Locating faults in Java implementations of algebraic specifications

Isabel Nunes and Filipe Luís

Faculty of Sciences, University of Lisbon
Lisboa, Portugal

Abstract. One of the most expensive and time consuming tasks of the software construction process is the location of the fault responsible for a runtime error, thereby any technique that makes it easier for the programmer to locate the faulty parts of her software is highly desirable. In this paper we focus on finding faults in object-oriented implementations of algebraic specifications, and present Flasji, a model-based approach that exploits the specification and corresponding models in order to interpret failures in implementations and discover their origin. Flasji builds a comprehensive JUnit test that compares runtime outcomes of operation implementations with expected ones, as specified by the model, and interprets the results in order to identify the method to blame. Results of comparative experiments 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, 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 the ConGu approach [16, 17] to the runtime verification of Java implementations of algebraic specifications, by giving it the capability of locating the methods that are responsible for detected faults. ConGu picks a module of axiomatic specifications, together with a Java implementation and a mapping refining specification sorts and operations to Java types and methods, and verifies that the implementation conforms to the specification. At present, the ConGu tool [5, 19] responds to an erroneous implementation by outputing the axiom that was violated; this is often insufficient to find the faulty method, because all involved methods become equally suspect. The ideal result would obviously be the exact localization of the fault, however this has proven a difficult task.

Several methods have been proposed in the literature to the diagnosis of test failures, that can be quite different in the way they approach the problem – static and dynamic program slicing [14, 23] were proposed in the eighties, and have since been used in several proposals; hit spectrum of executable statements from failed and passing tests is at the base of many approaches to fault-location as, e.g. [10, 12, 13, 20, 24]; specification-based approaches [6, 8, 11] to name just a few. In [25, 26] the authors classify, describe

and compare most of these approaches. In the related work section of this paper we will focus our discussion on relevant aspects related to the work here presented.

We present Flasji (read "flashy")[1], a fault-location technique that is model-based since it builds upon models of the specification to obtain a means to observe the implemented behaviour against the intended one. Unlike several existing approaches, Flasji does not 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.

The remainder of this paper is organized as follows: section 2 gives an overview of the runtime verification approach integrating ConGu and Flasji, while briefly introducing the main aspects of ConGu; section 3 focuses on the issues around our main goal – fault-location; an evaluation experiment of the proposed approach is presented in section 4 where results are compared with the ones obtained using two fault-location tools; section 5 discusses related work and, finally, section 6 concludes.

2 Approach Overview

In this section we give a general overview of the integrated approach we propose to fault-location. An example is also introduced that will be used throughout the paper.

Locating a runtime detected fault The ConGu/Flasji approach, illustrated in figure 1, integrates the ConGu tool, which builds instrumented Java classes that are able to verify, at runtime, whether the implementations satisfy the axioms of the specifications, with Flasji, which locates the method that is responsible for a given detected fault.

In a first phase, the ConGu tool is fed with all necessary artifacts and it outputs a collection of instrumented classes, which are the original ones equipped with contracts (pre and post-conditions), capable of throwing contract violation exceptions whenever an execution is not according to the specification. ConGu also outputs wrapper classes to be used in place of the original ones, allowing clients to obtain monitorized executions without the need to modify their own code. Test suites can be created and executed over these classes as if they were the original ones, because their interface is the same; the difference lies in the way their instances react to ill-implemented operations[2]. All details about the ConGu approach and tool are found elsewhere [5, 16, 17, 19].

In a second phase, client code is executed to exercize the monitorized classes, and if some contract violation exception is thrown, Flasji component steps in. Flasji picks part of the ConGu's input and output elements, together with the axiom that was violated (the methods involved in it become potential suspects), and locates the fault by generating and running special-purpose JUnit tests. These tests are model-based since they build upon models of the specification – Flasji uses a model generator to find a finite model of the specification which is used to define expected behaviours. Instances of the implementation under verification are created and their behaviour compared with the one expected. The results of these comparisons are interpreted in order to identify the faulty method.

[1] Fault-Location in Algebraic Specification Java Implementations

[2] A tool [2, 3] already exists – GenT – that builds comprehensive test suites for Java implementations of ConGu specifications.

