A fault-location technique for Java implementations of algebraic specifications

Isabel Nunes and Filipe Luís

DI–FCUL–TR–2012–02

DOI:10455/6809

(http://hdl.handle.net/10455/6809)

June 2012

Published at Docs.DI (http://docs.di.fc.ul.pt/), the repository of the Department of Informatics of the University of Lisbon, Faculty of Sciences.

A fault-location technique for Java implementations of algebraic specifications

Isabel Nunes and Filipe Luís

Faculty of Sciences, University of Lisbon,
Campo Grande, 1749–016 Lisboa, Portugal

Abstract. Executing comprehensive test suits allows programmers to strengthen the confidence on their software systems. However, given some failed test cases, finding the faults' locations is one of the most expensive and time consuming tasks, thereby any technique that makes it easier for the programmer to locate the faulty components is highly 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 the ConGu and GenT approaches, namely on the models for the specification under study and the corresponding generated JUnit test suits that cover all axioms of the specification, and present a collection of techniques and underlying methodology, that give the programmer a means to find the location of a fault that causes the implementation to violate the specification.

We propose Flasji, a stepwise process for finding the faulty method, which is transparent to the programmer, that applies the proposed techniques to find a collection of initial suspect candidates and to subsequently decide the prime suspect among them. We carried out an experiment to evaluate Flasji and obtained very encouraging results.

1 Introduction

Comprehensive testing 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. This aspect makes them suitable for describing abstract data types (ADTs) [8].

In this paper we capitalize on and enrich an integrated approach to specification-guided software development, which comprises the ConGu approach [18, 19] to the development of generic Java implementations of parameterized algebraic specifications, and the GenT tool [3, 4] that generates 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 comprehensive, i.e. that cover all axioms/properties of the specification, as ConGu and GenT do, 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 must be able to use their results to localize the faulty entities. At present, the ConGu tool output is often insufficient to the task of finding the faulty method; moreover, simply executing the JUnit tests generated by GenT fail to give the programmer clear hints about the faulty method – all methods used in failed tests are equally suspect. The ideal result of a test suite execution is the exact localization of the fault(s), however this has proven a very 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 [16, 24] 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. [12, 14, 15, 21, 25]; specification-based approaches [6, 10, 13] to name just a few, etc. In [26, 27] the authors classify, describe and compare most of these approaches. In this paper we will focus our discussion of the state of the art on some particular aspects related to the work here presented. Onde pour??

In a situation where all methods are equaly suspect, the comparison between abstract (correctly behaved) objects and concrete (observed behaviour) ones can be unworthy, due to the eventual lack of trust on the results given by observer methods.

Whenever we have a particular suspect, however, we may consider that it is relatively safe to use the other, not previously suspect, methods to inspect the concrete objects and, in that way, be able to compare their characteristics with the ones of the corresponding abstract objects. We propose Flasji (read "flashy")[1], a collection of techniques and underlying methodology, that allows to interpret and manipulate the results of ConGu/GenT 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 exclusively 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 through the application of two different techniques to those tests; that set of suspects is then reduced by the stepwise application of the other proposed techniques, eventually obtaining a unique suspect. Some of these techniques imply the automatic generation of new tests from the initial ones, that give new and useful information towards the goal of reducing the number of suspects.

Alike other fault-location tools and approaches, the unit of failure Flasji is able to detect is the method, leaving to the programmer the task of identifying the exact instruction within it that is responsible for the failure. If more than one failure exists, the repeated application of the approach, 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 sorts are not tested all together; instead, each one is tested conditionally, presuming all others from which it depends are correctly implemented (mock classes are used that are built according to the overall specification models, thereby correct). 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 2 gives an overview of the Flasji approach both in the broader context of ConGu and GenT approaches, and isolated; it also presents an example that will be used throughout the paper; section 3 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 4 whose main objective is to collect suspect methods and decide the guilty one; section 5 integrates these techniques resulting in the Flasji approach; an evaluation experiment of Flasji is presented in section 6 and results are compared with the ones obtained using two fault-location tools [22, 23]; section 7 discusses related work and, finally, section 8 concludes and identifies topics to be improved.

2 Approach Overview

In this section the reader will be given a general overview of the integrated approach we propose to fault-location, given a structured set of specifications defining an abstract data type, and a set of Java types implementing it. An example is also introduced that will be used throughout the paper to illustrate the several languages and techniques presented.

2.1 From failed tests to fault-location

We work with ConGu specifications and structure-defining modules (ahead in this text) [18, 19], and (refinement) mappings that define the correspondence between the sorts and operations of the specifications, and the Java types and methods of the implementations.

These elements are used as input to the GenT tool [3, 4], which generates additional elements that will be used by our fault-location approach – Flasji –, as illustrated in figure 1.

In a first phase, GenT is partially applied – the ConGu2Alloy part – to a structured set of ConGu specifications (.spc files containing the individual specifications, and .mod file defining the structure of the ADT). GenT translates the possibly parameterized data type to an equivalent Alloy specification (a file with .als extension) that also contains extra Alloy run commands for a thorough coverage of specification axioms.

In a second phase, the GenT Alloy2JUnit part is applied to this Alloy specification and to the ConGu refinement mapping and the Java types that implement the ConGu specification module (no source code needed,

[1] Fault-Location of Algebraic Specification Java Implementations

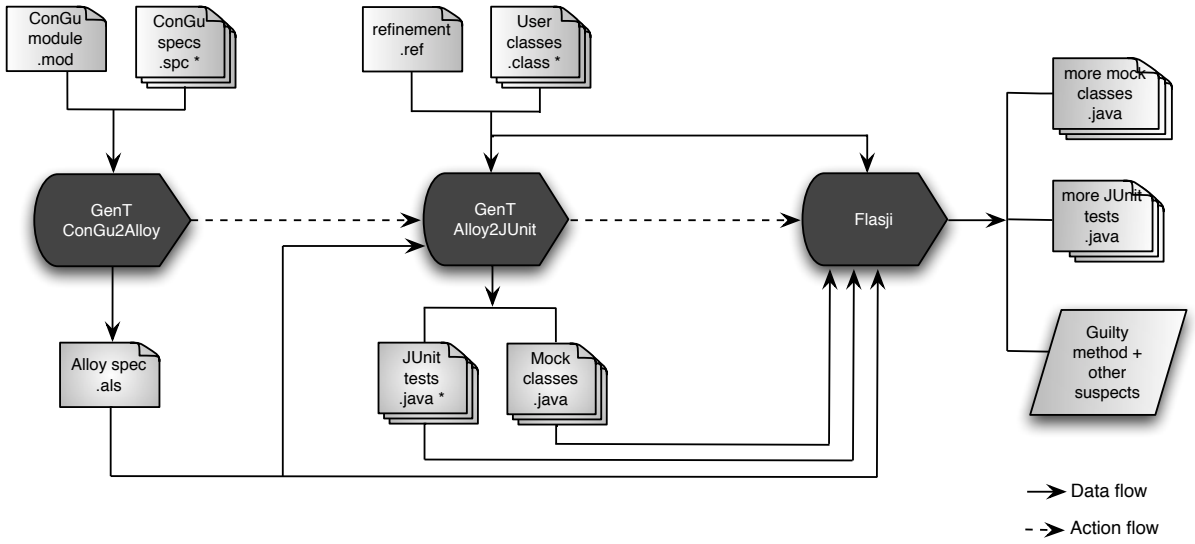

Fig. 1. Overview of the integrated approach.

only bytecode), resulting in the creation of comprehensive JUnit tests together with mock classes to be used for parameter types (see section 3 for a detailed explanation).

Finally, Flasji picks the generated Alloy specification, JUnit tests and mock classes, and the original refinement mapping and Java classes implementing the specification module, and applies the several techniques presented in this paper, in an integrated way, as briefly overviewed in figure 2.

In order to obtain a reasonable conviction of some method's guilt, departing from a situation in which one or more tests fail and all methods are suspect, we pick some particular failed tests and try 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 aditional significant information. The several techniques we present in this paper are used, as illustrated in figure 2, to identify possible suspects and to strengthen or weaken suspicions of guilt. The main objective is to end up identifying the strongest suspect and, eventually, other possible guilt implementations.

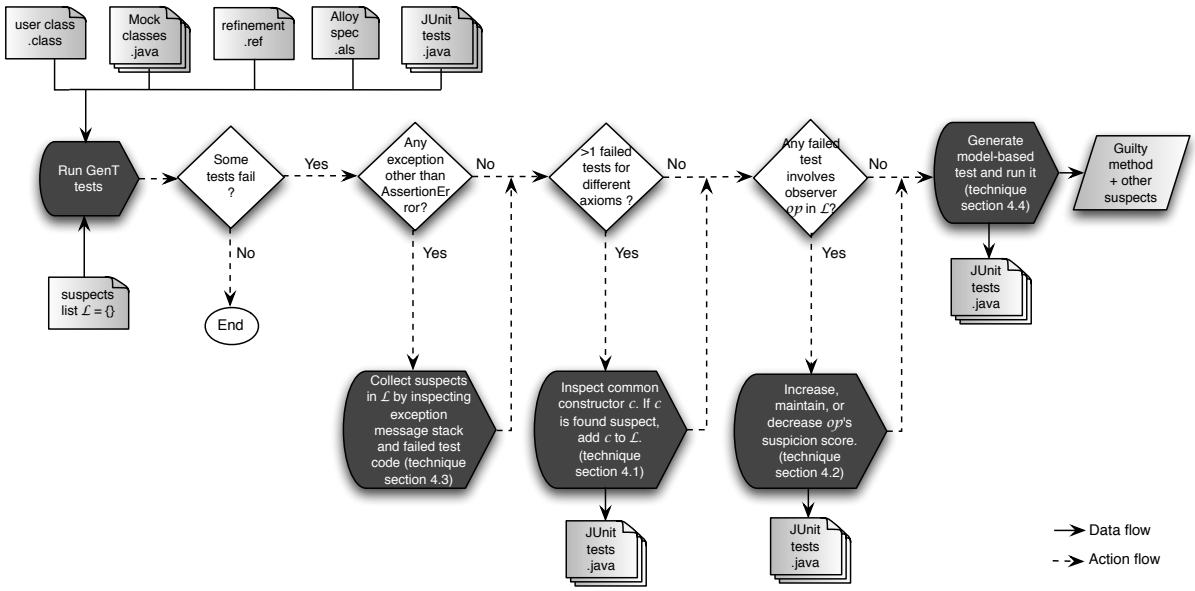

Fig. 2. Overview of the Flasji approach.

An initial important source of information about possible guilty operations may be found in JUnit tests that fail due to some exception other than `java.lang.AssertionError`. If a method throws a `NullPointerException`, for example, it may be because some (other) method was executed when it should not (its pre-condition evaluates to true instead of false), thus wrongly letting the system try other further operations. We devised an algorithm (section 4.3) to collect potential suspects out of this kind of errors.

Other two techniques (presented in sections 4.1 and 4.2) are devised that inspect both specification and Java code involved in certain failed tests, and automatically generate new tests that reveal some otherwise hidden information that is then used to strengthen/weaken guilt suspicions on specific operations implementation.

The fault-location process ends with the application of a fourth technique, that is presented in section 4.4, which generates new, thorough model-based tests, where the information about who are the suspect operations counts to constrain the confidence that can be put on their results; these tests compare the results obtained by application of the Java methods implementing specification operations against the results that they should obtain, as given by abstract information taken from models of the specification.

2.2 An example

We pick a classical yet rich parameterized data type – the sorted set – and begin presenting the ConGu specification module that specifies it, while explaining the characteristics of the ConGu specification and refinement languages. We also choose one of its many possible implementations to be used throughout the paper.

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 – detailed information about the ConGu approach can be found in [1, 18–20].

Listings 1.1 and 1.2 present the specifications of theSortedSet parameterized sort (the *core* sort) and of the simple sort Orderable (the *parameter* sort).

```
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
```

Listing 1.1. Specification of the parameterized sort (core) SortedSet

We have *constructor*, *observer* and *other* operations, where constructors compose the minimal set of operations that allow to build all instances of the sort, observers allow to analyse those instances, and the other operations are usually comparison operations or operations derived from the others.

```
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
```

Listing 1.2. Specification of the Orderable parameter sort used by SortedSet

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 —>?), in which case a *domains* section specifies the conditions that restrict their domain.

