System Validation 2014
Homework Part 2 - JML

Thom Ritterfeld - s1509101
Tom Vocke - s144002

PART 1 :: LOCK REQUIREMENTS

```
-- ================================================================
-- Always at least one door closed
-- ================================================================
can be defined as at any given time (ltlspec globally) either door1 or door2 or both should be
closed assuming atomic properties Door1_closed, Door2_closed

invariant ( Door1_Closed || Door2_Closed )
G ( Door1_closed || Door2_closed )

-- ================================================================
-- The doors may only be open when the water level on both sides is equal
-- ================================================================
can be defined as at any given time, when a door is open, the water level on that side of the
door should equal the lock waterlevel assuming atomic properties Door1_open, Doo2_open,
Side1_level, Side2_level, Lock_level

G ( Door1_open -> (Side1_level = Lock_level) )
G ( Door2_open -> (Side2_level = Lock_level) )

or:

#define lock1_check ( Door1_Closed && (Side1_level == Lock_level) )
#define lock2_check ( Door2_Closed && (Side2_level == Lock_level) )
invariant ( lock1_check || lock2_check )

-- ================================================================
-- It should be possible to lower and raise the water level
-- ================================================================

G ( LowerWaterLevel -> F waterLevelLow )
G ( IncreaseWaterLevel -> F waterLevelHigh )

or (in case waterlevel is changed using changeWaterLevel function)

/@ requires validValue ( newWaterLevel ) && doorsClosed == True
 @ ensures waterLevel == newWaterLevel
 @ diverges
 @/
changeWaterLevel ( int newWaterLevel) {
  ....
}
```

```
-- ================================================================
-- It should only be possible to raise/lower the water level when both doors are closed
-- ================================================================

G ( !doorsClosed -> (newWaterLevel == oldWaterLevel) )

or (in case waterlevel is changed using changeWaterLevel function)

/@ requires validValue ( newWaterLevel ) && doorsClosed == True
 @ ensures waterLevel == newWaterLevel
 @ diverges
 @/
changeWaterLevel ( int newWaterLevel) {
  ....
}


-- ================================================================
-- The doors should not close when there is a boat passing through
-- ================================================================

G (boatMoving -> !door1Closing && !door2Closing)
or
invariant ( (door1Closing || door2Closing) != boatMoving)
```

PART 2 :: SOKOBAN JML RAC

Problem 1.1 - Player is able to walk on walls:

```
sokoban/src/Game.java:119: JML invariant is false on leaving method Game.movePlayer(Position)
  boolean movePlayer (Position newPosition) {

sokoban/src/Game.java:30: Associated declaration: sokoban/src/Game.java:119:
            //@ public invariant board.isOpen(player.position);

sokoban/src/Game.java:119: JML postcondition is false
  boolean movePlayer (Position newPosition) {

sokoban/src/Game.java:116: Associated declaration: sokoban/src/Game.java:119:
            ensures (player.position == \old(player.position)) && (\result == false);

sokoban/src/Game.java:54: JML invariant is false on leaving method Game.wonGame()
  boolean wonGame () {

sokoban/src/Game.java:30: Associated declaration: sokoban/src/Game.java:54:
            //@ public invariant board.isOpen(player.position);
```

original (line 129, Game.java)

```
( !board.items[newPosition.x][newPosition.y].crate  )
```

fix (line 129, Game.java)

```
( board.isOpen(newPosition) )
```

Problem 1.2 - After the fix there is an error that it's not always a crate:

```
sokoban/src/Game.java:135: JML assertion is false
   //@ assert(board.items[newPosition.x][newPosition.y].crate);
```

```
if (board.isOpen(newPosition)) {
      player.setPosition (newPosition);
      return true;
   }else if(!board.items[newPosition.x][newPosition.y].crate){
          return false;
   }
```

Problem 2 - The player is able to walk diagonal

```
/media/sf_Software_Validation/assigments/assignment2/sokoban/src/Position.java:50: JML
postcondition is false
  boolean isValidNextPosition (Position newPosition) {
         ^
/media/sf_Software_Validation/assigments/assignment2/sokoban/src/Position.java:43:
Associated declaration:
/media/sf_Software_Validation/assigments/assignment2/sokoban/src/Position.java:50:
    @ ensures \result ==
     ^
```

original (line 59, Position.java)

```
if( dX >= -1 && dX <= 1 && dY >= -1 && dY <= 1) return true;
            return false;
```

fix (line 59, Position.java)

```
return ((dX == 0) && (dY == 1)) ||((dX == -1) && (dY == 0))||((dX == 1) && (dY == 0)) || ((dX
== 0) && (dY == -1));
```

