Testing Java implementations of algebraic specifications

Isabel Nunes and Filipe Luís
Lisbon University, Portugal

MBT 2013

Outline

- Introduction
- Some background
- Flasji explained
- Some issues
- Conclusions and further work

Introduction

- Flasji is
  - a model-based approach
  - to the location of faults in Java implementations of algebraic specifications
- Flasji picks
  - a ConGu specification module
  - Java classes implementing the module
  - a mapping from Congu types and operations to Java classes and methods
- Flasji suggests
  - a suspect method (or a list thereof)

    if the implementation violates the specification

A ConGu specification module

```
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];
        ...
        not isIn(empty(), E);
        isIn(insert(S,E), F) iff E = F or isIn(S, F);
        largest(insert(S, E)) = E if isEmpty(S);
        largest(insert(S, E)) = E if not isEmpty(S) and geq(E, largest(S));
        ...
end specification
```

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

An implementation

```
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() {
            ...
    }
    ...
}
```

```
public interface IOrderable<E> {
    boolean greaterEq (E e);
}
```

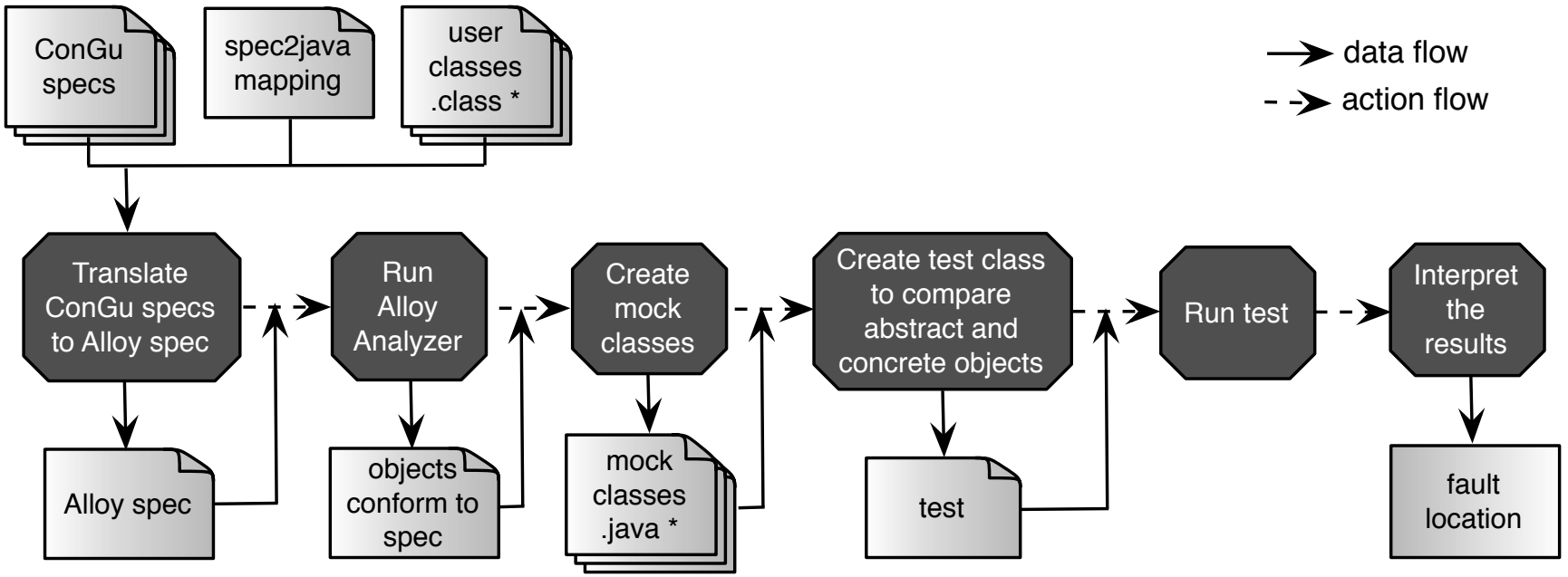
A refinement mapping

```
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
```

Flasji objectives

- Build a model-based test that, when executed:
  - creates “abstract” objects
    - that is, objects that behave according to the specification
  - and “concrete” objects
    - that is, instances of the implementing classes
  - and
    - compares their behaviours
- Execute the test
- Interpret the deviations

