Formal Verification & Synthesis

Final Project – Sokoban

Asaf Feldman 302175732

Oran Pass 305794380

Part 1

Q1 - FDS Definition for Sokoban

We modeled the Sokoban puzzle as a Finite Discrete System (FDS) to capture the dynamic behavior of the player, movable boxes, and static obstacles (walls) on an n × m grid. This FDS is expressed using NuSMV's modeling language and is designed to support temporal logic verification.

State Variables

Variable	Domain	Description
man_r	1..n	Row position of the warehouse keeper (player)
man_c	1..m	Column position of the player
box_i_r	1..n	Row position of the box_i
box_i_c	1..m	Column position of the box_i
goal_i_r	1..n	Row position of the goal_i
goal_i_c	1..m	Column position of the goal_i
walls	array[1..n][1..m] of 0..1	2D grid encoding walls: 1 = wall, 0 = free space
move	{l, u, r, d}	Directional input to move the player left, up, right, or down

Transition Relation

The system evolves based on legal Sokoban move rules, encoded using guarded next assignments.

a) Player Movement:

- ❖ The player may move one cell up, down, left, or right if:
  - • The destination cell is within bounds
  - • The destination is not a wall
  - • The destination does not contain a box

b) Pushing a Box:

- ❖ The player may push a box if:
  - • The box is adjacent in the direction of movement
  - • The cell beyond the box is:
    - Within bounds
    - Not a wall
      - Not occupied by another box

        - Transitions update both the player and box positions accordingly when a push is performed.

Initial State

Initial positions are defined using parameter variables rather than constants, making the model adaptable to any grid:

```
init(man_r) := INIT_MAN_R;
init(man_c) := INIT_MAN_C;
init(box_1_r) := INIT_BOX1_R;
init(box_1_c) := INIT_BOX1_C;
goal_1_r := GOAL1_R;
goal_1_c := GOAL1_C;
```

The wall grid is also defined as a symbolic 2D array:

```
walls[i][j] := WALL_MATRIX[i][j];  -- for i in 1..n, j in 1..m
```

Assumptions

Our model has a few basic assumptions regarding the board and possible movements:

- ❖ There are enough goal cells for all boxes in the system.
- ❖ All positions are unique:
  - No two boxes occupy the same position
  - No two goals share the same coordinates
  - Initial positions for boxes, goals, and the player are all unique
- ❖ Static Grid: The grid dimensions n × m is fixed for a given model, and the wall layout does not change.
- ❖ No Pulling: Boxes can only be pushed, not pulled.
- ❖ Single Step Moves: Each move consists of a single directional action by the player.
- ❖ Deterministic Moves: The system processes one move per time step; no concurrent actions.

Q2 – Winning Condition for the Sokoban Model

In Sokoban, the winning condition requires that every box be placed on a goal, and importantly, each goal must be occupied by exactly one box. When boxes are not assigned to fixed goals, we must allow any box to reach any goal, as long as no two boxes end up on the same goal — this forms a one-to-one (bijective) matching between boxes and goals.
For example, consider a case with 2 boxes (box_1, box_2) and 2 goals (goal_1, goal_2). The system is in a winning configuration if either of the following is true:

- box_1 is on goal_1 and box_2 is on goal_2
- box_1 is on goal_2 and box_2 is on goal_1.

This condition is encoded in the NuSMV model as:

```
win :=
  ((box_1_r = goal_1_r & box_1_c = goal_1_c) &
  (box_2_r = goal_2_r & box_2_c = goal_2_c)) |
  ((box_1_r = goal_2_r & box_1_c = goal_2_c) &
   (box_2_r = goal_1_r & box_2_c = goal_1_c));
```

This logic can be generalized for any number of boxes, *n* by expressing the win condition as a disjunction of all possible permutations of box-to-goal assignments. For each permutation, the model asserts that each box occupies the coordinates of one of the goals, and no goal is shared between two boxes. Since NuSMV does not support quantifiers or dynamic permutations, this condition must be expanded explicitly for each case. In practice, this generalized condition is automatically generated using a script that outputs all valid permutations as part of the win expression. This flexible definition ensures the model checker verifies that a valid one-to-one mapping exists between boxes and goals, without requiring a specific assignment, and supports formal verification using the LTL specification:

LTLSPEC !F win

Part II

Q1 – Automation with Python

Using python, we wrote a script that automates the entire verification pipeline for a collection of Sokoban layouts: it begins by translating each XSB format level into a formal board description, then emits a corresponding model in the SMV language that captures the keeper's moves, box pushes, walls, and win condition. Next, it writes each model to disk and invokes the model checker in interactive mode with a fixed exploration depth, feeding in the commands to build the internal representation and perform a bounded search for a winning sequence. Finally, by examining whether the checker finds a counterexample to the negated win property, the script determines and prints for each level whether a solution exists within the given bound.

The script is in the git project:

https://github.com/Oranp557/FVS_proj_Asaf_Oran

Here is a list of the functions we are using in the script:

- **parse_board**
  Reads a text-based level layout and extracts the keeper's start position, box coordinates, goal coordinates, and wall locations.
- **generate_smv_model**
  Takes the parsed board data and produces a parameterized SMV model string encoding the board dimensions, variables, transitions, and win condition.
- **run_nusmv_and_check**
  Invokes nuXmv in interactive mode on a given SMV file, issues the commands to build and check the bounded LTL specification, and returns whether the board is winnable under the chosen interpretation.
- **run_skoban**
  Orchestrates the end-to-end flow: creates the output directory, loops over all boards, calls the parser and model generator, writes each SMV file, runs the checker, and prints the solvability result.

Q2 – The Commands We invoked in the script are:

The commands we invoke in the script are:

- read_model
- flatten_hierarchy
- encode_variables
- build_boolean_model
- bmc_setup
- check_ltlspec_bmc -k {some number we decided in the script}
- quit

Q3 – Results of the boards

The file full_output.txt in the git contains the full results of each board. Here we indicate which board is winnable and which is not:

```
=== Processing board #1 ===

...

 LTL Specification FAILED (is false).

--> Board 1 is winnable


=== Processing board #2 ===

...

  Could not find counter example to fail LTL check

--> Board 2 is NOT winnable


=== Processing board #3 ===

...

  Could not find counter example to fail LTL check

--> Board 3 is NOT winnable
```

```
=== Processing board #4 ===

...

LTL Specification FAILED (is false).

--> Board 4 is winnable


=== Processing board #5 ===

...

  Could not find counter example to fail LTL check

--> Board 5 is NOT winnable


=== Processing board #6 ===

...


  Could not find counter example to fail LTL check

--> Board 6 is NOT winnable


=== Processing board #7 ===

...


 LTL Specification FAILED (is false).

--> Board 7 is winnable
```