PART 3 :: STATIC CHECKING

3.1 : wonGame ESC (nested)

Look for all board items if it is marked. If the board item is marked there should be a crate on it else it doesn't matter. By checking all we are checking this for every mark and know if we have won, because on every mark there is a crate.

```
requires true;
    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) || !board.items[x][y].marked
            ));
```

3.2 : wonGame ESC (loop invariant)

The static verification failed, although not on the loop we supplied, but on counter examples with invalid input parameters (negative x and y). We tried to solve this by supplying specifications for the relation between the board size and the size of the items list, both in invariants and in functions requirements in both Board.jaba and Game.java. (see included files)

Nevertheless, we did construct a loop invariant that should verify the function without the use of nested \forall models / specifications.

```
    /*@ loop_invariant
          ( (board.items[x][y].marked && board.items[x][y].crate) || !board.items[x][y].marked )
         && (x>=0 && x<board.xSize) && (y>=0 && y<board.ySize);
    @*/
```

3.3 : wonGame ESC (general case)

The general method we would describe, is that instead of specifying checking functions with nested loops similar to the loops, is to specify the loops in question using loop variants. The check in the \forall specification is very similar to the loop invariant.

PART 4 :: MODEL VARIABLES

4.1 : wonGame model

```
/*@ public model boolean wonGame;
        @ represents wonGame <- (\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) || !board.items[x][y].marked
        )
        );
        @*/
```

4.2 : gameStuck model

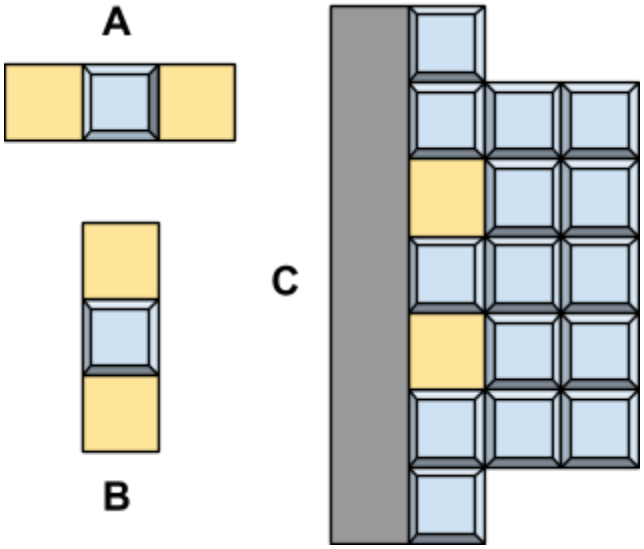
```
/*@ public model boolean gameStuck;
  @ represents gameStuck <- !(\exists int x; x >= 1 && x < board.xSize - 1;
        (\exists int y; y >= 1 && y < board.ySize - 1;
                (board.items[x][y].crate && (
                (board.items[x - 1][y].ground && !board.items[x - 1][y].crate && board.items[x +
1][y].ground && !board.items[x + 1][y].crate) ||
                (board.items[x][y - 1].ground && !board.items[x][y - 1].crate && board.items[x][y +
1].ground && !board.items[x][y + 1].crate) ))
        )
        );
  @*/
```

