Formal Verification and Synthesis

Erel Dekel 326064888
Boaz Gurevich 325813970

Date: 12.05.24

https://github.com/BoazGur/SokobanVerficiation

Part 1:

1. The FDS is the group: $D = \{V, \theta, \rho, J, C\}$

$V = \{board, x, y, turn, possible\_up, possible\_down, possible\_right, possible\_left\}$

When the variables are defined as:

board: array representation of the map, $n \times m$ matrix (XSB format) **in the nuXmv it's defined as a set of n*m variables (v_ij) which represent board[i][j], the nuXmv didn't work for us with a 2D array, also in the nuXmv we translated the XSB format to words:**
{@, +, $, *, #, ., -} -> {shtrudel, plus, dollar, star, solamit, dot, minus}

x: int, x-position of the keeper on board $s.t\ 0 \leq x < m$

y: int, y-position of the keeper on board $s.t\ 0 \leq y < n$

possible_u: boolean, true if the keeper can go up

possible_d: boolean, true if the keeper can go down

possible_r: boolean, true if the keeper can go right

possible_l: boolean, true if the keeper can go left

turn: enum, representation of action taken, $\{u, d, l, r, none\}$

***in the nuXmv we also defined 3 constants, n: # of rows, m: # of columns, done: boolean, true if there are no dollars (boxes).**

$\theta(starting\ board) = (turn = None) \wedge (board = input) \wedge (x = input.findcol(@ \vee +)) \wedge$
$\wedge (y = input.findrow(@ \vee +)) \wedge (possible\_u = up(x, y, board)) \wedge$
$\wedge (possible\_d = down(x, y, board)) \wedge (possible\_r = right(x, y, board)) \wedge$
$\wedge (possible\_l = left(x, y, board))$

Notice the board is given as an input to the nuXmv. The turn starts as None.

x and y are determined by the input where the keeper was found.

Let's define the functions up, down, right, and left. Which determine the behavior of possible_u, possible_d, possible_r, and possible_l respectively.

$possible\_u = !((y = 0) \vee ((y > 1) \wedge (input[y - 1][x] = \#)) \vee$
$\vee ((y > 1) \wedge (input[y - 1][x] \in \{\$, *\}) \wedge (input[y - 2][x] \in \{\$, *, \#\})))$

$possible\_d = !((y = n - 1) \vee ((y < n - 1) \wedge (input[y + 1][x] = \#)) \vee$
$\vee ((y < n - 2) \wedge (input[y + 1][x] \in \{\$, *\}) \wedge (input[y + 2][x] \in \{\$, *, \#\})))$

$possible\_l = !((x = 0) \vee ((x > 1) \wedge (input[y][x - 1] = \#)) \vee$
$\vee ((x > 1) \wedge (input[y][x - 2] \in \{\$, *\}) \wedge (input[y][x - 2] \in \{\$, *, \#\})))$

$possible\_r = !((x = m - 1) \vee ((x < m - 1) \wedge (input[y][x + 1] = \#)) \vee$
$\vee ((x < m - 2) \wedge (input[y][x + 1] \in \{\$, *\}) \wedge (input[y][x + 2] \in \{\$, *, \#\})))$

ρ consist of the transition of all the variables so we will defined each one here:

$possible\_u' = !((y = 0) \lor ((y > 1) \land (input[y - 1][x] = \#)) \lor$
$\lor ((y > 1) \land (input[y - 1][x] \in \{\$, *\}) \land (input[y - 2][x] \in \{\$, *, \#\})))$

$possible\_d' = !((y = n - 1) \lor ((y < n - 1) \land (input[y + 1][x] = \#)) \lor$
$\lor ((y < n - 2) \land (input[y + 1][x] \in \{\$, *\}) \land (input[y + 2][x] \in \{\$, *, \#\})))$

$possible\_l' = !((x = 0) \lor ((x > 1) \land (input[y][x - 1] = \#)) \lor$
$\lor ((x > 1) \land (input[y][x - 2] \in \{\$, *\}) \land (input[y][x - 2] \in \{\$, *, \#\})))$

$possible\_r' = !((x = m - 1) \lor ((x < m - 1) \land (input[y][x + 1] = \#)) \lor$
$\lor ((x < m - 2) \land (input[y][x + 1] \in \{\$, *\}) \land (input[y][x + 2] \in \{\$, *, \#\})))$

