Testing Java implementations of algebraic specifications

Isabel Nunes
Faculty of Sciences, University of Lisbon
Lisboa, Portugal
in@di.fc.ul.pt

Filipe Luís
Faculty of Sciences, University of Lisbon
Lisboa, Portugal
fluis@di.fc.ul.pt

In this paper we focus on exploiting a specification and the structures that satisfy it, to obtain a means of comparing implemented and expected behaviours and find the origin of faults in implementations. We present an approach to the creation of tests that are based on those specification-compliant structures, and to the interpretation of those tests' results leading to the discovery of the method responsible for an eventual test failure. Results of comparative experiments with a tool implementing this approach are presented.

1 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, finding the software element containing a given fault is highly desirable and several approaches exist that tackle this issue, that can be quite different in the way they approach the problem.

ConGu [14, 15] is both an approach and a tool for the runtime verification of Java implementations of algebraic specifications. It verifies that implementations conform to specifications by monitoring method executions in order to find any violation of automatically generated pre and post-conditions.

The ConGu tool [7] picks a module of axiomatic specifications, together with a Java implementation and a refinement that maps specifications to Java types, and responds to an erroneous implementation by outputing the specification constraint that was violated; this is often insufficient to find the faulty method, because all methods involved in the violated constraint become equally suspect.

A ConGu companion tool – the GenT tool [3, 4] – generates JUnit test cases from ConGu specifications. Generating test cases that are known to be comprehensive, i.e. that cover all constraints of the specification, as GenT does, is a very important activity, because the confidence we may gain on the correction of the software we use greatly depends on it. But, in order for these tests to be of effective use, we should be able to use their results to localize the faulty components. Here again, executing the JUnit tests generated by GenT fails to give the programmer clear hints about the faulty method – all methods used in failed tests are suspect. The ideal result of a test suite execution would be the exact localization of the fault.

In this paper we enrich ConGu, by giving it the capability of locating the methods that are responsible for detected faults. We present a technique that builds upon structures satisfying the specification to obtain a means to observe the implemented behaviour against the intended one, and to locate faulty methods in implementations. Unlike several existing approaches, ours does not inspect the executed code; instead, it exploits the specification and conforming structures in order to be able to interpret some failures and discover their origin.

A tool was built – the Flasji tool – that implements the presented technique, and a comparative experiment was undertaken to evaluate its results. A summary of these results, which were very encouraging, is presented in this paper.

A. Petrenko, H. Schlingloff (Eds.): Eighth Workshop on
Model-Based Testing (MBT 2013)
EPTCS 111, 2013, pp. 35–50, doi:10.4204/EPTCS.111.4

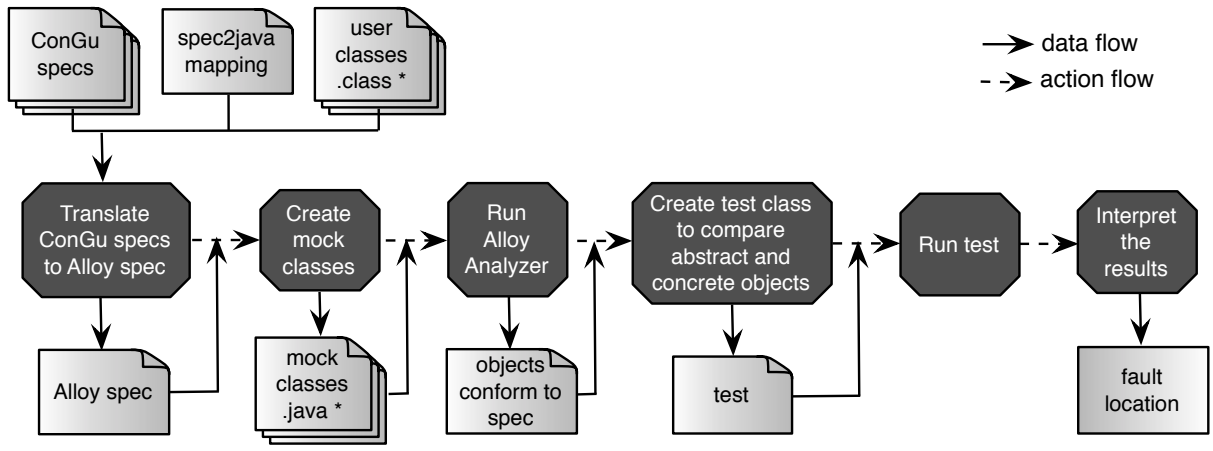

Figure 1: Overview of the Flasji approach.

The unit of fault Flasji is able to detect is the method, leaving to the programmer the task of identifying the exact instruction within it that is faulty. If more than one fault exists, the repeated application of the process, together with the correction of the identified faulty method, should be adopted. In what concerns integration testing strategy, Flasji applies 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 Java collection of classes that implements it.

The remainder of the paper is organized as follows: section 2 introduces the ConGu specification language through an example that will be used throughout the paper, and gives an overview of the Flasji approach; section 3 details every Flasji step, from picking a specification module and corresponding implementation, to the identification of the faulty method, explaining the several items Flasji produces; an evaluation experiment of the Flasji tool is presented in section 4 where results are compared with the ones obtained using two other tools; in section 5 we focus our discussion on relevant aspects related to the work presented in this paper; finally, section 6 concludes.

