Netta Budniski 215016239
Saggie Mishan 326574738

# Final Project – Formal Verification

## Part 1

1.

General definitions:

- B – represents a general n x m Sokoban board in XSB format.
- i,j – represents the coordinates of the keeper on the board.

$$0 \le i \le n-1\ , 0 \le j \le m-1$$

Our FDS:

$$D = \{V, \theta, \rho, J, C\}$$

$$V = \{B, i\ , j\ , up, down, left, right\}$$

$$\rho = \rho_{up} \vee \rho_{down} \vee \rho_{left} \vee \rho_{right}$$

Each $\rho_{direction}$ is dependent on the keeper location and its surroundings on the board. Possible scenarios: keeper is on a floor and a floor is next to him in a direction, keeper is on a floor and a wall is next to him etc. For each one of the cases we will address in the code and consider them before changing the keeper location.

We will define four instances of a Boolean value that will indicate if the keeper can go in a specific direction, one for each direction:

$$up(B,i,j) = !\left((i=0) \vee ((i \ge 1) \wedge (B[i-1][j] = \#)) \vee ((i \ge 2) \wedge (B[i-1][j] \in \{*,\$\}) \wedge (B[i-2][j] \in \{*,\$,\#\}))\right)$$

$$down(B,i,j) = !\left((i=n-1) \vee ((i \le n-2) \wedge (B[i+1][j] = \#)) \vee ((i \le n-3) \wedge (B[i+1][j] \in \{*,\$\}) \wedge (B[i+2][j] \in \{*,\$,\#\}))\right)$$

$$left(B,i,j) = !\left((j=0) \vee ((j \ge 1) \wedge (B[i\ ][j-1] = \#)) \vee ((j \ge 2) \wedge (B[i][j-1] \in \{*,\$\}) \wedge (B[i][j-2] \in \{*,\$,\#\}))\right)$$

$$right(B,i,j) = !\left((j=n-1) \vee ((j \le n-2) \wedge (B[i][j+1] = \#)) \vee ((j \le n-3) \wedge (B[i][j+1] \in \{*,\$\}) \wedge (B[i][j+2] \in \{*,\$,\#\}))\right)$$

2.

The temporal logic specification for a win of the Sokoban board is:

$$F(\$ \notin B)$$

Explanation: A win is achieved by making sure that every box is located on a goal. So if finally there are not any boxes left on the board not on a goal- we achieved a win.
In order to make this logic specification simple to automate it using Python, we will define it as such (for N boxes):

$$F\big((box_1\ is\ on\ goal) \wedge \ldots \wedge (box_N\ is\ on\ goal)\big)$$

Note: We will add a ! (not) before this expression in our Python file and then check if the outcome is false. That way if the output is false that the board is winnable, otherwise its

not. We decided to do so because in that way when the board is winnable we will get a possible route for solving it.

**Part 2**

1.

Our SMV files are in the GIT directory.

2+3.

The commend we use to run our nuXmv are:

nuxmv -int

read_model -i filename.smv

flatten_hierarchy

encode_variables

build_model

check_ltlspec

Board 1:

```
game = """\
#####
#@$.#
#####
"""
```

Output for board 1:

```
-- specification !( F pos_13 = 2)  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    pos_11 = 3
    pos_12 = 2
    pos_13 = 1
    steps_counter = 0
  -- Loop starts here
  -> State: 1.2 <-
    pos_11 = 1
    pos_12 = 3
    pos_13 = 2
    steps_counter = 1
  -- Loop starts here
  -> State: 1.3 <-
  -> State: 1.4 <-
```

We can see that we got false as an output, as we explained in part 1 when the output is false out boars is winnable. We also get the route for winning.

Board 2:

```
game = """\
#######
###.###
###$###
#.$@$.#
###$###
###.###
#######
"""
```

Output for board 2:

```
-- specification !( F (((pos13 = 2 & pos31 = 2) & pos35 = 2) & pos53 = 2))  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    pos13 = 1
    pos23 = 2
    pos31 = 1
    pos32 = 2
    pos33 = 3
    pos34 = 2
    pos35 = 1
    pos43 = 2
    pos53 = 1
    steps_counter = 0
  -- Loop starts here
  -> State: 1.2 <-
    pos13 = 2
    pos23 = 1
    pos31 = 2
    pos32 = 3
    pos33 = 1
    pos34 = 1
    pos35 = 2
    pos43 = 1
    pos53 = 2
    steps_counter = 1
  -- Loop starts here
  -> State: 1.3 <-
  -> State: 1.4 <-
```

We can see that we got false as an output, as we explained in part 1 when the output is false out boars is winnable. We also get the route for winning.

