2D-VEV: 2D Virtual Environment (Formal) Validation for Reinforcement Learning via PROMELA/SPIN

Francesco Brigante, 1987197

September 14, 2025

Abstract

The quality of the results after training autonomous agents with Reinforcement Learning highly depends on the virtual environments used during training, ensuring they are logically coherent, robust, well-designed and goal-reachable.
While generating such grid-based environments is computationally cheap, validating their correctness upfront avoids wasted training time and unwanted behaviors. This report presents a PROMELA/SPIN model that simulates an agent navigating 2D labyrinths with walls, keys, and doors, and formally verifies environment properties that ensure its correctness. We instantiate environments of increasing size ($W \times H$) and measure the state-space growth. Experiments reveal the expected state explosion for progressively larger mazes, with an interesting exception: a 35×35 case exhibits fewer states than a 25×25 case due to a simpler layout. This highlights that environment design—not only size—strongly influences verification complexity and, by extension, RL training difficulty.

1 Introduction

Reinforcement Learning relies on repeated interaction with an environment to learn goal-oriented policies. For practical training, environments must be:

(i) Structurally consistent (proper counts of start/goal/keys/doors),

(ii) Logically coherent (doors are only passable when the associated key has been collected and can't pass through walls or boundaries),

(iii) Actually solvable (a path exists from the start to the goal respecting constraints).

Failing these conditions can result in an incorrect environment design and a waste of significant training resources.

I propose a PROMELA model of 2D grid environments that encodes the agent dynamics and checks key Linear Temporal Logic properties with SPIN. By validating environments before RL training, we can filter out inconsistent or unsolvable instances and quantify their inherent complexity through state-space metrics.

Below an example of an environment implemented using ASCII characters, where S=Start, G=Goal, # = Wall, @ = Free, a,b,c keys to doors A,B,C:

ASCII

```
# # # # # # #
# S @ @ a @ #
# @ # # @ @ #
# @ @ A @ # #
# # @ # @ @ #
# @ @ @ @ G #
# # # # # # #
```

Render

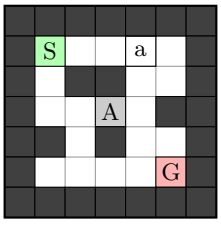


Figure 1: To the left ASCII implementation (S, G, a, A, #, @); to the right its rendering (`small.pml`, env1).

2 Model

Grid. An environment is a W × H grid with cell labels in {FREE (@), WALL (#), START (S), GOAL (G), KEY[1] (a), DOOR[1] (A), KEY[2] (b), DOOR[2] (B), KEY[3] (c), DOOR[3] (C)}.

Agent Actions. The agent moves in the cardinal directions: {N,S,E,W}.

State Space (LTS). States are tuples $(x, y, \text{key}[1..3])$ where key$[i] \in \{0, 1\}$ boolean vector indicating which keys we are carrying.
The initial state is $(x_0, y_0, [0, 0, 0])$ at the unique START.
We also have an auxiliary function to map (x, y) coordinates to state labels {FREE, WALL, ...}

Transition Relation. A move to (x', y') is allowed when:

(i) Bounds hold $0 \leq x' < W,\ 0 \leq y' < H$;

(ii) Target cell is not a wall;

(iii) If target is DOOR[i], then key$[i]$ must be 1 to pass;

(iv) If target is KEY[i], then key$[i]$ is set to 1 after the move.

Atomic Propositions. Atomic propositions include `at_goal`, `at_door_i`, `at_key_i`, and `has_key_i`.

Environment Assumptions.

- Exactly one START and one GOAL
- Correct KEY[i]-DOOR[i] matching

Environment Constraints.

1. Never walk through a wall
2. If a door is closed, you need the key to walk through it
3. Key preservation: $G(\texttt{has\_key\_i} \rightarrow X\texttt{has\_key\_i})$
4. Movement only within bounds
5. A path exists from START to GOAL

3 Implementation in PROMELA

We model the grid in a flattened 1D array and provide helper macros for correct indexing.
During a first naive implementation, using for the navigation process only an if block and letting SPIN decide the direction led to unfair choices, resulting in infinite loops. In order to mitigate this issue, I chose to prioritize the least-visited adjacent cell, providing a weak fairness heuristic for the exploration. The following block of code shows the defined macros and a `PASSABLE` function that checks if a cell is walkable or not.