The correspondence between sorts and Java types as well as between operations/predicates and those types' methods, can be described in terms of *refinement mappings*. Listing 1.4 shows a refinement mapping from the specifications in listings 1.1 and 1.2 to Java types `TreeSet` and `IOrderable` (listing 1.3).

```
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 1.3. Excerpt from a Java implementation of specificationsTotalOrder 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 1.4. 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 Test generation from ConGu specifications

The fault-location approach we present in this paper – Flasji – capitalizes on the GenT tool since many of GenT's artifacts constitute both its starting point and working material, namely the Alloy specification that GenT generates from the ConGu module, and the JUnit tests and parameter related mock classes.

In [4], where the GenT approach and tool are presented, the generation of Alloy [11] specifications from ConGu modules is introduced, followed by the construction of test cases from Alloy specifications and models, and ConGu refinement mappings. These tests cover all axioms of the data abstraction specification and, together with the Alloy specification, are key to the Flasji approach.

In this section we briefly present these subjects, considering that the original ConGu module respects the following: *(i)* it is satisfiable by finite models in order to be possible to use the Alloy Analyzer; *(ii)* the core sorts include at least a creator and a transformer.

3.1 Generation of Alloy specification and abstract tests

Listing 1.5 presents part of the Alloy specification that is generated from the ConGu module composed of SortedSet and Orderable sorts.

```
sig Element {}
sig SortedSet extends Element  {
        largestOp : lone Orderable,
        isInOp : (Orderable) −⟩ one BOOLEAN/Bool,
        isEmptyOp : one BOOLEAN/Bool,
        insertOp : (Orderable) −⟩ one SortedSet
}
sig Orderable extends Element {
        geqOp : (Orderable) −⟩ one BOOLEAN/Bool
}
one sig start  {
        emptyOp : one SortedSet
```

```
}
fact SortedSetConstruction {
        SortedSet in (start.emptyOp).*({coreSort : SortedSet, nextCoreSort : coreSort.insertOp[Orderable]})
}
fact OrderableUsedVariables {
        Orderable in (SortedSet.largestOp + SortedSet.isInOp.BOOLEAN/Bool + SortedSet.insertOp.SortedSet +
            Orderable.geqOp.BOOLEAN/Bool)
}
fact domainSortedSet0 {
        all SVar : SortedSet | SVar.isEmptyOp != BOOLEAN/True implies one SVar.largestOp else no SVar.
            largestOp
}
// (...)
fact axiomSortedSet3 {
        all EVar, FVar : Orderable, SVar : SortedSet | SVar.insertOp[EVar].isInOp[FVar] = BOOLEAN/True iff
            (EVar = FVar or SVar.isInOp[FVar] = BOOLEAN/True)
}
// (...)
fact axiomSortedSet5 {
        all EVar : Orderable, SVar : SortedSet | (SVar.isEmptyOp != BOOLEAN/True and EVar.geqOp[SVar.
            largestOp] = BOOLEAN/True) implies SVar.insertOp[EVar].largestOp = EVar
}
// (...)
```

Listing 1.5. Excerpt of the Alloy specification generated for the SortedSet module.

Axiom or composing Boolean expression	**Cases to practice (FDNF minterms)**
Simple conditional axiom: B if A	A and B not A and B not A and not B
Disjunctive logic: A or B	A and B A and not B not A and B
Biconditional axiom: A iff B	A and B not A and not B
Ternary conditional axiom: X = Y when A else Z	Apply previous rules to the pair: X = Y if A X = Z if not A
Multiple variables of the same type: Expression(A, B)	A = B and *Expression*(A, B) A != B and *Expression*(A, B)

Table 1. Cases to be practiced in conditional axioms and Boolean expressions.

In what concerns the syntactic part of the translation, sorts originate Alloy signatures (non-primitive signatures extend Element); operations that are not creators are translated to fields (relations) of corresponding Alloy signatures, omitting the self argument (partial operations originate lone relations – to 0 or 1); creator operations originate fields in the special signature start, which has a single instance and is the root of all instances in every model (see empty); predicates are treated as operations with boolean result; Alloy construction facts are generated in order to avoid junk in models, that is, the instances of the core sorts are restricted to the ones that can be represented by terms using only constructors (a creator followed by 0 or more transformers); Alloy usage facts are generated in order to avoid superfluous instances of non-core signatures (Element and Orderable in the example) in models, imposing each instance of a non-core signature to be used as input or output of a core signature relation, or to be an instance of a sub-signature of the non-core signature.

In what concerns the semantic part of the translation, domain restrictions that relate to partial operations are translated to Alloy facts that enforce the corresponding field to be defined inside the domain (see the case of largest); axioms are translated to Alloy facts, with universal quantification over corresponding free variables.

In order to fully test all axioms, the adopted coverage criterion implies the generation of as many test cases as there are minterms in each axiom, when represented in their full disjunctive normal form (FDNF) (see table 1). The FDNF of a logical formula that consists of Boolean variables connected by logical operators is a canonical DNF in which each Boolean variable appears exactly once (possibly negated) in every conjunctive term (called a minterm) [9].

// (...)

```
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

run axiomSortedSet3_1 {
        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

run axiomSortedSet3_2 {
        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

run axiomSortedSet3_3 {
        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
// (...)
```

Listing 1.6. Alloy run commands generated to test axiomSortedSet3 of Fig. 1.5.

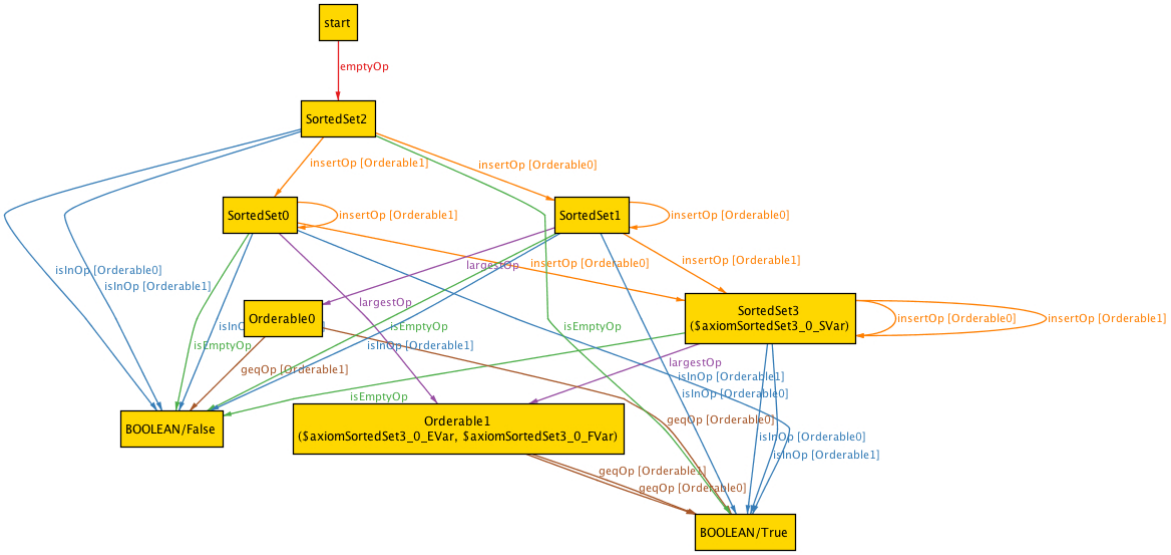

Fig. 3. A model for the axiomSortedSet3_0 run command found by the Alloy Analyzer.

Although not all minterms are necessarily satisfied, at least one minterm, for each axiom, must be so. Furthermore, in cases where there exist more than one free variable, the several combinations of equality are also covered. For this purpose, an Alloy run command is generated for each minterm and equality combination between free variables of an axiom, that allows to find, through the Alloy Analyzer, models and values for the free variables satisfying it (see listing 1.6).

Models satisfying the specification are generated by the Alloy Analyzer for each consistent run – figure 3 shows a model for the axiomSortedSet3_0 run command found by the Alloy Analyzer. These models define the objects that will be created in each JUnit test that GenT generates.

3.2 JUnit tests generated from Alloy models

The GenT tool then creates as many JUnit tests as consistent runs (for each axiom, one or more tests can be generated). All tests for a given axiom will test the same assertion – the axiom itself; the difference between them lies in the objects that are created, over which the `assert` statements are verified – each test verifies the axiom over a given instance of the model, which was found by Alloy Analyzer, that satisfied a given run of that axiom (remember each run corresponds to a clause/minterm of the FDNF of the axiom). Finite implementations of the

parameter sorts are automatically generated, 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 sort, which are independent from the models (see listing 1.7); *(ii)* mock objects (instances of mock classes) are created and initialized in each test, according to the corresponding model (see listing 1.8).

```
public class OrderableMock implements IOrderable⟨OrderableMock⟩{
        private HashMap⟨OrderableMock, Boolean⟩ greaterEqMap = new HashMap⟨OrderableMock, Boolean⟩();

        @Override public boolean greaterEq(OrderableMock e) {
                return greaterEqMap.get(e);
        }

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

Listing 1.7. Mock class corresponding to the Orderable parameter sort.

In each generated JUnit test for a given axiom we can identify a first part where the parameters of the generic ADT are created and initialized (mock objects) according to the Alloy model found for the corresponding run, followed by the configuration of the free variables used by the given axiom, according to the same model, and finally the axiom verification, where intermediate axiomatic terms are built and an assertion is verified which consists of the translation of the axiom expression. Comprehensive test cases result from this generation process insofar as all operations are tested for cases that cover every consistent minterm of all axioms.

```
@Test
public void axiomSortedSet3_0() {
    // IOrderable Mocks
    final OrderableMock orderableSort0 = new OrderableMock();
    final OrderableMock orderableSort1 = new OrderableMock();
    orderableSort0.add_greaterEq(orderableSort0, true);
    orderableSort0.add_greaterEq(orderableSort1, false);
    orderableSort1.add_greaterEq(orderableSort0, true);
    orderableSort1.add_greaterEq(orderableSort1, true);
    // Prepare Core Var Factories
    CoreVarFactory⟨TreeSet⟨OrderableMock⟩⟩ sVarFac = new CoreVarFactory⟨TreeSet⟨OrderableMock⟩⟩() {
        @Override
        public TreeSet⟨OrderableMock⟩ create() {
            TreeSet⟨OrderableMock⟩ s = new TreeSet⟨OrderableMock⟩();
            s.insert(orderableSort0);
            s.insert(orderableSort1);
            return s;
        }
    };
    // Test the Axiom
    axiomSortedSet3Tester(sVarFac, orderableSort1, orderableSort1);
}

private void axiomSortedSet3Tester(
          CoreVarFactory⟨TreeSet⟨OrderableMock⟩⟩ sVarFac, OrderableMock eVar,
                        OrderableMock fVar) {
    TreeSet⟨OrderableMock⟩ sVar_0 = sVarFac.create();
    TreeSet⟨OrderableMock⟩ sVar_1 = sVarFac.create();
    sVar_0.insert(eVar);
    assertTrue(sVar_0.isIn(fVar) == (eVar == fVar || sVar_1.isIn(fVar)));
    // isIn(insert(S,E), F) iff E = F or isIn(S, F);
}
```

Listing 1.8. JUnit test for the axiomSortedSet3_0 run command and respective model (fig. 3).

4 Finding suspects and strengthening/weakening suspicions

The techniques we present in this section depart from a situation where we have:

- an Alloy specification corresponding to the original ConGu module;
- the Java types that implement the module, where a fault exists in the class corresponding to the sort under study;
- a set of JUnit tests created by GenT, where some have failed, and the information about the exception they launched;
- mock classes for the parameter sorts.

Each proposed technique capitalizes, in its own way, on these "materials", trying to look at the specification operations, the corresponding Java methods and the failed tests in such a way that allows it to obtain more information about the location of the fault.

The tests we work with are conditional in the sense that they test the implementations of the sort under study – the *designated* sort, from now on – assuming that the implementations of the other sorts of the ConGu module are correct. We use GenT generated mock classes for parameter sorts and, if necessary, we generate mock classes for other non-designated, non-parameter sorts.

Our approach is not based on hit spectrum of executed statements; instead, it uses the specification and its models to try to find any deviation of the implemented methods to the expected results, both in terms of axiom compliance and individual method execution result. This means that we search and find guilty methods among the ones that implement the specification operations, not among private methods or others that are not specified in the initial ConGu specification module. If a given method m, that implements a given specification operation op, invokes a private method p that is faulty, our technique will point to m, meaning that it does not correctly implement op.

We propose two techniques that together can be used to find an initial list of possible suspects, departing from the set of failed tests, and two others that, given a suspect, can be used to strengthen or weaken the suspicions of guilt. In section 5 we delineate the stepwise approach that integrates these techniques.

Some useful concepts Before presenting our integrated approach to fault location, let us formalize some concepts that will be used in coming sections.

We consider each axiom (fact) of the Alloy specification in its full disjunctive normal form (FDNF) where each clause or minterm is a conjunction containing all literals of the axiom. For a given axiom, at least one of the minterms is satisfiable. More concretely, let our specification declare a set of axioms $\{ax\}$ where each ax is of the form:

$$\forall_{x_1:s_1,\ldots,x_n:s_n} \bigvee_{1\leq j\leq m} \varphi_j$$

for variables x_i and sorts s_i and where each of the m minterms of ax is of the form:

$$\varphi_j = \bigwedge_{1\leq k\leq l} a_{jk}$$

where a_{jk} represents a literal (an atom or its negation), and l is the number of literals in axiom ax.