$\rho_{turn} = ((done = true) \land (turn' = None)) \lor$
$\lor ((possible\_u' = true) \land (possible\_d' = true) \land (possible\_r' = true) \land (possible\_l' = true) \land$
$\lor (turn = \{none, u, d, r, l\})) \lor$
$\lor ((possible\_u' = true) \land (possible\_d' = true) \land (possible\_l' = true) \land (turn' = \{none, u, d, l\})) \lor$
$\lor ((possible\_u' = true) \land (possible\_d' = true) \land (possible\_r' = true) \land (turn' = \{none, u, d, r\})) \lor$
$\lor ((possible\_u' = true) \land (possible\_r' = true) \land (possible\_l' = true) \land (turn' = \{none, u, r, l\})) \lor$
$\lor ((possible\_d' = true) \land (possible\_r' = true) \land (possible\_l' = true) \land (turn' = \{none, d, r, l\})) \lor$
$\lor ((possible\_d' = true) \land (possible\_r' = true) \land (turn' = \{none, d, r\})) \lor$
$\lor ((possible\_d' = true) \land (possible\_l' = true) \land (turn' = \{none, d, l\})) \lor$
$\lor ((possible\_u' = true) \land (possible\_r' = true) \land (turn' = \{none, u, r\})) \lor$
$\lor ((possible\_u' = true) \land (possible\_l' = true) \land (turn' = \{none, u, l\})) \lor$
$\lor ((possible\_d' = true) \land (possible\_u' = true) \land (turn' = \{none, d, u\})) \lor$
$\lor ((possible\_l' = true) \land (possible\_r' = true) \land (turn' = \{none, l, r\})) \lor$
$\lor ((possible\_u' = true) \land (turn' = \{none, u\})) \lor$
$\lor ((possible\_d' = true) \land (turn' = \{none, d\})) \lor$
$\lor ((possible\_r' = true) \land (turn' = \{none, r\})) \lor$
$\lor ((possible\_l' = true) \land (turn' = \{none, l\})) \lor$
$\lor ((possible\_u' = false) \land (possible\_d' = false) \land (possible\_l' = false) \land (turn' = none))$

The idea of turn transition is that if done so turn is None, but if not done then we check cases:
If $possible\_u' = true$ then $u \in turn'$, If $possible\_d' = true$ then $d \in turn'$
If $possible\_r' = true$ then r', If $possible\_l' = true$ then $l \in turn'$
And always $None \in turn'$

$$\rho_x = ((turn' = r) \land (x < m - 1) \land (x' = x + 1)) \lor ((turn' = l) \land (x > 0) \land (x' = x - 1))$$

$$\rho_y = ((turn' = d) \land (y < n - 1) \land (y' = y + 1)) \lor ((turn' = u) \land (y > 0) \land (y' = y - 1))$$