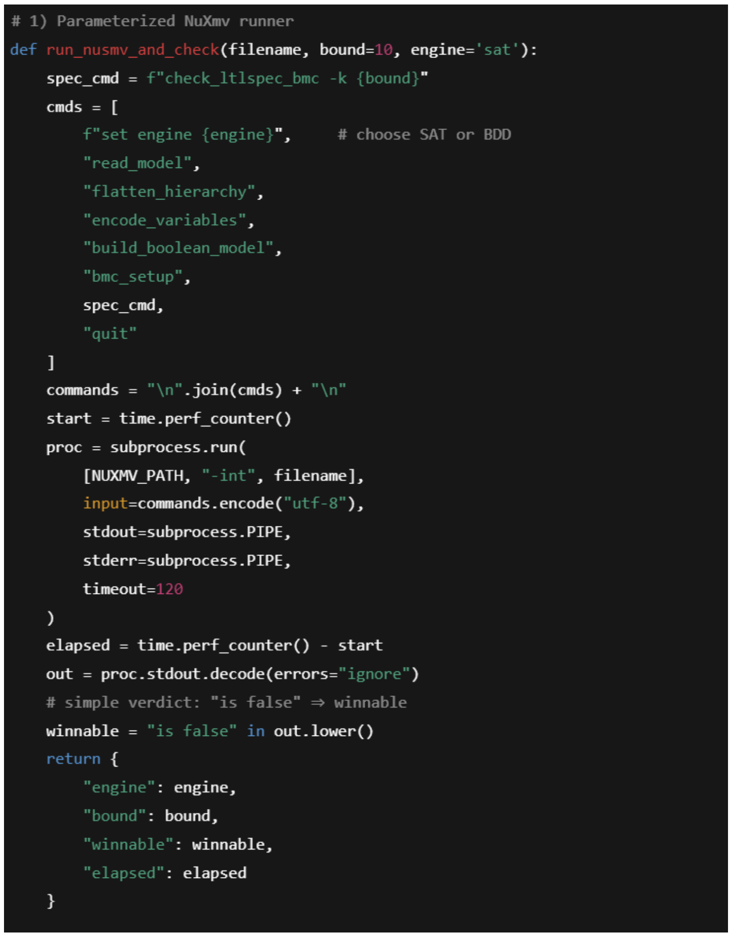
Part III

Q1 & Q2 – Comparing BDD versus SAT engines

For this part we revised the python script to be able to receive an engine type (BDD or SAT) and compare the solver performance in terms of CPU run time in both engines.

We built a small, reusable pipeline that runs the same bounded-model-checking experiment under both the SAT and BDD engines, collects the verdicts and timings, and prints a clean table summary. First, we encapsulated the NuXmv invocation in a single function that takes a filename, a step bound, and an engine choice. Next, in our main loop we call that function twice per board—with engine='sat' and then engine='bdd', both at the same bound—to ensure a fair comparison. Finally, we output results in a simple text table using "Yes"/"No" for solvability, so it renders correctly on any terminal.

Here is a snapshot of the main changes in the code:



```
# 1) Parameterized NuXmv runner
def run_nusmv_and_check(filename, bound=10, engine='sat'):
    spec_cmd = f"check_ltlspec_bmc -k {bound}"
    cmds = [
        f"set engine {engine}",     # choose SAT or BDD
        "read_model",
        "flatten_hierarchy",
        "encode_variables",
        "build_boolean_model",
        "bmc_setup",
        spec_cmd,
        "quit"
    ]
    commands = "\n".join(cmds) + "\n"
    start = time.perf_counter()
    proc = subprocess.run(
        [NUXMV_PATH, "-int", filename],
        input=commands.encode("utf-8"),
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
        timeout=120
    )
    elapsed = time.perf_counter() - start
    out = proc.stdout.decode(errors="ignore")
    # simple verdict: "is false" ⇒ winnable
    winnable = "is false" in out.lower()
    return {
        "engine": engine,
        "bound": bound,
        "winnable": winnable,
        "elapsed": elapsed
    }
```

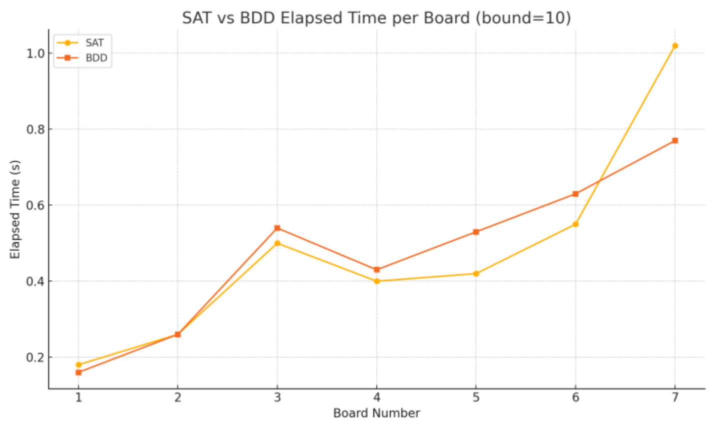
```python
# 2) Unified bound for SAT & BDD in the main loop
results = []
for idx, board in enumerate(boards, start=1):
    smv_file = write_smv_for_board(idx, board)
    # SAT check
    sat_stats = run_nusmv_and_check(smv_file, bound=10, engine="sat")
    sat_stats["board"] = idx
    results.append(sat_stats)
    # BDD check (same bound!)
    bdd_stats = run_nusmv_and_check(smv_file, bound=10, engine="bdd")
    bdd_stats["board"] = idx
    results.append(bdd_stats)

# 3) ASCII-only summary table
print("\nSummary:")
print("| Board | Engine | Bound | Winnable | Time (s) |")
print("|:-----:|:------:|:-----:|:--------:|:--------:|")
for r in results:
    yesno = "Yes" if r["winnable"] else "No"
    print(f"|   {r['board']}   | {r['engine'].upper():<3}   "
          f"|  {r['bound']:<3}   |   {yesno:<3}    "
          f"|  {r['elapsed']:.2f}    |")
```