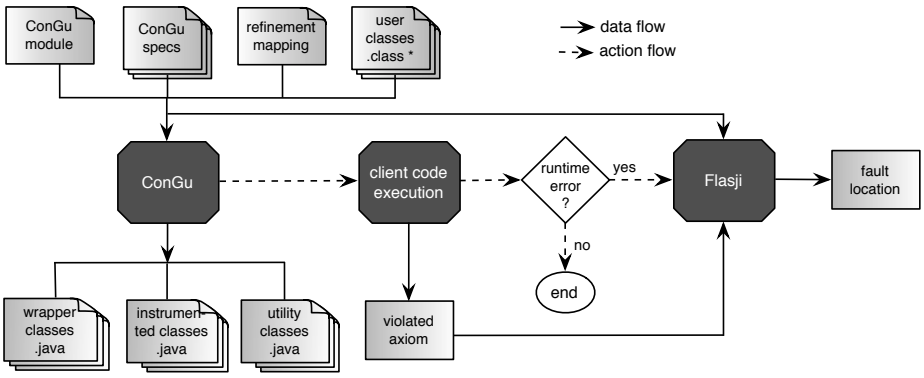

Fig. 1. Overview of the ConGu/Flasji integrated approach.

An example We present a classical yet rich parameterized data type SortedSet and explain the characteristics of the ConGu specification and refinement languages. Simple sorts, sub-sorts and parameterized sorts can be specified with the ConGu specification language, and mappings between those sorts and Java types can be defined using the ConGu refinement language. Listing 1.1 presents the specifications of the SortedSet parameterized sort and of the simple Orderable sort, which are identified as *core* and *parameter*, respectively, in the corresponding module structure.

```
specification SortedSet[TotalOrder]
  sorts
    SortedSet[Orderable]
  constructors
    empty: ⟶ SortedSet[Orderable];
    insert: SortedSet[Orderable] Orderable ⟶ SortedSet[Orderable];
  observers
    isEmpty: SortedSet[Orderable];
    isIn: SortedSet[Orderable] Orderable;
    largest: SortedSet[Orderable] ⟶? Orderable;
  domains
    S: SortedSet[Orderable];
    largest(S) if not isEmpty(S);
  axioms
    E, F: Orderable;  S: SortedSet[Orderable];
    ...
    isIn(insert(S,E), F) iff E = F or isIn(S, F);
    largest(insert(S, E)) = E if not isEmpty(S) and geq(E, largest(S));
    ...
end specification
```

```
specification TotalOrder
  sorts
    Orderable
  others
    geq: Orderable Orderable;
  axioms
    E, F, G: Orderable;
    E = F if geq(E, F) and geq(F ,E);
    ...
end specification
```

```
module
    core
        SortedSet
    parameter
        TotalOrder
end module
```

Listing 1.1. SortedSet and Orderable specifications and specification module defining the structure of the SortedSet parameterized datatype.

Depending on whether they have an argument of the sort under specification (self argument) or not, constructors are classified as *transformers* or *creators*. All non-constructor operations 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.

The correspondence between sorts and Java types is described in terms of *refinement mappings*. Listing 1.3 shows a refinement mapping from the specifications in listing 1.1 to Java types `TreeSet` and `IOrderable` (listing 1.2).

```
public interface IOrderable⟨E⟩ {
       boolean greaterEq(E e);
}

public class TreeSet⟨E extends IOrderable⟨E⟩⟩ {
       public TreeSet() {...}
       public void insert(E e) {...}
       public boolean isEmpty() {...}
       public boolean isIn(E e) {...}
       public E largest() {...}
       ...
}
```

Listing 1.2. Excerpt from a Java implementation of TotalOrder and SortedSet[TotalOrder]

```
refinement <E>
  TotalOrder is E {
    geq: Orderable e:Orderable is boolean greaterEq(E e);
  }
  SortedSet[TotalOrder] is TreeSet<E> {
    empty: ⟶ SortedSet[Orderable] is TreeSet();
    insert: SortedSet[Orderable] e:Orderable ⟶ SortedSet[Orderable] is void
        insert(E e);
    isEmpty: SortedSet[Orderable] is boolean isEmpty();
    isIn: SortedSet[Orderable] e:Orderable is boolean isIn(E e);
    largest: SortedSet[Orderable] ⟶? Orderable is E largest();
  }
end refinement
```

Listing 1.3. Refinement mapping from ConGu specifications to Java types

These mappings associate ConGu sorts and operations to Java types and corresponding methods. Notice that the parameter sort is mapped to a Java type variable that is used as the parameter of the generic `TreeSet` implementation.

3 Fault-location with Flasji

The fault-location approach we present in this paper (see figure 2) is a model-based one since it departs from models of the specification (and the abstract objects they define) and verifies whether corresponding concrete Java objects behave the same as abstract ones. Flasji interprets the deviations to the expected behaviour in order to find the location of the fault. It follows an incremental integration strategy since only one of the specification sorts is under verification – we call it the *designated sort* –, which is the one at the root of the class association graph; the others are considered as being correctly implemented (mock classes are generated that implement correct behaviour).