$\rho_{board[i][j]}$ = $((y = i) \land (x = j) \land (board[i][j] = @) \land (turn' \,!= None) \land (board[i][j]' =-)) \lor$
$((y = i) \land (x = j) \land (board[i][j] =+) \land (turn' \,!= None) \land (board[i][j]' =.)) \lor$
$\lor ((y = i) \land (x = j - 1 >- 1) \land (board[i][j] \in \{-, \$\}) \land (turn' = r) \land (board[i][j]' = @)) \lor$
$\lor ((y = i) \land (x = j - 1 >- 1) \land (board[i][j] \in \{., *\}) \land (turn' = r) \land (board[i][j]' =+)) \lor$
$\lor ((y = i) \land (x = j - 2 >- 2) \land (board[i][j - 1] \in \{*, \$\}) \land (turn' = r) \land (board[i][j] =-) \land$
$\land (board[i][j]' = \$)) \lor$
$\lor ((y = i) \land (x = j - 2 >- 2) \land (board[i][j - 1] \in \{*, \$\}) \land (turn' = r) \land (board[i][j] =.) \land$
$\land (board[i][j]' =*)) \lor$
$\lor ((y = i) \land (x = j + 1 < m) \land (board[i][j] \in \{-, \$\}) \land (turn' = l) \land (board[i][j]' = @)) \lor$
$\lor ((y = i) \land (x = j + 1 < m) \land (board[i][j] \in \{., *\}) \land (turn' = l) \land (board[i][j]' =+)) \lor$
$\lor ((y = i) \land (x = j + 2 > m) \land (board[i][j + 1] \in \{*, \$\}) \land (turn' = l) \land (board[i][j] =-) \land$
$\land (board[i][j]' = \$)) \lor$
$\lor ((y = i) \land (x = j + 2 > m) \land (board[i][j + 1] \in \{*, \$\}) \land (turn' = l) \land (board[i][j] =.) \land$
$\land (board[i][j]' =*)) \lor$
$\lor ((y = i - 1 >- 1) \land (x = j) \land (board[i][j] \in \{-, \$\}) \land (turn' = d) \land (board[i][j]' = @)) \lor$
$\lor ((y = i - 1 >- 1) \land (x = j) \land (board[i][j] \in \{., *\}) \land (turn' = d) \land (board[i][j]' =+)) \lor$
$\lor ((y = i - 2 >- 2) \land (x = j) \land (board[i - 1][j] \in \{*, \$\}) \land (turn' = d) \land (board[i][j] =-) \land$
$\land (board[i][j]' = \$)) \lor$
$\lor ((y = i - 2 >- 2) \land (x = j) \land (board[i - 1][j] \in \{*, \$\}) \land (turn' = d) \land (board[i][j] =.) \land$
$\land (board[i][j]' =*)) \lor$
$\lor ((y = i + 1 < n) \land (x = j) \land (board[i][j] \in \{-, \$\}) \land (turn' = u) \land (board[i][j]' = @)) \lor$
$\lor ((y = i + 1 < n) \land (x = j) \land (board[i][j] \in \{., *\}) \land (turn' = u) \land (board[i][j]' =+)) \lor$
$\lor ((y = i + 2 < n) \land (x = j) \land (board[i + 1][j] \in \{*, \$\}) \land (turn' = u) \land (board[i][j] =-) \land$
$\land (board[i][j]' = \$)) \lor$
$\lor ((y = i + 2 < n) \land (x = j) \land (board[i + 1][j] \in \{*, \$\}) \land (turn' = u) \land (board[i][j] =.) \land$
$\land (board[i][j]' =*))$

J are states that need to repeat an infinite amount of times, in our problem we want that?

C are pairs of states that if one repeat an infinite amount of times the other also, so in our problem we want that.

Part 1:
2. The LTLSPEC we defined for our problem is this: F(done), which means that for a win eventually done will be true.

Part 2:

1. The code and the nuXmv's and the output files can be found in this GitHub repo:
   https://github.com/BoazGur/SokobanVerficiation.

2. We converted all the board examples that was provided in the last page of the exercise to XSB format and run the Python script on them.
   Most of them worked great with just the command "nuXmv <file name>".
   Just the last board (board7) took too long so we ran it with BMC manually with the commands "nuXmv -int <file name>" and then "go_bmc" and then "check_ltlspec_bmc -k 15"

Board1.out:

```
27    -- specification !( F done)  is false
28    -- as demonstrated by the following execution sequence
29    Trace Description: LTL Counterexample
30    Trace Type: Counterexample
31      -> State: 1.1 <-
32        turn = none
33        possible_up = FALSE
34        possible_down = FALSE
35        possible_right = TRUE
36        possible_left = FALSE
37        y = 1
38        x = 1
39        v_00 = solamit
40        v_01 = solamit
41        v_02 = solamit
42        v_03 = solamit
43        v_04 = solamit
44        v_10 = solamit
45        v_11 = shtrudel
46        v_12 = dollar
47        v_13 = dot
48        v_14 = solamit
49        v_20 = solamit
50        v_21 = solamit
51        v_22 = solamit
52        v_23 = solamit
53        v_24 = solamit
54        done = FALSE
55        m = 5
56        n = 3
57      -> State: 1.2 <-
58        turn = r
59        x = 2
60        v_11 = minus
61        v_12 = shtrudel
62        v_13 = star
63        done = TRUE
64    -- Loop starts here
65      -> State: 1.3 <-
66        turn = none
67        possible_right = FALSE
68        possible_left = TRUE
69      -> State: 1.4 <-
70
```
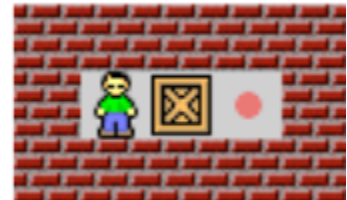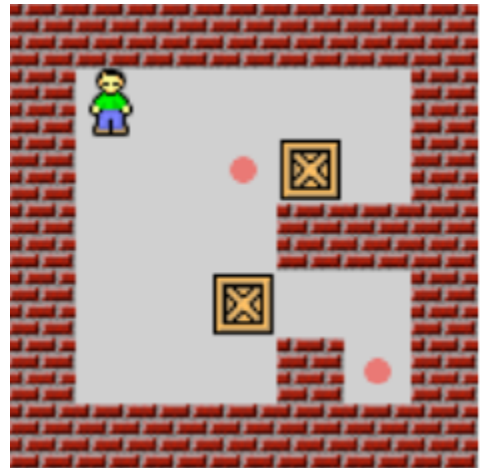
Board2.out:

```
27      -- specification !( F done)  is true
28
```


