System Validation 2016
Homework Part 1 - JML

Koen Mulder (s1757679) and Ruben van den Berg (s1354914)

October 13, 2016

Abstract

This homework assignment is done for the course System Validation. Here has been practiced with JML Annotations, Runtime Assertion Checking(from now on called RAC), Static Checking and Test Generation. In the first part JML annotations were made for informal array requirements. In the second part a game of Sokoban was used where informal requirements were made formal and were checked by JML Annotations, RAC, Static Checking and Test Generation. In the end the program behaved accordingly to the informal requirements.

1 Array Requirements

The first exercise relates to the array specification exercise as given in the first exercise session of the course. Below all informal requirements are given and a JML specification to solve this.

- Consider an array B with n elements: $B[0], ..., B[n-1]$. All elements in B are integers. Let j, k be two indices such that $0 \leq j < k < n$. With $B[j], ..., B[k]$ we denote the segment of B starting from index j and ending with index k. Which is in JML specifications

  ```
  /*@ spec_public */ private int[]B;       // The array
  /*@ spec_public */ private int n;        // Lenght of the array
  /*@ spec_public */ private int j;        // Index start
  /*@ spec_public */ private int k;        // Index end

  //@ public invariant j >= 0 && j < k && k < n;
  ```

- Any value that occurs in the segment $B[j], ..., B[k]$ also occurs outside that segment.

  ```
  //@ public invariant (\forall int i1; i1 >= j && i1 <= k;
  (\exists int i2; (i2 >= 0 && i2 < j) || (i2 > k && i2 < n); B[i1] == B[i2]));
  ```

  This is checked by going over this segment and see if there is a match for every element outside of $B[j], ..., B[k]$

- The array B does not contain duplicates when the segment $B[j], ..., B[k]$ is not considered.

  ```
  //@ public invariant (\forall int i1, i2; i1 >= 0 && i2 > i1 && i2 < n;
  (B[i1] == B[i2]) ==> ((i1 >= j && i1 <= k) || (i2 >= j && i2 <= k)));
  ```

  If there are two elements who are duplicates of each other it must mean one of the elements is inbetween $B[j], ..., B[k]$.

- Any value occurs at most twice in B.

  ```
  //@ public invariant (\forall int i1,i2,i3; i1>=0 && i2>i1 && i3>i2 && i3<n;
  B[i1] != B[i2] || B[i2] != B[i3] || B[i1] != B[i3]);
  ```

  If you have three values at least two values can not be equal to eachother.

- The highest value occuring in B occurs at least twice in B.

  ```
  //@ public invariant (\exists int i1, i2; i1 >= 0 && i2 > i1 && i2 < n;
  B[i1] == B[i2] && (\forall int i3; i3 >= 0 && i3 < n; B[i1] >= B[i3]));
  ```

  First we check if two elements are the same. If so we check for all other elements if they are equal or higher then those elements.

- The array B contains a palindrome of length k.

  ```
  //@ public invariant (\exists int i1, i2; i1 >= 0 && i2 == (i1+k-1) && i2 < n;
  (\forall int i3; i3 >= 0 && i3 < k; B[i1 + i3] == B[i2 - i3]));
  ```

  Index i1 and i2 are a segement with lenght k, after this we check for the palindrome. We consider the segment i1,i2 a sub array. We use two indices to check from back to front and the other way around. For every possible itteration the elements for these indices should be the same.

- After doing at most one swap the array B is sorted.

  ```
  /*@ public invariant (\exists int i1, i2; i1 >= 0 && i2 > i1 && i2 < n;
  (\forall int i3; i3 >= 1 && i3 < n;
  B[i3-1] <= B[i3] || (i3 == i1 && B[i3-1] <= B[i2]) ||
  (i3-1 == i2 && B[i1] <= B[i3]) || (i3-1 == i1 && i3 == i2 && B[i2] <= B[i1])));
  @*/
  ```

  We take two different indices(i1 and i2). Next we check every element in the array against the element in front. For every pair we check if the second element value is bigger or equal to the first. If this is not the case we check if the array will be sorted when i1 and i2 are swapped.

- There is a value that occurs more often in B than any other value.

  ```
  /*@ public invariant (\exists int i1; i1 >= 0 && i1 < n;
  (\forall int i2; i2 >= 0 && i2 < n && B[i1] != B[i2];
  (\sum int s1; s1 >= 0 && s1 < n && B[s1] == B[i1]; 1) >
  (\sum int s2; s2 >= 0 && s2 < n && B[s2] == B[i2]; 1) ) );
  @*/
  ```

  We take an index and check for every other index which does not have the same element value. If the first element value always occurs more often the statement holds.