2 Approach Overview

In this section we give a general overview of the Flasji approach. An example is introduced that will be used throughout the paper.

2.1 The approach in a nutshell

As illustrated in figure 1, the Flasji approach integrates a series of steps from an initial input comprising a module of ConGu specifications and corresponding Java implementations (together with a mapping defining correspondences between the two), to a final output comprising the method identified as the one containing the fault, whenever possible, and a list of other methods suspect of being faulty. The

whole process leading from the initial input to the final output is automated, without any further user intervention.

The main strategy underlying the Flasji process is the comparison between what we call "abstract" and "concrete" objects; the former are objects that are well-behaved in the sense that they conform to the specification, while the latter are objects that behave according to the classes implementing the specification, which we want to investigate for faults.

We capitalize on the Alloy Analyzer [1] tool which is capable of finding structures that satisfy a collection of constraints – a specification. The specifications this tool works with are written in the Alloy [11] language.

Flasji begins by translating the ConGu specification module into an Alloy specification, in order to be able to, in a posterior phase, generate structures satisfying it. It then creates Java classes whose instances will represent objects satisfying the specification (the "abstract" objects) – these classes are called "mock" classes; in order for "abstract" objects to represent structures that satisfy the specification, they are given the ability of storing and retrieving the results of applying each and every operation of the specification, as will be seen later.

A third step feeds the Alloy Analyzer tool with the specification, asking the tool for a collection of structures satisfying the specification. This collection will be used in the next step to define the abstract objects, which will present the expected, correct, behaviours.

In a fourth step, a test class is created that contains instructions to instantiate both the mock classes and the implementation classes given as input, and to compare the behaviour of the "concrete" objects against the "abstract" ones, in order to identify the faulty method. Flasji then executes this test class and interprets its results to obtain the faulty method.

Remember that all these steps are automatically processed, thus transparent to the Flasji user. Section 3 describes them in detail.

2.2 A specification and corresponding implementation

Simple sorts, sub-sorts and parameterized sorts can be specified with the ConGu specification language, and mappings between those sorts and Java types, and between those sorts' operations and Java methods, can be defined using the ConGu refinement language.

We present a classical yet rich example, of a ConGu specification of the SortedSet parameterized data type, representing a set of Orderable elements, together with the specification for its parameter (figure 2).

In a specification, we define *constructors*, *observers* and other operations, where constructors compose the minimal set of operations that allow to build all instances of the sort, 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 sort 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. Any function 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 – see, e.g., the axiom isEmpty(empty()); that specifies that the result of applying the observer operation isEmpty to a SortedSet instance obtained with the creator constructor empty is true, and the axiom **not** isEmpty(insert(S, E)); saying that the result of applying isEmpty to any SortedSet instance to which the transformer constructor

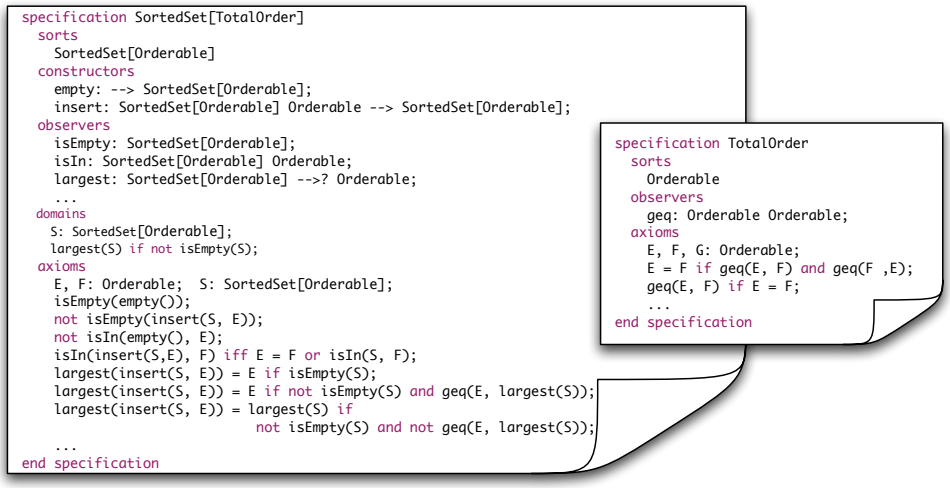

Figure 2: Parts of the ConGu specifications for the SortedSet parameterized data type and its parameter.

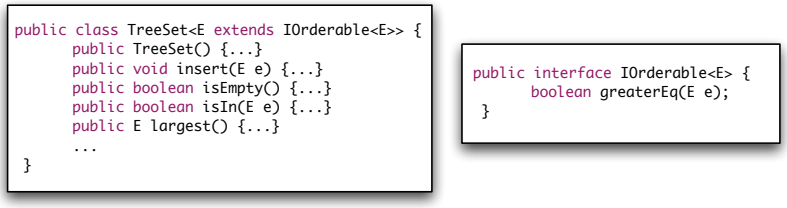

Figure 3: Excerpt from a Java implementation of the ConGu specification for SortedSet.

insert has been applied is false.

Generic Java class `TreeSet` in figure 3 represents a Java implementation of the SortedSet parameterized data type; in the same figure, interface `IOrderable` represents a Java type restraining the `TreeSet` parameter. We want to investigate the `TreeSet` class for faults, independent of any specific implementation of its parameter type.