4.3 : gameStuck invariant

```
//@ public invariant !gameStuck || wonGame;
```

4.4 : gameStuck model

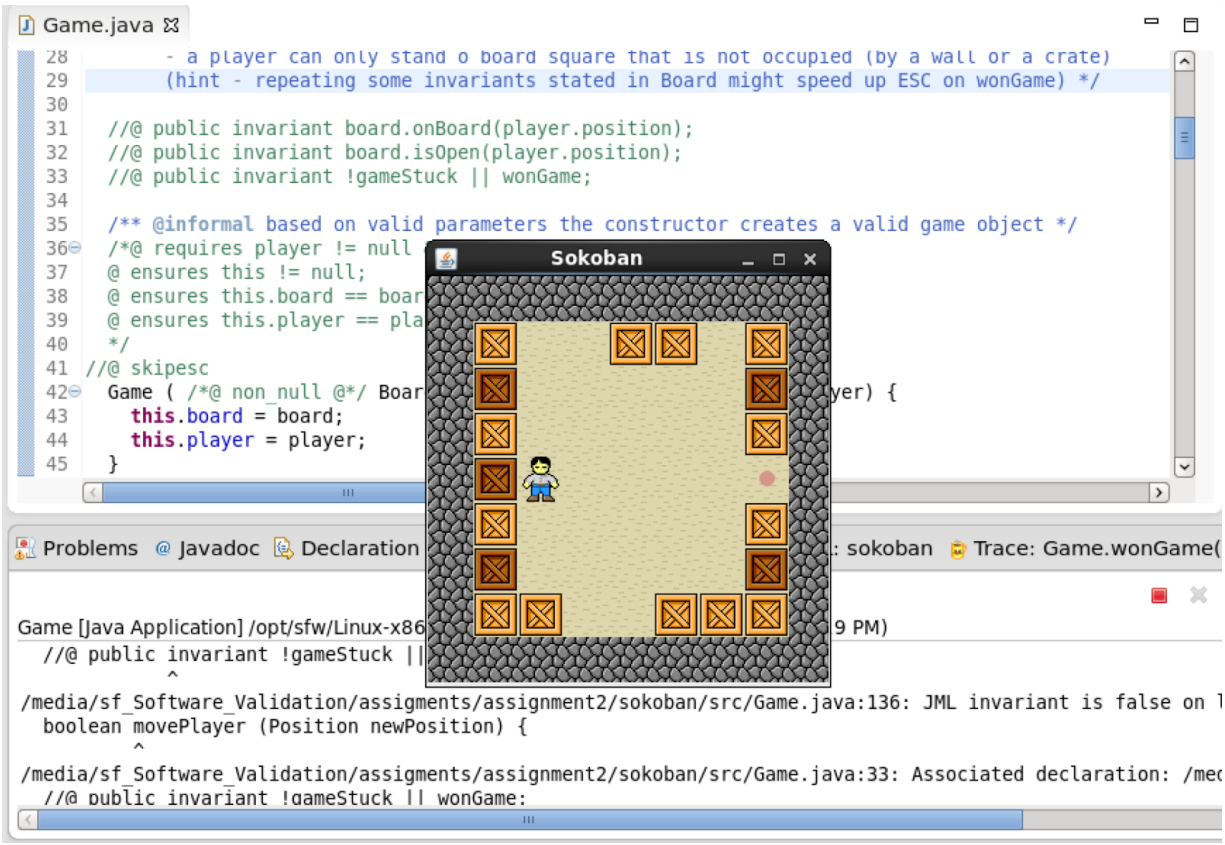
The simplest check we could think of was to check if there exists a crate on the board that is moveable. A crate is moveable when both horizontal neighbours are free (situation A) or both vertical neighbours are free (situation B). One is needed for the player, the other for the crate to move to. This is in theory not sufficient, since situations such a situation C can be created, where the game would still be stuck, but this cannot occur in the current layout of the game.



Trace :

```
sokoban/src/Game.java:146: JML caller invariant is false on leaving calling method (Caller:
Game.movePlayer(Position), Callee: Board.isOpen(Position))
    if (board.isOpen(newPosition)) {

sokoban/src/Game.java:33: Associated declaration: sokoban/src/Game.java:146:
  //@ public invariant !gameStuck || wonGame;

sokoban/src/Game.java:165: JML caller invariant is false on leaving calling method (Caller:
Game.movePlayer(Position), Callee: Board.isOpen(int,int))
    if (!board.isOpen(nX, nY)) {

sokoban/src/Game.java:33: Associated declaration: sokoban/src/Game.java:165:
  //@ public invariant !gameStuck || wonGame;

sokoban/src/Game.java:136: JML invariant is false on leaving method Game.movePlayer(Position)
  boolean movePlayer (Position newPosition) {

sokoban/src/Game.java:33: Associated declaration: sokoban/src/Game.java:136:
  //@ public invariant !gameStuck || wonGame;
```