2 JML Annotations and Runtime Assertion Checking

The goal in this part of the assignment was to turn informal requirements into JML specifications, these were marked with `@informal` in the inline comments in the code. For doing this we mainly used `invariant, requires, ensures` and `assignable` were mostly used to create the JML specifications. Only in the `boolean movePlayer (Position newPosition)` method `assert` is used to create assertion in methods itself. This part is split up into 2 parts, the JML annotations and the runtime assertion checkers. In the first part the most important annotations are shown per class, less interesting or basic annotations are only briefly mentioned. In the runtime assertion checking part the results of runing the program with these annotation is discussed.

2.1 JML Annotations

2.1.1 Position

The first assertions added are in the `class Position`, these are used mainly for checking if a positions is valid. This firstly includes invariants that state that x and y are always 0 or more. Similar assertions are used for checking the input in the constructor.

```
//@ invariant x >= 0 && y >= 0;
```

The other applications of assertions in the `class Position` is for the method `boolean isValidNextPosition (Position newPosition)`.

```
//@ requires Math.abs(newPosition.x - x) + Math.abs(newPosition.y - y) == 1;
//@ ensures \result == true;
//@ also
//@ requires Math.abs(newPosition.x - x) + Math.abs(newPosition.y - y) != 1;
//@ ensures \result == false;
```

Assertions are also added to the equals method and to the constructor, in which they state that the x and y values are set to the parameters.

2.1.2 Player

Assertions added the `class Player` are all basic. To the constructor assertions are added to state that the position parameter is the same as the position of the player. To `public void setPosition (Position newPosition)` annotations are added that state the new position is valid and that position of the player equals the parameter.

```
//@ requires position.isValidNextPosition (newPosition);
//@ ensures this.position.equals (position);
```

2.1.3 BoardItem

In `class BoardItem` invariants are added to state that if an item is a crate or is marked, it has to be ground. These specifications originate from the `class Board`.

```
        //@ public invariant crate ==> ground;
        //@ public invariant marked ==> ground;
```

2.1.4 Board

To the class `class Board` several assertions are added in ration to the size and state of the board.

```
//@ public invariant xSize > 0 && ySize > 0;
//@ public invariant xSize == ySize;
//@ public invariant xSize == items.length;
//@ public invariant \forall int x; x >= 0 && x < items.length;
                ySize == items[x].length;
//@ public invariant \nonnullelements (items);
//@ public invariant \forall int x; x >= 0 && x < items.length;
                \nonnullelements(items[x]);
```

It is stated that the xSize and ySize need to be equal and both greater then zero, the dimensions of the array need to match the xSize and ySize and finally the elements in the array are not allowed to be `null`. The last 3 of these annotations are added because of problems that occurred during the Static checking (3). Annotations are added to the constructor to ensure the dimensions are valid and the array is filled with walls.

```
//@ assignable items;
//@ assignable this.xSize;
//@ assignable this.ySize;
//@ requires xSize > 0 && ySize > 0 && xSize == ySize;
//@ ensures \forall int x; x >= 0 && x < xSize; \forall int y; y >= 0 && y < ySize;
                items[x][y].ground == false;
//@ ensures items.length == xSize && items[0].length == ySize;
```

Annotations added to `public boolean onBoard(Position p)`:

```
//@ ensures \result == (0 <= p.x && p.x < xSize && 0 <= p.y && p.y < ySize);
```

And annotations added to the `public boolean isOpen(Position p)`:

```
//@ requires onBoard(p);
//@ ensures \result == (items[p.x][p.y].ground && !items[p.x][p.y].crate);
//@ also
//@ requires !onBoard(p);
//@ ensures \result == false;
```

The annotations for both the `(Position p)` variant or the `(int x, int y)` variant are almost the same, with the only difference to the input variable against which is checked.

