java methods, can be defined using the ConGu refinement language. Detailed information about the ConGu approach can be found in [11], [12], [14].

With those input elements, in a first step, the 5-step process begins by invoking the GenT tool [2], [3], which creates thorough JUnit tests to be used to test the implementation against the specification.

```
sig SortedSet extends Element  {
       largestOp : lone Orderable,
       isInOp : (Orderable) -> one BOOLEAN/Bool,
       isEmptyOp : one BOOLEAN/Bool,
       insertOp : (Orderable) -> one SortedSet
}
//(...)
fact axiomSortedSet3  {
   all EVar, FVar : Orderable, SVar : SortedSet |
      SVar.insertOp[EVar].isInOp [FVar] = BOOLEAN/True iff
        (EVar = FVar or SVar.isInOp[FVar] = BOOLEAN/True)
}
//(...)
run axiomSortedSet3_0  {
   some EVar, FVar : Orderable, SVar : SortedSet |
      (SVar.insertOp[EVar].isInOp[FVar] = BOOLEAN/True and
         (EVar = FVar and SVar.isInOp[FVar] = BOOLEAN/True))
} for 6 but exactly 4 SortedSet
//(...)
```

Fig. 2. Part of the Alloy specification of the SortedSet module.

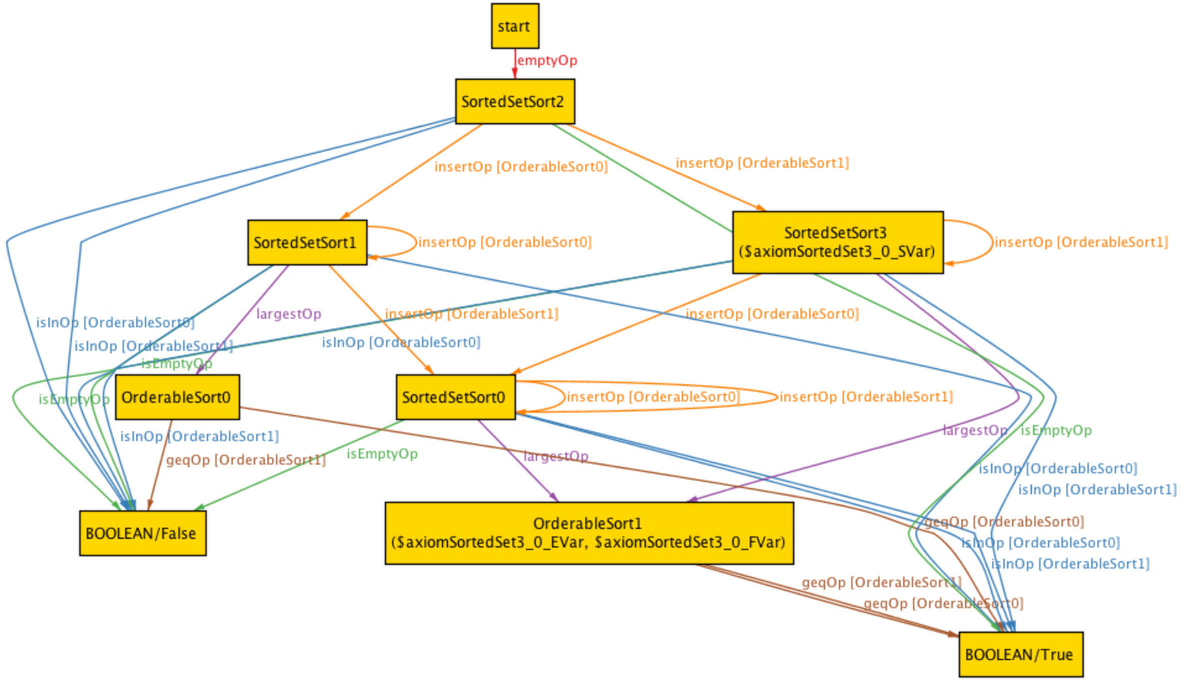


Fig. 3. A model for the axiomSortedSet3_0 run command found by the Alloy Analyzer.

In addition to these tests, GenT tool also generates additional elements that will be used by some of the other tasks of the 5-step process:

- an Alloy [8] specification that is equivalent to the original ConGu module, (see figure 2 for an excerpt, noting that fact axiomSortedSet3 is the Alloy counterpart of the ConGu axiom (isIn(insert(S,E),F) **iff** E=F **or** isIn(S,F) in the SortedSet specification of figure 1);
- extra Alloy *run* commands that thoroughly cover all specification axioms (figure 2 shows one of the *run* commands, one pertaining to the axiomSortedSet3 fact).

These *run* commands, from which the GenT tool generates the JUnit tests, correspond to parts of the full disjunctive normal form (FDNF) of each of the specification axioms $ax$ ($ax = \forall_{x_1:s_1,\ldots,x_n:s_n} \bigvee_{1\leq j\leq m} \varphi_j$) where each clause or minterm is a conjunction containing all literals of the axiom ($\varphi_j = \bigwedge_{1\leq k\leq l} a_{jk}$). More concretely, for each specification axiom $ax = \varphi_1 \vee \ldots \vee \varphi_m$, Alloy *run* commands $R_j$ are generated, at least one for each one of the $m$ clauses or minterms $\varphi_j$ of $ax$. For a given axiom, at least one of the minterms is satisfiable.

In order for the reader to understand the techniques that are based on these elements, we first briefly explain how the GenT tool generates the JUnit tests suite.

The GenT tool asks Alloy Analyzer, a tool that finds finite models of relational structures, to generate models for each and every *run* command, and uses these models to define the Java objects that the JUnit tests will use. Figure 3 shows an example of a model that was generated for the axiomSortedSet3_0 *run* command of figure 2 – this model instantiates the variable SVar with a SortedSet containing orderable1; the variable EVar is orderable1 and FVar is also orderable1; moreover, orderable1 is greater or equal to orderable0.

