Property-based Testing: Testing Java Using Erlang QuickCheck

Julio Mariño

January 15, 2014

Abstract

Property-based testing automates the task of writing tests by generating testing code from (logical) specifications. We show how to test Java code using Erlang QuickCheck by means of a library capable of invoking Java code from an Erlang interpreter. We will demonstrate the approach using one of the examples seen in the classroom: a full functional specification of list sorting and a Java implementation of the insertion sort algorithm.

1 Introduction

Property-based testing automates the task of writing tests by generating testing code from (logical) specifications. QuickCheck is a quite popular tool for generating black-box random tests from a quasi-logical notation. Originally developed for the Haskell functional programming language, the commercial (and most used) implementation of the system is based on the concurrent functional language Erlang.

Although black-box random testing has a number of known limitations, a key advantage of the approach is that, in principle, it is possible to adapt testing code written in a given language L to test the corresponding implementation in a different language L' provided that there is a pair of encoding functions that translate inputs from L data types to L', and outputs back from L' to L.

In order to show the approach we will be using a Java implementation of the insertion sort algorithm. This is shown in Fig. 1. A buggy version is shown in Fig. 2.

Remember that the functional specification of list sorting has two parts. The first one states that the output must be sorted. The second one states that the output must be a permutation of the input list.

2 Calling Java from Erlang

We will make use of the JavaErlang library (Fredlund 2013) to invoke Java code from Erlang. A sample session follows. First, we invoke the Erlang shell:

```
virgo:javatest xmc$ erl -sname virgo
Erlang R15B02 (erts-5.9.2) [source] [smp:2:2] [async-threads:0] [hipe]
```

```
public class insertionSort {

    public static int[] insertSort (int[] list) {
        // we make space for the new list that is returned, i.e. we do
        // not change the input list
        int size = list.length;
        int[] result = new int[size];

        // as result is originally filled with zeroes, and the input
        // list might contain negative values, or zeroes, we need to
        // keep in mind the "logical" length of the array
        int logical_size = 0;

        // we insert all list's elements in order in result
        for (int i = 0; i < size; i++) {
            // first, we skip smaller elements
            int pos = 0;
            for (int j = 0; j < logical_size && result[j] < list[i]; j++){
                pos = j+1;
            }
            // then, we shift the greater elements:
            for (int k = logical_size; k > pos; k--){
                result[k] = result[k-1];
            }
            logical_size++;
            // finally, we insert list[i]
            result[pos] = list[i];
        }
        return result;
    }
}
```

Figure 1: A Java implementation of insertion sort.

```
public class insertionSort_insertbefore {

    public static int[] insertSort (int[] list) {
        // we make space for the new list that is returned, i.e. we do
        // not change the input list
        int size = list.length;
        int[] result = new int[size];

        // as result is originally filled with zeroes, and the input
        // list might contain negative values, or zeroes, we need to
        // keep in mind the "logical" length of the array
        int logical_size = 0;

        // we insert all list's elements in order in result
        for (int i = 0; i < size; i++) {
            // first, we skip smaller elements
            int pos = 0;
            for (int j = 0; j < logical_size && result[j] < list[i]; j++){
                pos = j+1;
            }
            // we insert list[i]
            result[pos] = list[i];
            // then, we shift the greater elements:
            for (int k = logical_size; k > pos; k--){
                result[k] = result[k-1];
            }
            logical_size++;

        }
        return result;
    }
}
```

Figure 2: A buggy implementation of insertion sort. Can you spot the bug?

```
[kernel-poll:false]

Eshell V5.9.2  (abort with ^G)
```

Then, a Java server node is launched which will be in charge of executing code as requested from the Erlang side:

```
(virgo@virgo)2> {ok,NID} = java:start_node([{add_to_java_classpath,["."]}]).
{ok,101}
```

The node thus created is passed as parameter of the different commands that translate Erlang data into Java,

```
(virgo@virgo)4> Input = java:list_to_array(NID,[6,2,3,1,5,4],int).
{object,0,101}
```

call Java methods,

```
(virgo@virgo)6> Output = java:call_static(NID,insertionSort,insertSort,[Input]).
{object,1,101}
```

or translate Java data back to Erlang:

```
(virgo@virgo)7> java:array_to_list(Output).
[1,2,3,4,5,6]
```

3 Testing Java code using QuickCheck

The properties to be tested using Erlang QC will be written in an Erlang file, in this case named `sorting.erl`. First, we have a purely functional specification of sorting. i.e. there is nothing to do with Java or QC here. A predicate to state that a list of integers is sorted follows:

```
isSorted([]) -> true;
isSorted([_]) -> true;
isSorted([A,B|Zs]) -> (A =< B) and isSorted([B|Zs]).
```

Defining a predicate to check whether two lists are permutations of each other is only slightly harder:

```
occurs(_,[]) -> 0;
occurs(X,[H|Ts]) when X==H -> 1 + (occurs(X,Ts));
occurs(X,[_|Ts])           -> occurs(X,Ts).

permuts(As,Bs) ->
        lists:all(fun (X) -> occurs(X,As) == occurs(X,Bs) end, As++Bs).
```

Now, we define QuickCheck properties on Erlang lists for testing the effect of the Java sorting implementations on their corresponding Java relatives:

```
propSorted(Node) ->
        ?FORALL(Xs, nelist(int()),
                isSorted(java:array_to_list(
                        java:call_static(Node,
                                insertionSort,
                                insertSort,[java:list_to_array(Node,Xs,int)])))).

propSorted_insertbefore(Node) ->
        ?FORALL(Xs, nelist(int()),
                isSorted(java:array_to_list(
                        java:call_static(Node,
                                insertionSort_insertbefore,
                                insertSort,[java:list_to_array(Node,Xs,int)])))).

propPermuts(Node) ->
        ?FORALL(Xs, nelist(int()),
                permuts(Xs,java:array_to_list(
                        java:call_static(Node,
                                insertionSort,
                                insertSort,[java:list_to_array(Node,Xs,int)])))).

propPermuts_insertbefore(Node) ->
        ?FORALL(Xs, nelist(int()),
                permuts(Xs,java:array_to_list(
                        java:call_static(Node,
                                insertionSort_insertbefore,
                                insertSort,[java:list_to_array(Node,Xs,int)])))).
```

Let us try it! The “right” implementation passes all tests:

```
(virgo@virgo)2> eqc:quickcheck(sorting:propSorted(NID)).
Starting eqc mini version 1.0.1 (compiled at {{2010,6,13},{11,15,30}})
.....................................................................
.....................
OK, passed 100 tests
true
(virgo@virgo)3> eqc:quickcheck(sorting:propPermuts(NID)).
.....................................................................
.....................
OK, passed 100 tests
true
```

The buggy version passes the “isSorted” test (!!)

```
(virgo@virgo)5> eqc:quickcheck(sorting:propSorted_insertbefore(NID)).
.....................................................................
.....................
```

```
OK, passed 100 tests
true
```

but fails the “permuts” one:

```
eqc:quickcheck(sorting:propPermuts_insertbefore(NID)).
..........Failed! After 11 tests.
[0,-1]
false
```

4 References

The JavaErlang library can be obtained from `http://babel.ls.fi.upm.es/~fred/JavaErlang/`. A free version of Erlang QuickCheck, with limited functionality, but enough for running these tests can be downloaded from `http://www.quviq.com`.