Flasji picks a ConGu specification module, a Java implementation and the mapping from the former to the latter, and generates Java code that creates abstract and concrete

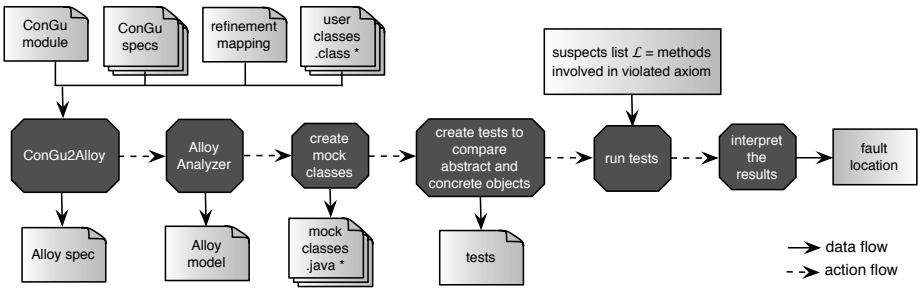

Fig. 2. The Flasji model-based approach to fault-location.

objects, according to a model of the specification that it also generates, and compares them. Flasji then runs the JUnit tests that accomplish those comparisons, and interprets their results.

3.1 Obtaining a model of the specification

In order to obtain a model of our specification, we capitalize on the Alloy Analyzer [9] which generates finite models from Alloy specifications. We may obtain Alloy specifications equivalent to our ConGu ones by using a component of the GenT tool [2, 3]. GenT's primary goal is the generation of tests suites from ConGu specifications and it has two components – ConGu2Alloy, that picks Congu specifications and generates equivalent Alloy specifications, and Alloy2JUnit, that generates JUnit tests.

So, Flasji executes ConGu2Alloy obtaining the corresponding Alloy specification ([2, 3] for details) and runs the Alloy Analyzer to generate a model that satisfies it. Figure 3 shows a model for the running example. This model consists of 2 Orderable objects and 4 SortedSet ones. The result of every applicable operation is defined for each of these objects (e.g. SortedSet2 contains the two Orderable objects, and results from inserting Orderable0 into SortedSet0 or else inserting Orderable1 into SortedSet1).

The objects on this model define correct, expected behaviour; Flasji uses the non-designated sort model objects (Orderable instances in the example) to create mock abstract objects that are used instead of concrete ones, and it uses the designated sort model objects (SortedSet instances in the example) to create mock abstract objects that will be compared with concrete ones (`TreeSet` in the example). In order to create those abstract objects, "mock" classes are generated, that are independent from the model.

```
public class OrderableMock implements IOrderable⟨OrderableMock⟩{
       private HashMap⟨OrderableMock, Boolean⟩ greaterEqResult =
                                          new HashMap⟨OrderableMock, Boolean⟩();
       public boolean greaterEq(OrderableMock e) {return greaterEqResult.get(e);}
       public void add_greaterEq(OrderableMock e, Boolean result) {
           greaterEqResult.put(e, result);}
}
```

Listing 1.4. Mock class corresponding to the Orderable parameter sort.

Listing 1.4 presents the mock class generated for the parameter sort of the running example. For each sort operation `X`, an attribute is defined to keep the information about

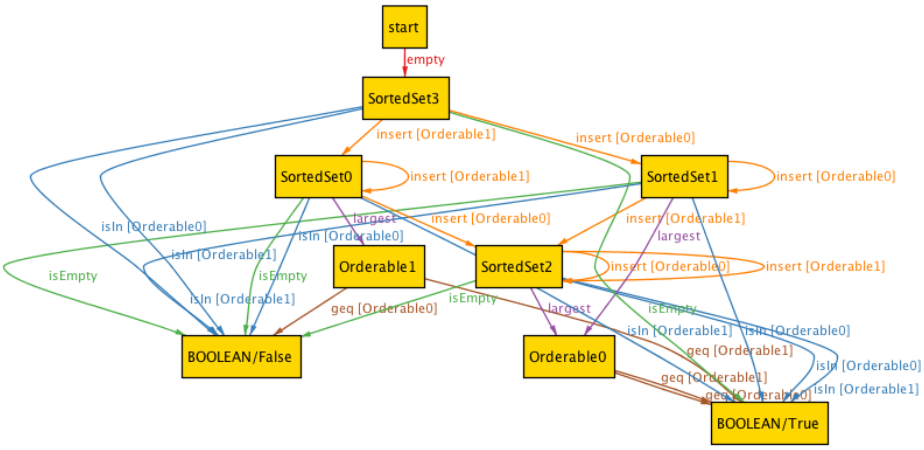

Fig. 3. A model for the SortedSet specification module found by the Alloy Analyzer.

the results of `X`, for every combination of the parameters; the `add_X` method "fills" that attribute, and the `X` method retrieves the result for given values of the parameters.