For parameterized specifications, GenT automatically generates finite implementations of the parameter types, for each test, from the models found by the Alloy Analyzer. This is done in two steps: *(i)* mock classes are created for each parameter type (see figure 4), whose instances will represent objects satisfying the corresponding parameter specification (these classes are independent from particular models, while their instances are created according to a given model); *(ii)* mock objects (instances of mock classes) are created and initialized in each test, according to the corresponding model (see figure 5).

```
1 public class OrderableMock implements IOrderable<OrderableMock>{
2    private HashMap<OrderableMock, Boolean> greaterEqMap =
                             new HashMap<OrderableMock, Boolean>();

3    public boolean greaterEq(OrderableMock e) {
4        return greaterEqMap.get(e);
5    }

6    public void add_greaterEq(OrderableMock e, Boolean result) {
7        greaterEqMap.put(e, result);
8    }
9 }
```

Fig. 4. Mock class for the Orderable parameter type.

Apart from methods with the same signature as the ones defined in the original Java interface that restricts the intended parameter implementation (see interface `IOrderable` in

figure 1), a mock class also contains methods that allow to define the results those original methods are supposed to deliver, according to a given model – see line 2 in figure 4, declaring and initializing a map that will contain the results of the `greaterEq` method for each value of its parameter, and lines 6 to 8 that declare a method whose purpose is to define those results. Furthermore, see lines 2 to 7 in figure 5, where two instances of the `OrderableMock` class are created and the results of the `greaterEq` method are "fed" into them according to the model in figure 3.

```
  @Test
1 public void axiomSortedSet3_0() {
      // IOrderable Mocks
2     final OrderableMock orderableSort0 = new OrderableMock();
3     final OrderableMock orderableSort1 = new OrderableMock();
4     orderableSort0.add_greaterEq(orderableSort0, true);
5     orderableSort0.add_greaterEq(orderableSort1, false);
6     orderableSort1.add_greaterEq(orderableSort0, true);
7     orderableSort1.add_greaterEq(orderableSort1, true);
      // Prepare Core Var Factories
8     CoreVarFactory<TreeSet<OrderableMock>> sVarFac =
              new CoreVarFactory<TreeSet<OrderableMock>>() {
9      public TreeSet<OrderableMock> create() {
10         TreeSet<OrderableMock> s = new TreeSet<OrderableMock>();
11         s.insert(orderableSort1);
12         return s;
13     }
14    };
     // Test the Axiom
15    axiomSortedSet3Tester(sVarFac, orderableSort1, orderableSort1);
16 }

17 private void axiomSortedSet3Tester(
        CoreVarFactory<TreeSet<OrderableMock>> sVarFac,
                        OrderableMock eVar, OrderableMock fVar) {
18    TreeSet<OrderableMock> sVar_0 = sVarFac.create();
19    TreeSet<OrderableMock> sVar_1 = sVarFac.create();
20    sVar_0.insert(eVar);
21    assertTrue(
      sVar_0.isIn(fVar) == (eVar == fVar || sVar_1.isIn(fVar)));
    // axiom isIn(insert(S,E), F) iff E = F or isIn(S, F);
22 }
```

Fig. 5. JUnit test for the axiomSortedSet3_0 run command and respective model (fig. 3).

All tests for a same axiom test the same assertion – the axiom itself; the difference between them lies in the objects that are used to verify the `assert` statements – each test verifies the axiom over a given model that satisfies a given run of that axiom (remember each run corresponds to a clause/minterm of the FDNF of the axiom).

In each generated JUnit test for a given axiom we can identify three parts:

- a first part where the parameters of the generic data type are created and initialized (mock objects) according to the Alloy model found for the corresponding run (lines 2 to 7 in figure 5, as already exemplified);
- a second part where factory objects capable of creating core type objects, according to the same model, for each free variable used by the given axiom (sVarFac in the example, created in lines 8 to 14, which is capable of building an object corresponding to the SortedSetSort3 element of the model in figure 3, which is identified by the Alloy Analyzer with the variable sVar in the *run* command of figure 2);
- a third part where a tester method for the givem axiom is invoked (line 15), that creates as many core type objects as needed to exercise the axiom (lines 18 and 19), prepares them (line 20), and tests the corresponding Java assertion (line 21).

The test cases that result from this generation process are thorough, since all operations are tested for cases that cover every consistent minterm of all axioms.The tools and theory behind them are not the novelty here; they were explained because the next section will capitalize on the underlying Alloy specification and extra *run* commands.

The 1st step of our 5-step approach finishes with the execution of these tests; the remaining four steps will only take place if some of these tests fail, in order to supply more clues about which method to blame for a given error.

The following abbreviations will be used in the next sections, considering fixed a specification, an implementation, a refinement, and corresponding Alloy runs and GenT generated tests: $\alpha(R)$ is the assertion verified in run $R$ (minterm of an axiom); $\alpha(t)$ is the assertion verified in test $t$ (it corresponds to the axiom that is at the base of test $t$); $t(R)$ is the Java test generated from run $R$.

## III. Four steps remaining

The four techniques we present in this section are aimed at identifying a faulty method in a Java class that implements what we call the *designated* type of the ConGu specification module; they assume that the implementations of the other types of the ConGu module are correct.

After having applied the GenT tool to obtain the Alloy specification and JUnit tests, and having run the tests, our 5-step approach picks, as illustrated in figure 6, (i) the Alloy specification equivalent to the original ConGu module; (ii) the Java types that implement the module, where a fault exists in the class corresponding to the designated type; (iii) the mapping defining a refinement of specification types and operations to Java elements; (iv) the set of JUnit tests created by GenT, where some have failed, and the information about the exceptions they launched; and (v) mock classes for the parameter types, and applies the several techniques presented in this section, in an integrated way, to identify possible suspects and to strengthen or weaken suspicions of guilt.