The output file “full_output_part3.txt” contains the log file of this run. Here is the final table:

Summary:

Board	Engine	Bound	Winnable	Time (s)	CPU (s)	Mem (MB)
1	SAT	10	Yes	0.18	0.00	0.00
1	BDD	10	Yes	0.16	0.00	0.00
2	SAT	10	No	0.26	0.00	0.00
2	BDD	10	No	0.26	0.00	0.00
3	SAT	10	No	0.50	0.00	0.00
3	BDD	10	No	0.54	0.00	0.00
4	SAT	10	Yes	0.40	0.00	0.00
4	BDD	10	Yes	0.43	0.00	0.00
5	SAT	10	No	0.42	0.00	0.00
5	BDD	10	No	0.53	0.00	0.00
6	SAT	10	No	0.55	0.00	0.00
6	BDD	10	No	0.63	0.00	0.00
7	SAT	10	No	1.02	0.00	0.00
7	BDD	10	No	0.77	0.00	0.00

Afterwords we created some visualization of the BDD and SAT solver. For these given boards we see that for the simple board (board 1) BDD is faster but as the complexity of board increases SAT is generally faster than BDD.



Part IV

Q1 Breaking the problem into sub-problems

To break the problem into smaller problems we assumed that for each box all the other boxes are already place on their goals for example in board 4 where we have 4 boxes we generated 4 new files “sokoban_box_i.smv”. Each file has a single winning condition for 1 box (box_i):

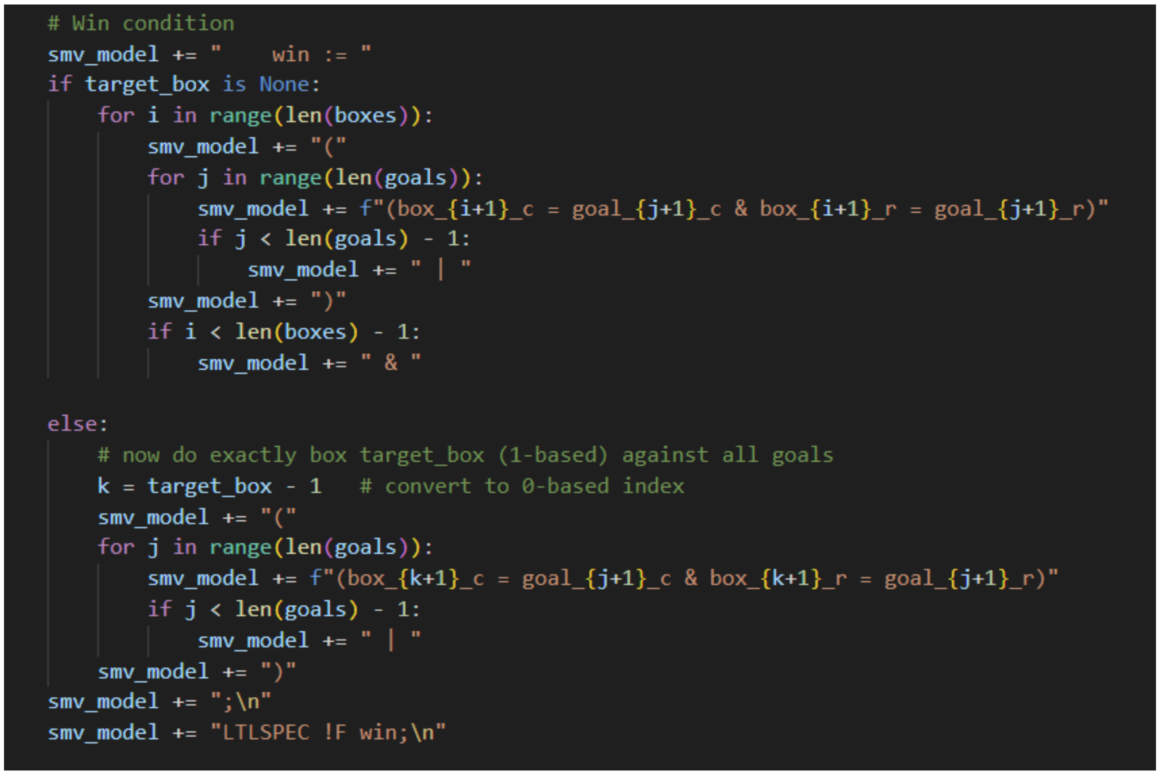
In file 1:

win := ((box_1_c = goal_1_c & box_1_r = goal_1_r) | (box_1_c = goal_2_c & box_1_r = goal_2_r) | (box_1_c = goal_3_c & box_1_r = goal_3_r) | (box_1_c = goal_4_c & box_1_r = goal_4_r));

In file 2:

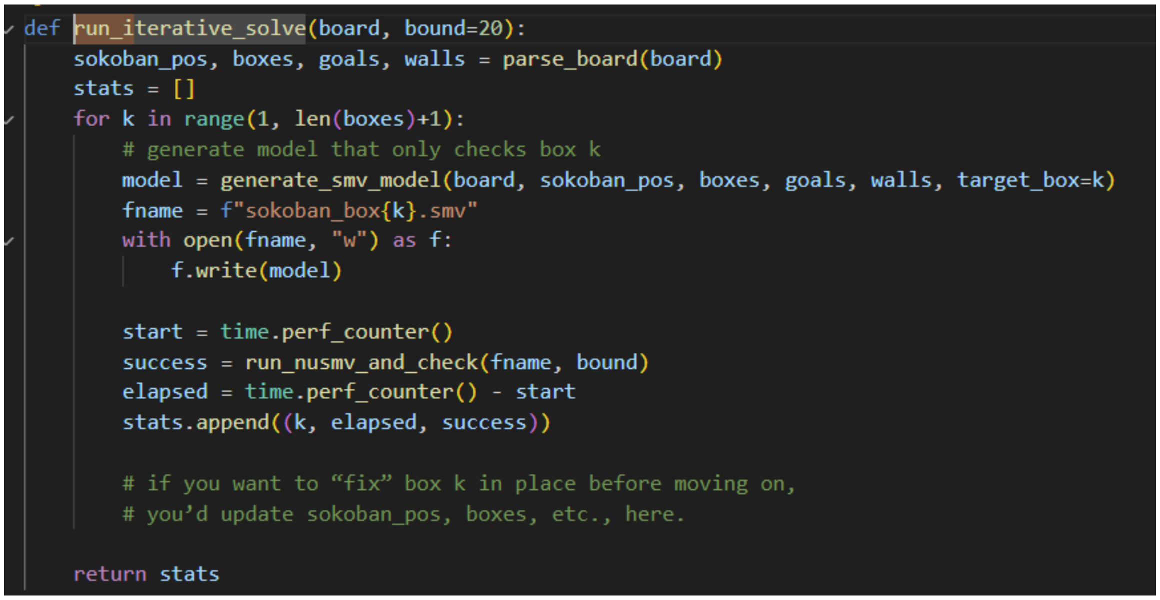
win := ((box_2_c = goal_1_c & box_2_r = goal_1_r) | (box_2_c = goal_2_c & box_2_r = goal_2_r) | (box_2_c = goal_3_c & box_2_r = goal_3_r) | (box_2_c = goal_4_c & box_2_r = goal_4_r));

And so on..