Board3.out:
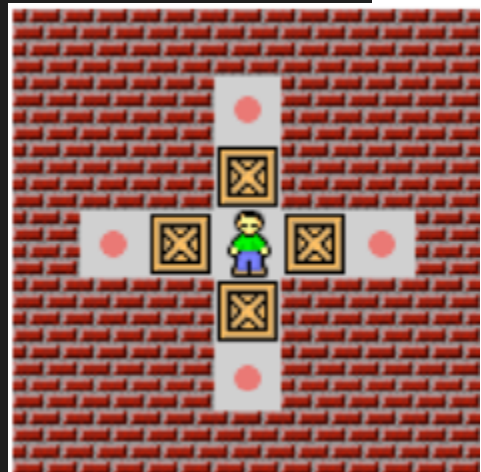
```
27      -- specification !( F done)  is true
28
```
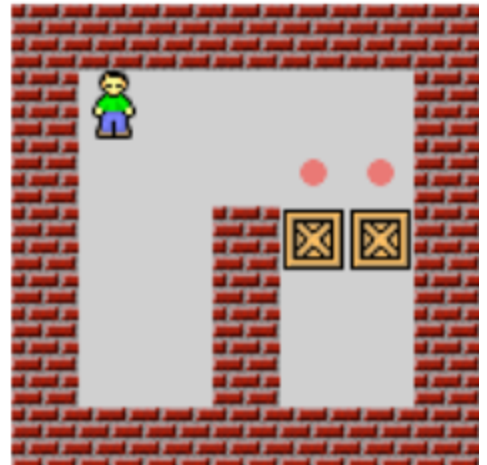
Board4.out:

```
27    -- specification !( F done)  is false
28    -- as demonstrated by the following ex
29    Trace Description: LTL Counterexample
30    Trace Type: Counterexample
31      -> State: 1.1 <-
32        turn = none
33        possible_up = TRUE
34        possible_down = TRUE
35        possible_right = TRUE
36        possible_left = TRUE
37        y = 3
38        x = 3
39        v_00 = solamit
40        v_01 = solamit
41        v_02 = solamit
42        v_03 = solamit
43        v_04 = solamit
44        v_05 = solamit
45        v_06 = solamit
46        v_10 = solamit
47        v_11 = solamit
48        v_12 = solamit
49        v_13 = dot
50        v_14 = solamit
51        v_15 = solamit
52        v_16 = solamit
53        v_20 = solamit
54        v_21 = solamit
55        v_22 = solamit
56        v_23 = dollar
57        v_24 = solamit
58        v_25 = solamit
59        v_26 = solamit
60        v_30 = solamit
61        v_31 = dot
62        v_32 = dollar
63        v_33 = shtrudel
64        v_34 = dollar
65        v_35 = dot
66        v_36 = solamit
67        v_40 = solamit
68        v_41 = solamit
69        v_42 = solamit
70        v_43 = dollar
71        v_44 = solamit
72        v_45 = solamit
73        v_46 = solamit
74        v_50 = solamit
75        v_51 = solamit
76        v_52 = solamit
77        v_53 = dot
78        v_54 = solamit
79        v_55 = solamit
```

```
80        v_56 = solamit
81        v_60 = solamit
82        v_61 = solamit
83        v_62 = solamit
84        v_63 = solamit
85        v_64 = solamit
86        v_65 = solamit
87        v_66 = solamit
88        done = FALSE
89        m = 7
90        n = 7
91   v  -> State: 1.2 <-
92        turn = d
93        y = 4
94        v_33 = minus
95        v_43 = shtrudel
96        v_53 = star
97   v  -> State: 1.3 <-
98        turn = u
99        possible_down = FALSE
100       possible_right = FALSE
101       possible_left = FALSE
102       y = 3
103       v_33 = shtrudel
104       v_43 = minus
105  v  -> State: 1.4 <-
106       possible_down = TRUE
107       possible_right = TRUE
108       possible_left = TRUE
109       y = 2
110       v_13 = star
111       v_23 = shtrudel
112       v_33 = minus
113  v  -> State: 1.5 <-
114       turn = d
115       possible_up = FALSE
116       possible_right = FALSE
117       possible_left = FALSE
118       y = 3
119       v_23 = minus
120       v_33 = shtrudel
121  v  -> State: 1.6 <-
122       turn = r
123       possible_up = TRUE
124       possible_right = TRUE
125       possible_left = TRUE
126       x = 4
127       v_33 = minus
128       v_34 = shtrudel
129       v_35 = star
```

```
130       -> State: 1.7 <-
131        turn = l
132        possible_up = FALSE
133        possible_down = FALSE
134        possible_right = FALSE
135        x = 3
136        v_33 = shtrudel
137        v_34 = minus
138       -> State: 1.8 <-
139        possible_up = TRUE
140        possible_down = TRUE
141        possible_right = TRUE
142        x = 2
143        v_31 = star
144        v_32 = shtrudel
145        v_33 = minus
146        done = TRUE
147     -- Loop starts here
148       -> State: 1.9 <-
149        turn = none
150        possible_up = FALSE
151        possible_down = FALSE
152        possible_left = FALSE
153       -> State: 1.10 <-
154
```
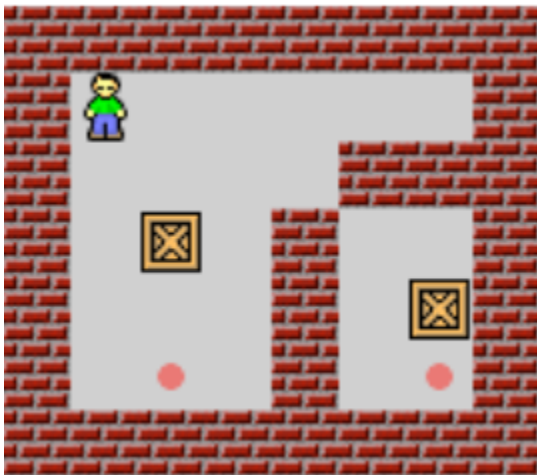
Board5.out:

```
27    -- specification !( F done)  is true
28
```