The underlying methodology departs from a situation in which one or more tests fail and all methods are equally suspect, picks some particular failed tests and tries to look at the behaviour of the involved operations in a different perspective, as a way to unveil other facets of those operations and obtain additional significant information. We begin (according to figure 6) by reasoning about the clues we may get from a test failure due to an unchecked exception as, for example, `NullPointerException`, `ArrayIndex OutOfBounds Exception`, etc. Afterwards, we will study JUnit tests failures due to `AssertionError`. Finally, we will see an alternative way of using the specification models to obtain more information about the location of the fault, and interpret the results.

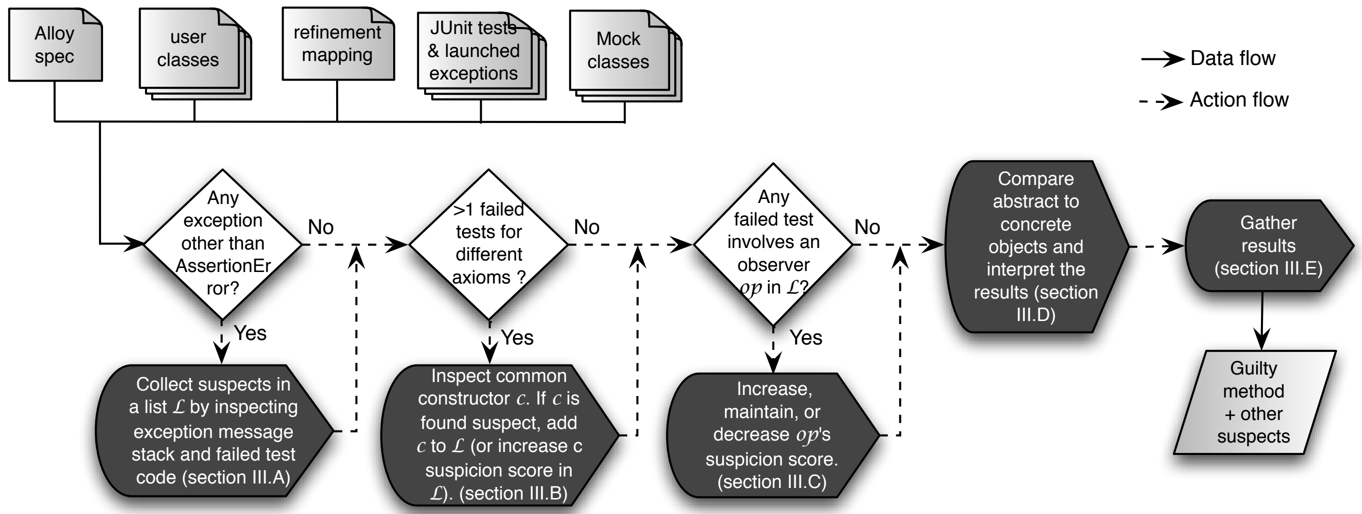


Fig. 6. The four techniques are put to work if initial tests detect some failure.

### A. *Failure caused by unchecked exceptions*

Whenever an unchecked exception arises, apart from the obvious suspect (if it exists) – the method in the class implementing the designated type that threw the exception –, other methods can be suspect candidates. For example, if the methods that are used in its pre-condition are ill-implemented, it can be the case that a well–implemented method throws an exception because it is invoked in a state where it should not.

Departing from a situation where the source code of the class under study is not available, but the failed test code and the exception messages stack are, we must observe the latter to find the method where the exception was raised, and investigate its pre-condition because the operations that are involved in it become suspicious. If the method that raised the exception is not a candidate for the suspects list – for example, it is a method belonging to a class that is not the one that implements the designated type – we must investigate the parameters used in its invocation. Furthermore, if we have other tests that also abort from some unchecked exception, we may try to find the most probable suspect by intersecting the obtained sets of suspects.

The following algorithm describes the necessary steps to build the suspects list, having in mind the above discussed cases:

1) for each test that fails with an exception different from `AssertionError`:
   a) inspect the first line in the message stack, identifying the method $m^*$ that launched the exception:
      i) if $m^*$ is of the designated type, make $m$ this method
      ii) else, find the() method(s) of the designated type that is(are) used to compose the parameter(s) of $m^*$ in the exception-throwing invocation and make $m_{(i)}$ this(these) method(s)
   b) add $m_{(i)}$ to the suspects list;
   c) if (each) $m_{(i)}$ has a pre-condition $p$:
      i) add the methods that are used in $p$ to the suspects list;
2) in the end, intersect the suspects lists of all the failed tests to obtain the most probable guilty operations.

The succession of steps 1.(a).i.(b), for example, covers the cases where the method $m$ that launched the exception is ill-implemented; the succession of steps 1.(a).i.(b).(c).i covers the cases where the method $m$ that launched the exception failed because it was incorrectly invoked, and a defect in some method involved in its pre-condition caused this to be evaluated to true instead of false.

EXAMPLE – the guilty operation is used in the pre-condition of the method that launched the exception:

Let method `isEmpty` of the `TreeSet` implementation be faulty such that it returns false for empty sets; moreover, let method `largest`, whose pre-condition is `!isEmpty`, be implemented in such a way that it launches `ArrayIndexOutOfBoundsException` when the set is empty.

Some tests fail with that exception, and the `largest` method is the one that the exception message stack points to. According to the algorithm above, the `isEmpty` method, as well as `largest`, are added to the suspects list.

### B. *Working on constructor operations*

Following figure 6, we now show how we can gather new suspects to the suspects list from methods implementing constructor operations, by inspecting specific pairs of failed tests.

The idea here is to pick every pair of failed tests that test different axioms but use the same constructor, and mix the two to try to unveil more information about that constructor. Because they originate from different axioms, there is a possibility that they only have the constructor operation in common.