2.1.5 Game

The most important annotations int `class Game` are related to the `boolean wonGame ()` and `movePlayer (Position newPosition)`. The annotations added to `boolean wonGame ()` check if the result is true if all marked positions also are a crate. The annotation used in this methods are further explained in 3 Static Checking.

```
//@ ensures \result == (\forall int x; x >= 0 && x < board.xSize;
        (\forall int y; y >= 0 && y < board.ySize;
                board.items[x][y].marked ==> board.items[x][y].crate));
```

The annotations to `movePlayer (Position newPosition)` are the most extensive annotations of the project, other then pre and post conditions multiple assertions are added to the body of the method.

```
//@ normal_behaviour
//@ requires !player.position.isValidNextPosition (newPosition);
//@ ensures \result == false;
//@ also normal_behaviour
//@ requires !board.onBoard(newPosition);
//@ ensures \result == false;
//@ also normal_behaviour
//@ requires !board.items[newPosition.x][newPosition.y].ground;
//@ ensures \result == false;
//@ also normal_behaviour
//@ requires board.isOpen(newPosition);
//@ requires player.position.isValidNextPosition (newPosition);
//@ requires board.onBoard(newPosition);
//@ ensures \result == true;
//@ also normal_behaviour
//@ requires board.onBoard(newPosition);
//@ requires board.items[newPosition.x][newPosition.y].crate;
//@ requires player.position.isValidNextPosition (newPosition);
//@ requires board.isOpen(newPosition.x + (newPosition.x - player.position.x),
        newPosition.y + (newPosition.y - player.position.y));
//@ ensures \result == true;
//@ also normal_behaviour
//@ requires board.onBoard(newPosition);
//@ requires board.items[newPosition.x][newPosition.y].crate;
//@ requires player.position.isValidNextPosition (newPosition);
//@ requires !board.isOpen(newPosition.x + (newPosition.x - player.position.x),
        newPosition.y + (newPosition.y - player.position.y));
//@ ensures \result == false;
```

The annotations check for all the different situations that can happen within the method such as invalid positions, walls, boxes and they can be pushed or not.

2.2 Runtime Assertion Checking

The following problems were found during runtime checking:

- Players could move through walls.
- Diagonal movement was a valid next position.
- The current position is a valid next position.

The first problem found was that players had the ability to move through walls. This problem was detected both visually and by the JML checker. The problem occurs in the `class Game` inside the method `boolean movePlayer (Position newPosition)`. This class method has a list of mutiple JML specifications for 6 different situations. The following specification detected the problem:

```
//@ requires !board.items[newPosition.x][newPosition.y].ground;
//@ ensures \result == false;
```

This states that a move to a position that is not ground is not a valid move and will not be executed. At runtime it was found that this was not the case and that a player was able to move to a position that is not ground. After adding an check in `boolean movePlayer (Position newPosition)` which validates that the `newPosition` should be on a ground position the code functions according to the specification.

The next problem was ability of the player to move diagonally to another position. This is in contract with the specification which states: ***a valid next position is always one move horizontally or vertically from the current one***. As the previous problem this was also detected visually and by the JML checker. This problem was detected in the `class Position` in the method `boolean isValidNextPosition (Position newPosition)`. Part of the JML specification is as follows:

```
//@ requires Math.abs(newPosition.x - x) +
Math.abs(newPosition.y - y) == 1;
//@ ensures \result == true;
```

There also is an additional part in which the requires expression is not equal to 1, in that case it should return false. By changing the way the method checks if a position is valid of not this problem was was repaired.

The final problem was discovered based on the same specification as the second problem. According the code the current position of the player was also a valid next position, but according to the specification this is not the case. This problem was solved when the second problem was solved.

3 Static Checking

Within this section we will perform formal verification on the method `wonGame` in the `Game` class. The `wonGame` method is responsible for checking the winning situation on the game board after every move.
We implemented four `loop invariants`, two per for loop. One describing if the *x* or *y* element was within the correct boundaries, the other is a nested `forall` loop checking the result. The loop invariant for the outer loop iterating of the *x* was as following:

```
//@ loop_invariant x >= 0 && x <= board.xSize;
//@ loop_invariant result == \forall int i; i >= 0 && i < x;
        (\forall int j; j >= 0 && j < board.ySize;
                !board.items[i][j].marked || board.items[i][j].crate);
//@ decreases board.xSize - x;
```

The loop invariant for the inner loop iterating of the *y* was as following:

```
//@ loop_invariant y >= 0 && y <= board.ySize;
//@ loop_invariant rowresult == (\forall int i; i >= 0 && i < y;
        !board.items[x][i].marked || board.items[x][i].crate);
//@ decreases board.ySize - y;
```