Board6.out:

```
27    -- specification !( F done)  is true
28
```

Board7.out:

```
nuXmv > check_ltlspec_bmc -k 15
-- no counterexample found with bound 0
-- no counterexample found with bound 1
-- no counterexample found with bound 2
-- no counterexample found with bound 3
-- no counterexample found with bound 4
-- no counterexample found with bound 5
-- no counterexample found with bound 6
-- no counterexample found with bound 7
-- no counterexample found with bound 8
-- no counterexample found with bound 9
-- no counterexample found with bound 10
-- specification !( F done)    is false
-- as demonstrated by the following execu
Trace Description: BMC Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    turn = none
    possible_up = TRUE
    possible_down = TRUE
    possible_right = FALSE
    possible_left = TRUE
    y = 4
    x = 3
    v_00 = solamit
    v_01 = solamit
    v_02 = solamit
    v_03 = solamit
    v_04 = solamit
    v_05 = solamit
    v_06 = solamit
    v_07 = solamit
    v_08 = solamit
    v_09 = solamit
    v_010 = solamit
    v_10 = solamit
    v_11 = minus
    v_12 = minus
    v_13 = minus
    v_14 = minus
    v_15 = minus
    v_16 = minus
    v_17 = solamit
    v_18 = solamit
    v_19 = solamit
    v_110 = solamit
    v_20 = solamit
    v_21 = minus
    v_22 = minus
    v_23 = solamit
    v_24 = minus
    v_25 = minus
    v_26 = minus
    v_27 = dot
    v_28 = solamit
    v_29 = solamit

    v_210 = solamit
    v_30 = solamit
    v_31 = minus
    v_32 = dollar
    v_33 = minus
    v_34 = minus
    v_35 = minus
    v_36 = minus
    v_37 = minus
    v_38 = solamit
    v_39 = solamit
    v_310 = solamit
    v_40 = solamit
    v_41 = minus
    v_42 = minus
    v_43 = shtrudel
    v_44 = solamit
    v_45 = minus
    v_46 = minus
    v_47 = minus
    v_48 = solamit
    v_49 = solamit
    v_410 = solamit
    v_50 = solamit
    v_51 = solamit
    v_52 = solamit
    v_53 = minus
    v_54 = minus
    v_55 = minus
    v_56 = minus
    v_57 = star
    v_58 = solamit
    v_59 = solamit
    v_510 = solamit
    v_60 = solamit
    v_61 = solamit
    v_62 = solamit
    v_63 = solamit
    v_64 = solamit