Let $t(R_f)$ and $t(R_g)$ be two failed tests. Moreover, let runs $R_f$ and $R_g$ from which the two tests originate be from different axioms and involve the same constructors.

Because $t(R_f)$ and $t(R_g)$ test different axioms, they pottentially test different observers over the same constructors. Let $R_{fg}$ be a new, compound, run that results from joining the assertions from the two runs corresponding to the two failed tests. The assertion for this new run is $\alpha(R_{fg}) = \alpha(R_f) \wedge \alpha(R_g)$.

Let the Alloy Analyzer generate models for this new run. Then, generate the test $t(R_{fg})$ as usual (as GenT does), but using a model that ensures the test's failure. As expected, this new test is such that $\alpha(t_{fg}) = \alpha(t_f) \wedge \alpha(t_g)$.

Because the failing test $t_{fg}$ is built over a set of objects that corresponds to a model for run $R_{fg}$, and the constructor is the only operation we can guarantee the original tests have in common, then: **(1)** if $\alpha(t_f)$ and $\alpha(t_g)$ fail, we can strengthen the suspicion of guilt of the common constructor; **(2)** if $\alpha(t_f)$ fails and $\alpha(t_g)$ passes, or vice-versa, the guilt should be sought for some other operation but the constructor.

We must be able to find which is the case (1 or else 2). The following reasoning accomplishes the task: To find if both $\alpha(t_f)$ and $\alpha(t_g)$ fail, we build a new test, over the same objects, where we negate both $\alpha(t_f)$ and $\alpha(t_g)$, and run it; if it passes, then (1) is the case. To find if one sub-expression fails while the other passes, we can use the same above built test where $\alpha(t_f)$ and $\alpha(t_g)$ were negated; if it fails, then (2) is the case.

So, we build test $t_{f0g0}$ from $t_{fg}$ using the same model instance satisfying $R_{fg}$, such that $\alpha(t_{f0g0}) = \neg\alpha(t_f) \wedge \neg\alpha(t_g)$, and execute it. If it passes, we conclude that the original problem may be caused by the constructor used in $t_{fg}$, (*strengthens* the guilt); if it fails we conclude that the original problem should not be caused by the constructor used in $t_{fg}$ (*weakens* the guilt);

EXAMPLE We "plant" a fault in the method that implements the SortedSet insert constructor (the method with the same name), in order to exemplify the application of this technique.

As a result of this fault, some of the GenT tests generated in step 1 of the 5-step process fail; among them, the test for run axiomSortedSet3_0 (that gives semantic to operation isIn):

```
run axiomSortedSet3_0  {
        some EVar, FVar : Orderable, SVar : SortedSet | (SVar.insertOp[EVar].isInOp
                [FVar] = BOOLEAN/True and (EVar = FVar and SVar.isInOp[FVar] =
                BOOLEAN/True))
} for 6 but exactly 4 SortedSet
```

that corresponds to (a minterm of the FDNF of) axiom 3:

```
fact axiomSortedSet3  {
        all EVar, FVar : Orderable, SVar : SortedSet | SVar.insertOp[EVar].isInOp[
                FVar] = BOOLEAN/True iff (EVar = FVar or SVar.isInOp[FVar] = BOOLEAN/
                True)
}
```

and the test for run axiomSortedSet5_0 (that gives semantic to operation largest):

```
run axiomSortedSet5_0  {
        some EVar : Orderable, SVar : SortedSet | ((SVar.isEmptyOp != BOOLEAN/True
                and EVar.geqOp[SVar.largestOp] = BOOLEAN/True) and SVar.insertOp[EVar
                ].largestOp = EVar)
} for 6 but exactly 4 SortedSet
```

that corresponds to (a minterm of the FDNF of) axiom 5:

```
fact axiomSortedSet5  {
        all EVar : Orderable, SVar : SortedSet | (SVar.isEmptyOp != BOOLEAN/True
                and EVar.geqOp[SVar.largestOp] = BOOLEAN/True) implies SVar.insertOp[
                EVar].largestOp = EVar
}
```

Since they involve the same constructor operation – insert –, we will join these two runs and create a compound run as explained above. A new run axiomSortedSet3_0_5_0 is created:

```
run axiomSortedSet3_0_5_0  {
        some EVar, FVar : Orderable, SVar : SortedSet |
                (SVar.insertOp[EVar].isInOp[FVar] = BOOLEAN/True and (EVar = FVar
                        and SVar.isInOp[FVar] = BOOLEAN/True))
        and
                ((SVar.isEmptyOp != BOOLEAN/True and EVar.geqOp[SVar.largestOp] =
                        BOOLEAN/True) and SVar.insertOp[EVar].largestOp = EVar)
} for 6 but exactly 4 SortedSet
```

We execute the Alloy Analyzer for this run in order to find models satisfying it, and we generate a test for it based on an Alloy model that guarantees this test's failure. Figure 7 shows such a test where the underlying model defines that SVar is a SortedSet into which orderable0 and orderable1 are inserted; EVar is orderable1 and FVar is also orderable1; moreover, orderable1 is greater or equal to orderable0.

```
@Test
public void axiomSortedSet3_0_5_0(){
   // Mocks' setup
   OrderableMock orderableSort0 = new OrderableMock();
   OrderableMock orderableSort1 = new OrderableMock();
   orderableSort0.add_greaterEq(orderableSort0, true);
   orderableSort0.add_greaterEq(orderableSort1, false);
   orderableSort1.add_greaterEq(orderableSort0, true);
   orderableSort1.add_greaterEq(orderableSort1, true);
   // Factories core var setup
   CoreVarFactory<TreeSet<OrderableMock>> sVarFac =
                  new CoreVarFactory<TreeSet<OrderableMock>>() {
      public TreeSet<OrderableMock> create() {
         TreeSet<OrderableMock> s = new TreeSet<OrderableMock>();
         s.insert(orderableSort0);
         s.insert(orderableSort1);
         return s;
      }
   };
   // Test the Axiom
   axiomSortedSet3_0_5_0Tester(sVarFac, orderableSort1, orderableSort1);
}

private void axiomSortedSet3_0_5_0Tester(
      CoreVarFactory<TreeSet<OrderableMock>> sVarFac,
                  OrderableMock eVar, OrderableMock fVar) {
   TreeSet<OrderableMock> sVar_0 = sVarFac.create();
   TreeSet<OrderableMock> sVar_1 = sVarFac.create();
   TreeSet<OrderableMock> sVar_2 = sVarFac.create();
   TreeSet<OrderableMock> sVar_3 = sVarFac.create();
   TreeSet<OrderableMock> sVar_4 = sVarFac.create();
   //axiom 3
   sVar_0.insert(eVar);
   assertTrue(sVar_0.isIn(fVar) == (eVar == fVar || sVar_1.isIn(fVar))
   //axiom 5
   if(!sVar_2.isEmpty() && eVar.greaterEq(sVar_3.largest())) {
      sVar_4.insert(eVar);
      assertTrue(sVar_4.largest().equals(eVar));
   }
}
```