The correspondence between ConGu and Java types must be defined in order for implementations to be checked. This correspondence is described in terms of *refinement mappings*; figure 4 shows a refinement mapping from the specifications SortedSet and TotalOrder (figure 2) to Java types `TreeSet` and `IOrderable` (figure 3).

These mappings associate ConGu sorts and operations to Java types and corresponding methods. The insert operation of sort SortedSet is mapped to the `TreeSet` class method with the same name with the signature **`void`** `insert(E e)`. Notice that the TotalOrder parameter sort is mapped to a Java type variable that is used as the parameter of the generic `TreeSet` implementation; this specific mapping is interpreted as constraining any instantiation of the `TreeSet` parameter to a Java type `Some` containing a method with signature **`boolean`** `greaterEq(Some e)`.

Detailed information about the ConGu approach can be found in [14, 15].

```
refinement <E>
  SortedSet[TotalOrder] is TreeSet<E> {
     empty: --> SortedSet[Orderable] is TreeSet();
     insert: SortedSet[Orderable] e:Orderable --> SortedSet[Orderable] is void insert(E e);
     ...
  }
  TotalOrder is E {
     geq: Orderable e:Orderable is boolean greaterEq(E e);
  }
end refinement
```

Figure 4: Refinement mapping from ConGu specifications to Java types.

3 Flasji step-by-step

As already said in the previous section, the main goal of the Flasji approach is to verify whether "concrete" Java objects behave the same as corresponding "abstract" ones; the deviations to the expected behaviour are interpreted in order to find the location of the faulty method.

Only one of the implementing classes is under verification – the *core* type, that is, the one that implements the *core* sort –, which is the one at the root of the class association graph (the `TreeSet` class in the example). Thus, both "abstract" and "concrete" objects will be created for this type, in order to compare behaviours. This does not apply for non-core types since they are not under verification. However, as we shall see, "abstract" parameter objects must be created.

Let us now detail the several steps of the Flasji approach.

3.1 Translating the ConGu specification module

Flasji creates an Alloy specification equivalent to the ConGu specification module, in order to be able, ahead in the process, to obtain a collection of objects that conform to the specification, thus defining expected, correct behaviour.

This step capitalizes on already existing work, referred to in the introduction of this paper, namely the GenT tool [3, 4], of which Flasji uses the ConGuToAlloy module.

3.2 Creating mock classes

Flasji creates mock classes for the Java types implementing the specification core and parameter sorts; these classes' instances will represent the "abstract" objects. In the running example, two mock classes must be created, one corresponding to the `TreeSet` class – `TreeSetMock` –, and other corresponding to the `IOrderable` interface – `OrderableMock`.

This mock class will be used to generate parameter objects that will be inserted not only in "abstract" sorted sets (`TreeSetMock` instances as explained below), but also in "concrete" ones (`TreeSet` instances); the idea, as said before, is to test the implementation of the core signature for any parameter instantiation that correctly implements the Orderable sort.

Each instance of a mock class defines an object conforming to the specification, including its "behaviour", that is, the results of applying to it all the operations of the type (respecting the corresponding domain conditions). Since we only compare "abstract" and "concrete" objects of the core type, fundamental differences exist between the mock class for this core type and the others. Let us see first the mock class for the Orderable parameter of the running example, which is a non-core type.

```
1 public class OrderableMock implements IOrderable⟨OrderableMock⟩{
2   private HashMap⟨OrderableMock, Boolean⟩ greaterEqResult = new HashMap⟨OrderableMock,
        Boolean⟩();
3   public boolean greaterEq(OrderableMock e) {
4     return greaterEqResult.get(e);}
5   public void add_greaterEq(OrderableMock e, Boolean result) {
6     greaterEqResult.put(e, result);
7   }
8 }
```

Listing 1: Mock class corresponding to the Orderable parameter sort.

For each method `X` corresponding to a specification operation X, an attribute is defined to keep the information about the results of `X`, for every combination of the method's parameters (see line 2 for the method `greaterEq` in interface `Iorderable`, corresponding to operation geq in sort Orderable); the `add_X` method "fills" that attribute (lines 5 and 6), and the `X` method retrieves the result for given values of the method's parameters (lines 3 and 4).

The class that represents "abstract" objects of the core type (class `TreeSetMock` in the example) is also generated. The idea here is not to use these "abstract" objects on both abstract and concrete contexts, as we do with parameter ones, but to use them to inform us, for every operation, of the results we should expect when applying corresponding methods to corresponding "concrete" objects in order to verify whether the latter behave as they should.