This mock class will be used to generate parameter objects that will be inserted not only in abstract `TreeSet`s representing the model objects, but also in concrete ones; the idea is to test the implementation of the designated sort for any parameter instantiation that correctly implements the Orderable sort specification.

Flasji also generates a class to represent abstract objects of the designated sort, which is not really a "mock" class since the signatures of the methods are not exactly the same as in the concrete class. The idea here is not to use these abstract objects on both abstract and concrete contexts, as we do with non-designated sort ones, but to use them for what they really are – Java representations of the objects of the model – and, as such, to compare them with concrete ones. Nevertheless, we will use the term "mock".

3.2 Comparing abstract with concrete objects

Flasji generates a Java mock class for the type implementing the designated sort, and a JUnit test class that (i) creates abstract objects corresponding to the objects defined by the Alloy model; (ii) creates the designated sort concrete objects corresponding to the abstract ones (by using the corresponding concrete constructors); and (iii) compares the behaviour of those abstract and concrete objects.

A mock class is thus generated for the SortedSet implementation:

```
public class TreeSetMock ⟨T⟩{

        private HashMap⟨T,Boolean⟩ isInResult = new HashMap⟨T,Boolean⟩();
        private boolean isEmptyResult;
        private T largestResult;

        public boolean isIn (T e){return isInResult.get(e);}
        public void add_isIn (T e, Boolean result){isinResult.put(e, result);}
        public boolean isEmpty(){return isEmptyResult;}
```

```
        public void add_isEmpty(boolean result){isEmptyResult = result;}
        public T largest(){return largestResult;}
        public void add_largest(T result){largestResult = result;}
...
        // to be continued
}
```

Listing 1.5. Part of the mock class corresponding to the SortedSet sort.

Observer operations with non-designated sort results Let us first see how we can use observer operations whose result type is different from the designated sort to compare abstract with concrete objects, using the Alloy model of figure 3.

Flasji generates a test class (see listing 1.6) where the "abstract Java world" is first populated with the abstract objects: 2 `OrderableMock` instances and 4 `TreeSetMock` ones, as defined by the Alloy model.

```
@Test
public void abstractVSconcreteTest () {
        //IOrderable Mocks
        OrderableMock orderable0 = new OrderableMock();
        OrderableMock orderable1 = new OrderableMock();
        orderable0.add_greaterEq(orderable0, true);
        orderable0.add_greaterEq(orderable1, true);
        orderable1.add_greaterEq(orderable0, false);
        orderable1.add_greaterEq(orderable1, true);

        //Abstract objects TreeSet
        TreeSetMock ⟨OrderableMock⟩ absTree3 = new TreeSetMock ⟨OrderableMock⟩();
        absTree3.add_isEmpty(true);
        absTree3.add_isIn(orderable0, false);
        absTree3.add_isIn(orderable1, false);
        TreeSetMock⟨OrderableMock⟩ absTree0= new TreeSetMock ⟨OrderableMock⟩();
        absTree0.add_isEmpty(false);
        absTree0.add_largest(orderable1);
        absTree0.add_isIn(orderable0, false);
        absTree0.add_isIn(orderable1, true);
        TreeSetMock ⟨OrderableMock⟩ absTree1 = new TreeSetMock ⟨OrderableMock⟩();
        absTree1.add_isEmpty(false);
        absTree1.add_largest(orderable0);
        absTree1.add_isIn(orderable0, true);
        absTree1.add_isIn(orderable1, false);
        TreeSetMock ⟨OrderableMock⟩ absTree2 = new TreeSetMock ⟨OrderableMock⟩();
        absTree2.add_isEmpty(false);
        absTree2.add_largest(orderable0);
        absTree2.add_isIn(orderable0, true);
        absTree2.add_isIn(orderable1, true);
        // to be continued ...
```

Listing 1.6. (Part of) the test comparing abstract with concrete objects – building the abstract objects.