Fig. 7. JUnit test for the joint run.

Next we generate the modified test for this run (the one that negates both axiom assertions), as explained above, and execute it. The part of this new test method that creates the objects over which the test works is obviously similar because it uses the same model. The difference lies in the tester method it calls (see figure 8).

```
private void axiomSortedSet3_0N_5_0NTester(
      CoreVarFactory<TreeSet<OrderableMock>> sVar_Factory,
                   OrderableMock eVar, OrderableMock fVar) {
   // create the treesets
   ...
   // test the negation of the axioms
   //axiom 3 negated
   sVar_0.insert(eVar);
   assertTrue(sVar_0.isIn(fVar) != (eVar == fVar || sVar_1.isIn(fVar)));
   //axiom 5 negated
   sVar_4.insert(eVar);
   assertTrue(!sVar_2.isEmpty() &&  eVar.greaterEq(sVar_3.largest()) &&
                                  !sVar_4.largest().equals(eVar));
}
```

Fig. 8. Tester method with negated assertions.

To simplify reading the example, we did not use the FDNF of axioms in this test but, instead, a direct negation of the axioms (from `==` to `!=` and from (*B if A*) to (*A and not B*)).

This test passed, so we can strengthen the suspicions of guilt of the constructor – insert – that is common to both axioms.

### C. Working on non-constructor operations

Recalling figure 6, here the idea is to try to discover whether the tests fail due to a wrong implementation of a given non-constructor, observer operation *op* or not. Apart from failed tests, some prior suspicion on a given operation *op* (e.g., by technique in section III-A) gives us a means to better focus our effort.

Let $t$ be a failed test and *op* a non-constructor operation that was pointed as a suspect earlier, for which we want to strengthen/weaken that suspicion. Let further $t$ contain operation *op*, $t$ originate from run $R$ which corresponds to a given, particular, axiom, whose FDNF has $m$ minterms, and $\alpha(t) = \mathcal{T}(\alpha(R) \bigvee_{1 \geq j \geq (m-1)} \alpha(R_j))$, where $\mathcal{T}$ denotes the result of the translation from the Alloy run to the JUnit test (this translation includes all Java instructions and methods that are necessary to create the objects implementing the Alloy model entities, and the JUnit expressions that test the above assertions). In what follows, we supress the translation function $\mathcal{T}$, considering it implicit in all cases where it should apply.

Because Alloy run commands from which GenT tests originate are in FDNF, we know that each disjunct corresponds to a clause or minterm, that is, to a conjunction of literals. Some of these literals involve *op*, others do not, and the idea is to separate ones from the others in the distinguished disjunct $\alpha(R)$ and treat them differently in what follows. In order to simplify, and without loss of generality, we follow by exemplifying the case where the number $m$ of clauses or minterms of the given axiom is equal to 2, that is, $\alpha(t) = \alpha(R) \vee \alpha(R_1)$

We can now represent $\alpha(R)$ as a conjunction of two possible conjunctions – one that contains terms containing *op*, which we call $o$, and other with no terms containing *op*, which we call $n$, that is, $\alpha(t) = (o \wedge n) \vee \alpha(R_1)$

Remember test $t$ failed, thus all its disjuncts are necessarily false. Because $t$ is built over a set of objects that correspond to a model for run $R$, then: **(1)** if all the conjuncts in $o$ are false, while the ones in $n$ are true, we can strengthen the suspicion of guilt of the method implementing operation *op*; **(2)** if $o$ is true, the guilt should be sought for some other operation but *op*. For test $t$ to fail and $o$ to be true, $n$ must be false.

We must be able to find which is the case (1 or else 2). The following reasoning accomplishes the task: We build a new test, over the same objects, where we negate all the conjuncts of $o$, and run it; if it passes, then (1) is the case; We build yet another test, over the same objects, where we negate all the conjuncts of $n$, and run it; if it passes, then (2) is the case.

So, we build tests $t_o$ and $t_n$ from $t$ using the same model instance satisfying $R$, such that $\alpha(t_o) = (\eta o \wedge n) \vee \alpha(R_1)$, where $\eta o$ denotes the conjunction of the negation of the terms in $o$ (remember $o$ is a conjunction of terms involving operation *op*), and $\alpha(t_n) = (o \wedge \eta n) \vee \alpha(R_1)$, where $\eta n$ denotes the conjunction of the negation of the terms in $n$ (remember $n$ is a conjunction of terms not involving operation *op*).

Next we execute tests $t_o$ and $t_n$. If $t_o$ passes, we conclude that the original problem may be caused by the method implementing operation *op* (*strengthens* the guilt). If $t_n$ passes, we conclude that the original problem should not be caused by the method implementing operation *op* (*weakens* the guilt). The ideal, effective, results are the ones where one of the two tests succeeds and the other fails; any other paired results are inconclusive.

