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

Francesco Brigante, 1987197

September 23, 2025

Abstract

The quality of the results after training autonomous agents with Reinforcement Learning highly depends on the digital 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.

To validate the effectiveness of our formal verification approach, we conducted a complete RL training phase using PPO on two distinct datasets: 100 formally verified environments and 100 unverified environments. The trained models were then evaluated on a test set of 20 environments. Results demonstrate that agents trained on verified environments achieve significantly better performance, with an average of 169.7 steps to reach the goal compared to 1581.9 steps for agents trained on unverified environments—a remarkable 9.3x improvement. This substantial performance difference validates the importance of formal environment verification before RL training.

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:

2 Model

Grid. An environment is a $W \times H$ 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}.

ASCII

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

Render

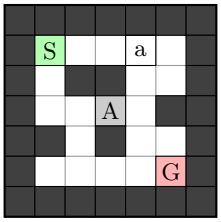


Figure 1: To the left ASCII implementation (S, G, a, A, #, @); to the right its rendering (`small.pml`, env1).

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

In this section, we analyze the correctness of the LTL property implementation and examine state-space characteristics of different environment sizes. Additionally, we validate the practical effectiveness of the approach through RL training experiments.

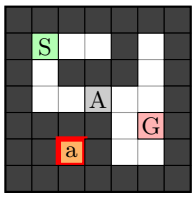
4.1 LTL Property Validation

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.

4.2 State-Space Explosion Analysis

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

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.

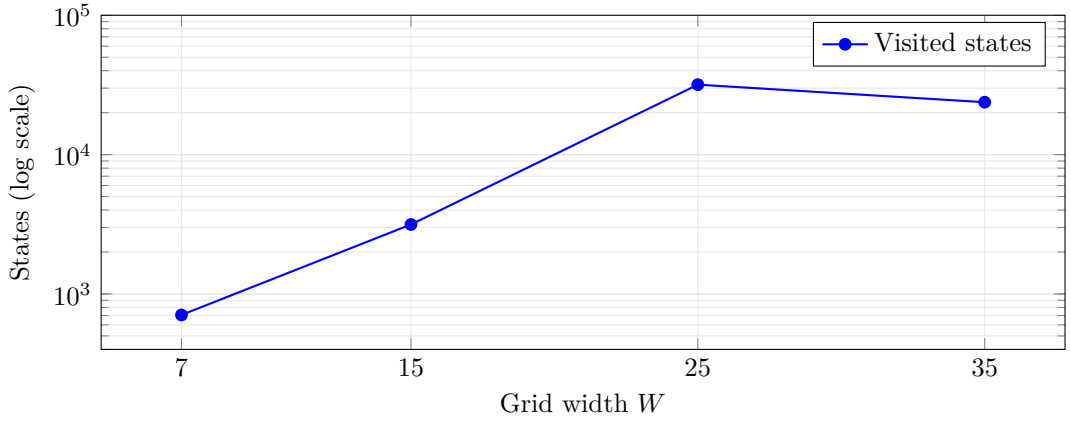
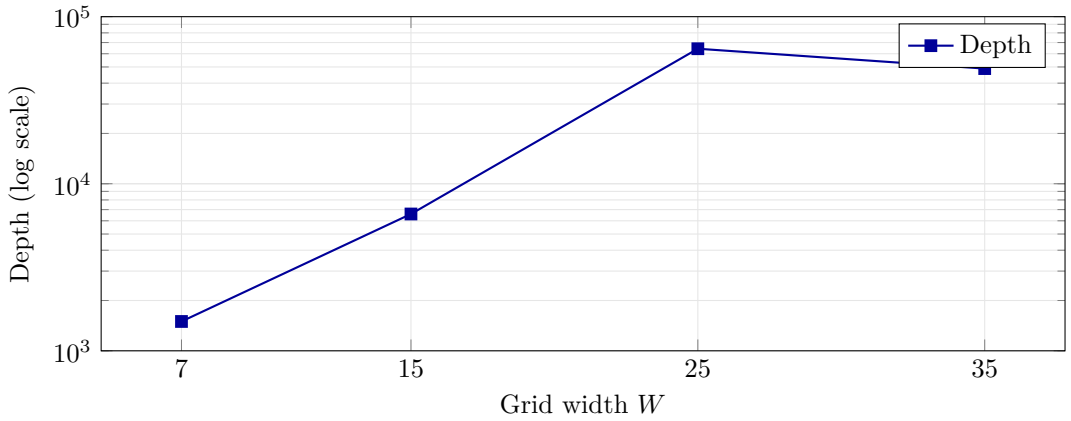


Figure 2: State-space growth vs. grid width. The 35×35 case is lower than 25×25 due to an easier layout.