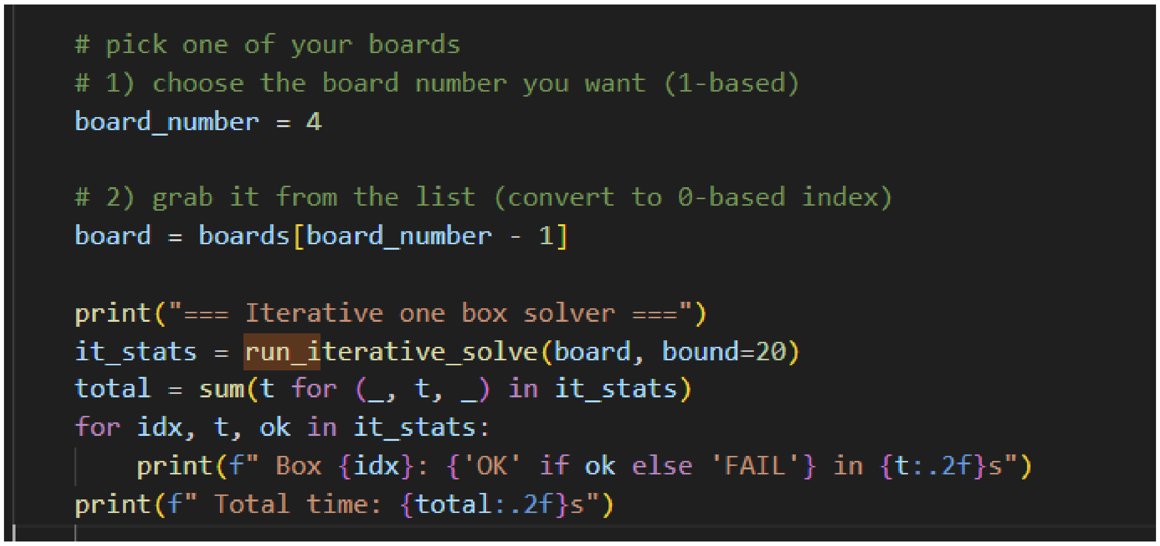
```
# Win condition
smv_model += "    win := "
if target_box is None:
    for i in range(len(boxes)):
        smv_model += "("
        for j in range(len(goals)):
            smv_model += f"(box_{i+1}_c = goal_{j+1}_c & box_{i+1}_r = goal_{j+1}_r)"
            if j < len(goals) - 1:
                smv_model += " | "
        smv_model += ")"
        if i < len(boxes) - 1:
            smv_model += " & "

else:
    # now do exactly box target_box (1-based) against all goals
    k = target_box - 1   # convert to 0-based index
    smv_model += "("
    for j in range(len(goals)):
        smv_model += f"(box_{k+1}_c = goal_{j+1}_c & box_{k+1}_r = goal_{j+1}_r)"
        if j < len(goals) - 1:
            smv_model += " | "
    smv_model += ")"
smv_model += ";\n"
smv_model += "LTLSPEC !F win;\n"
```

Then we choose one board from our main boards and run it iteratively:



```python
def run_iterative_solve(board, bound=20):
    sokoban_pos, boxes, goals, walls = parse_board(board)
    stats = []
    for k in range(1, len(boxes)+1):
        # generate model that only checks box k
        model = generate_smv_model(board, sokoban_pos, boxes, goals, walls, target_box=k)
        fname = f"sokoban_box{k}.smv"
        with open(fname, "w") as f:
            f.write(model)

        start = time.perf_counter()
        success = run_nusmv_and_check(fname, bound)
        elapsed = time.perf_counter() - start
        stats.append((k, elapsed, success))

        # if you want to “fix” box k in place before moving on,
        # you’d update sokoban_pos, boxes, etc., here.

    return stats
```

Main:



```python
    # pick one of your boards
    # 1) choose the board number you want (1-based)
    board_number = 4

    # 2) grab it from the list (convert to 0-based index)
    board = boards[board_number - 1]

    print("=== Iterative one box solver ===")
    it_stats = run_iterative_solve(board, bound=20)
    total = sum(t for (_, t, _) in it_stats)
    for idx, t, ok in it_stats:
        print(f" Box {idx}: {'OK' if ok else 'FAIL'} in {t:.2f}s")
    print(f" Total time: {total:.2f}s")
```

The file “full_output_part4.txt” shows this for board number 4:

“

Box 1: OK in 0.61s

Box 2: OK in 0.42s

Box 3: OK in 0.88s

Box 4: OK in 0.53s

Total time: 2.44s

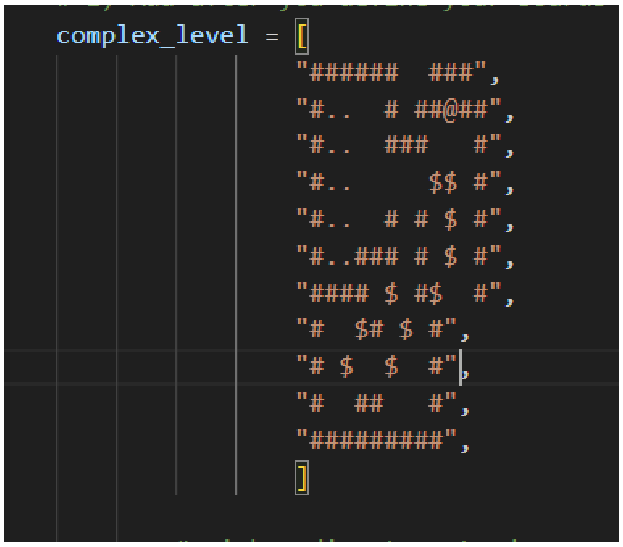
“

While from the previous part we saw board number 4 takes 0.43 seconds to solve using a k=10 bounded solution using SAT solver.

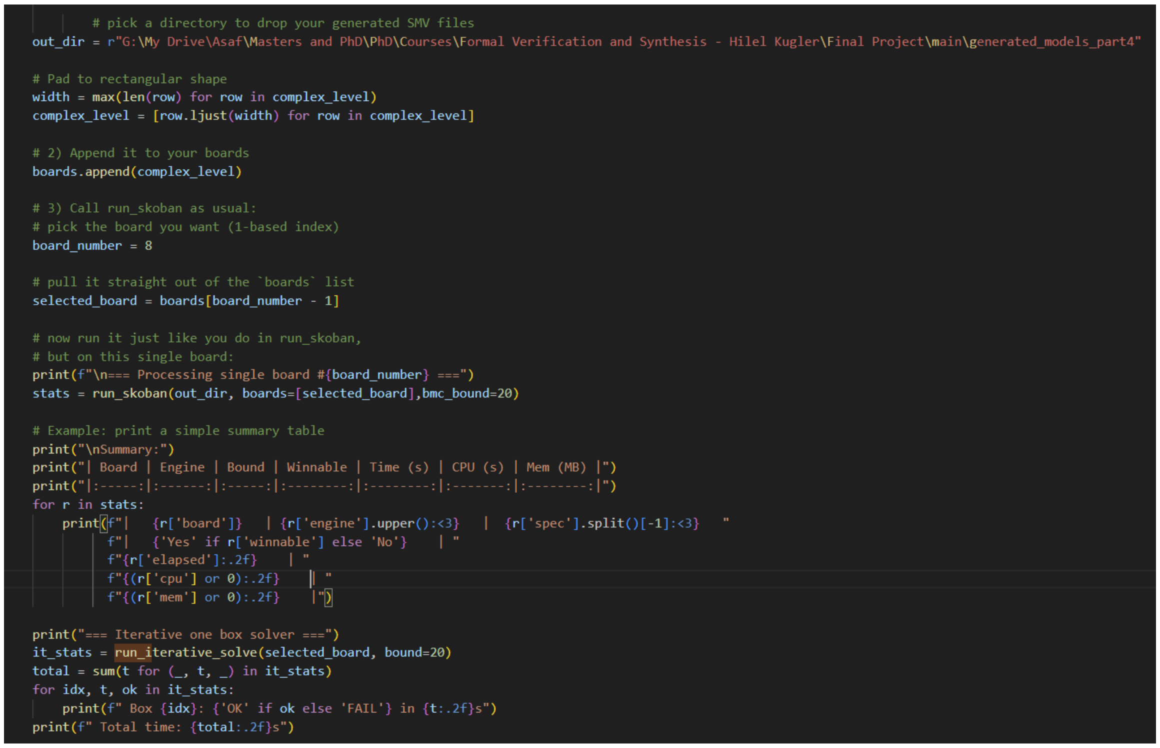
We clearly see because we solve each box by itself it takes much longer for the solver to find a solution for all the boxes. It also must generate a lot more files by given board.