EXAMPLE This case exemplifies the situation where there is a fault in an already suspicious observer, which is the method `isEmpty`. As a result of the existing fault, some of the GenT generated tests fail; among them, the test for run axiomSortedSet4_2, that includes the suspect `isEmpty` observer:

```
run axiomSortedSet4_2  {
      some EVar : Orderable , SVar : SortedSet | (SVar.isEmptyOp != BOOLEAN/True
             and SVar.insertOp[EVar].largestOp != EVar)
} for 6 but exactly 4 SortedSet
```

and that corresponds to axiom 4:

```
fact axiomSortedSet4  {
      all EVar : Orderable , SVar : SortedSet | SVar.isEmptyOp = BOOLEAN/True
             implies SVar.insertOp[EVar].largestOp = EVar
}
```

The model used for this test defines that SVar contains two orderables, and EVar is the smallest of those two orderables. We pick the failed test (see figure 9), where axiom 4 is tested in its FDNF form), and from it we create two new tests, which are presented in figure 10, according to the technique presented in this section.

Notice that the changes were made to the second minterm of the disjunction (lines 12 and 26 in figure 10), which is the one that corresponds to the run that originated the failed test. In the $t_o$ test, we negated the literals that include the suspect observer (`isEmpty`). In $t_n$ we negated the literals that do not include `isEmpty`.

Test $t_o$ succeeds and test $t_n$ fails, so we strengthen the suspicions of guilt of `isEmpty`.

### D. Comparing abstract with concrete objects

The next step, according to figure 6, is the comparison of the results one obtains by invocation of the methods implementing the specification operations, to expected results, that is, to

```
private void axiomSortedSet4Tester(
      CoreVarFactory<TreeSet<OrderableMock>> sVarFac, OrderableMock eVar) {
   TreeSet<OrderableMock> sVar_0 = sVarFac.create();
   TreeSet<OrderableMock> sVar_1 = sVarFac.create();
   TreeSet<OrderableMock> sVar_2 = sVarFac.create();
   TreeSet<OrderableMock> sVar_3 = sVarFac.create();
   TreeSet<OrderableMock> sVar_4 = sVarFac.create();
   TreeSet<OrderableMock> sVar_5 = sVarFac.create();
   sVar_1.insert(eVar);
   sVar_3.insert(eVar);
   sVar_5.insert(eVar);
   assertTrue((sVar_0.isEmpty() && sVar_1.largest().equals(eVar))
      || (!sVar_2.isEmpty() && !sVar_3.largest().equals(eVar))
      || (!sVar_4.isEmpty() && sVar_5.largest().equals(eVar)));
}
```

Fig. 9. Test for run axiomSortedSet4_2.

```
1 private void axiomSortedSet4Tester0(
      CoreVarFactory<TreeSet<OrderableMock>> sVarFac, OrderableMock eVar) {
2        TreeSet<OrderableMock> sVar_0 = sVarFac.create();
3        TreeSet<OrderableMock> sVar_1 = sVarFac.create();
4        TreeSet<OrderableMock> sVar_2 = sVarFac.create();
5        TreeSet<OrderableMock> sVar_3 = sVarFac.create();
6        TreeSet<OrderableMock> sVar_4 = sVarFac.create();
7        TreeSet<OrderableMock> sVar_5 = sVarFac.create();
8        sVar_1.insert(eVar);
9        sVar_3.insert(eVar);
10       sVar_5.insert(eVar);
11       assertTrue((sVar_0.isEmpty() && sVar_1.largest().equals(eVar))
12             || (!(!sVar_2.isEmpty()) && !sVar_3.largest().equals(eVar))
13             || (!sVar_4.isEmpty() && sVar_5.largest().equals(eVar)));
14 }

15 private void axiomSortedSet4TesterN(
      CoreVarFactory<TreeSet<OrderableMock>> sVarFac, OrderableMock eVar) {
16       TreeSet<OrderableMock> sVar_0 = sVarFac.create();
17       TreeSet<OrderableMock> sVar_1 = sVarFac.create();
18       TreeSet<OrderableMock> sVar_2 = sVarFac.create();
19       TreeSet<OrderableMock> sVar_3 = sVarFac.create();
20       TreeSet<OrderableMock> sVar_4 = sVarFac.create();
21       TreeSet<OrderableMock> sVar_5 = sVarFac.create();
22       sVar_1.insert(eVar);
23       sVar_3.insert(eVar);
24       sVar_5.insert(eVar);
25       assertTrue((sVar_0.isEmpty() && sVar_1.largest().equals(eVar))
26             || (!sVar_2.isEmpty() && !(!sVar_3.largest().equals(eVar)))
27             || (!sVar_4.isEmpty() && sVar_5.largest().equals(eVar)));
28 }
```

Fig. 10. New tests for run axiomSortedSet4_2, according to the technique in section III-C.

ones as defined by the specification model. Unlike the three previous steps, this one does not work on the initial GenT tests; it nevertheless uses the Alloy specification created by GenT (the one that is equivalent to the original ConGu specification module).

We use the terms "abstract" and "concrete" to name the objects that behave according to the model, and objects that are instances of the classes implementing the specification, respectively.

In this step, mock Java classes whose instances will represent the abstract objects are automatically created; they must give their instances the ability to store and retrieve the results of applying all observers and non-creator constructors, as defined by a model. In the running example, we use the already presented `OrderableMock` class (recall figure 4) to obtain abstract `Orderable` objects, as well as a new `TreeSetMock` one, whose instances will be abstract `TreeSet` objects.

Then, a new model-based test is automatically generated that *(i)* creates as many instances of these mock classes as the number of objects of each type defined by the model, as well as corresponding concrete objects for the designated type, and *(ii)* verifies whether the results of invoking the implementing methods on the concrete objects are as expected (that is, correspond to the ones obtained by calling the corresponding methods over corresponding abstract objects).