3.1 Grid encoding and passability

```
#define IDX(row,col) ((row)*W + (col))
#define CELL(row,col) env[IDX(row,col)]

#define AT_WALL   (CELL(row,col)=='#')
#define AT_GOAL   (CELL(row,col)=='G')
#define AT_KEY_A  (CELL(row,col)=='a')
#define AT_KEY_B  (CELL(row,col)=='b')
#define AT_KEY_C  (CELL(row,col)=='c')
```

```
#define AT_DOOR_A (CELL(row,col)=='A')
#define AT_DOOR_B (CELL(row,col)=='B')
#define AT_DOOR_C (CELL(row,col)=='C')

#define PASSABLE(r,c) (CELL(r,c)!='#' && \
                       (CELL(r,c)!='A' || has_a) && \
                       (CELL(r,c)!='B' || has_b) && \
                       (CELL(r,c)!='C' || has_c))
```

3.2 Exploration process

Least-visited-neighbor exploration code:

```
proctype Robot() {
    int steps = 0;
    short visited[N_CELLS];
    // initialize visited[][] to 0 ...

    // wait until (row,col) is set to START and assert not wall
    (row != 0 || col != 0 || CELL(row,col) == 'S');
    assert(!AT_WALL);

    do
    :: (steps >= 70000) -> break
    :: AT_GOAL -> goal_reached = 1 /* reached */
    :: else ->
        visited[IDX(row,col)] = visited[IDX(row,col)] + 1;
        byte min_visit = 255; byte move_choice = 0; // 1:N 2:S 3:E 4:W

        // North
        if
        :: (row>0   && PASSABLE(row-1,col)) ->
             if :: visited[IDX(row-1,col)] < min_visit ->
                    min_visit = visited[IDX(row-1,col)]; move_choice = 1
                :: else -> skip fi
        :: else -> skip fi;
        // similarly for S, E, W ...

        if
        :: move_choice==1 -> row = row-1
        :: move_choice==2 -> row = row+1
        :: move_choice==3 -> col = col+1
        :: move_choice==4 -> col = col-1
        :: else -> break fi;

        if :: (CELL(row,col)=='a') -> has_a=1 :: else -> skip fi;
        if :: (CELL(row,col)=='b') -> has_b=1 :: else -> skip fi;
        if :: (CELL(row,col)=='c') -> has_c=1 :: else -> skip fi;
        steps++
    od;
}
```

3.3 LTL properties

We check necessary properties to accept an environment instance:

```
ltl no_wall        { [] !AT_WALL }
ltl door_a_closed  { [] ( AT_DOOR_A -> has_a ) }
ltl door_b_closed  { [] ( AT_DOOR_B -> has_b ) }
ltl door_c_closed  { [] ( AT_DOOR_C -> has_c ) }
ltl keep_a         { [] ( has_a -> X has_a ) }
ltl keep_b         { [] ( has_b -> X has_b ) }
```

```
ltl keep_c          { [] ( has_c -> X has_c ) }
ltl bounds_ok       { [] ( row < H && col < W ) }
ltl reach_goal      { <> goal_reached }
```

These align with the 5 Environment Constraints defined in section 2.

4 Experimental Setup

We evaluate four environments of increasing size: 7×7 (`small.pml`), 15×15 (`medium.pml`), 25×25 (`large.pml`), and 35×35 (`mega.pml`). For each, I compiled and run SPIN, checking all the LTL Properties and printing traces to collect data.

Table 1: State-space metrics collected with SPIN

File	W	H	Area	States	Depth	Memory (MB)	Time (ms)
small.pml	7	7	49	707	1498	0.2	10
medium.pml	15	15	225	3153	6590	3	20
large.pml	25	25	625	31782	64278	78	300
mega.pml	35	35	1225	23775	48894	113	420

5 Results

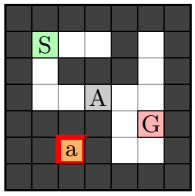
5.1 LTL Property Validation

To validate the correctness of the PROMELA implementation, I tested the LTL properties on both well-formed and malformed environments.
All LTL properties (no_wall, door_a_closed, keep_a, bounds_ok, reach_goal) are satisfied for properly constructed environments like `env1` in our test suite. To further verify SPIN's error detection capabilities, I also defined deliberately malformed mazes: `env2` with an unreachable goal due to wall placement, and `env3` with an unreachable key 'a' that blocks access to door 'A'. For example, in `env3` (shown below), the key 'a' is surrounded by walls, making it impossible to collect and thus preventing passage through door 'A' to reach the goal:

env3 - ASCII

```
# # # # # # #
# S @ @ # @ #
# @ # # # @ #
# @ @ A @ @ #
# # # # @ G #
# # a # @ @ #
# # # # # # #
```

env3 - Render



SPIN correctly identified these structural errors, failing the `reach_goal` property for both malformed cases, thus demonstrating the robustness of the proposed verification approach.