For each axiom $ax = \varphi_1 \vee \ldots \vee \varphi_m$, Alloy run commands R_j exist, at least one for each one of the m clauses or minterms of ax. In addition, when there is more than one free variable of the same type in an axiom, the various equality combinations among them (equal or diferent) are also covered. Some of the runs will eventually be inconsistent, but at least one from each axiom is consistent.

Generated from model instances for the consistent R_j runs, JUnit tests are created by the GenT tool that allow to test axiom ax for each generated model instance. All these tests verify the disjunction of the same `assert` statements (corresponding to the m minterms φ_j of axiom ax). They only differ one from another in the objects that they create and manipulate, which correspond to the model instances generated for the corresponding run commands.

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:

- α_R is the assertion verified in run R (minterm)
- α_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

4.1 Working on constructor operations

The objects involved in a given GenT test are created according to an Alloy model for the overall specification added with a specific run. The `assert` statement that is verified over those objects corresponds to an axiom of the specification (the one from which the given run originated). This means that the result of t_{R_1} and t_{R_2} is not necessarily the same as the result of an hypothetical test $t_{R_1\&R_2}$ (generated from the run that joins α_{R_1} and α_{R_2} using conjunction). This is because the tests t_{R_1} and t_{R_2} have `assert` statements over potentially different objects, while the test $t_{R_1\&R_2}$ would test the conjunction of the same assertions over the objects that correspond to an instance of the Alloy model for the compound run.

Although the failure of two or more tests, each one over a specific set of objects, gives us valuable information about the involved operations, knowing the behaviour of suspicious operations when applied over the same set of objects, and looking at that behaviour in a different perspective, can bring aditional significant information. We propose gathering the runs that originate the failed tests, using the strategy described ahead, as a way to unveil the potentially guilty operations.

The idea here is to pick two failed tests that test different axioms but use the same constructor, and mix them 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.

The Strategy We gather the runs that originated those two failed tests into a new run that joins (using conjunction) the two runs' assertions, and generate a new model that satisfies this new run. We want this model to be such that, if we generate a test – call it t_{11} – from this new, compound, run, the test fails. Although it may seem obvious, this is not always the case: on the one hand, the model we use to build the new test is not necessarily the same as the ones that were used for the original failed tests; on the other hand, test t_{11} corresponds to a conjunction of disjunctions of clauses – from all these clauses, only two of them correspond to the two clauses that form the new, compound, run.

Once we are certain that the test t_{11} generated from this new run fails, we generate a further new test – call it t_{00} – that negates both assertions. This new test is built over the same objects used by t_{11}, which we know to represent a model that satisfies the above new, compound, run.

We claim that, if this new test passes, we can increase the guilt of the constructors – since the two original failed tests test for diferent axioms, the constructors are the only operations we can ensure they have in common. So, we can see here a further clue to increase suspicion of their guilt. The fact that the model that is used in these new tests is such that it should verify both axioms, we can say that, if the test negating both assertions passes, we have a very good clue leading to a deficiency in the implementation of the constructor that both assertions have in common.

If this new test fails we can be sure that one of the assertions of the original compound test had to be true in order to obtain a falsity when negating both assertions (the other was necessarily false – remember that test t_{11} above, which operated over the same objects, failed). Both assertions use the same constructors but one succeeds and the other does not, so, we claim that, in this case, the guilt should not rely on the common constructor.

Putting it to work Let t_f and t_g be two failed tests, such that:

- t_f and t_g test different axioms
- t_f and t_g involve the same constructors
- t_f and t_g originate from runs R_f and R_g

Because t_f and t_g test different axioms, they pottentially test diferent 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 run is $\alpha_{R_{fg}} = \alpha_{R_f} \wedge \alpha_{R_g}$.

Let Alloy generate models for this new run. Then, generate the test t_{fg} as usual, using a model that ensures its failure. As expected, this new test is such that $\alpha_{t_{fg}} = \alpha_{t_f} \wedge \alpha_{t_g}$.

Because 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 α_{t_f} and α_{t_g} fail, we can strengthen the suspicion of guilt of the common constructor;
2. if α_{t_f} fails and α_{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 whether both α_{t_f} and α_{t_g} fail, we build a new test, over the same objects, where we negate both α_{t_f} and α_{t_g}, and run it; if it passes, then (1) is the case;
- To find whether one sub-expression fails while the other passes, we can use the same above built test where α_{t_f} and α_{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 $\alpha_{t_{f0g0}} = \neg\alpha_{t_f} \wedge \neg\alpha_{t_g}$
Next we execute test t_{f0g0}:

– If it passes:
  • the original problem may be caused by the constructors used in t_{fg}, (**strengthens** the guilt).
– If it fails:
  • the original problem should not be caused by the constructors used in t_{fg} (**weakens** the guilt);

If this new test fails due to some exception other than `java.lang.AssertionError` (e.g. `NullPointerException`, `ArrayIndexOutOfBoundsException`, etc), we cannot conclude nothing more about the constructor involved. We can identify three possible reasons for such a case:

– The characteristics of the short-circuit Java operators `&&` and `||` – they only evaluate the second argument if the first is true or false, respectively – makes the order in which disjuncts and conjuncts are picked revelant;
– The new Alloy model for the compound run may be such that some pre-condition be violated in the Java implementation, that was not violated before.

Some examples We modify a particular implementation of the SortedSet ADT by "planting" some faults in order to exemplify the application of this technique.

EXAMPLE 1 – THE PRIVATE `insert` CONSTRUCTOR OF `SortedSet` IS ILL-IMPLEMENTED.

```
private Node⟨E⟩ insert(Node⟨E⟩ node, E e) {
          if (node == null)
              return new Node⟨E⟩(null, e, null);
          if (e.equals(node.data))
              return node;
          if (e.greaterEq(node.data))
              // node.right = insert(node.right, e);  this is right
              node.left = insert(node.left, e);   // fault here!
          else
              // node.left = insert(node.left, e);     this is right
              node.right = insert(node.right, e);  // fault here!
          return node;
}
```

As a result of this fault, some of the GenT generated tests fail; among them, the tests for runs axiomSortedSet3_0 (that gives semantic to operation isInOp):

```
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
```

and 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 correspond to 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 axiom 5, respectively:

```
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 – insertOp –, 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. The chosen 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 OrderableSort_0 = new OrderableMock();
        OrderableMock OrderableSort_1 = new OrderableMock();
        OrderableSort_0.add_greaterEq(OrderableSort_0, true);
        OrderableSort_0.add_greaterEq(OrderableSort_1, false);
        OrderableSort_1.add_greaterEq(OrderableSort_0, true);
        OrderableSort_1.add_greaterEq(OrderableSort_1, true);
        // Factories core var setup
        CoreVarFactory⟨TreeSet⟨OrderableMock⟩⟩ SVar_Factory = new CoreVarFactory⟨TreeSet⟨OrderableMock⟩⟩()
             {
                @Override
                public TreeSet⟨OrderableMock⟩ create() {
                        TreeSet⟨OrderableMock⟩ __var__0 = new TreeSet⟨OrderableMock⟩();
                        __var__0.insert(OrderableSort_0);
                        __var__0.insert(OrderableSort_1);
                        return __var__0;
                }
        };
        // Test the Axiom
        axiomSortedSet3_0_5_0Tester(SVar_Factory, OrderableSort_1, OrderableSort_1);
}

private void axiomSortedSet3_0_5_0Tester(
        CoreVarFactory⟨TreeSet⟨OrderableMock⟩⟩ sVar_Factory,
                OrderableMock eVar, OrderableMock fVar) {
        TreeSet⟨OrderableMock⟩ sVar_0 = sVar_Factory.create();
        TreeSet⟨OrderableMock⟩ sVar_1 = sVar_Factory.create();
        TreeSet⟨OrderableMock⟩ sVar_2 = sVar_Factory.create();
        TreeSet⟨OrderableMock⟩ sVar_3 = sVar_Factory.create();
        TreeSet⟨OrderableMock⟩ sVar_4 = sVar_Factory.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));
        }
}
```

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, which is the following:

```
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));
}
```

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. In the implementation of the SortedSet specification that is used in this example, the public insert constructor almost only invokes the private insert one, so, making public insert a suspect also makes the private insert a suspect.

EXAMPLE 2 – THE PUBLIC `insert` CONSTRUCTOR OF `SortedSet` IS ILL-IMPLEMENTED. If we test the implementation with the following fault in the public constructor `insert`:

```
public void insert(E e) {
        if (root == null || root.data.greaterEq(e))  // fault here
                root = insert(root, e);
}
```

we obtain the same tests failing as above – the ones that correspond to runs 3_0 and 5_0 – and we have to try with more than one of the model instances that Alloy gives us for the compound run until we obtain one that makes the corresponding test fail. Some of them would originate an error on the test (an exception other than `java.lang.AssertionError`). The final result tells us that we can strengthen the suspicions of guilt of the constructor, that is, the test with the negated axioms passes.

EXAMPLE 3 – WHEN TEST t_{fx0gy0} FAILS. The next case exemplifies the situation where the fault does not lie in the constructor implementation, and test t_{fx0gy0} fails, as it should, weakening the suspicions. This example is taken from another ConGu module and its respective Java implementation – `MapChain` specification and a `HashTableChain` implementation.

The fault lies in the `get` observer:

```
public Object get(Object key) {
        int index = find(key);
        if (table[index] != null) {
            return table[index].get(0); //   fault here; should be following cycle instead
            /* Search the list at table[index] to find the key.
            for (Entry nextItem : table[index]) {
                if (nextItem.key.equals(key))
                    return nextItem.value;
            } */
        }
        return null;
    }
```

As a result of this fault, some of the GenT generated tests fail; among them, the tests for runs axiomMapChain4_1 (that gives semantic to operation remove):

```
run axiomMapChain4_1  {
        some MVar : MapChain, VVar : Value, KVar, K1Var : Key | (KVar != K1Var and MVar.putOp[KVar][VVar].
            removeOp[K1Var] = MVar.removeOp[K1Var].putOp[KVar][VVar])
} for 7 but exactly 4 MapSort
```

and axiomMapChain8_0 (that gives semantic to operation put):

```
run axiomMapChain8_0  {
        some MVar : MapChain, VVar, V1Var : Value, KVar, K1Var : Key | (KVar != K1Var and MVar.putOp[KVar][
            VVar].putOp[K1Var][V1Var] = MVar.putOp[K1Var][V1Var].putOp[KVar][VVar])
} for 7 but exactly 4 MapSort
```

that correspond to axiom 4:

```
fact axiomMap4  {
        all MVar : MapChain, VVar : Value, KVar, K1Var : Key | KVar = K1Var implies MVar.putOp[KVar][VVar].
            removeOp[K1Var] = MVar.removeOp[K1Var] else MVar.putOp[KVar][VVar].removeOp[K1Var] = MVar.
            removeOp[K1Var].putOp[KVar][VVar]
}
```

and axiom 8, respectively:

```
fact axiomMap8  {
        all MVar : MapChain, VVar, V1Var : Value, KVar, K1Var : Key | KVar != K1Var implies MVar.putOp[KVar
            ][VVar].putOp[K1Var][V1Var] = MVar.putOp[K1Var][V1Var].putOp[KVar][VVar]
}
```

The reason why the execution of the corresponding tests fails is that the method `equals` of class `HashTableChain` is used to compare `HashTableChain` instances, and it uses method `get`.

Since the tests involve the same constructor operation – put –, we will join these two runs and create a compound run as explained above. A new run axiomMapChain4_1_8_0 is created:

```
run axiomMapChain4_1_8_0  {
        some MVar : MapChain, VVar, V1Var : Value, KVar, K1Var : Key |
        (KVar != K1Var and MVar.putOp[KVar][VVar].removeOp[K1Var] = MVar.removeOp[K1Var].putOp[KVar][VVar])
and
        (KVar != K1Var and MVar.putOp[KVar][VVar].putOp[K1Var][V1Var] = MVar.putOp[K1Var][V1Var].putOp[KVar
            ][VVar])
} for 7 but exactly 4 MapSort
```

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 its failure. This chosen model defines that MVar is an empty MapChain; both VVar and V1Var are Value0, and KVar is Key0 and K1Var is Key1.