THE ALGORITHM To summarize, the following algorithm describes the necessary steps to build the test class that compares concrete with abstract objects:

- Generate an instance of the Alloy model that satisfies all facts of the Alloy specification (use an empty run command in Alloy Analyzer);
- Generate mock classes (whose instances will be the "abstract objects") for the types defined by the specifications in the module;
- Generate a test class to compare the abstract with the concrete objects:
  - compose the abstract objects, as defined by the Alloy model:
    - as many abstract objects are created as there are instances of the specified types in the Alloy model;
    - they are created by instantiating the corresponding mock classes and by applying them the non-creator constructors defined by the shortest path in the model;
    - they are "fed" with the information about its behaviour, as defined by the Alloy model;
  - create the corresponding concrete objects:
    - concrete objects $concX_{ij}$ are created of the designated type – at least one for each abstract object $absX_i$ of the designated type –, by using the concrete class constructors corresponding to the ones used to define $absX_i$;
    - in order to avoid side effects, the number of $concX_{ij}$ concrete objects of the designated type that must be created for a given $i$ depends on the number of observer applications that must be executed in order to compare this concrete instance with the abstract $absX_i$ one;
  - compare the abstract with the concrete objects:
    - apply to the concrete objects all the observer methods (including the non-creator constructors) that should be possible to apply, as indicated by the Alloy model, while comparing (using `assertTrue` statements) the results with the corresponding mock methods' results when executed over corresponding abstract objects;
    - in order to be able to reason about the results of all these comparisons, each `assertTrue` invocation is enclosed in a **`try-catch`** block.

INTERPRETING THE RESULTS The interpretation of this test's results is based upon the following observations: (i)

whether several and varied observers fail or only one fails – this is important to decide whether to blame a constructor or a given, specific, observer; (ii) whether varied observers fail when applied to concrete objects created only by the constructor-creator, or when applied to objects that were also the target of non-creator constructors – this is important to decide which constructor is the faulty one.

The result interpretation algorithm inspects three data structures containing data collected during the execution of the test (whenever an `assertTrue` command fails):

- $L_1$ - Set of pairs $< obs; obj >$ that register that differences occurred between expected behaviour and actual behaviour, for given observer $obs$ and object $obj$;
- $L_2$ - Set of pairs $< ncc; obj >$ that register that differences occurred between expected behaviour and actual behaviour, for given non-creator constructor $ncc$ and object $obj$;
- $L_3$ - Set of pairs $< cc; n >$ that register for every creator constructor $cc$ the number of failed observations over objects uniquely built with $cc$;

**if** $(L_1 \cup L_2)$ contains pairs for more than 1 observer, **then**
  **if** there exists $< cc, i >$ in $L_3$ with $i > 0$, **then**
    **if** that pair $< cc, i >$ with $i > 0$ is unique, **then**
      $cc$ is guilty;
    **else**
      inconclusive;
    **endIf**
  **else**
    **for** each non-creator constructor $ncc_j$ **do**
      $L_{nccj} \leftarrow$ sub-set of $L_2$ containing only pairs from $L_2$ whose first element is $ncc_j$;
      Delete from $L_{nccj}$ the pairs whose $obj$ was not built using only $ncc_j$ and a creator constructor;
      **if** $L_{nccj}$ is not empty, **then**
        add $ncc_j$ to the final set of suspects (FSS);
      **endIf**
    **endFor**
  **endIf**
  **if** #FSS = 1 **then**
    the guilty is the sole element of FSS;
  **else**
    inconclusive;
  **endIf**
**else**
  **if** $(L_1 \cup L_2)$ is empty, **then**
    inconclusive;
  **else**
    the guilty is the sole observer in $(L_1 \cup L_2)$;
  **endIf**
**endIf**

### *E. Final list of suspects*

If the previous algorithm elected a guilty method in the end, then this method is identified as the most probable guilty.

In either case, a ranked list of (other) suspects is presented which is the list FSS above added with the suspects identified/strengthened by the previous three techniques, knowing that, whenever techniques in sections III-B and III-C strengthened (weakened) the suspicions of some operator $op$, $op$ ranking increased (decreased).

## IV. EVALUATION

To evaluate the effectiveness of our approach, we applied it to two case studies – this paper's SortedSet running example, and a MapChain specification module and corresponding implementations; the Java classes implementing the designated sorts of both case studies where seeded with faults covering all the specification operations.

We put the presented 5-step approach to work and registered the results.

We also tested those defective classes in the context of two existing fault-location tools – GZoltar [1], [15] and EzUnit4 [4], [16] –, that give as output a list of methods suspect of containing the fault, ranked by probability of being faulty. The tests suites we used were the ones generated by the GenT tool, exactly the same that were obtained in step 1 of our 5-step approach.

For each seeded fault we reported whether the faulty method was ranked, by each tool, as most probable guilty, second most probable guilty or third or less probable. A fourth type of result happened whenever the guilty method was not ranked as suspect at all.

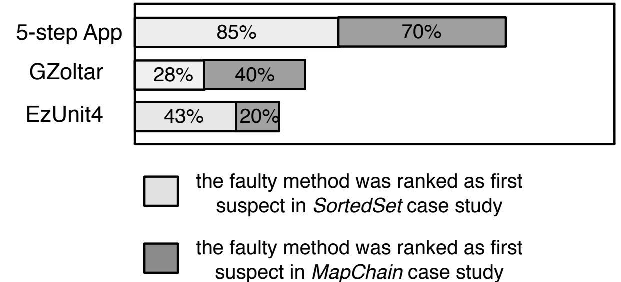


Fig. 11. Summary of the experiment. The bars measure the success of each approach in ranking the faulty method as first suspect.

Figure 11 shows very encouraging results for our approach, which leads us to continue working on it, trying to identify and ammeliorate negative aspects and weaknesses.