5.2 State-Space Explosion Analysis

Figure 2 shows the number of visited states as a function of the grid width W; a logarithmic y-axis highlights the growth rates. From $7 \to 15 \to 25$, we observe an exponential-like increase in the explored state space. Surprisingly, the 35×35 case has fewer states than the 25×25 case. This is maily because the larger environment is structurally simpler (fewer walls, tight corridors and bottlenecks, mostly open), yielding lower branching; hence, the state space is smaller despite the larger area, indicating that maze complexity is another variable to take in consideration.

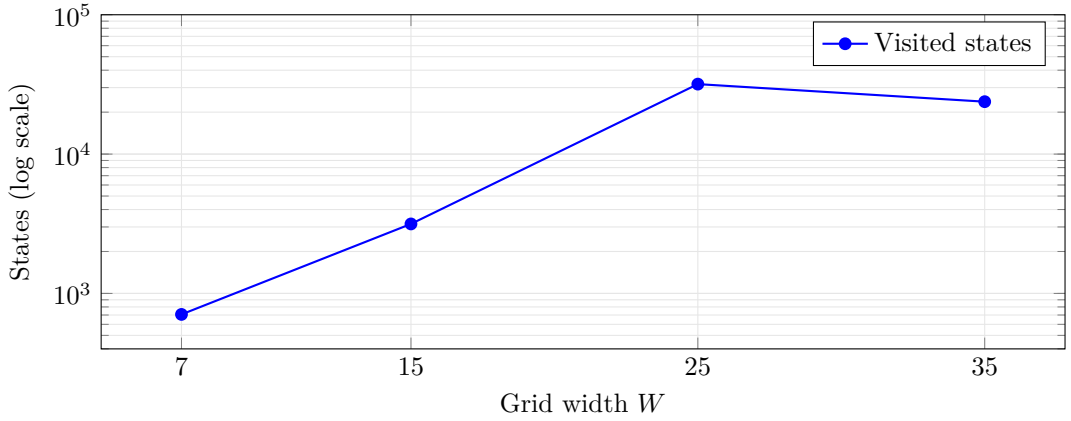
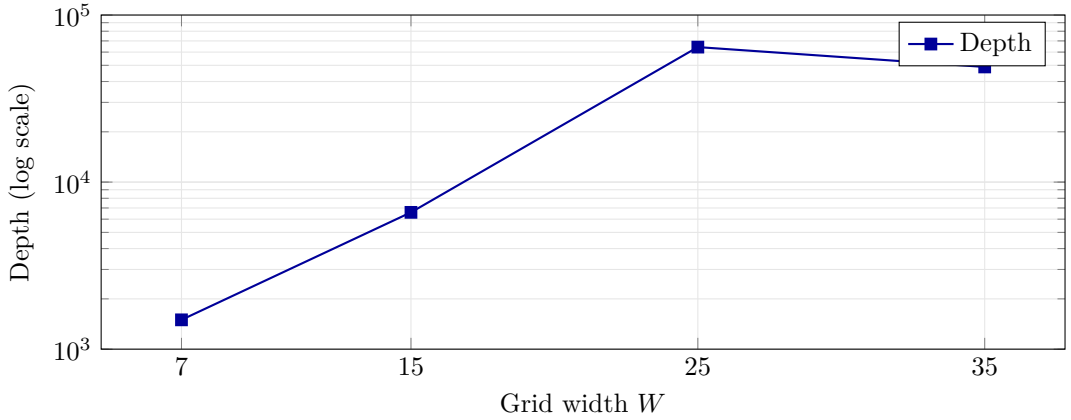


Figure 2: State-space growth vs. grid width. The 35×35 case is lower than 25×25 due to an easier layout.

Depth follows a similar trend, broadly increasing with size but not strictly monotonic, while memory gets an exponential increase in the first 3 files and a much smaller one for the 35×35 environment.



In addition to states and depth, memory usage (Table 1) generally increases exponentially with W (and area), but the `mega.pml` case again shows that structure can modulate cost: despite the largest area, its simpler layout and fewer explored states reduce both depth and effective memory growth compared to the other cases.

Key Takeaways. (i) Grid *size* correlates with verification difficulty, often exponentially; (ii) Grid *structure* (e.g., narrow passages, door placements, key access patterns) can dominate size effects if its design is complex; (iii) Environment design is a crucial factor, both for formal verification and for RL training difficulty.

6 Conclusions

This report introduced a PROMELA/SPIN model to validate 2D environments with walls, keys, and doors before RL training, which can be easily extended and improved. The experiments demonstrate state-space explosion with increasing W and H, alongside a salient counterexample where a larger but simpler environment yields fewer states. This underscores the importance of environment design beyond raw dimensions.