A problem we encountered during the implementation of the nested `forall` loop was the nonzero error. Within the loop elements could not be zero. We fixed this by adding the following specifications to `Board.java`.

```
//@ public invariant \forall int x; x >= 0 && x <
    items.length; ySize == items[x].length;
//@ public invariant \nonnullelements (items);
//@ public invariant \forall int x; x >= 0 && x <
    items.length; \nonnullelements(items[x]);
```

Here we state that every item can not be `null`. Same goes for all the items in the sub-array. Next we state that the `ySize` needs to be equal to `items[x].length` when *x* is between zero and `items.length`.
We did not encounter any more errors afterwards in the `wonGame()` method. However still some errors occurred in the `movePlayer(Position)` method when static checked. This is acceptable since this method is fully checked during runtime without any errors. Due to the complexity of the method and the specification we are not sure how to solve these errors given by the static checker at the `movePlayer(Position)` method.

If we would get a similar assignment in which we had to specify loop invariants we would probably do it the same way. In the case of multiple loops we would works from inside out, starting with most inner loop, annotating it, moving on the the loop around it. In each of these iterations you would reason about what will make the check either true or false and turn this into loop invariants together with the limiting invariants and decreases.

4 Test Generation

The goal of this part of the exercise was generate tests for the JML specifications and provide test data to feed these generates tests. For this part of the exercise we did not create any additional JML annotations in the code as we did already create the required specification in 2.1 *JML Annotations and Runtime Assertion Checking: JML Annotations*. The first created a set of different `Game` objects with a `Board` and `Player`, each `Game` object was created for a different situation, those are summed below:

- moving to ground position
- moving to a marked ground position
- moving to a wall position
- moving towards a wall in different situations
  - a wall with ground behind it
  - a wall with marked ground behind it
  - a wall with a wall behind it
  - a wall with an other box behind it

All the `Boards` had a size of 7 by 7 with the player starting in the center.

The next step was to create position data to feed to the `movePlayer(Position newPosition)`. For this we just created all possible combination for x and y with the value 0 to 7(inclusive), resulting in 64 positions. This means that we provide more positions then the test boards contains. By using these positions we also run all kinds different scenarios such as diagonal movement, movement too far away and movement to invalid positions. No negative positions were used as this is in conflict with the specification of `Position`. We ended up with 196 successfully completed tests.

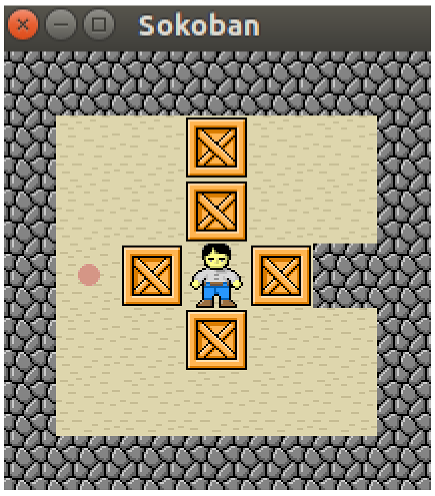
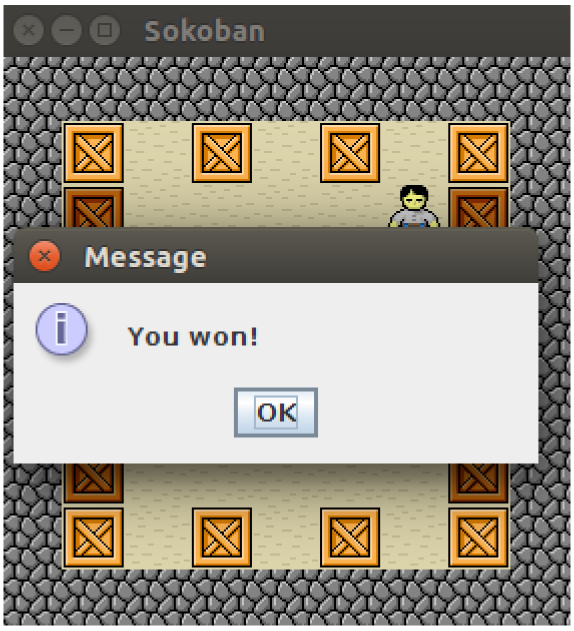


Figure 1: A test board for Sokoban (crates)



You won!

Figure 2: Sokoban win