The fundamental difference lies in the information that core type "abstract" objects keep for operations whose result is of the core type (insert in the example). Since we want to be able to know whether the "concrete" `TreeSet` object that results from applying the `insert` method of the `TreeSet` class to a "concrete" object `concObj` is the correct one, we give the corresponding "abstract" object `absObj` information that allows us to verify it – we "feed" `absObj` with the "concrete" object that should be expected when applying that operation. Ahead in this paper we show how this is achieved; for now, we just present the `TreeSetMock` mock class in listing 2, where attribute and methods in lines 19 to 24 allow "abstract" objects to keep and inform about "concrete", expected results of applying `insert` for different values of the method's parameter:

```
1 public class TreeSetMock ⟨T⟩{
2
3   // operations whose result is of a non-core type
4   private HashMap⟨T,Boolean⟩ isInResult = new HashMap⟨T,Boolean⟩();
5   private boolean isEmptyResult;
6   private T largestResult;
7
8   public boolean isIn (T e){return isInResult.get(e);}
9   public void add_isIn (T e, Boolean result){
10    isinResult.put(e, result);}
11  public boolean isEmpty(){return isEmptyResult;}
12  public void add_isEmpty(boolean result){
13    isEmptyResult = result;}
14  public T largest(){return largestResult;}
15  public void add_largest(T result){
16    largestResult = result;}
17
18  // operation whose result is of the core type
19  private HashMap⟨T,TreeSet⟨T⟩⟩ insertResult = new HashMap⟨T,TreeSet⟨T⟩⟩ ();
20
21  public TreeSet⟨T⟩ insert (T e){
22    return insertResult.get(e); }
23  public void add_insert (T e, TreeSet⟨T⟩ concVal){
24      insertResult.put(e, concVal); }
```

```
25 }
```

Listing 2: Mock class corresponding to the SortedSet sort.

Line 19 declares and initializes the attribute that will store the information about the results of method `insert` – for each value of the parameter `T e`, it will store a "concrete" object. Methods `insert` and `add_insert`, in lines 21 to 24, allow to retrieve and define, respectively, the result of `insert` for every value of the operation's parameter.

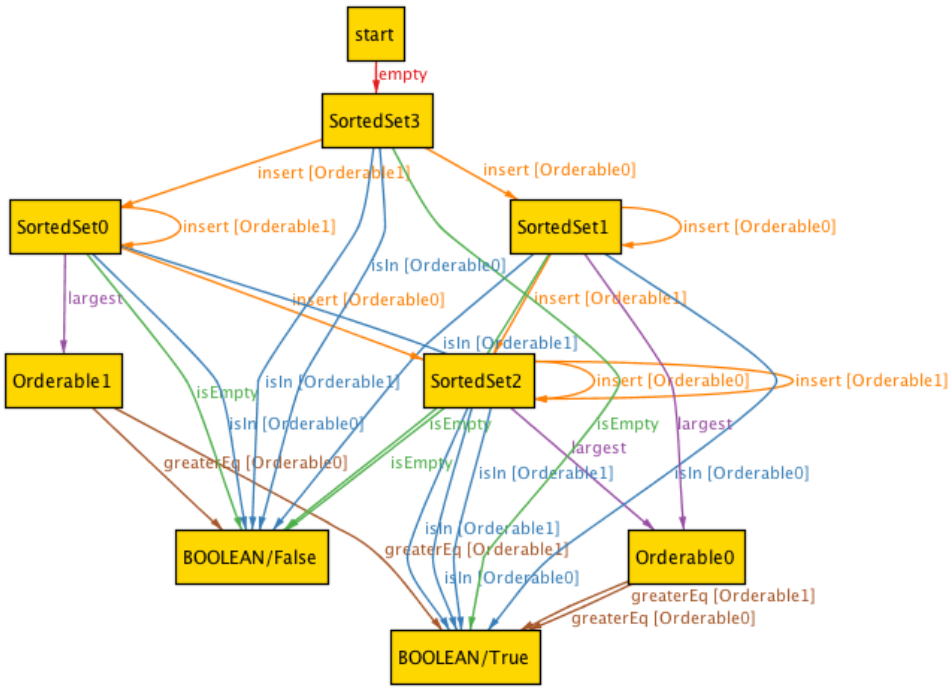

Figure 5: A structure satisfying the SortedSet specification module, found by the Alloy Analyzer.

3.3 Obtaining a collection of instances satisfying the specification

In order to obtain a collection of instances of the specification sorts that conform to our specification, Flasji capitalizes on the Alloy Analyzer tool [11] which, if such a finite collection exists, is capable of generating it (such a finite collection does not exist e.g. in the case of a specification of an unbounded stack).

The results presented in this paper assume Flasji asks the Alloy Analyzer to generate a structure with a fixed number of objects of the core type; work is under way to optimize the determination of the number of objects that should be considered of each type.

Figure 5 shows an example for the SortedSet specification. This collection consists of 2 Orderable instances and 4 SortedSet ones. The result of every applicable operation is defined for each of these instances (e.g. SortedSet2 contains the two Orderables, and it is obtained by inserting Orderable0 into SortedSet0 or else from inserting Orderable1 into SortedSet1).

These instances define correct, expected behaviour; in the next steps, Flasji will use the non-core sort ones (Orderable instances in the example) to create mock "abstract" objects that will represent well-

behaved objects, and uses the core sort ones (SortedSet instances in the example) to create mock "abstract" objects (`TreeSetMock` and "concrete" corresponding ones (`TreeSet` instances in the example) instances in the example) that will be compared. Let us see how.

3.4 Creating the test class

Flasji generates a test class containing instructions to:

1. create "abstract" objects corresponding to the objects composing the Alloy structure that conforms to the specification;
2. create "concrete" objects of the core type that correspond to the "abstract" ones (by using the corresponding concrete constructors); and
3. compare the behaviour of these "abstract" and "concrete" objects, by observing them in equal circumstances, that is, by applying corresponding methods and comparing the results.