Concrete objects corresponding to the ones of the designated sort will also be built, using the corresponding methods, and comparisons will be made (see listing 1.7). For now, we only present part of the test class, the one concerning the observers with non-designated sort result. Whenever there are several ways to build an object, we choose the shortest path in the model. Then we apply the observer operations to their abstract and concrete versions. Notice that, in order to cope with undesired side effects of methods, as many copies of a given concrete object are created as methods applied to it.

```
@Test
public void abstractVSconcreteTest () {
   ...
        //Concrete objects TreeSet
        //Create and compare with corresponding abstract
        TreeSet⟨OrderableMock⟩ concTree3 = new TreeSet ⟨OrderableMock⟩();
        TreeSet⟨OrderableMock⟩ concTree3_1 = new TreeSet ⟨OrderableMock⟩();
        TreeSet⟨OrderableMock⟩ concTree3_2 = new TreeSet ⟨OrderableMock⟩();
        TreeSet⟨OrderableMock⟩ concTree3_3 = new TreeSet ⟨OrderableMock⟩();
        assertTrue(concTree3_1.isEmpty() == absTree3.isEmpty());
        assertTrue(concTree3_2.isIn(orderable1) == absTree3.isIn(orderable1));
        assertTrue(concTree3_3.isIn(orderable0) == absTree3.isIn(orderable0));
        //Create and compare with corresponding abstract
        TreeSet⟨OrderableMock⟩ concTree0 = new TreeSet ⟨OrderableMock⟩();
        TreeSet⟨OrderableMock⟩ concTree0_1 = new TreeSet ⟨OrderableMock⟩();
        TreeSet⟨OrderableMock⟩ concTree0_2 = new TreeSet ⟨OrderableMock⟩();
        TreeSet⟨OrderableMock⟩ concTree0_3 = new TreeSet ⟨OrderableMock⟩();
        TreeSet⟨OrderableMock⟩ concTree0_4 = new TreeSet ⟨OrderableMock⟩();
        concTree0.insert(orderable1);
        concTree0_1.insert(orderable1);
        concTree0_2.insert(orderable1);
        concTree0_3.insert(orderable1);
        concTree0_4.insert(orderable1);
        assertTrue(concTree0_1.isEmpty() == absTree0.isEmpty());
        assertTrue(concTree0_2.largest() == absTree0.largest());
        assertTrue(concTree0_3.isIn(orderable1) == absTree0.isIn(orderable1));
        assertTrue(concTree0_4.isIn(orderable0) == absTree0.isIn(orderable0));
        // two more to go...
```

Listing 1.7. Continuing... building the concrete objects and comparing with the corresponding abstract ones.

Since the ultimate goal of this 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 allows to collect all results which will help composing a final test diagnosis. Moreover, in order to benefit from the information ConGu gives about potential suspects, the test class under construction is given the name of a method (through System.getProperties, a way to input information into JUnit tests) – it denotes a method that is suspect of containing the fault; the idea is to run the test class for each suspect in the suspects list. Violations in `assertTrue` statements involving the suspect operation are signaled to allow them to be interpreted accordingly.

Observer operations with designated sort result Let S be the designated sort of a Congu specification module and `C` the Java class that implements it. Let observer obsA, whose result is of sort S, be implemented by method `C mObsA(...)`. Consider the case where we want to compare abstract and concrete objects, whose references are kept in variables `absT1` and `concT1`, respectively, by applying to them the available observers. So, we want to compare the result of applying `mObsA` to `concT1` – call it X – with the one that should be expected – call it Y. In this case, both X and Y are concrete objects, instances of class `C`, and will be compared using `equals`.

We obtain X with `concT1.mObsA(...)`. How to obtain Y? The model gives, for any of the sorts' instances, the result of applying all operations (given all combinations of parameters within the model), so we should ask `absT1` (which is the "mock" object

that represents the corresponding abstract object) for the expected result. For this to work, we must "feed" `absT1` with that knowledge. How?

As can be observed in listings 1.6 and 1.7, the references to the concrete objects that are created are kept within variables whose names are paired to the ones of the variables that refer to corresponding abstract objects. So, if according to the model the result of applying observer obsA to `absT1` is the abstract object referenced by `absT2`, then, we say that the expected result Y of `concT1.mObsA(...)` is the concrete object `concT2`. We "feed" `absT1` with this knowledge in order to keep it available. Let us see how.

Let our SortedSet specification also define removeLargest: SortedSet $\longrightarrow$SortedSet, an observer operation that has a result of the designated type SortedSet and has the domain condition removeLargest(S) **if not** isEmpty(S). As we know, an observer whose result is of the designated sort may be implemented as a method with any return type: (i) **`void`** (tipically used in mutable implementations); (ii) `TreeSet` (tipically used in immutable implementations, but others also apply) and (iii) another different type (this case should be treated as the others since we cannot know, at the abstract level, what the result of the concrete method is supposed to be).

We only compare the relevant objects, which are (a) the concrete object corresponding to the abstract object that the Alloy model determines as being the result of the operation, and (b) either the new state of the target concrete object after the concrete method has been executed (for cases i and iii and also in mutable versions of ii) or the concrete object that is the result of the concrete method execution (for immutable versions of ii).