Board 3:

```
game = """\
########
#@__$$##
########
#___..##
########
"""
```

Output for board 3:

```
-- specification !( F (pos34 = 2 & pos35 = 2))  is true
```

As we can see we got true as an output- which means the board is not winnable.

**Part 3**

1.

Board 1:

```
game = """\
#####
#$@.#
#####
"""
```

Output for board 1:

```
Output is in the file: sokoban_bdd.out
Time: BDD engine: 0.07 seconds
Output is in the file: sokoban_sat.out
Time: SAT engine: 0.05 seconds

Performance Comparison:
BDD Engine: 0.07 seconds
SAT Engine: 0.05 seconds
SAT has better performences.
```

Board 2:

```
game = """\
#######
###.###
###$###
#.$@$.#
###$###
###.###
#######
"""
```

Output for board 2:

```
Output is in the file: sokoban_bdd.out
Time: BDD engine: 0.13 seconds
Output is in the file: sokoban_sat.out
Time: SAT engine: 0.12 seconds

Performance Comparison:
BDD Engine: 0.13 seconds
SAT Engine: 0.12 seconds
SAT has better performences.
```

Board 3:

```
game = """\
########
#@__$$##
########
#___..##
########
"""
```

Output for board 3:

```
Output is in the file: sokoban_bdd.out
Time: BDD engine: 0.13 seconds
Output is in the file: sokoban_sat.out
Time: SAT engine: 0.13 seconds

Performance Comparison:
BDD Engine: 0.13 seconds
SAT Engine: 0.13 seconds
SAT has better performences.
```

2.

As we can see in our measurements, BDD is faster or equal to the SAT in all of our measurements. Therefore, it is better to use BDD engine when time is a priority of the developers.

## Part 4

1.

In order to break the problem into sub problems we will solve the board for each box at a time. Each box will have its own loop, and each iteration has its own temporal logic formulae. For iteration k:

$$F(box_k \ is \ on \ goal)$$

Explanation: we will iterate on the board N times - for N boxes. Each iteration we will focus on one box only and treat the other N-1 ones as walls.

After iterating N times, we will check the output of each iteration of its own. If at least one iteration was that the board is not winnable- then we can declare that the board is not winnable. Otherwise, we declare it is winnable.

2.

For a given board with N boxes on it – there would be N iterations.

Board 1:

```
game = """\
#####
#@$.#
#####
"""
```

Output for board 1:

```
Output saved to sokoban_temp.out
For loop number 1: the game with box at (1, 1) to goal at (1, 3) - is winnable!
For loop number 1 the runtime is: 0.27 seconds
number of loops: 1
Total Runtime: 0.28 seconds
```

As we can see we have one iteration as expected and the board is indeed winnable.

Board 2:

```
game = """\
#######
###.###
###$###
#.$@$.#
###$###
###.###
#######
"""
```

Output for board 2:

```
Output saved to sokoban_temp.out
For loop number 1: the game with box at (2, 3) to goal at (1, 3) - is winnable!
For loop number 1 the runtime is: 0.09 seconds
number of loops: 1
Total Runtime: 0.12 seconds
```

```
Output saved to sokoban_temp.out
For loop number 2: the game with box at (3, 2) to goal at (3, 1) - is winnable!
For loop number 2 the runtime is: 0.07 seconds
number of loops: 2
Total Runtime: 0.19 seconds
```

```
Output saved to sokoban_temp.out
For loop number 3: the game with box at (3, 4) to goal at (3, 5) - is winnable!
For loop number 3 the runtime is: 0.14 seconds
number of loops: 3
Total Runtime: 0.34 seconds
```

```
Output saved to sokoban_temp.out
For loop number 4: the game with box at (4, 3) to goal at (5, 3) - is winnable!
For loop number 4 the runtime is: 0.11 seconds
number of loops: 4
Total Runtime: 0.48 seconds
```

As we can see there were four iterations – as expected. Each one of them is winnable- so the board is winnable.