Then we generate the modified test for this run and execute it. This test fails, so we weaken the suspicions of guilt of the constructor that is common to both axioms – put.

```
@Test
public void axiomMapChain4_1N_8_0N() {
                // Mocks' setup
                final ObjectMock Value0 = new ObjectMock();
                final ObjectMock Key0 = new ObjectMock();
                final ObjectMock Key1 = new ObjectMock();
                // Factories core var setup
                CoreVarFactory⟨HashtableChain⟩ MVar_Factory = new CoreVarFactory⟨HashtableChain⟩() {
                        @Override
                        public HashtableChain create() {
                                HashtableChain __var__0 = new HashtableChain();
                                return __var__0;
                        }
                };
                // Test the Axiom
                axiomMap4_1N_8_0NTester(MVar_Factory, Value0, Value0, Key0, Key1);
}

private void axiomMap4_1N_8_0NTester(
                CoreVarFactory⟨HashtableChain⟩ MVar_Factory, ObjectMock VVar,
                        ObjectMock V1Var, ObjectMock KVar, ObjectMock K1Var) {
                //axiom4 part1 negated - not (B if A) = A and not B
                HashtableChain MVar1 = MVar_Factory.create();
                MVar1.put(KVar, VVar);
                MVar1.remove(K1Var);
                HashtableChain MVar2 = MVar_Factory.create();
                MVar2.remove(K1Var);
                assertTrue((KVar == K1Var) && !(MVar1.equals(MVar2)));
                //axiom4 part2 negated - not (C if not A) = not A and not C
                HashtableChain MVar3 = MVar_Factory.create();
                MVar3.put(KVar, VVar);
                MVar3.remove(K1Var);
                HashtableChain MVar4 = MVar_Factory.create();
                MVar4.remove(K1Var);
                MVar4.put(KVar, VVar);
                assertTrue((KVar != K1Var) && !(MVar3.equals(MVar4)));
                // axiom8 negated - not (B if A) = A and not B
                HashtableChain MVar5 = MVar_Factory.create();
                MVar5.put(KVar, VVar);
                MVar5.put(K1Var, V1Var);
                HashtableChain MVar6 = MVar_Factory.create();
                MVar6.put(K1Var, V1Var);
                MVar6.put(KVar, VVar);
                assertTrue((KVar != K1Var) && !MVar5.equals(MVar6));
}
```

4.2 Working on non-constructor operations

Here we do not build a new, compound run, but instead we manipulate and modify parts of a failed test in order to try to extract further information. The idea is to try to discover whether the tests fail due to a wrong implementation of a given non-constructor operation *op* or not.

Apart from failed tests, some prior suspicion on a given operation *op* gives us a means to better focus our effort. That can be obtained, e.g., by using the strategy described in section 4.3.

The Strategy From the originally failed test – call it t –, we generate a further new test – call it t_{op} – resulting from modifying the assertions in t in such a way that the ones involving *op*, in the disjunct corresponding to the run that originated t, are negated. This new test is built over the same objects used by t, which we know to represent a model instance that satisfies the original run corresponding to t.

We claim that, if this new test passes, we can increase the suspicion of guilt of operation *op*, because it shows that the main cause of failure of the original tests was really *op*. We may also generate a further test – call it t_n – from t, by negating the assertions in t that do not involve *op* in the disjunct corresponding to the run that originated t. If this new test passes, we can weaken the suspicion of guilt of operation *op*.

Putting it to work Let t be a failed test and *op* an operation that was pointed as a suspect, for which we want to strengthen/weaken that suspicion. Let further:

– t contains operation *op*

– t originates from run R which corresponds to a given, particular, axiom, whose FDNF has m minterms
– $\alpha_t = \mathcal{T}(\alpha_R \bigvee_{1 \geq j \geq (m-1)} \alpha_{R_j})$

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). For the rest of the paper, we will 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 $Assert_R$ 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.

$$\alpha_t = \alpha_R \vee \alpha_{R_1}$$

We can now represent $Assert_{R_x}$ 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.

$$\alpha_t = (o \wedge n) \vee \alpha_{R_1}$$

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_{op} and t_n from t using the same model instance satisfying R, such that:

$$\alpha_{t_{op}} = (\eta o \wedge n) \vee \alpha_{R_1}$$

where ηo denotes the conjunction of the negation of the terms in o (remember o is a conjunction of terms involving operation *op*).

$$\alpha_{t_n} = (o \wedge \eta n) \vee \alpha_{R_1}$$

where ηn 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_{op} and t_n:

– If t_{op} passes:
  • the original problem may be caused by the method implementing operation *op* (strengthens the guilt).
– If t_n passes:
  • 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.

Some examples We give two examples where we succeed in strengthening and weakening, respectively, the guilt of an observer operation implementation, by applying the technique presented in this section.

EXAMPLE 4 – AN EXAMPLE WHERE TEST t_{op} PASSES AND t_n FAILS (CORRECTLY STRENGTHENS SUSPICION OF GUILT) This case exemplifies the situation where the fault lies in the suspected observer, which is the method `isEmpty`, and test t_{op} succeeds and test t_n fails, as it should be, strengthening the suspicions on `isEmpty`.

```
public boolean isEmpty() {
    return root != null;      // fault here! should be: return root == null;
}
```

As a result of this fault, some of the GenT generated tests fail; among them, the test for run axiomMapChain4_1, 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.

```
private void axiomSortedSet4Tester(CoreVarFactory⟨TreeSet⟨OrderableMock⟩⟩ sVarFac, OrderableMock eVar) {
        TreeSet⟨OrderableMock⟩ sVar0 = sVarFac.create();
        TreeSet⟨OrderableMock⟩ sVar1 = sVarFac.create();

        if(sVar0.isEmpty()) {
                sVar1.insert(eVar);
                assertTrue(sVar1.largest().equals(eVar));
        }
}
```

Now we transform the tested axiom to its full disjunctive normal form:

```
private void axiomSortedSet4Tester(CoreVarFactory⟨TreeSet⟨OrderableMock⟩⟩ sVarFac, OrderableMock eVar) {
        TreeSet⟨OrderableMock⟩ sVar0 = sVarFac.create();
        TreeSet⟨OrderableMock⟩ sVar1 = sVarFac.create();
        TreeSet⟨OrderableMock⟩ sVar2 = sVarFac.create();
        TreeSet⟨OrderableMock⟩ sVar3 = sVarFac.create();
        TreeSet⟨OrderableMock⟩ sVar4 = sVarFac.create();
        TreeSet⟨OrderableMock⟩ sVar5 = 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)));
}
```

and then create two new tests according to the technique presented in this section:

```
private void axiomSortedSet4TesterOp(CoreVarFactory⟨TreeSet⟨OrderableMock⟩⟩ sVarFac, OrderableMock eVar) {
        TreeSet⟨OrderableMock⟩ sVar0 = sVarFac.create();
        TreeSet⟨OrderableMock⟩ sVar1 = sVarFac.create();
        TreeSet⟨OrderableMock⟩ sVar2 = sVarFac.create();
        TreeSet⟨OrderableMock⟩ sVar3 = sVarFac.create();
        TreeSet⟨OrderableMock⟩ sVar4 = sVarFac.create();
        TreeSet⟨OrderableMock⟩ sVar5 = 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)));
}

private void axiomSortedSet4TesterN(CoreVarFactory⟨TreeSet⟨OrderableMock⟩⟩ sVarFac, OrderableMock eVar) {
        TreeSet⟨OrderableMock⟩ sVar0 = sVarFac.create();
        TreeSet⟨OrderableMock⟩ sVar1 = sVarFac.create();
        TreeSet⟨OrderableMock⟩ sVar2 = sVarFac.create();
        TreeSet⟨OrderableMock⟩ sVar3 = sVarFac.create();
        TreeSet⟨OrderableMock⟩ sVar4 = sVarFac.create();
        TreeSet⟨OrderableMock⟩ sVar5 = 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)));
}
```

Notice that the changes were made to the second minterm of the disjunction, which is the one that corresponds to the run that originated the failed test. In the t_{op} 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_{op} succeeds and test t_n fails, so we strengthen the suspicions of guilt of `isEmpty`.

EXAMPLE 5 –AN EXAMPLE WHERE t_{op} FAILS AND TEST t_n PASSES (CORRECTLY WEAKENS GUILT) This case exemplifies the situation where the fault does not lie in the suspected observer, which is the method `largest`, and test t_{op} fails and test t_n passes, as it should be, weakening the suspicions on `largest`.

We pick the exact same case as in example 4, where the real guilty method is `isEmpty`. All steps are the same except for the one where tests t_{op} and t_n are created. In t_{op} we negate, on the second minterm, the literals that include `largest`, because this is the suspect now, and in t_n we negate the literals where `largest` does not appear:

```
private void axiomSortedSet4TesterOp(CoreVarFactory⟨TreeSet⟨OrderableMock⟩⟩ sVarFac, OrderableMock eVar) {
        TreeSet⟨OrderableMock⟩ sVar0 = sVarFac.create();
        TreeSet⟨OrderableMock⟩ sVar1 = sVarFac.create();
        TreeSet⟨OrderableMock⟩ sVar2 = sVarFac.create();
        TreeSet⟨OrderableMock⟩ sVar3 = sVarFac.create();
        TreeSet⟨OrderableMock⟩ sVar4 = sVarFac.create();
        TreeSet⟨OrderableMock⟩ sVar5 = 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)));
}

private void axiomSortedSet4TesterN(CoreVarFactory⟨TreeSet⟨OrderableMock⟩⟩ sVarFac, OrderableMock eVar) {
        TreeSet⟨OrderableMock⟩ sVar0 = sVarFac.create();
        TreeSet⟨OrderableMock⟩ sVar1 = sVarFac.create();
        TreeSet⟨OrderableMock⟩ sVar2 = sVarFac.create();
        TreeSet⟨OrderableMock⟩ sVar3 = sVarFac.create();
        TreeSet⟨OrderableMock⟩ sVar4 = sVarFac.create();
        TreeSet⟨OrderableMock⟩ sVar5 = 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)));
}
```

Notice that, by coincidence, these tests are perfectly equal to the ones in the previous example (except for their meaning to the results). This happens because the literals where `largest` appears are exactly the same as the ones where `isEmpty` does not appear and vice-versa.

Test t_{op} fails and test t_n succeeds, so we weaken the suspicions of guilt of `largest`.

4.3 Test failure caused by an exception other than `java.lang.AssertionError`

Here we reason about the clues we may get from a test failure due to an unchecked exception as, for example, `NullPointerException`, `ArrayIndex OutOfBounds Exception`, etc. Apart from the obvious suspect (if it exists) – the method belonging to the class implementing the designated sort 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 corrected 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, the following reasoning may apply:

– the method where the exception is thrown is ill-implemented or
– some method used in its pre-condition is ill-implemented causing the pre-condition to evaluate to true instead of false, thus letting the system try to execute another operation when it should not, causing it to abort, or
– some method that is used to calculate some of the arguments of the aborted method is ill-implemented or wrongly used, giving a result that the latter cannot absorb.

We must observe the exception messages stack 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 sort, thus a mock, correct implementation is used – we must investigate the parameters used in its invocation.

Furthermore, if we have other tests that also abort from some unexpected 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_1 that launched the exception:
      i. if m_1 is of the designated sort, make m this method
      ii. else, find the(a) method(s) that is(are) used to compose the parameter(s) of m_1 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 wrong tests to obtain the most probable guilty operations.

The succession of steps 1.(a).i.(b) covers the cases where the method m that launched the exception is ill-implemented (example 6 below); 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 7 below); the succession of steps 1.(a).ii.(b).(c).i covers the cases in which method m failed because other methods that were invoked before it were wrongly invoked (their pre-conditions were satisfied when they should not) and their anomalous results originated an *a posteriori* exception (example 8 below).

Some examples Some examples follow that cover the cases referred above.

Example 6 – the fault lies in the method where the exception is launched If we test the `TreeSet` implementation with the following fault in `largest`:

```
public E largest() {
    if (root == null)
        return null;
    Node⟨E⟩ rightMost = root;
    while (rightMost != null) // fault here! - should be: while (rightMost.right != null)
        rightMost = rightMost.right;
    return rightMost.data;
}
```

we only obtain failures caused by exceptions other than `java.lang.AssertionError`. The specific exception in this case is `NullPointerException` in tests 4_0, 5_0, 5_2, 5_5, 6_0, 6_1 and 6_2. All these tests invoke the `largest` method, and they all point to this method as the source of the exception:

```
Failure Trace
java.lang.NullPointerException
        at SortedSet.TreeSet.largest(TreeSet.java:66)
        at SortedSet.TreeSet2.axiomSortedSet4Tester(TreeSetTest.java:198)
        at SortedSet.TreeSet2.test0_axiomSortedSet4_0(TreeSetTest.java:198)
```

According to the algorithm above, the `largest` method is added to the suspects list.

Example 7 – the guilty operation is used in the pre-condition of the method that launched the exception In the SortedSet example, let method `isEmpty` be faulty and method `largest`, whose pre-condition is `!isEmpty`, be implemented in such a way that it launches `ArrayIndexOutOfBoundsException` when the set is empty.

If the fault is such that the `isEmpty` method gives a **`false`** result when executed over an empty set, some test may fail with that exception, and the `largest` method is the one that the exception message stack points to. According to the algorithm above, the `largest` and `isEmpty` methods are added to the suspects list.