For this purpose, the `TreeSetMock` class must declare variables and methods to store the (abstract) results that the Alloy model establishes for this operation given the combination of existing parameters, as well as the `TreeSet` concrete objects that correspond to those abstract objects. Whatever the case (i, ii or iii), those are the needs.

In order to add these new observations to our running example, we add to the `TreeSetMock` class an inner class `Pair_TreeSetMock_TreeSet` and define two mock observer "versions" of the `removeLargest` method in order to be possible to obtain the expected abstract and concrete results of the operations.

```
public class TreeSetMock⟨T⟩{
...
        private class Pair_TreeSetMock_TreeSet ⟨T⟩{
                TreeSetMock⟨T⟩ aVal;
                TreeSet⟨T⟩ cVal;
                void setAbsVal (TreeSetMock⟨T⟩ av) { aVal = av;}
                void setConcVal (TreeSet⟨T⟩ cv) { cVal = cv;}
        }

        private Pair_TreeSetMock_TreeSet removeLargestResult = new
             Pair_TreeSetMock_TreeSet ();

        public void add_removeLargest (TreeSetMock⟨T⟩ absVal, TreeSet⟨T⟩ concVal){
               removeLargestResult.setAbsVal(absVal);
               removeLargestResult.setConcVal(concVal);
        }
        public TreeSetMock⟨T⟩ removeLargest (){return removeLargestResult.aVal;}
        public TreeSet⟨T⟩ removeLargest_C (){return removeLargestResult.cVal;}
...
        private HashMap⟨T,Pair_TreeSetMock_TreeSet⟩ insertResult = new HashMap⟨T,
             Pair_TreeSetMock_TreeSet⟩ ();

        public void add_insert (T e, TreeSetMock⟨T⟩ absVal, TreeSet⟨T⟩ concVal){
                Pair_TreeSetMock_TreeSet p = new Pair_TreeSetMock_TreeSet();
```

```
                p.setAbsVal(absVal);
                p.setConcVal(concVal);
                insertResult.put(e, p);
        }
        public TreeSetMock<T> insert (T e){return insertResult.get(e).aVal;}
        public TreeSet<T> insert_C (T e){return insertResult.get(e).cVal;}
...
```

Listing 1.8. `TreeSetMock` class with observers with designated sort result.

Why is `insert` also added to the mock class? The applications of the constructor operations that were not covered by the shortest path adopted to build the objects, should also be exercized as observations in order to be able to test the dynamics of the construction process that leads to a given object – e.g. suppose a table constructor that, given a key k and a value v, adds the pair $\langle k, v \rangle$ to the table object if it does not contain any pair with that same key already, and substitutes the old value by the new one, otherwise. Moreover, suppose the constructor implementation does not perform the substitution when it should. If we only pick the shortest path objects, that fault will not arise because none of those objects will be the target of two applications of the constructor with the same key. So, we should be able to test all applications of constructors present in the model, even if they do not generate new, different, objects: we will treat non-creator constructors as observers whose result is of the designated sort.

Let us refactor our example in listing 1.6 to cope with the existence of this kind of observers. We must define the results of observers over the abstract objects using the methods `add_xObserver` of the mock classes. The abstract behaviour of the removeLargest observer (not part of the original example) is: `removeLargest(absTree3)` is not defined, `removeLargest(absTree0)` and `removeLargest(absTree1)` have both `absTree3` as result, and `absTree0` is the result of `removeLargest(absTree2)`.

```
//Adding results of observers to abstract objects
...
//absTree3 does not satisfy domain condition of removeLargest
//the two inserts were already covered (to obtain absTree0 and absTree1)
...
absTree0.add_removeLargest(absTree3, concTree3);
// insert of orderable0 was already covered (to obtain absTree2)
absTree0.add_insert(orderable1, absTree0, concTree0);
...
absTree1.add_removeLargest(absTree3, concTree3);
absTree1.add_insert(orderable0, absTree1, concTree1);
absTree1.add_insert(orderable1, absTree2, concTree2);
...
absTree2.add_removeLargest(absTree0, concTree0);
absTree2.add_insert(orderable0, absTree2, concTree2);
absTree2.add_insert(orderable1, absTree2, concTree2);
```

Listing 1.9. Adding results of the remaining observers to abstract objects.

Finally, we compare the abstract and concrete objects using all possible observers (remember listing 1.7 already shows the comparisons using the other observers).