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.

4.3 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.
These insights validate our approach: formal verification not only ensures environment correctness but also provides complexity metrics that predict training difficulty. The practical effectiveness of this verification approach is further demonstrated in the following sections.

5 RL Training and Evaluation

To validate the practical benefits of the formal verification approach, I conducted a comprehensive RL training pipeline comparing agents trained on verified versus unverified environments.

5.1 Dataset Creation with BMC Verification

File `bmc_verifier.py` represents a Python-based BMC (Bounded Model Checking) verifier that integrates with the PROMELA models to automatically verify environments and compute minimum steps necessary to reach goal. Unlike traditional model checking, BMC searches for counterexamples within a bounded depth, making it particularly suitable for finding optimal solutions.

Negated LTL Property for BMC. To find the minimum number of steps to reach the goal, and thus the counterexample, we negate the reachability property in PROMELA:

```
/* LTL Properties -> negated for BMC */
ltl reach_goal { [] !goal_reached }
```

This property states that the goal is *never* reached. When SPIN finds a counterexample to this negated property, it has actually found a path where the goal *is* reachable. The counterexample trace provides the exact sequence of steps needed to reach the goal.

Incremental Depth Search. The core of our BMC verification is the `find_minimum_steps` function that systematically searches for the minimum solution:

```
def find_minimum_steps(self, maze: List[List[str]], max_depth: int = 50) -> Tuple[bool, str,
    int]:
    try:
        with tempfile.TemporaryDirectory() as temp_dir:
            # try each depth incrementally
            for depth in range(1, max_depth + 1):
                is_reachable, message = self._run_bmc_at_depth(temp_dir, maze, depth)

                if is_reachable:
                    return True, f"Goal reachable in minimum {depth} steps", depth

            return False, f"Goal not reachable within {max_depth} steps", -1
    except Exception as e:
        return False, f"BMC verification error: {str(e)}", -1
```

This incremental approach is crucial for finding the *minimum* number of steps. By starting from depth 1 and incrementally increasing, we ensure that the first successful verification corresponds to the optimal solution.

The bounded search is performed using SPIN's `-u` flag:

```
# run verification with bounded search
result = subprocess.run(
    [f'./pan_{depth}', f'-u{depth}', '-N', 'reach_goal', '-e'],
    cwd=work_dir,
    capture_output=True,
    text=True,
    timeout=60
    )
```

Where `-u{depth}` limits the search depth, `-N reach_goal` specifies which LTL property to check, and `-e` requests counterexample generation.

Counterexample Interpretation. The BMC result interpretation handles the negated property logic:

```
def _extract_bmc_results(self, stdout: str, stderr: str) -> Tuple[bool, str]:
    stdout_lower = stdout.lower()

    # given a negated LTL property, counterexample means goal is reachable
    if "assertion violated" in stdout_lower:
        return True, "Goal reachable (counterexample found)"

    if "acceptance cycle" in stdout_lower:
        return True, "Goal reachable (acceptance cycle found)"
```

```
    if "errors: 0" in stdout_lower:
        return False, "Goal not reachable at this depth"
```

When SPIN reports "assertion violated" or finds an "acceptance cycle", it has discovered a counterexample to our negated property, meaning the goal *is* reachable. This counterintuitive but powerful approach allows us to use BMC for optimal path finding rather than just safety verification.

The dataset creation process incorporates several important features:

- **Skip-trivial environments**: Automatically filters out environments requiring fewer than 4 steps to complete, ensuring meaningful learning challenges.
- **Unsolvable environment handling**: When creating the verified dataset, if BMC verification determines an environment is unsolvable, it is automatically discarded and the generator continues until the required number of verified environments is generated.
- **Verification overhead**: Creating a dataset of verified environments requires significantly more computational effort as the system must generate and test multiple candidates until finding sufficient solvable instances.

Using this infrastructure, we generated two distinct datasets: 100 formally verified environments and 100 unverified environments, both using 7×7 grids to balance computational feasibility with meaningful complexity.