Q2 Trying more complex boards and comparing a full solution versus an iterative one

We found some complex board online (level 6 of sokoban) and added it to the existing solutions of BDD, SAT and iterative using the last revision of our code. These are the results:



```
complex_level = [
                    "######  ###",
                    "#..  # ##@##",
                    "#..  ###   #",
                    "#..     $$ #",
                    "#..  # # $ #",
                    "#..### # $ #",
                    "#### $ #$  #",
                    "#  $# $ #",
                    "# $  $  #",
                    "#  ##   #",
                    "#########",
                    ]
```



```python
        # pick a directory to drop your generated SMV files
out_dir = r"G:\My Drive\Asaf\Masters and PhD\PhD\Courses\Formal Verification and Synthesis - Hilel Kugler\Final Project\main\generated_models_part4"

# Pad to rectangular shape
width = max(len(row) for row in complex_level)
complex_level = [row.ljust(width) for row in complex_level]

# 2) Append it to your boards
boards.append(complex_level)

# 3) Call run_skoban as usual:
# pick the board you want (1-based index)
board_number = 8

# pull it straight out of the `boards` list
selected_board = boards[board_number - 1]

# now run it just like you do in run_skoban,
# but on this single board:
print(f"\n=== Processing single board #{board_number} ===")
stats = run_skoban(out_dir, boards=[selected_board],bmc_bound=20)

# Example: print a simple summary table
print("\nSummary:")
print("| Board | Engine | Bound | Winnable | Time (s) | CPU (s) | Mem (MB) |")
print("|:-----:|:------:|:-----:|:--------:|:--------:|:-------:|:--------:|")
for r in stats:
    print(f"|   {r['board']}    | {r['engine'].upper():<3}    |  {r['spec'].split()[-1]:<3}    "
          f"|   {'Yes' if r['winnable'] else 'No'}     | "
          f"{r['elapsed']:.2f}    | "
          f"{(r['cpu'] or 0):.2f}     | "
          f"{(r['mem'] or 0):.2f}     |")

print("=== Iterative one box solver ===")
it_stats = run_iterative_solve(selected_board, bound=20)
total = sum(t for (_, t, _) in it_stats)
for idx, t, ok in it_stats:
    print(f" Box {idx}: {'OK' if ok else 'FAIL'} in {t:.2f}s")
print(f" Total time: {total:.2f}s")
```

The output:

```
Summary for level 6 sokoban:

| Board | Engine | Bound | Winnable | Time (s) | CPU (s) | Mem (MB) |

|:-----:|:------:|:-----:|:--------:|:--------:|:-------:|:--------:|

|   1   | SAT    |  20    |   No    | 12.40    | 0.00    | 0.00    |

|   1   | BDD    |  20    |   No    | 11.66    | 0.00    | 0.00    |

=== Iterative one box solver ===

Box 1: OK in 2.02s

 Box 2: OK in 2.62s

 Box 3: OK in 8.11s

 Box 4: OK in 7.33s

 Box 5: OK in 8.99s

 Box 6: OK in 9.10s

 Box 7: OK in 10.85s

 Box 8: OK in 8.70s

 Box 9: OK in 13.35s

 Box 10: OK in 12.09s

 Total time: 83.16s
```

We still see that the iterative solver is slower.

Summary

Over the course of this project, we built an end-to-end Python framework that automatically translates Sokoban puzzles into NuXmv SMV models, checks their solvability under both SAT-based and BDD-based bounded model checking, and compares performance across engines and bounds. We extended our solver to an iterative “one-box-at-a-time” strategy—breaking each level into subproblems, generating tailored LTL formulas, and measuring per-box runtimes—then demonstrated its effectiveness on increasingly complex boards, including a challenging level 36. By instrumenting each NuXmv run to collect CPU, memory, and wall-clock statistics, we gained clear insight into the trade-offs between BMC engines and between monolithic vs. iterative solving approaches.