Example 8 – the guilty operation is used in the pre-condition of a method that is used to obtain a parameter of the method where the exception is launched In the SortedSet example, let method `isEmpty` be faulty and method `largest`, whose pre-condition is `!isEmpty`, be implemented in such a way that it returns **`null`** whenever `isEmpty` returns true. If an empty set appears to be non-empty due to the faulty `isEmpty` method, an exception may subsequently occur, not when calling the `largest` method, but when its **`null`** result is compared with some other `orderable` value. In this case a `NullPointerException` is launched while executing the comparison method, which has no pre-conditions.

If we test the implementation with the following fault in `isEmpty`:

```
public boolean isEmpty() {
    return root != null;      // fault here! should be: return root == null;
}
```

we obtain `NullPointerException` in tests 5_5 and 6_4. These tests have their assertion testing protected by a condition. For example, the tester for axiom 5 is:

```
private void axiomSortedSet5Tester(CoreVarFactory⟨TreeSet⟨OrderableMock⟩⟩ sVarFac,OrderableMock eVar) {
        TreeSet⟨OrderableMock⟩ sVar_0 = sVarFac.create();
        TreeSet⟨OrderableMock⟩ sVar_1 = sVarFac.create();
        TreeSet⟨OrderableMock⟩ sVar_2 = sVarFac.create();

        if(!sVar_0.isEmpty() && eVar.greaterEq(sVar_1.largest())) {
                sVar_2.insert(eVar);
                assertTrue(sVar_2.largest().equals(eVar));
        }
}
```

According to the exception message stack,

```
Failure Trace
java.lang.NullPointerException
        at SortedSet.OrderableMock.greaterEq(OrderableMock.java:17)
        at SortedSet.TreeSet2.axiomSortedSet5Tester(TreeSetTest.java:293)
        at SortedSet.TreeSet2.test0_axiomSortedSet5_5(TreeSetTest.java:379)
```

the exception is raised in the second condition of the **`if`** guard – `eVar.greaterEq(sVar_1.largest())`. The operation `greaterEq` is not a suspect because it is not a designated sort operation, meaning that we are only using a mock implementation of it (which, by definition, is corrrect). So, the guilty method cannot be `greaterEq` and we turn our suspicion to the parameter that is fed to it, which involves the `largest` method. As a consequence, we add `largest` to the suspects list.

Furthermore, the operation `largest` has a pre-condition involving `isEmpty`, which says that the operation `largest` is defined when the sorted set is not empty, and undefined otherwise. This leads us to add `isEmpty` to the suspects list.

The exact same behaviour happens in test 6_4. The tester for this axiom is:

```
private void axiomSortedSet6Tester(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();

        if(!sVar_0.isEmpty() && !eVar.greaterEq(sVar_1.largest())) {
                sVar_2.insert(eVar);
                assertTrue(sVar_2.largest().equals(sVar_3.largest()));
        }
}
```

and the raised exception is the same. The reasoning is also the same as above.

4.4 Abstract VS Concrete objects

As already discussed, in a situation where all methods are equaly suspect, the comparison between abstract (correctly behaved) objects and concrete (observed behaviour) ones can be unworthy, due to the eventual lack of trust on the results given by observer methods.

Whenever we have a particular suspect, however, we may consider that it is relatively safe to use the other, not previously suspect, methods to inspect the concrete objects and, in that way, be able to compare their characteristics with the ones of the corresponding abstract objects.

The Strategy We want to be able to compare the Java objects obtained from executing the methods that implement a given ADT's operations, with corresponding abstract objects that result from applying those abstract operations. So, we must build the abstract objects and be able to inspect them too; we want to do this in the most cost effective way.

The idea here is to use the models found by the Alloy Analyzer that satisfy the specification, to obtain the information we need to verify the correctness of the corresponding Java objects.

A Java (mock) class is created whose instances represent Alloy model objects of the designated sort, that has the same public, specified, methods as the class that implements that sort (although some of them have different signatures as we will see ahead). As many instances of that class are built as the number of objects of this sort defined by the model. The methods, when invoked over an instance of this class, will give the same results as the ones determined by the Alloy model.

We then build the concrete objects that correspond to the ones defined by the Allloy model, using the Java methods that implement ADT operations, and compare them, step by step, with the abstract corresponding ones.

We use ordinary mock classes for all the non-designated sorts of the specification module (some of them will have already been created by the GenT tool, namely the ones for the parameter sorts). During the comparison, we only use concrete objects (ones created and manipulated by the Java concrete implementation methods) for the designated sort; any object from other sort will be a mock one, therefore no concrete versus abstract comparison will be executed for its implementation.

Putting it to work A new test is built that compares abstract with concrete objects. For this, we must first generate mock classes for the sorts that compose the specification module. If the module contains parameterized specifications, we already have mock classes for the parameter sorts. The mock class for the designated sort is different from the others because the signatures of some of its methods are different from the ones of the corresponding concrete methods. No problem comes from that because its instances are only used in the context of this kind of tests.

As an example, a mock class is generated for the SortedSet implementation – `TreeSet`.

```
public class TreeSetMock ⟨T⟩{
        private HashMap⟨T,Boolean⟩ isInResult = new HashMap⟨T,Boolean⟩();
        private boolean isEmptyResult;
        private T largestResult;

        @Override
        public boolean isIn (T e){
                return isInResult.get(e);
        }

        public void add_isIn (T e, Boolean result){
                isinResult.put (e, result);
        }

        @Override
        public boolean isEmpty(){
                return isEmptyResult;
        }

        public void add_isEmpty(boolean result){
                isEmptyResult = result;
        }

        @Override
        public T largest(){
                return largestResult;
        }

        public void add_largest(T result){
                largestResult = result;
        }

        // to be continued
}
```

Listing 1.9. Part of the mock class corresponding to the SortedSet sort.

This mock class brings nothing new in what concerns methods that implement ADT observer operations whose result type is different from the designated sort. The novelty lies in methods that implement non-creator constructors and observers whose result is of the SortedSet type. Before entering that subject, let us see how we can use the simple, non-problematic observers to compare abstract with concrete objects.

We generate an Alloy model for the whole Alloy specification; we do this by executing the Alloy run command with an empty body. Figure 4 shows the first model obtained by executing the Alloy command: run {} for 6 but exactly 4 SortedSet.

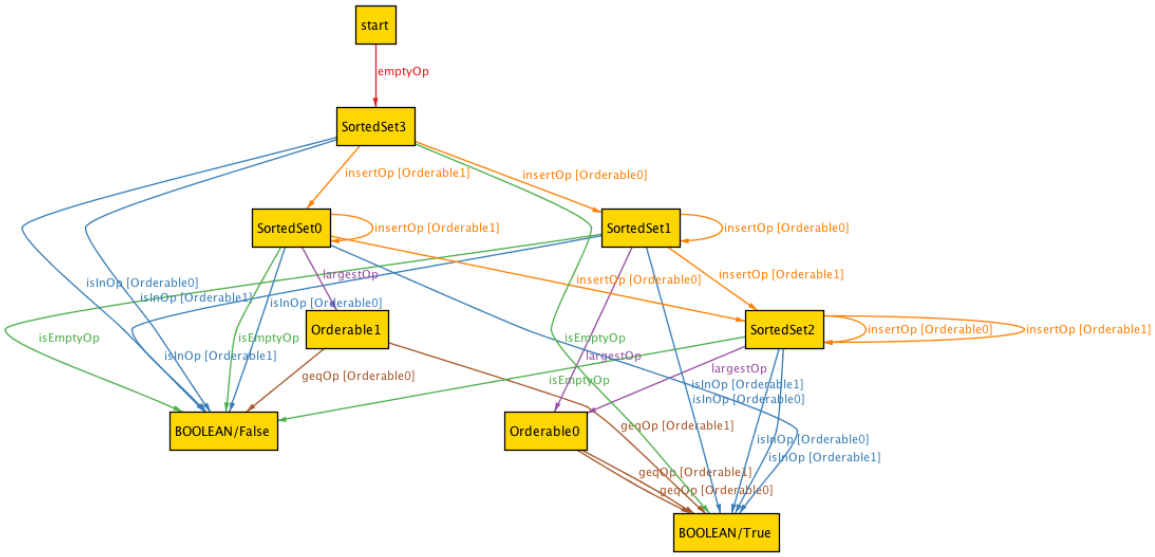

Fig. 4. A model for the SortedSet specification found by the Alloy Analyzer.

A test class is generated that creates, inspects and compares the abstract and concrete objects corresponding to the obtained model. The abstract Java world is first populated with the abstract objects: two `OrderableMock` instances and four `TreeSetMock` ones, as indicated by the Alloy model below, reflecting each and every object of the model and their inter-relationships. The concrete Java world objects will also be built, using the corresponding methods, and comparisons will be made between each pair ⟨abstract, concrete⟩ of objects of the designated sort. For now, we only present part of the test class, the one that concerns the above observers:

```
@Test
public void test0 ()
{
        //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);

        //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⟩();
        ...
            // to be continued
}
```

Listing 1.10. (Part of) Abstract versus concrete objects test.

In order to be possible to give adequate feedback to the user concerning the differences between abstract and concrete objects, that is, between the results concrete Java methods should obtain and the results they really obtain, we must be able to test all the `assert` commands. A possible solution is to enclose each `assertTrue` invocation in a **`try-catch`** block that collects all results which will help composing a final test diagnosis.

Now we see how to cope with methods implementing non-creator constructors and SortedSet result type observers.

Constructor operations The GenT tool generates a test for each of the runs discussed in section 3.2, where objects found to verify the assertion are built using the corresponding methods in Java. Whenever there are several ways to build an object, GenT chooses the shortest path in the model. Here, we also pick the shortest path to build the designated sort objects of the model, and, by applying the observer operations to their abstract and concrete versions, test the several methods (see listings 1.10 and 1.11). Notice that, in order to cope with undesired side effects of methods, as many copies of a given concrete object are created as mehods applied to it.

```
@Test
public void test0 ()
{
    ...
        //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));

        //Create and compare with corresponding abstract
        TreeSet⟨OrderableMock⟩ concTree1 = new TreeSet ⟨OrderableMock⟩();
        TreeSet⟨OrderableMock⟩ concTree1_1 = new TreeSet ⟨OrderableMock⟩();
        TreeSet⟨OrderableMock⟩ concTree1_2 = new TreeSet ⟨OrderableMock⟩();
        TreeSet⟨OrderableMock⟩ concTree1_3 = new TreeSet ⟨OrderableMock⟩();
        TreeSet⟨OrderableMock⟩ concTree1_4 = new TreeSet ⟨OrderableMock⟩();
        concTree1.insert(orderable0);
        concTree1_1.insert(orderable0);
        concTree1_2.insert(orderable0);
        concTree1_3.insert(orderable0);
        concTree1_4.insert(orderable0);
        assertTrue(concTree1_1.isEmpty() == absTree1.isEmpty());
        assertTrue(concTree1_2.largest() == absTree1.largest());
        assertTrue(concTree1_3.isIn(orderable1) == absTree1.isIn(orderable1));
        assertTrue(concTree1_4.isIn(orderable0) == absTree1.isIn(orderable0));

        //Create and compare with corresponding abstract
        TreeSet⟨OrderableMock⟩ concTree2 = new TreeSet ⟨OrderableMock⟩();
        TreeSet⟨OrderableMock⟩ concTree2_1 = new TreeSet ⟨OrderableMock⟩();
```

```
        TreeSet⟨OrderableMock⟩ concTree2_2 = new TreeSet ⟨OrderableMock⟩();
        TreeSet⟨OrderableMock⟩ concTree2_3 = new TreeSet ⟨OrderableMock⟩();
        TreeSet⟨OrderableMock⟩ concTree2_4 = new TreeSet ⟨OrderableMock⟩();
        concTree2.insert(orderable1);
        concTree2.insert(orderable0);
        concTree2_1.insert(orderable1);
        concTree2_1.insert(orderable0);
        concTree2_2.insert(orderable1);
        concTree2_2.insert(orderable0);
        concTree2_3.insert(orderable1);
        concTree2_3.insert(orderable0);
        concTree2_4.insert(orderable1);
        concTree2_4.insert(orderable0);
        assertTrue(concTree2_1.isEmpty() == absTree2.isEmpty());
        assertTrue(concTree2_2.largest() == absTree2.largest());
        assertTrue(concTree2_3.isIn(orderable1) == absTree2.isIn(orderable1));
        assertTrue(concTree2_4.isIn(orderable0) == absTree2.isIn(orderable0));

}
```

Listing 1.11. (Part of) Abstract versus concrete objects test.