    v_65 = minus
    v_66 = minus
    v_67 = minus
    v_68 = minus
    v_69 = minus
    v_610 = solamit
    v_70 = solamit
    v_71 = solamit
    v_72 = solamit
    v_73 = solamit
    v_74 = solamit
    v_75 = solamit
    v_76 = solamit
    v_77 = minus
    v_78 = minus
    v_79 = minus
    v_710 = solamit

    v_80 = solamit
    v_81 = solamit
    v_82 = solamit
    v_83 = solamit
    v_84 = solamit
    v_85 = solamit
    v_86 = solamit
    v_87 = minus
    v_88 = minus
    v_89 = minus
    v_810 = solamit
    v_90 = solamit
    v_91 = solamit
    v_92 = solamit
    v_93 = solamit
    v_94 = solamit
    v_95 = solamit
    v_96 = solamit
    v_97 = solamit
    v_98 = solamit
    v_99 = solamit
    v_910 = solamit
    done = FALSE
    m = 11
    n = 10
  -> State: 1.2 <-
    turn = l
    x = 2
    v_42 = shtrudel
    v_43 = minus
  -> State: 1.3 <-
    possible_down = FALSE
    possible_right = TRUE
    x = 1
    v_41 = shtrudel
    v_42 = minus
  -> State: 1.4 <-
    turn = u
    possible_left = FALSE
    y = 3
    v_31 = shtrudel
    v_41 = minus
  -> State: 1.5 <-
    turn = r
    possible_down = TRUE
    x = 2
    v_31 = minus
    v_32 = shtrudel
    v_33 = dollar
  -> State: 1.6 <-
    possible_left = TRUE
    x = 3
    v_32 = minus
    v_33 = shtrudel
    v_34 = dollar

    -> State: 1.7 <-
      possible_up = FALSE
      x = 4
      v_33 = minus
      v_34 = shtrudel
      v_35 = dollar
    -> State: 1.8 <-
      possible_up = TRUE
      possible_down = FALSE
      x = 5
      v_34 = minus
      v_35 = shtrudel
      v_36 = dollar
    -> State: 1.9 <-
      possible_down = TRUE
      x = 6
      v_35 = minus
      v_36 = shtrudel
      v_37 = dollar
    -> State: 1.10 <-
      turn = d
      possible_right = FALSE
      y = 4
      v_36 = minus
      v_46 = shtrudel
    -> State: 1.11 <-
      turn = r
      possible_right = TRUE
      x = 7
      v_46 = minus
      v_47 = shtrudel
    -> State: 1.12 <-
      turn = u
      possible_right = FALSE
      y = 3
      v_27 = star
      v_37 = shtrudel
      v_47 = minus
      done = TRUE
nuXmv > |
```
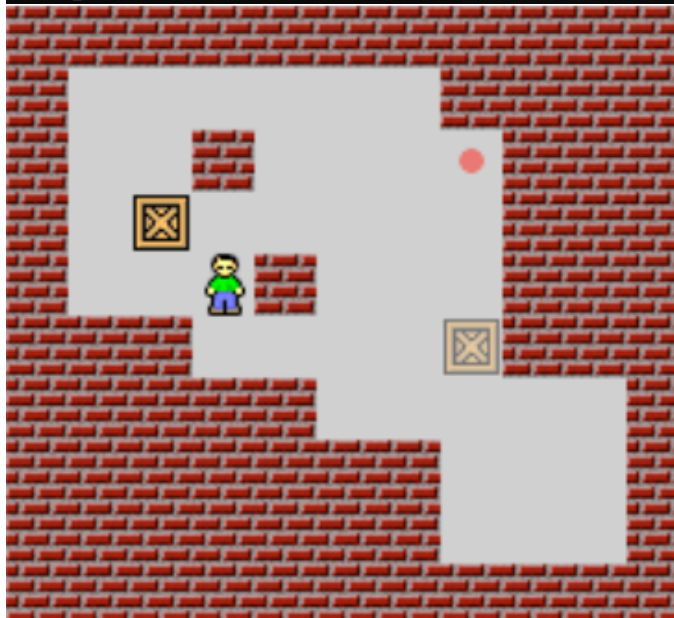
Part 2:

   3. For each board we found was solvable we'll define the winning moves:

      Board1: r
      Board4: d, u, u, d, r, l, l
      Board7: l, l, u, r, r, r, r, r, d, r, u

Part 3:

Board	SAT[sec]	BDD[sec]
board1	0.07	0.07
board2	0.58	0.07
board3	8.54	3.92
board4	0.66	3.51
board5	5.83	1.02
board6	3.66	0.9
board7	12.94	1023

When running the SVM's we used interactive mode and different commands for BDD and SAT:
BDD: commands = go -> check_ltlspec -> quit
SAT: commands = go_bmc -> check_ltlspec_bmc -k 15 -> quit

From those tests, we can understand that BDD solved faster for boards which isn't solvable probably because we forced the SAT to continue and stop after 15 moves. But we noticed that for solvable boards (1, 4, 7) the SAT solved them faster, and for board7 that is bigger than the others the SAT solved it much faster than the BDD.
To conclude, in our opinion the SAT is better than BDD, especially in solvable boards.

Part 4:

For this part, we added some new functionality to the SMVWriter class using the SMVWriterIterative class which inherits from SMVwriter. In this class, we also follow the position of one specific box in each iteration. For that we had modified 'done' to check if the chosen box had reached a target. At the end of each iteration we try to get our new board state. To do that, first of all, we check if our specification is true. If so, we know our board is unsolvable and we return None. otherwise, we inspect all the last changes of each place of the board (aka. the board state) and then return the board after the iteration.

For each iteration, we run the nuXmv in SAT mode, using these commands:
SAT: commands = go_bmc -> check_ltlspec_bmc -k 15 -> quit
For each board, we saved the total time that took the nuXmv to run, and these are the results:

Board	ITERATIVE_SAT[sec]	ITERATIONS	TIME PER INTERATION[sec]
board1	0.08	1	[0.08]
board2	0.81	1	[0.81]
board3	1.71	2	[0.63, 1.08]
board4	0.75	4	[0.17, 0.19, 0.19, 0.2]
board5	4.84	2	[1.17, 3.67]
board6	4.37	2	[0.35, 4.01]
board7	11.94	1	[11.94]

As for the times of the iterative algorithm, we got pretty similar results to the regular SAT mode. We can see a dramatic change in runtime in the board3 solution which is caused by the error we will explain in the next paragraph.

In some cases, two boxes can satisfy the specification although the board isn't solved. What's happening is that the new box pushed the first from the target and now the box that was on the target is not solved. This makes the program run another iteration as there is now 'another' box (which we solved in previous iterations). To solve that we found the number of boxes in the beginning and then we checked if our iteration number surpassed the number of boxes in the beginning. This solution doesn't really solve the problem, it just indicates that if our algorithm can get in a loop then it will stop it and define the board as unsolvable, which of course isn't always the true assumption.

Another problem we encountered when we tried to solve bigger, more complex boards, was the fact that when we solved for a specific box we could accidentally push another box to a corner or someplace it couldn't reach a target, to solve this problem we just avoided it:)

We created a new bigger solvable board, especially for this part which in XSB format looks like this:

```
###########
#--$--.####
##-#---.###
#--$----###
#--@#---###
###----$###
#####-----#
#######---#
#######.--#
###########
```

Which is like board7, but with more boxes to make it harder to solve. We checked the times in the SAT model, and our iterative_SAT, and got these results:

Regular SAT: 121.58 sec, and solved in 23 moves (The full output file can be seen in the GitHub repo under outputSAT/board8.out).
Iterative_SAT: 13.3 sec, and 3 iterations (The full output file can be seen in the GitHub repo under outputIterative/board8_time.out,board8_box_iteration1-3.out).
The iteration times are: [2.51, 9.84, 0.95]

As we can see, using our Iterative solution on big and complex boards yields way better results in time, as expected.