## V. RELATED WORK

Several approaches to testing implementations of algebraic specifications exist, that cover test generation, and many compare the two sides of equations where variables have been substituted by ground terms ([5], [6], [7], [9] to name a few); differences exist in the way ground terms are generated, and in the way comparisons are made. The gap between algebraic specifications and implementations makes the comparison between concrete objects difficult, giving rise to what is known as the *oracle problem*, more specifically, the search for

reliable decision procedures to compare results computed by the implementation. Whenever one cannot rely on the `equals` method, there should be another way to investigate equality between concrete objects. Several works have been proposed that deal with this problem, e.g. [6], [10], [17].

Concerning the fourth technique presented in this paper, in [13] we tackle this issue by presenting two alternative ways of comparing concrete objects, one that relies only in observers whose result is not of the designated type. In some way this complies with the notion of observable contexts in [6] – all observers but the ones whose result is of the designated type constitute observable contexts.

A critical issue w.r.t. our approach is the one concerned with private methods. Private methods are not identified as suspects by our approach because they do not correspond to any specification operation as defined by the refinement mapping from specifications to implementations; instead, the public, specified, methods that invoke them are identified. In our experiment, both GZoltar and EzUnit4 identified a faulty private method (in second and in seventh place of the rankings, respectively), while our approach ignored it and identified the public method that invoked it instead, as the first suspect.

## VI. Conclusions

We presented a 5-step approach for finding faults in Java implementations of algebraic specifications, that capitalyzes on GenT and Alloy Analyzer tools and applies four techniques aiming at the discovery of the method containing the fault.

The application of these techniques departs from a situation in which one or more tests (generated by the GenT tool) fail, and all methods are suspect; some particular failed tests are picked and the behaviour of the involved operations is studied in several different perspectives, as a way to unveil other facets of those operations and obtain aditional significant information.

Models of the specification are at the basis of these techniques, since they are the departing point for the generation of thorough tests, at several phases of the process, which are key to the main goal of the 5-step approach.

Our approach was compared with two other fault-location tools over two case studies where faults have been seeded in several implementing Java classes, and results were very encouraging.

Further work – tool building – focuses mainly in the last step of this approach, that is, in the comparison between abstract and concrete objects, due to the fact that its results alone showed great effectiveness and resources are limited. Several improvements are foreseen as, for example, testing the implementation of the `equals` method, even if the specification module does not specify it, in order to be able to better rely on its results.

## Acknowledgments

This work was partially supported by Fundação para a Ciência e Tecnologia under contract (PTDC/EIA-EIA/103103/2008).

## References

[1] R. Abreu, P. Zoeteweij, and A.J.C. van Gemund. Spectrum-based multiple fault localization. In *Proc. 24th IEEE/ACM ASE*, pages 88–99. IEEE Computer Society, 2009.

[2] F.R. Andrade, J.P. Faria, A. Lopes, and A.C.R. Paiva. Specification-driven test generation for Java generic classes. In *IFM 2012*, volume 7321 of *LNCS*, pages 296–311. Springer-Verlag, 2012.

[3] F.R. Andrade, J.P. Faria, and A. Paiva. Test generation from bounded algebraic specifications using Alloy. In *Proc. ICSOFT 2011*, volume 2, pages 192–200. SciTePress, 2011.

[4] P. Bouillon, J. Krinke, N. Meyer, and F. Steimann. EzUnit: A framework for associating failed unit tests with potential programming errors. In *8th XP*, volume 4536 of *LNCS*, pages 101–104. Springer, 2007.

[5] R.K. Doong and P.G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM TOSEM*, 3(2):101–130, April 1994.

[6] M.C. Gaudel and P.L. Gall. Testing data types implementations from algebraic specifications. *Formal Methods and Testing*, 2008.

[7] M. Hughes and D. Stotts. Daistish: systematic algebraic testing for OO programs in the presence of side-effects. In *Proc. ISSTA96, ACM Press*, pages 53–61, January 1996.

[8] D. Jackson. *Software Abstractions - Logic, Language, and Analysis, Revised Edition*. MIT Press, 2012.

[9] L. Kong, H. Zhu, and B. Zhou. Automated testing EJB components based on algebraic specifications. In *COMPSAC 2007*, pages 717–722, July 2007.

[10] P.D.L. Machado and D. Sanella. Unit testing for CASL architectural specifications. In *Proc. 27th MFCS*, volume 2420 of *LNCS*, pages 506–518. Springer, 2002.

[11] I. Nunes, A. Lopes, and V. Vasconcelos. Bridging the gap between algebraic specification and object-oriented generic programming. In *Runtime Verification*, volume 5779 of *LNCS*, pages 115–131. Springer, 2009.

[12] I. Nunes, A. Lopes, V. Vasconcelos, J. Abreu, and L. Reis. Checking the conformance of Java classes against algebraic specifications. In *Proc. 8th ICFEM*, volume 4260 of *LNCS*, pages 494–513. Springer, 2006.

[13] I. Nunes and F. Luís. A fault-location technique for Java implementations of algebraic specifications. Technical Report 02, Faculty of Sciences of the University of Lisbon, 2012. available at http://hdl.handle.net/10455/6809.

[14] L.S. Reis. ConGu v.1.50 users guide. 2007.

[15] A. Riboira and R. Abreu. The GZoltar project: A graphical debugger interface. In L. Bottaci and G. Fraser, editors, *TAIC PART*, volume 6303 of *LNCS*, pages 215–218. Springer, 2010.

[16] F. Steimann and M. Bertschler. A simple coverage-based locator for multiple faults. In *IEEE ICST*, volume 4536 of *LNCS*, pages 101–104. Springer, 2009.

[17] H. Zhu. A note on test oracles and semantics of algebraic specifications. In *QSIC 2003*, pages 91–99. IEEE Computer Society, 2003.