The reader might wonder why is it that we compare `OrderableMock` objects by using == instead of `equals`, knowing that their type is not a primitive one? The answer is simple to understand, and is given by steps. Knowing that:

- in the generic class `TreeSet⟨T⟩`, that implements the designated sort SortedSet, there is no way to create new parameter objects unless `clone` is used, because it is not possible to have **`new`** `T()` (this means that, in cases where parameter types are not obliged to be cloneable, there is no way a generic class can create new parameter objects);
- we use the `OrderableMock` type to instantiate the parameter type variables of both `TreeSetMock` and `TreeSet` objects, that is, all parameter objects that populate the sets in the test are of type `OrderableMock`;
- we create only as many parameter objects as defined by the Alloy model (two in this example test: `orderable0` and `orderable1`)

we may conclude that:

- all methods, abstract and concrete, that implement observers whose result is of Orderable sort, return `OrderableMock` objects;
- all these methods can only return one of `orderable0` and `orderable1`. So, we may compare `OrderableMock` abstract and concrete results by using ==.

Returning to the question of the constructor operations, we should also test their other applications in order to be able to test the dynamics of the construction process that leads to a given object – 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 that the constructor is ill-implemented in such a way that it 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; more generally, no constructor is used that results in the same object to which it is applied to.

Since the other applications of the constructors do not aim at the construction of new, different, objects, we are going to treat them as observations. In this way, we can adopt the same reasoning as the one we will apply ahead for observers with designated sort result.

So, let us now study the case, not covered above, where the result of an observer is from a type implementing the designated sort of the ConGu module.

Observer operations with designated sort result The Alloy model defines exactly what the objects are that result from each operation defined in the several sorts of the specification, when applied to each object of those sorts. Because we create a distinct variable to hold the reference for each of the abstract objects, we may consider that the name of the variable identifies the abstract object it refers to (none of these variables and objects are ever modified).

The references of the concrete objects that are created are kept within variables whose names can be paired to the ones of the variables that refer to corresponding abstract objects. Then, it is enough to check whether the concrete objects we get as a result of Java observers are equal (using Java method `equals`) to the concrete

ones whose references are held by the variables whose names correspond to the ones that hold the corresponding abstract results.

As an example, 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 several observers, namely `obsA` observer and its Java implementation `mObsA`. If the chosen Alloy model instance determines that `obsA` result, when applied to the abstract object whose reference is kept in variable `absT1`, is the abstract object whose reference is kept in variable `absT2`, then we will test that the result of the Java method `mObsA`, when applied to the object referred to by variable `concT1`, is equal to the one whose reference is kept in variable `concT2`, that is, we will test whether `(concT1.mObsA()).equals(concT2)` is true.

We are not comparing whether the `concT1.mObsA()` result object and the one referred to by `concT2` are *observationally* the same, that is, we do not explicitly compare the results of applying the observers to each one of them – we use `equals` instead. Observational equivalence is applied only between abstract and concrete objects (in this case, between `absT1` and `concT1`), and it is inquired at the first level of depth only.

Finishing the work In Java there is no way to declare a local variable whose name is not known at compile time, not even using reflection. But we need to decide, at runtime, what the concrete object is we must inspect, based on the name of a variable containing the reference for an abstract object. So, whenever we store the abstract object that results from applying a given observer to a given abstract object, in cases where the given observer result is of the designated sort, we must also store the corresponding concrete object. Instead of storing the name of the corresponding concrete variable, we store the reference to the concrete object itself.

Suppose our SortedSet ADT defines observer operation removeLargest: SortedSet ⟶SortedSet, 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 like the one we have been using); *(ii)* `TreeSet` (tipically used in immutable implementations) and *(iii)* another different type (e.g. **`int`** in order to return some error code, `T` to return the element that is removed from the set, etc). This last 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. In what concerns case *(ii)*, a mutable implementation can also apply – for example, method `removeLargest` changes the state by removing the largest element and it returns the previous state as its result – and ConGu allows the implementer to define which one should apply.

We only compare what is relevant for our purpose, which are the abstract object that the Alloy model determines as being the result of the operation, and 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 case *ii*, in an immutable implementation).

For this purpose, the `TreeSetMock` class must declare variables and methods to store the (abstract) result that the Alloy model establishes for this operation for each instance of SortedSet, as well as the `TreeSet` concrete object that corresponds to that abstract object. Whatever the case (*i*, *ii* or *iii*), those are the needs.

The technique that follows also applies to the observations we decided to perform over SortedSet and corresponding `TreeSet` objects using the non-creator constructor insert. So, the examples will also include this method, showing how it can be treated as a designated sort observer.

In order to add the constructor and the new observer to our running example, we modify the `TreeSetMock` class by adding an inner class `Pair_TreeSetMock_TreeSet` to it. Furthermore, we define two mock observer "versions" of the new methods 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 ();
        private HashMap⟨T,Pair_TreeSetMock_TreeSet⟩ insertResult = new HashMap⟨T,Pair_TreeSetMock_TreeSet⟩
             ();
...
        @Override
        public TreeSet⟨T⟩ removeLargest_C (){
                return removeLargestResult.cVal;
```

```
        }
        public TreeSetMock⟨T⟩ removeLargest_A (){
                return removeLargestResult.aVal;
        }
        public void add_removeLargest (TreeSetMock⟨T⟩ absVal, TreeSet⟨T⟩ concVal){
                removeLargestResult.setAbsVal(absVal);
                removeLargestResult.setConcVal(concVal);
        }
        @Override
        public TreeSet⟨T⟩ insert_C (T e){
            return insertResult.get(e).cVal;
        }
        public TreeSetMock⟨T⟩ insert_A (T e){
            return insertResult.get(e).aVal;
        }
        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);
        }
...
} // end of class TreeSetMock
```

Listing 1.12. `TreeSetMock` class with a designated sort observer and a non-creator constructor.

Let us refactor our example in listing 1.10 to make it general enough to cope with the existence of this kind of observers. Because we need abstract and concrete objects when we add results of operations to the mock objects, then, unlike the way we did before, here we must first create all abstract and concrete objects, and only after this can we add the results of abstract operations to the mock objects. As before, initial concrete objects of the designated sort are built according to the shortest path to the Alloy model objects.

```
//Abstract and concrete objects TreeSet3
TreeSetMock ⟨OrderableMock⟩ absTree3 = new TreeSetMock ⟨OrderableMock⟩();
TreeSet⟨OrderableMock⟩ concTree3 = new TreeSet ⟨OrderableMock⟩();
.... //Create other copies of concTree3

//Abstract and concrete objects TreeSet0
TreeSetMock⟨OrderableMock⟩ absTree0= new TreeSetMock ⟨OrderableMock⟩();
TreeSet⟨OrderableMock⟩ concTree0 = new TreeSet ⟨OrderableMock⟩();
concTree0.insert(orderable1);
.... //Create other copies of concTree0

//Abstract and concrete objects TreeSet1
TreeSetMock ⟨OrderableMock⟩ absTree1 = new TreeSetMock ⟨OrderableMock⟩();
TreeSet⟨OrderableMock⟩ concTree1 = new TreeSet ⟨OrderableMock⟩();
concTree1.insert(orderable0);
.... //Create other copies of concTree1

//Abstract and concrete objects TreeSet2
TreeSetMock ⟨OrderableMock⟩ absTree2 = new TreeSetMock ⟨OrderableMock⟩();
TreeSet⟨OrderableMock⟩ concTree2 = new TreeSet ⟨OrderableMock⟩();
concTree2.insert(orderable1);
concTree2.insert(orderable0);
.... //Create other copies of concTree2
```

Listing 1.13. Creating all abstract and concrete objects.

Next we must define the results of observers over the abstract objects using the `add_xObserver` methods of the mock classes. The operation removeLargest was not part of the original running example due to simplification issues. In the meanwhile, we introduced it as a way to exemplify the case of an observer with a designated sort result type, without modifying the presented specifications. However, in order to continue with the present explanation, we need to define the abstract behaviour of this observer; we do it by giving its expected results when applied to the designated sort abstract objects composing the model:

- `removeLargest(absTree3)` is not defined
- `removeLargest(absTree0)` is equal to `absTree3`
- `removeLargest(absTree1)` is equal to `absTree3`
- `removeLargest(absTree2)` is equal to `absTree0`

According to this and the original example:

```
//Adding results of observers to abstract objects
absTree3.add_isEmpty(true);
//absTree3 does not satisfy domain condition of largest
absTree3.add_isIn(orderable0, false);
```

```
absTree3.add_isIn(orderable1, false);
//absTree3 does not satisfy domain condition of removeLargest
//the two inserts were already applied; no need to further observe

absTree0.add_isEmpty(false);
absTree0.add_largest(orderable1);
absTree0.add_isIn(orderable0, false);
absTree0.add_isIn(orderable1, true);
absTree0.add_removeLargest(absTree3, concTree3);
// insert of orderable0 was already applied; no need to further observe
absTree0.add_insert(orderable1, absTree0, concTree0);

absTree1.add_isEmpty(false);
absTree1.add_largest(orderable0);
absTree1.add_isIn(orderable0, true);
absTree1.add_isIn(orderable1, false);
absTree1.add_removeLargest(absTree3, concTree3);
absTree1.add_insert(orderable0, absTree1, concTree1);
absTree1.add_insert(orderable1, absTree2, concTree2);

absTree2.add_isEmpty(false);
absTree2.add_largest(orderable0);
absTree2.add_isIn(orderable0, true);
absTree2.add_isIn(orderable1, true);
absTree2.add_removeLargest(absTree0, concTree0);
absTree2.add_insert(orderable0, absTree2, concTree2);
absTree2.add_insert(orderable1, absTree2, concTree2);
```

Listing 1.14. Adding results of observers to abstract objects.

Finally, we compare the abstract and concrete objects using all possible observers. Remember that if a given observer is the suspect one, we must treat it differently, that is, the way `assertTrue` violations are interpreted must have this in mind.

```
//Comparing results of observer isEmpty()
assertTrue(concTree3_1.isEmpty() == absTree3.isEmpty());
assertTrue(concTree0_1.isEmpty() == absTree0.isEmpty());
assertTrue(concTree1_1.isEmpty() == absTree1.isEmpty());
assertTrue(concTree2_1.isEmpty() == absTree2.isEmpty());

//Comparing results of observer largest()
assertTrue(concTree0_2.largest() == absTree0.largest());
assertTrue(concTree1_2.largest() == absTree1.largest());
assertTrue(concTree2_2.largest() == absTree2.largest());

//Comparing results of observer isIn()
assertTrue(concTree3_2.isIn(orderable1) == absTree3.isIn(orderable1));
assertTrue(concTree3_3.isIn(orderable0) == absTree3.isIn(orderable0));
assertTrue(concTree0_3.isIn(orderable1) == absTree0.isIn(orderable1));
assertTrue(concTree0_4.isIn(orderable0) == absTree0.isIn(orderable0));
assertTrue(concTree1_3.isIn(orderable1) == absTree1.isIn(orderable1));
assertTrue(concTree1_4.isIn(orderable0) == absTree1.isIn(orderable0));
assertTrue(concTree2_3.isIn(orderable1) == absTree2.isIn(orderable1));
assertTrue(concTree2_4.isIn(orderable0) == absTree2.isIn(orderable0));

// to be continued...
```

Listing 1.15. Comparing results of some of the 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.

We must follow these steps:

- first we invoke the concrete method (which has **`void`** return type) using the concrete object as a target;
- then we compare (using `equals`) the new state of the concrete object (it has eventually been modified by the observer method) with the concrete object that, according to the mock corresponding object, should be the correct result (in the `removeLargest` example, we invoke the mock `TreeSet⟨T⟩removeLargest_C()` method to obtain this object).

```
//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 1.16. Comparing results of the designated sort observers.

As expected, although it is not shown in any of the previous listings, the exact same steps that we followed to obtain the original concrete objects, are taken to obtain the copies of the original concrete objects we use as targets for the application of these observers.