By compiling and executing this test class, Flasji will be able to get information that it will interpret in order to find the faulty method, as explained ahead. First let us see how Flasji acomplishes this test class creation task.

3.4.1 Creating abstract objects

Listing 3 shows part of the generated test class, namely the creation of the "abstract" objects according to the Alloy structure defined in figure 5: two `OrderableMock` instances (lines 4 and 5) and four `TreeSetMock` ones (lines 11 to 14).

Lines 6 to 9 show `OrderableMock` objects being initialized – because the only method of this type is `greaterEq`, only the method `add_greaterEq` is invoked over each "abstract" object, for every possible value of its parameter, in order to give these objects the information about the expected, correct, results.

We postpone the initialization of the `TreeSetMock` objects because it implies previous creation of the corresponding concrete objects.

```
1  @Test
2  public void abstractVSconcreteTest () {
3    //IOrderable Mocks
4    OrderableMock orderable0 = new OrderableMock();
5    OrderableMock orderable1 = new OrderableMock();
6    orderable0.add_greaterEq(orderable0, true);
7    orderable0.add_greaterEq(orderable1, true);
8    orderable1.add_greaterEq(orderable0, false);
9    orderable1.add_greaterEq(orderable1, true);
10   //Abstract objects TreeSet
11   TreeSetMock ⟨OrderableMock⟩ sortedSet3 = new TreeSetMock ⟨OrderableMock⟩();
12   TreeSetMock ⟨OrderableMock⟩ sortedSet0 = new ...;
13   TreeSetMock ⟨OrderableMock⟩ sortedSet1 = new ...;
14   TreeSetMock ⟨OrderableMock⟩ sortedSet2 = new ...;
```

Listing 3: (Part of) the test class – building the "abstract" objects (incomplete).

3.4.2 Creating concrete objects

Flasji also builds "concrete" objects for the core sort. For each "abstract" object of the core sort there will exist a corresponding "concrete" one, which will be built using the corresponding constructor methods (see listing 4).

For example, according to the structure in figure 5, the sortedSet0 instance of sort SortedSet can be obtained by application of the creator constructor empty followed by application of the transformer constructor insert with parameter orderable1; complying with this (see lines 7 and 8), we build the corresponding "concrete" object `concSortedSet0` using the java constructor `TreeSet⟨OrderableMock⟩()`, which corresponds to the creator constructor empty, and apply to it the method `insert`, which corresponds to the transformer constructor insert, with parameter `orderable1`. Whenever there are several ways to build an object, the shortest path is chosen.

```
1 @Test
2 public void abstractVSconcreteTest () {
3   ...
4   //Create concrete objects TreeSet
5   TreeSet⟨OrderableMock⟩ concSortedSet0 = new TreeSet ⟨OrderableMock⟩();
6   concSortedSet0.insert(orderable1);
7   TreeSet⟨OrderableMock⟩ concSortedSet0_1 = new ...;
8   concSortedSet0_1.insert(orderable1);
9   ...
10  TreeSet⟨OrderableMock⟩ concSortedSet0_5 = new ...;
11  concSortedSet0_5.insert(orderable1);
12  // three more to go (concSortedSet3, 1 and 2)...
```

Listing 4: Continuing... building the "concrete" objects (incomplete).

Notice that, since methods will be applied to these "concrete" objects in order to verify their behaviour, as many copies of a given "concrete" object are created as methods applied to it, in order to cope with undesired side effects.

3.4.3 Back to abstract objects

Now that "concrete" objects are already created, we can initialize the `TreeSetMock` objects:

```
1 @Test
2 public void abstractVSconcreteTest () {
3   ...
4   //Initializing sortedSet abstract objects
5   sortedSet0.add_isEmpty(false);
6   sortedSet0.add_largest(orderable1);
7   sortedSet0.add_isIn(orderable0, false);
8   sortedSet0.add_isIn(orderable1, true);
9   sortedSet0.add_insert(orderable0, concSortedSet2);
10  sortedSet0.add_insert(orderable1, concSortedSet0);
11 // three more to go (sortedSet3, 1 and 2)...
```

Listing 5: Continuing... initializing `TreeSetMock` objects.

Lines 5 to 8 "feed" the `sortedSet0` object with the information that it represents a sorted set that is not empty, whose largest element is the `orderable1` object, and that it contains `orderable1` but not `orderable0`, as would be expected by inspection of the structure in figure 5. In the case of object `sortedSet3`, which is empty as can be seen in figure 5, the instruction invoking the method `add_largest()` over it would not be generated, since the operation `largest` is undefined for that object.

These informations will be used later on to obtain the values that are expected to be the results of the corresponding methods when applied to the "concrete" object corresponding to `sortedSet0`, which, by construction, is `concSortedSet0` (or any of its copies).

Line 9 "feeds" `sortedSet0` with the information about which "concrete" object should be expected after inserting `orderable0` in the "concrete" object that corresponds to `sortedSet0` (`concSortedSet0`

or any of its copies) – the expected result is `concSortedSet2`. In the same way, in line 10, the expected result of inserting `orderable1` in the "concrete" object that corresponds to `sortedSet0` is defined to be itself.