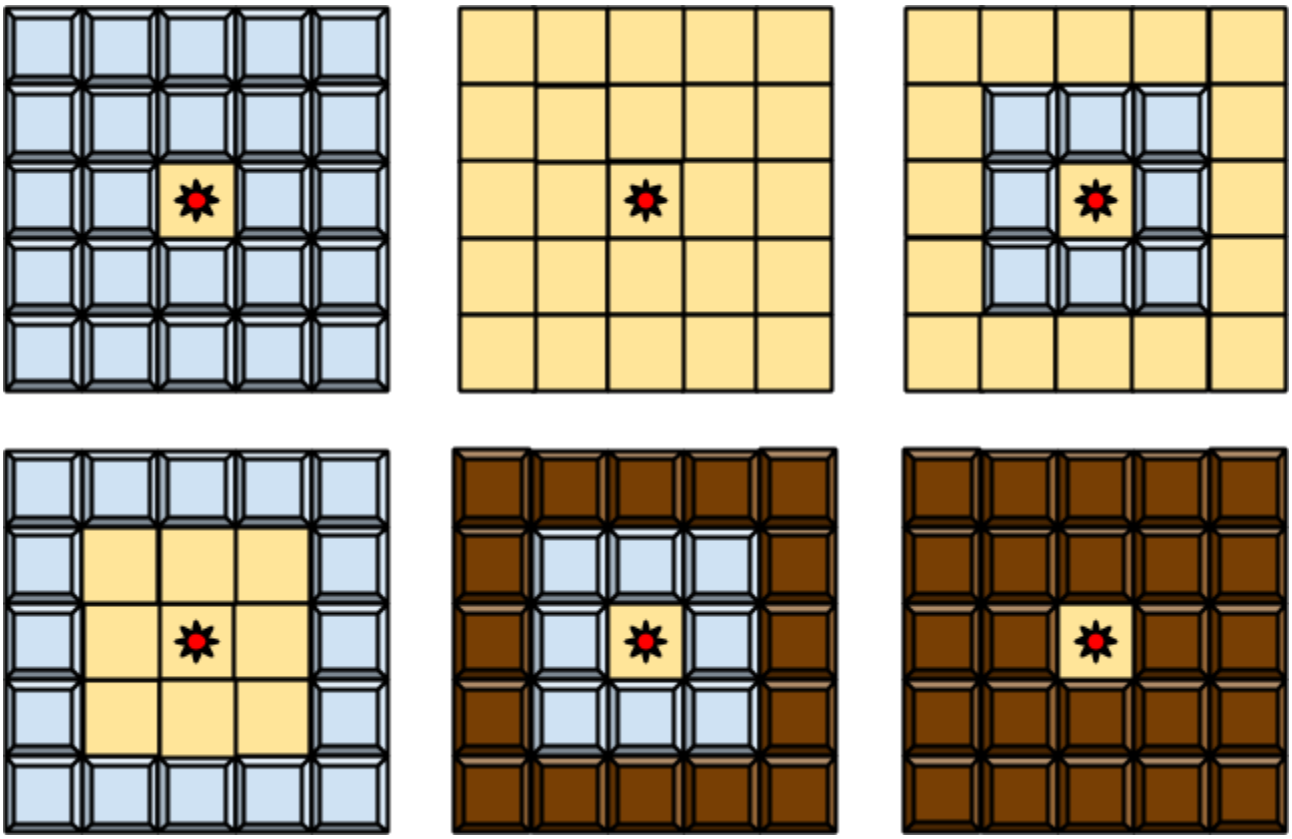
Screenshot:


Game.java
- a player can only stand o board square that is not occupied (by a wall or a crate)
(hint - repeating some invariants stated in Board might speed up ESC on wonGame) */
//@ public invariant board.onBoard(player.position);
//@ public invariant board.isOpen(player.position);
//@ public invariant !gameStuck || wonGame;
/** @informal based on valid parameters the constructor creates a valid game object */
/*@ requires player != null
@ ensures this != null;
@ ensures this.board == boar
@ ensures this.player == pla
*/
//@ skipesc
Game (/*@ non_null @*/ Boar
yer) {
this.board = board;
this.player = player;
}
Sokoban
Problems
Javadoc
Declaration
: sokoban
Trace: Game.wonGame(
Game [Java Application] /opt/sfw/Linux-x86
9 PM)
//@ public invariant !gameStuck ||
^
/media/sf_Software_Validation/assigments/assignment2/sokoban/src/Game.java:136: JML invariant is false on
boolean movePlayer (Position newPosition) {
^
/media/sf_Software_Validation/assigments/assignment2/sokoban/src/Game.java:33: Associated declaration: /me
//@ public invariant !gameStuck || wonGame;

PART 5 :: TEST GENERATION

4.1 : Test Strategy

We wanted to test the following situations. Here the yellow fields squares are open sections, the blue squares are crates, and the brown squares are walls. We want to test the moves to all possible 9 surrounding squares of the player, and a move of more than 1 square.



For this we altered added these test factories:

Game_InstanceStrategy.java:

```
/**
 * @return local-scope instances of Game.
 */
public RepeatedAccessIterator<?> localValues() {
        return new ObjectArrayIterator<Object>
        (new Object[]
        { genGame (0), genGame(1), genGame(2), genGame(3), genGame(4) , genGame(5)});
}
```

Here genGame(n) is a factory that instantiates a board in the situations given above, 0 being the left top most case, 5 the right down most case etc. The altered files is included with this document. It was validated using a println statement to print a few states of the board for each call of genGame.