The algorithm To conclude, 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 sorts of the specifications in the module (the ones corresponding to parameter sorts were already built by GenT):
  - mock classes must give their instances the ability to store and retrieve the results of applying all observers;
  - mock classes for non-designated non-parameter sorts are built as usual; the mock class for the designated sort is somewhat different since it must be prepared to facilitate the comparison between concrete objects:
    - for all observers whose result is of the designated sort as well as for non-creator constructors, mock objects must be able to store and retrieve not only the specific abstract result but also the corresponding concrete object;
- Generate a test class to compare the abstract with the concrete objects, for a given suspect method $spct$:
  - compose the abstract objects, as defined by the Alloy model:
    - as many abstract objects are created as there are instances of the specified sorts in the Alloy model;
    - they are created by instantiating the corresponding mock classes and by applying them the constructors defined by the shortest path in the model;
    - they are "fed" with the information about its behaviour, as defined by the Alloy model (for the designated sort, this exercise depends on concrete objects having already been created);
  - create the corresponding concrete objects:
    - concrete objects $concX_{ij}$ are created of the designated sort – at least one for each abstract object $absX_i$ of the designated sort –, by using the concrete class constructors corresponding to the ones used to define $absX_i$;
    - the number of $concX_{ij}$ concrete objects of the designated sort that must be created depends on the number of observer applications that must be executed in order to compare this concrete instance with the abstract one;
    - there are no concrete objects of non-designated sorts, only mock ones;
  - compare the abstract with the concrete objects:
    - apply to the concrete objects all the observer methods (including the non-creator constructor) that should be possible to apply, as indicated by the Alloy model, while comparing the results with the corresponding mock methods' results when executed over abstract objects;
    - comparisons involving observers whose result is of a primitive sort, or of any of the specified sorts which are not the designated sort, are made using == between the results of applying the observer to the concrete and to the abstract objects;
    - comparisons involving observers whose result is of the designated sort, or non-creator constructors, are made using `equals` between the result of applying the method, and the concrete object stipulated by the mock abstract object as being a correct result for that method;
    - be aware of the suspect method $spct$: if it is an observer, the comparisons involving it must supply a distinctive output (because it is, *a priori*, less reliable)
- Run the test class for a given suspect method

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 the guilt lies in the constructor or in 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* that should be used is given by the following:

- If only a given observer ob fails:
  - decide ob is the faulty method
- If several and varied observers fail:
  - if those failures also happen when observers are applied to a constructor-creator cc method:
    - decide cc is the faulty method
  - else (*observers do not fail when applied to any constructor-creator*) if some of the observations made by a non-creator constructor ncc (when used as an observer) fail:
    - decide ncc is the faulty method
  - else
    - inconclusive

If a constructor-creator cc (`TreeSet()` that implements the empty creator operation in the running example) is faulty, it is reasonable to think that the application of the other constructors over an object created by cc will most probably result in non-conformant objects, because the initial object was ill-built.

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

This reasoning must account for the initial suspect that was input to the test; if it is an observer, its results should be deemed as non-reliable.

An example We applied this tecnique to all our case studies with excellent results. As an example, the case where the `insert` operation was ill-implemented and for which there were two suspects – methods `isEmpty` and `insert`.

We applied the abstract-concrete comparison presented in this section to the two suspects, independently from each other, and got an unequivocal result that pointed to a faulty implementation of the `insert` operation.

None of the direct observations over concrete objects freshly created by the Java constructor `TreeSet()` (the one that implements the constructor-creator empty) failed.

When applied to concrete objects that resulted from the `insert` constructor, all observers gave results that were different from the ones expected, that is, from the ones obtained by the application of their corresponding mock methods to the corresponding abstract objects.

This tells us that the concrete objects that were built by application of the `insert` method were wrongly built, and that the problem does not come from some particular way we look at the objects. If the implementation of `isEmpty` were wrong, one would not expect all comparisons to be false, but only the ones involving `isEmpty`.

What if one cannot trust `equals`? The technique just presented depends on the correctness of the `equals` method. If there are reasons to suspect that it is ill-implemented or that it may depend on some other suspicious method, using it to evaluate other methods' correctness can be misleading (see the discussion on the *oracle problem* in section 7). As an example, we obtained poor results when using this technique over a faulty implementation of the MapChain case study, where a fault in method `get` affected the results of `equals` due to the fact that the latter used the former (see section 6).

We consider here an alternative way of comparing designated sort concrete objects, that does not rely on the correcteness of `equals`. The idea is to only use non-designated sort observers over concrete objects; the results of these observers can be compared using == because we work with a finite set of immutable mock objects, as already explained.

The difference lies essentially in the way we ensure that designated sort concrete objects resulting from concrete operations are the correct ones: we observe them with non-designated sort observers only. The concrete objects we must observe are:

- initial objects obtained by following minimum paths in the model;

– results of designated sort observers;
– objects obtained from additional applications of non-creator constructors (the ones corresponding to non-minimum paths).

We work with a finite universe of designated sort objects – the ones defined by the Alloy model of the specification –, so the correct outcomes of all operations whose result is of the designated sort must be within this universe. Since we apply all operations defined in the model (not just the ones corresponding to the minimum path for each object), then if we observe all corresponding concrete operations applications against corresponding abstract objects, we obtain the greatest coverage that is possible to obtain with the available observers.

Initial objects obtained by following minimum paths in the model are compared with corresponding abstract ones exactly as shown in listing 1.15. The abstract correspondents of the other designated sort concrete objects are obtained by invoking the "A" versions of the proper mock class operations. For example, if we want to ensure that the state of concrete object `concTree0_0` (one of `concTree0` "clones") after application of `insert(orderable0)`, is as expected, we should compare it to the corresponding abstract object, which we obtain by applying `insert_A(orderable0)` to `absTree0` (the abstract correspondent of `concTree0`).

Applying this reasoning to our running example would entail the replacement of the instructions in listing 1.16 by the ones in listing 1.17. For space reasons we simplify this example by sequentially applying several observer operations to the same concrete object; however, that is not the real procedure – as already presented, each observer operation is to be applied to a copy of the concrete object in question.

```
// Comparing results for insert
//conc0_0
concTree0_0.insert(orderable1);
assertTrue((concTree0_0.isIn(orderable0) == absTree0.insert_A(orderable1).isIn(orderable0)));
assertTrue((concTree0_0.isIn(orderable1) == absTree0.insert_A(orderable1).isIn(orderable1)));
assertTrue((concTree0_0.isEmpty() == absTree0.insert_A(orderable1).isEmpty()));
if (absTree0.insert_A(orderable1).largest()) != null)
    assertTrue((concTree0_0.largest() == absTree0.insert_A(orderable1).largest()));
//conc1_0
concTree1_0.insert(orderable0);
assertTrue((concTree1_0.isIn(orderable0) == absTree1.insert_A(orderable0).isIn(orderable0)));
assertTrue((concTree1_0.isIn(orderable1) == absTree1.insert_A(orderable0).isIn(orderable1)));
assertTrue((concTree1_0.isEmpty() == absTree1.insert_A(orderable0).isEmpty()));
if (absTree1.insert_A(orderable0).largest()) != null)
    assertTrue((concTree1_0.largest() == absTree1.insert_A(orderable0).largest()));
//conc1_1
concTree1_1.insert(orderable1);
assertTrue((concTree1_1.isIn(orderable0) == absTree1.insert_A(orderable1).isIn(orderable0)));
assertTrue((concTree1_1.isIn(orderable1) == absTree1.insert_A(orderable1).isIn(orderable1)));
assertTrue((concTree1_1.isEmpty() == absTree1.insert_A(orderable1).isEmpty()));
if (absTree1.insert_A(orderable1).largest()) != null)
    assertTrue((concTree1_1.largest() == absTree1.insert_A(orderable1).largest()));
//conc2_0
concTree2_0.insert(orderable0);
assertTrue((concTree2_0.isIn(orderable0) == absTree2.insert_A(orderable0).isIn(orderable0)));
assertTrue((concTree2_0.isIn(orderable1) == absTree2.insert_A(orderable0).isIn(orderable1)));
assertTrue((concTree2_0.isEmpty() == absTree2.insert_A(orderable0).isEmpty()));
if (absTree2.insert_A(orderable0).largest()) != null)
    assertTrue((concTree2_0.largest() == absTree2.insert_A(orderable0).largest()));
//conc2_1
concTree2_1.insert(orderable1);
assertTrue((concTree2_1.isIn(orderable0) == absTree2.insert_A(orderable1).isIn(orderable0)));
assertTrue((concTree2_1.isIn(orderable1) == absTree2.insert_A(orderable1).isIn(orderable1)));
assertTrue((concTree2_1.isEmpty() == absTree2.insert_A(orderable1).isEmpty()));
if (absTree2.insert_A(orderable1).largest()) != null)
    assertTrue((concTree2_1.largest() == absTree2.insert_A(orderable1).largest()));
// Comparing results for removeLargest
//conc0_2
concTree0_2.removeLargest();
assertTrue((concTree0_2.isIn(orderable0) == absTree0.removeLargest_A().isIn(orderable0)));
assertTrue((concTree0_2.isIn(orderable1) == absTree0.removeLargest_A().isIn(orderable1)));
assertTrue((concTree0_2.isEmpty() == absTree0.removeLargest_A().isEmpty()));
if (absTree0.removeLargest_A().largest()) != null)
    assertTrue((concTree0_2.largest() == absTree0.removeLargest_A().largest()));
//conc1_2
concTree1_2.removeLargest();
assertTrue((concTree1_2.isIn(orderable0) == absTree1.removeLargest_A().isIn(orderable0)));
assertTrue((concTree1_2.isIn(orderable1) == absTree1.removeLargest_A().isIn(orderable1)));
assertTrue((concTree1_2.isEmpty() == absTree1.removeLargest_A().isEmpty()));
if (absTree1.removeLargest_A().largest()) != null)
    assertTrue((concTree1_2.largest() == absTree1.removeLargest_A().largest()));
//conc2_2
concTree2_2.removeLargest();
assertTrue((concTree2_2.isIn(orderable0) == absTree2.removeLargest_A().isIn(orderable0)));
assertTrue((concTree2_2.isIn(orderable1) == absTree2.removeLargest_A().isIn(orderable1)));
```

```
assertTrue((concTree2_2.isEmpty() == absTree2.removeLargest_A().isEmpty()));
if (absTree2.removeLargest_A().largest()) != null)
    assertTrue((concTree2_2.largest() == absTree2.removeLargest_A().largest()));
```

Listing 1.17. Comparing results of the designated sort operations

As before, the `insert(orderable0)` operation is not applied to `concTree0` because that would repeat the process with which `concTree2` was obtained in the first place (see listing 1.11) – inserting `orderable1` and `orderable0`, in that order, to the empty sorted set. The same applies to `concTree3`.

Being a partial operation, the `largest` observer should never be applied to empty sorted sets as already shown for `concTree3` in listing 1.15. The same applies to `removeLargest` that is only applied to `concTree0`, `concTree1`, `concTree2` and their copies. This is not difficult to ensure because these objects were built according to the model, which defines which operations apply and which do not. But in cases where we have to apply partial operations to objects that are the result of some other operation, we should guard the application of every partial operation with a condition that somehow verifies its domain. This is to say that the several applications of the `largest` observer in listing 1.17 must be guarded by a condition that decides whether that observer may apply (the corresponding abstract object knows the answer, and all we have to do is to ask it).

Section 6 presents results of applying both versions of this abstract VS concrete technique – the `equals`-based comparison and the observer only-based comparison ones – to two case studies.

5 Flasji – integrating the 4 techniques

As already referred in section 2 (recall figure 2), the Flasji approach is the result of integrating the techniques presented in the last sections. Now that the reader has a deeper understanding of these techniques we look back at the steps to be taken to discover a fault in a given Java implementation of a ConGu module. We depart from a situation where GenT tests have just been executed and some have failed, and try to reach the goal of presenting the faulty method.

5.1 The algorithm

Given the artifacts generated by the GenT tool – a suite of JUnit tests, an Alloy specification corresponding to the original ConGu module, and mock classes for the parameters of the module –, and the Java class implementing the designated sort:

- Execute the test suite;
- Try to execute techniques 4.3 and 4.1 to build an initial list of suspects where all the observer suspects ranking position is the same;
- For every observer obs in the suspects list try to apply technique 4.2:
  - if the guilt of obs is strengthened, increase its observer ranking position in the suspects list;
  - if the result is inconclusive, maintain obs in the suspects list as is;
  - if the guilt of obs is weakened, subtract obs from the suspects list;
- If the suspects list is not empty:
  - For each suspect $spct$ in the list apply the test built by using the technique 4.4, giving $spct$ as input to the test, and taking into account the suspiciousness ranking of each $spct$ in the list;
- Else (*the suspects list is empty*):
  - For each method m involved in the failed tests, apply the test built by using the technique 4.4, giving m as input to the test;
- Output the result, which consists of:
  - ”Most probable faulty method” - one only method name, which is the one these last tests agree on; if these tests’ results are inconclusive, no faulty method can be elected here;
  - ”Other plausible ones” - the other method names in the suspects list, where constructors are at the same level of plausibility than best ranked observers.

5.2 An example

As an example, the case where the `insert` operation was ill-implemented and for which we began by applying the technique in section 4.3 – there were tests failing with `NullPointerException` –, and the technique in 4.1 – there were two tests for different axioms, involving the same constructor, that failed. The former exercise led us to add `isEmpty` and `largest` to the suspects list. The latter exercise led us to add `insert` to the list.

We then applied the technique in 4.2 to both `isEmpty` and `largest` and we were able to discard `largest`, while keeping `isEmpty` as a suspect.

Finally we applied the abstract-concrete comparison presented in section 4.4 to the two remaining suspects, independently from each other, and got an unequivocal result that pointed to a faulty implementation of the `insert` operation.

6 Evaluation

To evaluate the effectiveness of our Flasji approach, we applied it to two preliminary case studies – this paper SortedSet running example, and MapChain specifications and corresponding implementations; the Java classes implementing the designated sorts of both case studies where seeded with several and diverse errors covering all the specification operations.