3.4.4 Comparing abstract and concrete objects

In a next step, Flasji generates instructions in the test class that invoke all possible operations over the "abstract" and "concrete" objects and compare the results:

```
1  @Test
2  public void abstractVSconcreteTest () {
3    ...
4    //Compare concrete with corresponding abstract
5    assertTrue(concSortedSet0_1.isEmpty() == sortedSet0.isEmpty());
6    assertTrue(concSortedSet0_2.largest() == sortedSet0.largest());
7    assertTrue(concSortedSet0_3.isIn(orderable1) == sortedSet0.isIn(orderable1));
8    assertTrue(concSortedSet0_4.isIn(orderable0) == sortedSet0.isIn(orderable0));
9    concSortedSet0_5.insert (orderable1);
10   assertTrue(concSortedSet0_5.equals(sortedSet0.insert (orderable1)));
11   //three more to go (concSortedSet3, 1 and 2)...
```

Listing 6: Continuing... comparing "abstract" and "concrete" objects (incomplete).

The JUnit method `assertTrue` is used to generate an `AssertionError` exception whenever the behaviour of the "concrete" objects is not as expected, that is, whenever the results of methods invoked over "concrete" objects are different from the ones indicated by their "abstract" counterparts.

Lines 5 to 8 show the comparison between the `sortedSet0` "abstract" object and its "concrete" counterpart `concSortedSet0` using each `TreeSet` method whose result type is not `TreeSet` nor **`void`**. Since all these results are of primitive types or of the parameter type `OrderableMock`, Flasji uses == to compare between `sortedSet0` and `concSortedSet0` results.

Lines 9 and 10 show the comparison between "abstract" and "concrete" objects using (the only) operation with a core result type – `insert`. As already referred, to verify whether a given operation whose result is of the core type is well implemented, we compare the "concrete" object the method returns, with the "concrete" object that it should return. Since `insert` is implemented with a **`void`** result type, we must first invoke the method using the "concrete" object as a target, and then we compare (using `equals`) its new state with the "concrete" object that, according to the "abstract" corresponding object, should be the correct result.

Since the ultimate goal of this test class is to find the method containing the fault, it should be possible to reason about the results of all these comparisons, so we must be able to test all the `assert` commands. Although we do not show it in this paper due to space limitations, enclosing each `assertTrue` invocation in a **`try`**–**`catch`** block that catches `AssertionError` exceptions, allows to collect all results which will help composing a final test diagnosis.

A final note before continuing: whenever the module of Congu specifications input to Flasji includes more than one non-parameter type, e.g., the case where the input includes a core sort C and one non-core, non-parameter sort N, the class implementing C is verified for faults considering that the class implementing N is correct. No mock class is built for N, hence no "abstract" `N` objects are created; only "concrete" `N` objects are. For methods that return `N` type results, "abstract" `C` objects are "fed" with the information about which `N` "concrete" object should be the expected result. Thus, comparison between actual and expected results relating these methods are achieved using `equals`. The running SortedSet example does not cover this kind of situation.

3.5 Running the test and interpreting the results

As soon as the test class is generated, Flasji compiles it and executes it. Then, it interprets the results of the tests. The interpretation is based upon the following observations:

1. whether several and varied observers (non-constructor operations) fail or only one fails – this is important to decide whether to blame a constructor or a given, specific, observer;
2. 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 and actual behaviour, for given observer *obs* and object obj;
- L_2 - Set of pairs $< ncc; obj >$ that register that differences occurred between expected and actual behaviour, for given non-creator (transformer) 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, when applied to concrete objects, more than one observer methods present results that are different from the ones expected ($(L_1 \cup L_2)$ contains pairs for more than one observer), we may infer that the method(s) used to build those concrete objects are ill-implemented, and that the problem does not come from some particular way of inspecting the objects. If the implementation of a given observer is wrong, one would not expect problems when inspecting the objects using the other observers, but only in the observations involving that particular one.

If a constructor-creator *cc* (in the running example, `TreeSet()` is the *cc* that implements the empty creator operation) is faulty, it is reasonable to think that the application of the other constructors over an object created with *cc* will most probably result in non-conformant objects, because the initial object is already ill-built. The information in L_3 allows us to focus on creator-constructors.

When no problems arise when observing a freshly created object, but they do arise when observing those objects after being affected by a given non-creator constructor *ncc* (`insert` in the running example), then one may point the finger to *ncc*.