To verify whether a given observer whose result is of the designated sort is well implemented, we compare the concrete object that the concrete method returns with the concrete object that it should return were the observer operation correctly implemented. We consider here that the given observer is implemented with a **`void`** result type, which is consistent with the mutable characteristic of the original example, so we must first invoke the concrete 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.

```
//Comparing results of removeLargest
concTree0_5.removeLargest ();
assertTrue(concTree0_5.equals(absTree0.removeLargest_C ()));
concTree1_5.removeLargest ();
assertTrue(concTree1_5.equals(absTree1.removeLargest_C ()));
concTree2_5.removeLargest ();
assertTrue(concTree2_5.equals(absTree2.removeLargest_C ()));

//Comparing results of insert
cconcTree0_6.insert (orderable1);
assertTrue(concTree0_6.equals(absTree0.insert_C (orderable1)));
concTree1_6.insert (orderable0);
assertTrue(concTree1_6.equals(absTree1.insert_C (orderable0)));
concTree1_7.insert (orderable1);
assertTrue(concTree1_7.equals(absTree1.insert_C (orderable1)));
concTree2_6.insert (orderable0);
assertTrue(concTree2_6.equals(absTree2.insert_C (orderable0)));
concTree2_7.insert (orderable1);
assertTrue(concTree2_7.equals(absTree2.insert_C (orderable1)));
```

Listing 1.10. Comparing results of the designated sort observers.

Interpreting the results The interpretation of these tests' 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 directly created by the constructor-creator, or when applied to objects that were the target of non-creator constructors – this is important to decide which constructor is the faulty one. The *modus operandi* used is:

- if only a given observer ob fails $\rightarrow$ decide ob is the faulty method
- else *(several observers fail)*
  - if those failures (also) happen when observers are applied to an object directly created by a constructor-creator cc method $\rightarrow$ decide cc is the faulty method
  - else
    - if some of the observations made by a non-creator constructor ncc fail $\rightarrow$ decide ncc is the faulty method
    - else $\rightarrow$ inconclusive

The relevance of being a suspect lies in the (reduced) confidence one may have in its results; thus, less weight is given to observations using a suspect method.

4 Evaluation

To evaluate the effectiveness of our Flasji 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 also picked two fault-location tools – GZoltar [1, 21] and EzUnit4 [4, 22] – that work

	Faulty method	# tests	failed tests	Faulty method ranked: Flasji	EzUnit4	GZoltar
SortedSet	isEmpty	20	5	1_{st}	n_{th}	2_{nd}
	isEmpty	20	1	1_{st}	1_{st}	2_{nd}
	isIn	20	1	1_{st}	n_{th}	1_{st}
	largest	20	7	1_{st}	1_{st}	1_{st}
	largest	20	1	1_{st}	1_{st}	2_{nd}
	private insert	20	2	No	n_{th}	2_{nd}
	public insert	20	5	1_{st}	n_{th}	2_{nd}
MapChain	get	17	4	No	n_{th}	1_{st}
	get	17	3	No	2_{nd}	1_{st}
	get	17	4	No	n_{th}	n_{th}
	isEmpty	17	2	1_{st}	2_{nd}	1_{st}
	isEmpty	17	1	1_{st}	n_{th}	1_{st}
	put	17	0	1_{st}	No	No
	put	17	1	1_{st}	1_{st}	2_{nd}
	put	17	2	1_{st}	1_{st}	2_{nd}
	remove	17	1	1_{st}	2_{nd}	2_{nd}
	remove	17	1	1_{st}	2_{nd}	2_{nd}

Table 1. Results of comparative experiments. "Flj", "EzU" and "GZ" stand for Flasji, EzUnit4 and GZoltar respectively. "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.

on JUnit tests and, alike Flasji, have methods as the units under test, and compared the three approaches' results.

For this evaluation task we used tests generated by the GenT tool [2, 3] for the two case studies, which exercise all the methods implementing specified operations and all axioms of the specification. In what concerns our approach, these tests executions correspond to the "Client code execution" box of figure 1 in section 2. Runtime errors were detected and Flasji was put to work: it generated new, model-based tests and executed them in order to find guilty methods. The other two tools – EzUnit4 and GZoltar – inspected the several executions of the GenT tests in order to create a ranking of suspects.

For each of several faulty 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 results of executing the GenT tests (20 tests for the SortedSet case, and 17 for the MapChain). For each seeded fault we show the number of failed GenT tests and 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 a summary in figure 4). 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). Applying an alternative method of observation (see [18]) – one

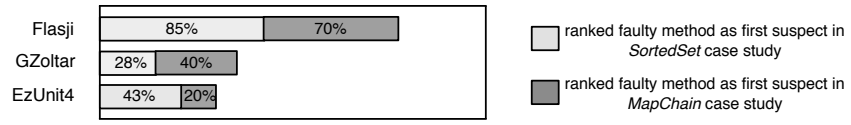

Fig. 4. Summary of the evaluation experiment. The bars measure the success of each approach in ranking the faulty method as first suspect.

where the `equals` method is not used and, instead, only the outcomes of observers whose result is not of the designated 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 faulty methods `put iii` and `remove i` got worse – they are ranked second instead of first. These particular cases indicated `isEmpty` as prime suspect because the seeded fault of both `put iii` and `remove i` was the absence of change in the number of elements in the map whenever insertion/removal happens, which made `isEmpty` fail.