Flasji in a nutshell

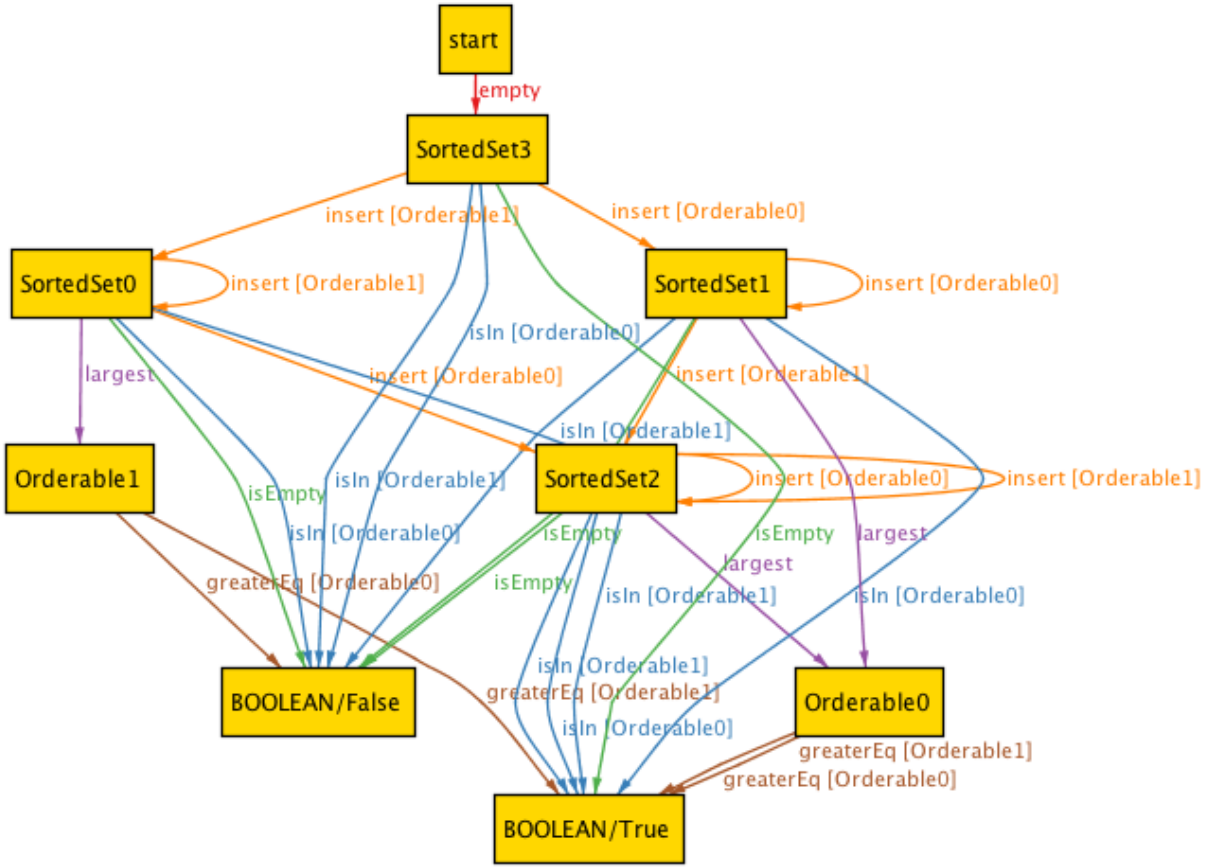


Abstract objects

- Each abstract object
  - represents an instance of a specification sort in a model of the spec module
  - can be queried in order to know
    - what the correct result should be
      of invoking each method
      over the corresponding concrete counterpart
  - so, it must be “injected” with
    - the information about the correct results
      for each applicable pair <method , argument_value>
- For this, Flasji creates “mock” classes whose instances will be the abstract objects. It does so for:
  - The designated sort and
  - The parameter sorts, if the previous is a parameterized sort

Abstract objects – obtaining a model

- To obtain a collection of instances of the specification sorts, Flasji:
  - translates the Congu spec module to an Alloy specification
  - uses the Alloy Analyzer to find a model



Abstract objects – "mock" classes

- Mock classes implement methods to:
  - inject abstract objects with expected results on concrete counterparts
  - query those results

Example:

- the mock class for the ***parameter*** sort

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

    public void add_greaterEq(OrderableMock e, boolean result) {
            greaterEqResult.put(e, result);
    }

    public boolean greaterEq(OrderableMock e) {
            return greaterEqResult.get(e);
    }
}
```

Abstract objects – "mock" classes

- the mock class for the ***designated*** sort

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

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

    public boolean isEmpty() {
            return isEmptyResult;
    }

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

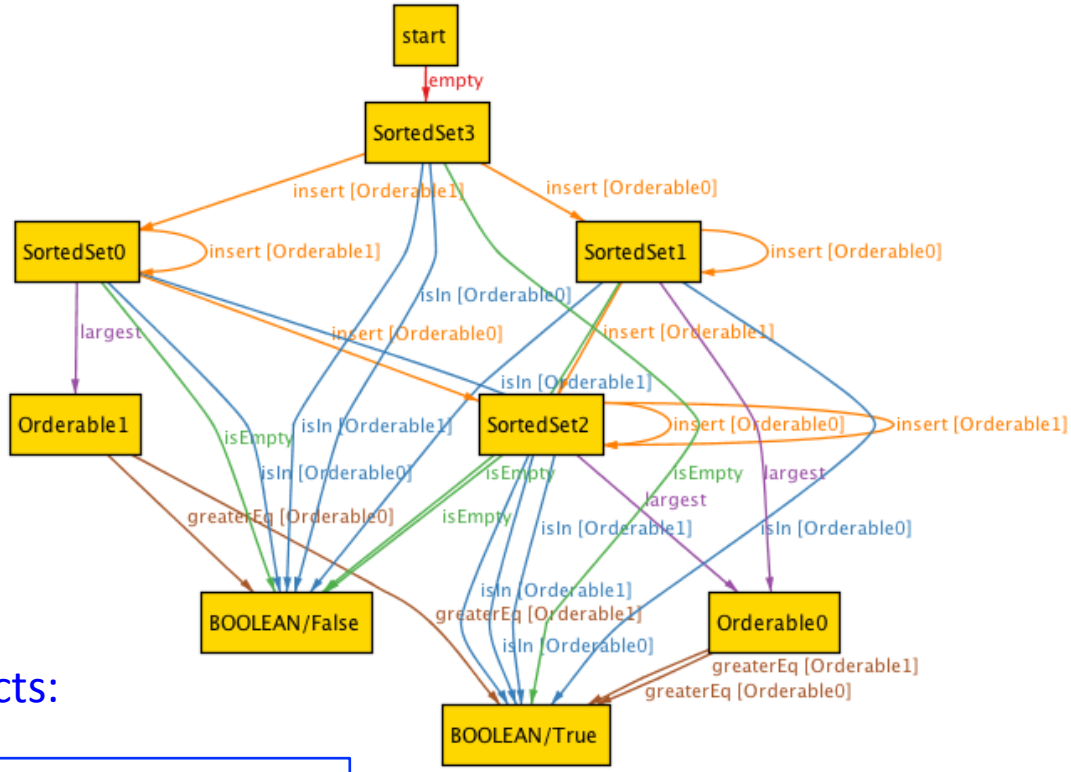
```
    public T largest()  {
            return largestResult;
    }

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

    private HashMap<T,TreeSet<T>> insertResult =
            new HashMap<T,TreeSet<T>> ();