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 (L1 ∪ L2) is empty, then
      inconclusive;
    else
      the guilty is the sole observer in (L1 ∪ L2);
    endIf
  endIf
```

If the algorithm elects a guilty method in the end, then the user is given the identified method as the most probable guilty. In either case, the set FSS of (other) suspects is presented.

4 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 Flasji to run for every defective class, and registered the outputs.

We also tested those defective classes in the context of two existing fault-location tools – GZoltar [2, 17] and EzUnit4 [5, 18] –, 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 generated by the GenT [3, 4] tool (already referred to in this paper), from the ConGu specifications and refinement mappings; under given restrictions (e.g., the specification has finite models) GenT generates comprehensive test suites, that cover all specification axioms. GenT generated 20 test cases for the SortedSet case, and 17 for the MapChain one.

Finally we compared the three tools' results for every defective variation of each case study.

For each of the defective versions of the designated sorts implementations (for example, two different faults were seeded in SortedSet `isEmpty` method, three in MapChain `get` method, etc) table 1 shows:

- the number of tests (among the 20 JUnit tests that were generated by GenT for the SortedSet case, and 17 for the MapChain one) that failed when both GZoltar and EzUnit4 run them;
- whether the faulty method was ranked, by each tool, as most probable guilty (1_{st}), second most probable guilty (2_{nd}) or third or less probable (n_{th}). A fourth type of result – "No" – means the guilty method has not been ranked as suspect at all.

Flasji provided very accurate results in general (see also a summary in figure 6). The bad results in the three faults for method `get` of the MapChain case study (there were no suspects found whatsoever) are due to the fact that `equals` uses the `get` method, therefore becoming unreliable whenever method `get` is faulty. This case exemplifies the *oracle problem* (see section 5).

	Faulty method	failed tests	Faulty method ranked: Flasji	EzUnit4	GZoltar
SortedSet	isEmpty	5	1_{st}	n_{th}	2_{nd}
	isEmpty	1	1_{st}	1_{st}	2_{nd}
	isIn	1	1_{st}	n_{th}	1_{st}
	largest	7	1_{st}	1_{st}	1_{st}
	largest	1	1_{st}	1_{st}	2_{nd}
	private insert	2	No	n_{th}	2_{nd}
	public insert	5	1_{st}	n_{th}	2_{nd}
MapChain	get	4	No	n_{th}	1_{st}
	get	3	No	2_{nd}	1_{st}
	get	4	No	n_{th}	n_{th}
	isEmpty	2	1_{st}	2_{nd}	1_{st}
	isEmpty	1	1_{st}	n_{th}	1_{st}
	put	0	1_{st}	No	No
	put	1	1_{st}	1_{st}	2_{nd}
	put	2	1_{st}	1_{st}	2_{nd}
	remove	1	1_{st}	2_{nd}	2_{nd}
	remove	1	1_{st}	2_{nd}	2_{nd}

Table 1: Results of comparative experiments. "1_{st}", "2_{nd}" and "n_{th}" stand for first, second and third or worse, respectively. "No" means the faulty method has not been ranked as suspect.

Applying an alternative method of observation (see [16]) – one where the `equals` method is not used and, instead, only the outcomes of observers whose result is not of the core sort are used in comparisons – we obtain the right results for this case, i.e. `get` is ranked as prime suspect. However, the good results we had for the 3rd faulty `put` method and the 1st `remove` got worse – they are ranked second instead of first. These particular cases indicated `isEmpty` as prime suspect because the seeded fault of both those methods was the absence of change in the number of elements in the map whenever insertion/removal happens, which made `isEmpty` fail.

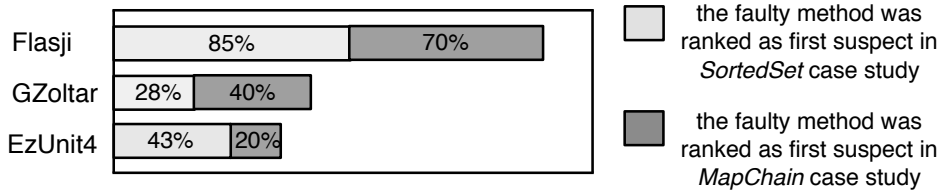

Figure 6: Summary of the evaluation experiment. The bars measure the success of each approach in ranking the faulty method as first suspect.

Another critical issue w.r.t. our approach is the one concerned with private methods. The fault in *private* method `insert` of the SortedSet case study, caused Flasji to rank the *public* `insert` method, instead, as the most probable suspect (in the particular implementation used, the public `insert` method is composed of one only statement which invokes the private `insert` method). As expected, private methods are not identified as suspects by Flasji because they do not directly refine any specification

operation (as defined in the refinement mapping from specifications to implementations); instead, the public, specified, methods that invoke them are identified.

A case worth mentioning is the one corresponding to the first seeded fault in the `put` method of the MapChain case study, where none of the seventeen GenT tests fail (the particular case that causes the error was not covered). As a consequence, neither EzUnit4 and GZoltar detected the fault; on the contrary, Flasji succeeded in detecting the guilty method.

5 Related work

The approach presented in this paper relies on the existence of structures satisfying the specification to supply the behaviour of objects to be used in tests. The structured nature of specifications, where functions and axioms are defined sort by sort, and where the latter are independently implemented by given Java types, is essential to the incremental integration style of Flasji.

Several approaches to testing implementations of algebraic specifications exist, that cover test generation ([6, 8, 9, 10, 12] to name a few), and many compare the two sides of equations where variables have been substituted by ground terms – 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. [9, 13, 19]. In [16] we tackle this issue by presenting an alternative way of comparing concrete objects, one that relies only in observers whose result is of a non-core sort. In some way this complies with the notion of observable contexts in [9] – all observers but the ones whose result is of the designated sort constitute observable contexts.