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 (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. As a consequence, neither EzUnit4 and GZoltar detected the fault; the same happened with our approach because Flasji was not applied (remember figure 1 in section 2). However, putting Flasji to work led to the identification of the guilty method.

5 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 ([6–8, 11] 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. [7, 15, 27]. In [18] 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-designated sort. In some way this complies with the notion of observable contexts in [7] – all observers but the ones whose result is of the designated sort constitute observable contexts.

Many other fault-location approaches have been proposed over the years that are not based on specifications (in [25, 26] the authors present a very comprehensive survey where they classify fault-location approaches). Among these, program spectrum-based methods (e.g. [4, 10, 12, 13, 20–22, 24]) from which we have chosen two tools to be part of our evaluation experiment, as presented in section 4.

Program spectrum-based methods record information about several aspects of the execution of tests and use it to identify most probably faulty pieces of code. Differences between approaches include the use of failed tests only or both passed and failed ones, the way they calculate the suspiciousness of each piece of code, the unit of suspicion (basic statements, methods), etc. The tools we used in section 4 – GZoltar [1, 21] and EzUnit4 [4, 22] – compute the probability of Java methods being faulty given their participation in a number of passing and failing test cases, differing essentially in the way they calculate the suspiciousness of each executed method.

6 Conclusions

We presented Flasji, a model-based approach to the location of Java methods that incorrectly implement algebraic specifications. Flasji capitalyzes on ConGu, GenT and Alloy Analyzer tools and builds JUnit tests whose goal is the identification of the faulty method. These tests are based in models of the specification and aim at comparing the behaviour of instances of concrete implementations with the one that is expected as defined by the model. The results of the comparisons are interpreted in order to find the method responsible for the fault.

The structured nature of ConGu 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 the Flasji fault-location technique – a Java type implementing a specification sort is tested conditionally, i.e. presuming all others from which it depends are correctly implemented.

An evaluation experiment is presented where Flasji is compared with two other fault-location tools over two case studies where faults have been seeded in the implementing Java classes.

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. P. Crispim, A. Lopes, and V. Vasconcelos. Runtime verification for generic classes with ConGu2. In *Proc. SBMF 2010*, volume 6527 of *LNCS*, pages 33–48. Springer-Verlag, 2011.

6. R.K. Doong and P.G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM TOSEM*, 3(2):101–130, April 1994.
7. M.C. Guadel and P.L. Gall. Testing data types implementations from algebraic specifications. *Formal Methods and Testing*, 2008.
8. 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.
9. D. Jackson. *Software Abstractions - Logic, Language, and Analysis, Revised Edition*. MIT Press, 2012.
10. J.A. Jones and M.J. Harrold. Empirical evaluation of the Tarantula automatic fault- localization technique. In *Proc. 20th IEEE/ACM ASE*, pages 273–282, December 2005.
11. L. Kong, H. Zhu, and B. Zhou. Automated testing EJB components based on algebraic specifications. In *COMPSAC 2007*, pages 717–722, July 2007.
12. B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proc. 2005 ACM SIGPLAN PLDI*, pages 15–26, June 2005.
13. C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE TOSE*, 32(10):831–848, October 2006.
14. J.R. Lyle and M. Weiser. Automatic program bug location by program slicing. In *Proc. 2nd International Conference on Computer and Applications*, pages 877–883, June 1987.
15. 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.
16. 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.
17. 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.
18. 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.
19. L.S. Reis. ConGu v.1.50 users guide. 2007.
20. M. Renieris and S.P. Reiss. Fault localization with nearest neighbor queries. In *Proc. 18th IEEE ASE*, pages 30–39, October 2003.
21. 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.
22. 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.
23. M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.
24. W. E. Wong, V. Debroy, and Choi B. A family of code coverage-based heuristics for effective fault localization. *Journal of Systems and Software*, 83(2):188–208, February 2010.
25. W.E. Wong and V. Debroy. A survey of software fault localization. Technical Report UTDCS-45-09, Dept. Computer Science, University of Texas at Dallas, November 2009.
26. W.E. Wong and V. Debroy. Software fault localization. *IEEE Transactions on Reliability*, 59(3):473–475, September 2010.
27. H. Zhu. A note on test oracles and semantics of algebraic specifications. In *QSIC 2003*, pages 91–99. IEEE Computer Society, 2003.