    public TreeSet<T> insert (T e) {
             return insertResult.get(e);
    }

    public void add_insert (T e, TreeSet<T> concVal){
                    insertResult.put(e, concVal);
    }
}
```

The test class

- Flasji generates a class containing instructions to:
  - create abstract objects, that is, instances of the mock classes:
    - for the parameter types
  - create concrete objects, that is, instances of the implementing classes:
    - for the designated sort and other non-parameter sorts
  - create abstract objects
    - for the designated sort
  - invoke all methods that correspond to the specification operations:
    - over abstract designated sort objects
    - over concrete designated sort objects
    - for all combinations of argument values

- All the above objects are created as defined by the model:
  - in number
  - by following the operations application path for each one

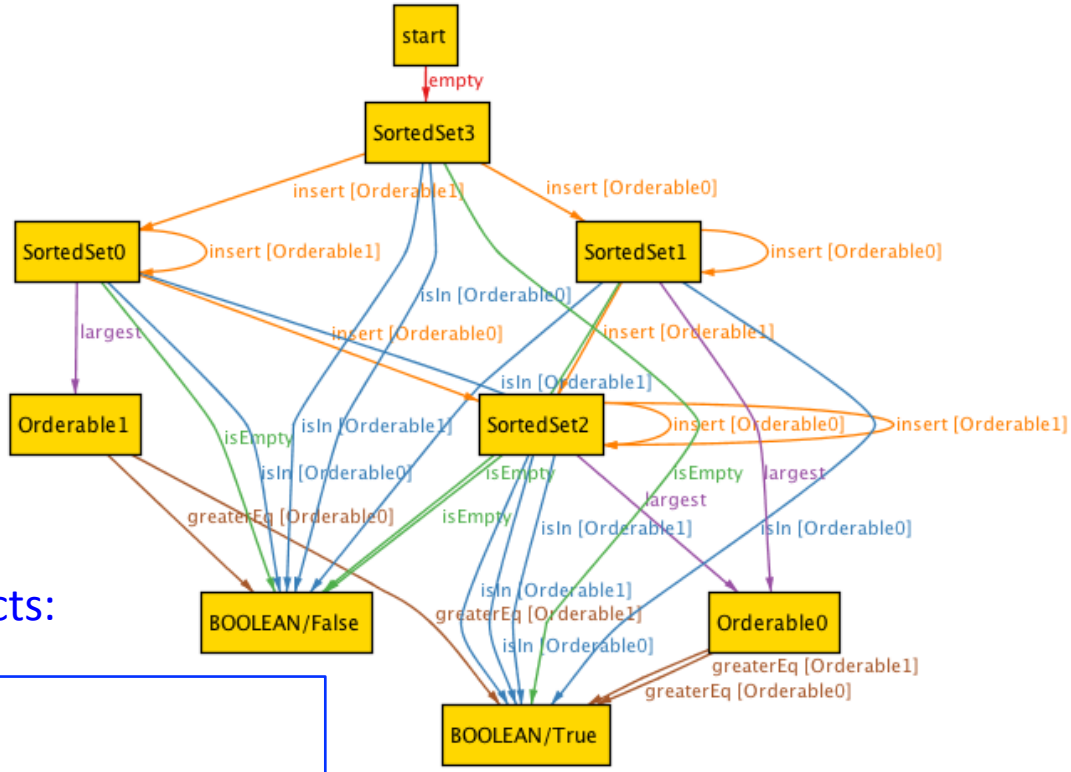
The test class



Creating parameter sort abstract objects:

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

The test class



Creating designated sort concrete objects:

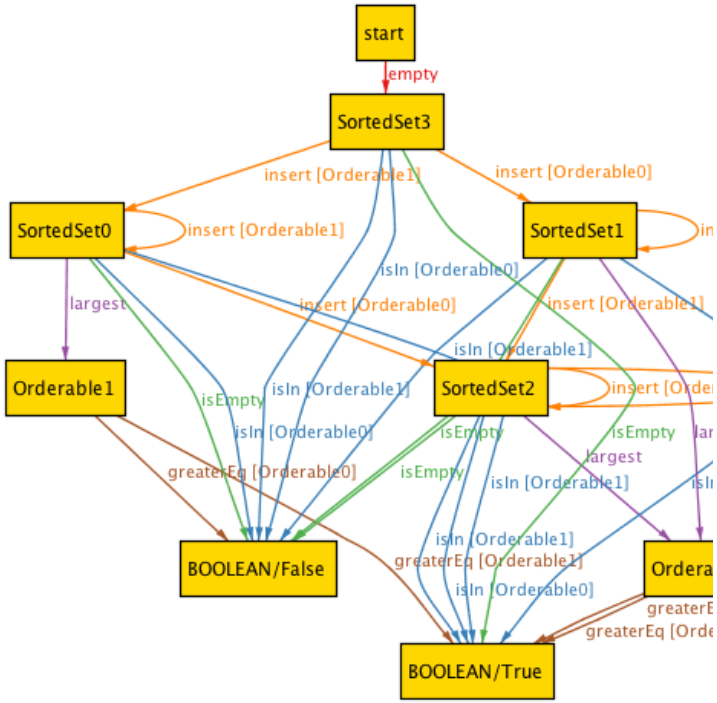
```
...
//Create concrete objects TreeSet
TreeSet<OrderableMock> concSortedSet0 = new
                              TreeSet <OrderableMock>();
concSortedSet0.insert(orderable1);
TreeSet<OrderableMock> concSortedSet0_1 = new ...;
concSortedSet0_1.insert(orderable1);
...
TreeSet<OrderableMock> concSortedSet0_6 = new ...;
concSortedSet0_6.insert(orderable1);
// three more to go (concSortedSet 1, 2 and 3)...
```

Concrete objects would also be created for any other non-parameter type.

The test class

Creating designated sort abstract objects:

```
...
//Create abstract objects TreeSetMock
TreeSetMock <OrderableMock> sortedSet3 = new
                    TreeSetMock <OrderableMock>();
TreeSetMock <OrderableMock> sortedSet0 = new ...;
TreeSetMock <OrderableMock> sortedSet1 = new ...;
TreeSetMock <OrderableMock> sortedSet2 = new ...;
// inject information about expected results
sortedSet0.add_isEmpty(false);
sortedSet0.add_largest(orderable1);
sortedSet0.add_isIn(orderable0, false);
sortedSet0.add_isIn(orderable1, true);
sortedSet0.add_insert(orderable0, concSortedSet2);
sortedSet0.add_insert(orderable1, concSortedSet0);
// three more to go (sortedSet 1, 2 and 3)...
```



The test class

Comparing concrete with expected behaviours:

```
...
//Compare concrete with corresponding abstract
assertTrue (concSortedSet0_1.isEmpty() == sortedSet0.isEmpty());
assertTrue (concSortedSet0_2.largest() == sortedSet0.largest());
assertTrue (concSortedSet0_3.isIn(orderable1) == sortedSet0.isIn(orderable1));
assertTrue (concSortedSet0_4.isIn(orderable0) == sortedSet0.isIn(orderable0));

concSortedSet0_5.insert (orderable1);
assertTrue (concSortedSet0_5.equals(sortedSet0.insert (orderable1)));
concSortedSet0_6.insert (orderable0);
assertTrue (concSortedSet0_6.equals(sortedSet0.insert (orderable0)));
// three more to go (sortedSet 1, 2 and 3)...
```

assertTrue are enclosed in *try-catch* blocks to collect all deviations

Running the test

- Flasji runs the test it created, and interprets the results
  - The interpretation is based upon:
    - whether several and varied observers fail or only one fails (important do decide whether to blame a constructor or a given observer);
    - whether varied observers fail when applied to objects built only with the creator constructor or else also with other constructors
  - Results are collected into:
    - L1 – Set of pairs < `obs` ; `obj`> registering deviations for given observer `obs` and object `obj`
    - L2 – Set of pairs < `ncc` ; `obj`> registering deviations for given non-creator constructor `ncc` and object `obj`
    - L3 – Set of pairs < `cc` ; `n`> registering, for every creator constructor , the number of deviations over objects uniquely built with `cc`

Algorithm

Several observers fail and just one creator constructor is suspect

Several observers fail and no creator constructor is guilty

Only one observer fails

Only one non-creator constructor is suspect

```
if (L1 ∪ L2) contains pairs for more than 1 observer, then
    if there exists < cc, i > in L3 with i > 0, then
        if that pair < cc, i > with i > 0 is unique, then
            cc is guilty;
        else
            inconclusive;
        endIf
    else
        for each non-creator constructor nccj do
            Lnccj ← sub-set of L2 containing only pairs from
                L2 whose first element is nccj;
            Delete from Lnccj the pairs whose obj was not
                built using only nccj and a creator constructor;
            if Lnccj is not empty, then
                add nccj to the final set of suspects (FSS);
            endIf
        endFor
    endIf
    if #FSS = 1 then
        the guilty is the sole element of FSS;
    else
        inconclusive;
    endIf
else
    if (L1 ∪ L2) is empty, then
        inconclusive;
    else
        the guilty is the sole observer in (L1 ∪ L2);
    endIf
endIf
```

Evaluation

- Two case studies – SortedSet and MapChain
- Flasji was put to run for several deffective implementing classes, and outputs were registered
- Two fault-location tools – GZoltar and EzUnit4 – were put to run with several tests covering all specification axioms (generated with a companion tool – GenT)
- Comparison between the results of the three tools
  - Flasji pointed to the guilty method 70% to 85% of the times
  - the other tools correctly identified the faulty method as first suspect around 40% of the times

Some issues

- Private methods are not identified by Flasji
- Flasji relies upon the equals method
- Number of objects of each sort in the model
- Inconclusive results – new model ; join observations

Conclusions and further work

- Presented a model-based approach to location of faults in java implementations of algebraic specifications
- An Eclipse plug-in is already available but still under improvement
  - verify the equals method even if the specification does not define it
  - optimize the dimensions of the model
  - allow for the verification of the other non-parameter types

Thank you