The unreliability of `equals` can also affect the effectiveness of the GenT tests [3] since this method is used whenever concrete objects of the same type are compared. One of the improvements we intend to make is to give Flasji the ability to test the `equals` method in order to make its use more reliable.

6 Conclusions

We presented Flasji, a technique whose goal is to test Java implementations of algebraic specifications and find the method that is responsible for some deviation of the expected behaviour.

Flasji capitalyzes on ConGu, namely using ConGu specification and refinement languages, and enriches it with the capability of finding faulty methods. It accomplishes the task through the generation of tests that are based on structures satisfying the specification. The behaviour of instances of the implementation is compared with the one expected, as given by those specification-compliant structures. The results of the comparisons are interpreted in order to find the method responsible for the fault.

An evaluation experiment was presented where Flasji results over two case studies, for which faults have been seeded in the implementing Java classes, are compared with two other tools' results when executed over comprehensive suites of tests. The encouraging results obtained in comparative studies led us to continue working on it, with the purpose of improving some negative aspects and weaknesses, some of which already identified and reported in this paper.

The following improvements, among others, are planned: (i) 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, (ii) optimizing the determination of the number of objects of each type that an Alloy structure conforming to the specification should contain (the results here presented assumed Flasji asks the Alloy Analyzer to generate a structure with a fixed number of objects of the core type), and (iii) whenever there are several non-parameter types, apply the process several times, each considering one of them as the core type, and integrate the results (special cases as e.g. inter-dependent types, deserve attention).

References

[1] *Alloy Analyzer*. `http://alloy.mit.edu/alloy/`.

[2] R. Abreu, P. Zoeteweij & A.J.C. van Gemund (2009): *Spectrum-based Multiple Fault Localization*. In: *Proc. 24th IEEE/ACM ASE*, IEEE Computer Society, pp. 88–99, doi:10.1109/ASE.2009.25.

[3] F.R. Andrade, J.P. Faria, A. Lopes & A.C.R. Paiva (2012): *Specification-Driven Unit Test Generation for Java Generic Classes*. In: *IFM 2012*, *LNCS* 7321, Springer-Verlag, pp. 296–311, doi:10.1007/978-3-642-30729-4.

[4] F.R. Andrade, J.P. Faria & A. Paiva (2011): *Test Generation from Bounded Algebraic Specifications using Alloy*. In: *Proc. ICSOFT 2011*, 2, SciTePress, pp. 192–200.

[5] P. Bouillon, J. Krinke, N. Meyer & F. Steimann (2007): *EzUnit: A Framework for associating failed unit tests with potential programming errors*. In: *8th XP*, *LNCS* 4536, Springer, pp. 101–104, doi:10.1007/978-3-540-73101-6_14.

[6] K. Claessen & J. Hughes (2000): *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*. In: *Proc. of ICFP*, ACM SIGPLAN Notices, pp. 268–279, doi:10.1145/351240.351266.

[7] P. Crispim, A. Lopes & V. Vasconcelos (2011): *Runtime verification for generic classes with ConGu2*. In: *Proc. SBMF 2010*, *LNCS* 6527, Springer-Verlag, pp. 33–48, doi:10.1007/978-3-642-19829-8_3.

[8] R.K. Doong & P.G. Frankl (1994): *The ASTOOT Approach to Testing Object-Oriented Programs*. *ACM TOSEM* 3(2), pp. 101–130, doi:10.1145/192218.192221.

[9] M.C. Gaudel & P.L. Gall (2008): *Testing data types implementations from algebraic specifications*. *Formal Methods and Testing*, doi:10.1007/978-3-540-78917-8_7.

[10] M. Hughes & D. Stotts (1996): *Daistish: systematic algebraic testing for OO programs in the presence of side-effects*. In: *Proc. ISSTA96, ACM Press*, pp. 53–61, doi:10.1145/229000.226301.

[11] D. Jackson (2012): *Software Abstractions - Logic, Language, and Analysis, Revised Edition*. MIT Press.

[12] L. Kong, H. Zhu & B. Zhou (2007): *Automated Testing EJB Components Based on Algebraic Specifications*. In: *COMPSAC 2007*, pp. 717–722, doi:10.1109/COMPSAC.2007.82.

[13] P.D.L. Machado & D. Sanella (2002): *Unit testing for CASL architectural specifications*. In: *Proc. 27th MFCS*, *LNCS* 2420, Springer, pp. 506–518, doi:10.1007/3-540-45687-2_42.

[14] I. Nunes, A. Lopes & V. Vasconcelos (2009): *Bridging the Gap between Algebraic Specification and Object-Oriented Generic Programming*. In: *Runtime Verification*, *LNCS* 5779, Springer, pp. 115–131, doi:10.1007/978-3-642-04694-0_9.

[15] I. Nunes, A. Lopes, V. Vasconcelos, J. Abreu & L. Reis (2006): *Checking the conformance of Java classes against algebraic specifications*. In: *Proc. 8th ICFEM*, *LNCS* 4260, Springer, pp. 494–513, doi:10.1007/11901433_27.

[16] I. Nunes & F. Luís (2012): *A fault-location technique for Java implementations of algebraic specifications*. Technical Report 02, Faculty of Sciences of the University of Lisbon.

[17] A. Riboira & R. Abreu (2010): *The GZoltar Project: A Graphical Debugger Interface*. In L. Bottaci & G. Fraser, editors: *TAIC PART*, *LNCS* 6303, Springer, pp. 215–218. Available at `http://dx.doi.org/10.1007/978-3-642-15585-7_25`.

[18] F. Steimann & M. Bertschler (2009): *A simple coverage-based locator for multiple faults*. In: *IEEE ICST*, *LNCS* 4536, Springer, pp. 366–375, doi:10.1109/ICST.2009.24.

[19] H. Zhu (2003): *A note on test oracles and semantics of algebraic specifications*. In: *QSIC 2003*, IEEE Computer Society, pp. 91–98, doi:10.1109/QSIC.2003.1319090.