We also picked two program spectrum-based tools – GZoltar [22] and EzUnit4 [5, 23] – that, alike Flasji, work on JUnit tests and have methods as the units under test, and compared the three approaches' results. Both tools give a ranking of suspect fault locations as a result, where suspicioussness is measured through percentages.

The two tools were given as input the GenT generated tests for the two case studies, and were put to work while those tests executed over several faulty versions of the designated sort implementation. Their results were registered and used to compare with Flasji's, for both versions of the abstract VS concrete technique – the `equals`-based comparison version (see sections 4.4, 4.4 and 4.4) and the observer only-based comparison one (see section 4.4).

The artifacts generated by the GenT tool – a suite of JUnit tests, an Alloy specification corresponding to the original ConGu module, and mock classes for the parameters of the module – were given as input to Flasji, and it was put to work, as described in section 5, successively for each of the same faulty classes. According to the presented Flasji algorithm, the outcome of our tests is of two kinds: *(i)* the most probable suspect (if it can be found) and *(ii)* other suspects (with a ranking for observers, and constructors ranked as the most well-ranked observers).

Table 2 displays the results of the three tools over the two case studies (the two versions of our approach are referred as "Fe" and "Fo", for `equals`-based comparison and observer only-based comparison, respectively); Twenty JUnit tests were generated by GenT for the SortedSet case, and seventeen for the MapChain one. For each seeded fault, we show the number of failed tests and whether the faulty method was ranked, by each tool, as most probable guilty, second most probable guilty or third or less probable. Being ranked as second probable guilty means that the tool wrongly elected another method as the most probable one. The percentages presented for EzUnit4 and GZoltar results are the ones that these tools calculate for the given method suspiciousness (for a given test suite, a 100% suspicion for a method m may coexist with other, less suspicious methods m_i, ranked with varied other percentages).

This experiment is very favorable to Flasji's both versions since they provided very accurate results in general. The bad results (there were no suspects found whatsoever) for the `equals`-based comparison version ("Fe" in the table) in the three faults for method `get` of the MapChain case study 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 7). Applying an alternative way of observation – the one implemented by the observer only-based comparison version ("Fo" in the table) – we obtain the right results for this case, i.e. `get` 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 was interpreted by the EzUnit4 tool in such a way that it ranked the guilty private `insert` in 7th place with a suspiciousness percentage of 26; GZoltar ranked it in second with 58%. Flasji ranked the *public* `insert` method as the most probable suspect, and it never identified the private method as

	Faulty op.	# tests		Faulty method detected as:											
				1st suspect				2nd suspect				3rd or worse			
		all	fail	Fe	Fo	EzU	GZ	Fe	Fo	EzU	GZ	Fe	Fo	EzU	GZ
SortedSet:	isEmpty *i*	20	5	yes	yes						65			50	
	isEmpty *ii*		1	yes	yes	43					28				
	isIn		1	yes	yes		100							21	
	largest *i*		7	yes	yes	64	100								
	largest *ii*		1	yes	yes	62					32				
	private insert		2	*pub*	*pub*						58			26	
	public insert		5	yes	yes						58			32	
MapChain:	get *i*	17	4		yes		100							52	
	get *ii*		3		yes		87			58					
	get *iii*		4		yes									44	82
	isEmpty *i*		2	yes	yes		100			42					
	isEmpty *ii*		1	yes	yes		100							39	
	put *i*		0	yes	yes										
	put *ii*		1	yes	yes	55					58				
	put *iii*		2	yes		61			yes		53				
	remove *i*		1	yes					yes	44	71				
	remove *ii*		1	yes	yes					44	71				

Table 2. Results of comparative experiments. "Fe" and "Fo" stand for the two variants of our Flasji approach w.r.t. the comparison between concrete objects – the one using `equals` and the one using observers only. "EzU" and "GZ" stand for EzUnit4 and GZoltar respectively.

suspect; the public `insert` method is composed of one only statement – `root = insert(root, e);` – which invokes the private `insert` method. As explained in section 4.4, private methods are not identified as suspects by Flasji 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.

There can be cases where the models chosen by the GenT approach in the process of generating the test suit fail in exposing some fault because they simply do not cover any designated sort instance where the error could emerge. As an example, the faulty implementation of method `put i` made none of the seventeen GenT tests fail. As a consequence, Flasji integrated approach was not put to work and the fault stayed uncaught. EzUnit4 and GZoltar were also neutral because none of the tests failed. Knowing that the error existed, we applied Flasji concrete VS abstract technique and identified the guilty method.

We intend to do mutation testing but, since we still lack a tool that thoroughly generates the new test classes (on-going work), the experimentation of many different faults turns out to be excessively time-consuming.

7 Related work

The fault-location approach presented in this paper relies on the existence of models of the specification to supply the objects (and their behaviour) 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 the adopted GenT test generation, and the here presented fault-location techniques – remember parameter sorts are implemented as mock classes in the GenT approach, and mock objects are also used in place of all non-designated sorts implementations in the presented fault-location approach.

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, 7, 10, 13] 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, 17, 28].

In the context of the work presented in this paper, we tackle both the case where `equals` is reliable and the opposite (see section 4.4). In the first case, we compare some of the designated sort objects using `equals` and all

non-designated sort objects using == (the problem does not apply to these sorts because we work with a finite set of immutable mock objects, as explained in section 4.4).

The technique we propose for the case where `equals` is not reliable compares the results of applying the reliably-comparable observers over abstract and concrete objects (here again, we profit from the reliability of == over non-designated sort objects). 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. Although observers of designated sort and non-creator constructors cannot be used to observe concrete objects (unlike the case where we can rely on `equals`), nevertheless they still contribute to the final comparison because we not also create all objects that constitute a model of the specification, but apply to them all those transformer operations as defined in the model, obtaining a very rich set of objects, both abstract and concrete, that are then observed through the reliably-comparable observers.

Despite having an alternative way of comparing designated sort objects for cases where `equals` is unreliable, this unreliability can still affect the integrated fault-location approach presented in this paper – the suspects list that feeds the concrete VS abstract technique is built from failed GenT tests and their manipulations, and GenT tests use `equals` whenever concrete objects of the same type are compared. Tipically, this may cause otherwise unsuspected operations be accused of being suspect, although it may also happen that a defect in `equals` masks a defect in other operation, therefore classifying it as non-suspect. One of the topics GenT authors point as further work is the need to be able to test the `equals` method in order to make its use more reliable.

Many other fault-location approaches have been proposed over the years that are not based on specifications (in [26, 27] the authors present a very comprehensive survey where they classify fault-location approaches). Among these, program spectrum-based methods ([5, 12, 14, 15, 21–23, 25] to cite a few) from which we have chosen two tools to be part of our evaluation experiment, as presented in section 6.

Program spectrum-based methods record information about several aspects of the execution of tests and use it to identify most probable pieces of code that are suspect of being faulty. 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 6 – GZoltar [2, 22] and EzUnit4 [5, 23] –, alike our approach, work on JUnit tests and have methods as the units under test. They both 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.

8 Conclusions and further work

We presented Flasji, an integrated approach to the location of Java methods that incorrectly implement algebraic specifications of data types. The adopted testing strategy is one of incremental integration meaning that the Java type implementing the main sort of the specification is tested on the assumption that all others on which it depends, are correctly implemented.

We capitalyze on ConGu specifications and on the GenT tool outputs – the Alloy specification corresponding to the ConGu original one, and the JUnit tests covering all axioms of the specification – and inspect failed tests, eventually building new ones, in a process leading to the identification of the most probable guilty method.

Alloy models of the specification supply the material that allows us to create mock classes that represent correctly behaved objects, with which new tests are built that allow to verify the correctness of implementations for the covered operations applications.

Flasji integrates four different techniques, namely two that are used to find initial suspects, and two others to restrict the suspects list. 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.

There are some topics that we would like to further investigate, some of them essentially depending on GenT evolution. The latter include, e.g. coping with specifications that have infinite models only, by transforming them in order to have finite models only, amenable by Alloy. The generation of tests that allow to gain more confidence on the `equals` method (as discussed in section 7), which is used in GenT tests to compare Java objects, is another objective of the GenT team.

In the context of the approach presented in this paper, we are already working on the development of a tool that implements the proposed fault-location process. As soon as we have a working prototype, we intend to perform mutation testing in order to support the already carried out evaluation.

We also intend to study the way succeeded tests could be used to best contribute to the process of fault-location. As an example of a question worthy of being investigated, the one of whether it is relevant to know, in a passing test, if the true clauses include the clause corresponding to the one in the run that originated the test, and, if that happens in all run/tests from a given axiom, whether that can be seen as a strong evidence that the operations involved are well-implemented.

Full integration of the Flasji approach with ConGu is being investigated, in order to obtain a runtime verification framework – ConGu instrumented classes verify specification-induced contracts at runtime – with fault-location capabilities. Presently, the information given by the ConGu tool is often insufficient to the desired goal of finding the guilty part of the implementation.

References

1. J. Abreu. ConGu v.1.50 the specification and the refinement languages. 2007.
2. R. Abreu, P. Zoeteweij, and A.J.C. van Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 24th International Conference on Automated Software Engeneering (ASE'09)*, pages 88–99. IEEE Computer Society, 2009.
3. F.R. Andrade, J.P. Faria, A. Lopes, and A.C.R. Paiva. Specification-driven test generation for java generic classes. In *IFM 2012 - integrated Formal Methods; accepted*, 2012.
4. F.R. Andrade, J.P. Faria, and A. Paiva. Test generation from bounded algebraic specifications using alloy. In *ICSOFT2011, 6th International Conference on Software and Data Technology*, January 2011.
5. P. Bouillon, J. Krinke, N. Meyer, and F. Steimann. Ezunit: A framework for associating failed unit tests with potential programming errors. In *8th International Conference on Agile Processes in Software Engineering and eXtreme Programming (XP)*, volume 4536 of *Lecture Notes in Computer Science*, pages 101–104. Springer, 2007.
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. J. Guttag. Abstract data types, then and now. *Software pioneers: contributions to software engineering*, pages 442–452, 2002.
9. J.L. Hein. *Discrete Structures, Logic, and Computability*. Jones & Bartlett, 2009.
10. 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.
11. D. Jackson. *Software Abstractions - Logic, Language, and Analysis, Revised Edition*. MIT Press, 2012.
12. J.A. Jones and M.J. Harrold. Empirical evaluation of the tarantula automatic fault- localization technique. In *Proceedings of the 20th IEEE/ACM Conference on Automated Software Engineering*, pages 273–282, December 2005.
13. L. Kong, H. Zhu, and B. Zhou. Automated testing ejb components based on algebraic specifications. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, pages 717–722, July 2007.
14. B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 15–26, June 2005.
15. C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on Software Engineering*, 32(10):831–848, October 2006.
16. J.R. Lyle and M. Weiser. Automatic program bug location by program slicing. In *Proceedings of the 2nd International Conference on Computer and Applications*, pages 877–883, June 1987.
17. P.D.L. Machado and D. Sanella. Unit testing for casl architectural specifications. In *Proc. 27th Intl. Symp. on Mathematical Foundations of Computer Science*, volume 2420 of *Lecture Notes in Computer Science*, pages 506–518. Springer, 2002.
18. I. Nunes, A. Lopes, and V. Vasconcelos. Bridging the gap between algebraic specification and object-oriented generic programming. In *Runtime Verification*, volume 5779 of *Lecture Notes in Computer Science*, pages 115–131. Springer, 2009.
19. I. Nunes, A. Lopes, V. Vasconcelos, J. Abreu, and L. Reis. Checking the conformance of java classes against algebraic specifications. In *Proc. of 8th International Conference on Formal Engineering Methods*, volume 4260 of *Lecture Notes in Computer Science*, pages 494–513. Springer, 2006.
20. L.S. Reis. ConGu v.1.50 users guide. 2007.
21. M. Renieris and S.P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pages 30–39, October 2003.
22. André Riboira and Rui Abreu. The gzoltar project: A graphical debugger interface. In Leonardo Bottaci and Gordon Fraser, editors, *TAIC PART*, volume 6303 of *Lecture Notes in Computer Science*, pages 215–218. Springer, 2010.
23. F. Steimann and M. Bertschler. A simple coverage-based locator for multiple faults. In *IEEE International Conference on Software Testing Verification and Validation*, volume 4536 of *Lecture Notes in Computer Science*, pages 101–104. Springer, 2009.
24. M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.

25. 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.
26. 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.
27. W.E. Wong and V. Debroy. Software fault localization. *(Section Authors of the IEEE Reliability Society Annual Technical Report), IEEE Transactions on Reliability*, 59(3):473–475, September 2010.
28. H. Zhu. A note on test oracles and semantics of algebraic specifications. In *QSIC 2003*, pages 91–99. IEEE Computer Society, 2003.