Game_movePlayer__Position_newPosition__0__newPosition.java:

```
public Position genPos (int x, int y) {
        return new Position (x,y);
}

public RepeatedAccessIterator<?> localValues() {
        return new ObjectArrayIterator<Object>
        (new Object[]
        { genPos(3,3), genPos(3,4), genPos(3,5), genPos(4,3), genPos(4,5), genPos(5,3), genPos(5,4),
          genPos(5,5), genPos(6,6), genPos(-1,-1),genPos(0,0) });
}
```

Here genPos(x,y) is a factory that instantiates a position object with position (x,y) the player starts at position (4,4), so the generated positions are all the directly surrounding positions of the player, plus a position that is 2 fields away, and 0 fields away. Negative values already violate the position constructor specifications, and can therefor not be generated without tricks outside the scope of this project.

4.2 : Test Specifications

The intention of the test specifications was to introduce 5 valid cases, in which the player position is updated, and the board is updated correctly. For all other cases, the player and the board should not change, and the result should be false.

This written using model variables to shorten the specifications a little. These are given below, and represent a test of whether the player is next to a crate: :

```
/*@ model boolean leftCrate;
@ represents leftCrate <- (board.items[player.position.x - 1][player.position.y].crate);
@*/
/*@ model boolean rightCrate;
@ represents rightCrate <- (board.items[player.position.x + 1][player.position.y].crate);
@*/
/*@ model boolean upCrate;
@ represents upCrate <- (board.items[player.position.x][player.position.y - 1].crate);
@*/
/*@ model boolean downCrate;
@ represents downCrate <- (board.items[player.position.x][player.position.y + 1].crate);
@*/
```

The function specification then becomes:

```
/*@
normal_behaviour
requires board.isOpen(newPosition);
requires player.position.isValidNextPosition(newPosition);
ensures player.position == newPosition;
also
normal_behaviour
requires player.position.isValidNextPosition(newPosition);
requires !board.isOpen(newPosition) && leftCrate && (newPosition.x - player.position.x) == -1 && (newPosition.y
- player.position.y) == 0 && board.isOpen(player.position.x - 2, player.position.y);
ensures player.position == newPosition;
ensures board.items[newPosition.x - 1][newPosition.y].crate;
also
normal_behaviour
requires player.position.isValidNextPosition(newPosition);
requires !board.isOpen(newPosition) && rightCrate && (newPosition.x - player.position.x) == 1 &&
(newPosition.y - player.position.y) == 0 && board.isOpen(player.position.x + 2, player.position.y);
ensures player.position == newPosition;
ensures board.items[newPosition.x + 1][newPosition.y].crate;
also
normal_behaviour
requires player.position.isValidNextPosition(newPosition);
requires !board.isOpen(newPosition) && upCrate && (newPosition.y - player.position.y) == -1 && (newPosition.x
- player.position.x) == 0 && board.isOpen(player.position.x, player.position.y - 2);
ensures player.position == newPosition;
ensures board.items[newPosition.x][newPosition.y - 1].crate;
also
normal_behaviour
requires player.position.isValidNextPosition(newPosition);
requires !board.isOpen(newPosition) && downCrate && (newPosition.y - player.position.y) == 1 &&
(newPosition.x - player.position.x) == 0 && board.isOpen(player.position.x, player.position.y + 2);
ensures player.position == newPosition;
ensures board.items[newPosition.x][newPosition.y + 1].crate;
also
normal_behaviour
requires !(board.isOpen(newPosition));
requires !(!board.isOpen(newPosition) && leftCrate && (newPosition.x - player.position.x) == -1 &&
(newPosition.y - player.position.y) == 0 && board.isOpen(player.position.x - 2, player.position.y));
requires !(!board.isOpen(newPosition) && rightCrate && (newPosition.x - player.position.x) == 1 &&
(newPosition.y - player.position.y) == 0 && board.isOpen(player.position.x + 2, player.position.y));
requires !(!board.isOpen(newPosition) && upCrate && (newPosition.y - player.position.y) == -1 &&
(newPosition.x - player.position.x) == 0 && board.isOpen(player.position.x, player.position.y - 2));
requires !(!board.isOpen(newPosition) && downCrate && (newPosition.y - player.position.y) == 1 &&
(newPosition.x - player.position.x) == 0 && board.isOpen(player.position.x, player.position.y + 2));
ensures (player.position == \old(player.position)) && (\result == false);
ensures (board == \old(board));
 @*/

 boolean movePlayer ( /*@ non_null @*/ Position newPosition) { ...
```

When this is compiled and run, we get 67 tests (10 positions x 6 game objects, + 6 null objects + 1 racEnabled test), all of which pass except for the null object tests which were skipped legaly.