5.2 Environment Implementation with Gymnasium

The maze environments were implemented using the Gymnasium framework, providing a standardized RL interface.
The designed reward function is simple yet effective: it encourages reaching goal while exploring cells.

```
    def _calculate_reward(self) -> float:
        if self.goal_reached:
            return 100.0  # big reward for reaching goal

        reward = -0.1  # small step penalty for each move

        # reward for collecting keys (only when first collected)
        current_cell = self.maze[self.agent_row][self.agent_col]
        if current_cell in ['a', 'b', 'c'] and current_cell not in self.keys_collected:
            reward += 10.0  # good reward for keys

        # small bonus for exploration
        current_pos = (self.agent_row, self.agent_col)
        if current_pos not in self.visited_cells:
            reward += 0.1  # exploration bonus if visiting new cell

        return reward
```

This function structure provides strong positive reward for goal achievement (100.0), moderate rewards for key collection (10.0), exploration bonuses (0.1) when visiting new cells (to avoid re-visiting), and small penalties for each step (-0.1) to encourage efficiency.
It is worth noting that this reward function is quite simple and could be further refined to achieve even better results. More sophisticated reward shaping, longer training periods, or larger datasets could potentially enhance performance further. However, the remarkable 9.3× improvement achieved with this basic reward function demonstrates that the benefits of formal verification are robust and do not depend on complex reward engineering, in fact the quality of the verified environments alone drives substantial performance gains.

5.3 Training Protocol

Both datasets were used to train separate PPO (Proximal Policy Optimization) agents with identical hyperparameters:

- Training steps: 100,000
- Algorithm: PPO
- Environment batches for efficient training across all environments

After training, both models were evaluated on a common test set of 20 solvable environments, with each environment tested over 20 episodes to ensure statistical reliability.

5.4 Evaluation Results

The evaluation results demonstrate the significant advantage of training on verified environments. Table 2 shows the comprehensive comparison:

Table 2: RL Evaluation Results: Verified vs Unverified Training

Env ID	Min Steps	Verified Model		Unverified Model		Improvement
	(Optimal)	Success	Avg Steps	Success	Avg Steps	Factor
env_0001	8	1.0	245.8	1.0	3204.2	13.0×
env_0002	5	1.0	76.1	1.0	323.9	4.3×
env_0003	1	1.0	40.6	1.0	125.5	3.1×
env_0004	7	1.0	124.8	1.0	4306.0	34.5×
env_0005	7	1.0	258.6	1.0	4042.2	15.6×
env_0006	6	1.0	52.1	1.0	84.2	1.6×
env_0007	4	1.0	41.0	1.0	430.8	10.5×
env_0008	6	1.0	156.8	1.0	162.4	1.0×
env_0009	5	1.0	203.3	1.0	65.2	0.3×
env_0010	1	1.0	61.1	1.0	76.0	1.2×
env_0011	5	1.0	442.6	1.0	1805.7	4.1×
env_0012	3	1.0	195.5	1.0	3259.9	16.7×
env_0013	8	1.0	160.3	1.0	38.6	0.2×
env_0014	2	1.0	384.1	1.0	7.6	0.02×
env_0015	5	1.0	58.1	1.0	865.2	14.9×
env_0016	5	1.0	82.0	1.0	336.6	4.1×
env_0017	3	1.0	32.4	1.0	261.5	8.1×
env_0018	9	1.0	234.1	1.0	5644.5	24.1×
env_0019	11	1.0	539.9	1.0	6595.7	12.2×
env_0020	1	1.0	4.1	1.0	2.8	0.7×
AVERAGES	**5.1**	**1.0**	**169.7**	**1.0**	**1581.9**	**9.3×**

Key Performance Metrics:

- **Overall improvement**: Agents trained on verified environments achieve goals in an average of 169.7 steps compared to 1581.9 steps for unverified training—a remarkable **9.3× improvement**.
- **Success rate**: Both models achieved 100% success rate, demonstrating that verification improves efficiency rather than just solvability in this simple scenario.
- **Consistency**: The verified model shows more consistent performance across different environment types.

Notable Cases: Several environments demonstrate particularly dramatic improvements:

- **env_0004**: 34.5× improvement (124.8 vs 4306.0 steps)
- **env_0005**: 15.6× improvement (258.6 vs 4042.2 steps)
- **env_0018**: 24.1× improvement (234.1 vs 5644.5 steps)
- **env_0019**: 12.2× improvement (539.9 vs 6595.7 steps)

Analysis of Outlier Cases: Interestingly, three environments show counter-intuitive results where the unverified model performs better:

- **env_0009**: 0.3× (203.3 vs 65.2 steps) - The unverified model accidentally learned a more direct strategy for this specific simple layout, while the verified model developed a more cautious exploration pattern.
- **env_0013**: 0.2× (160.3 vs 38.6 steps) - Similar pattern here.
- **env_0014**: 0.02× (384.1 vs 7.6 steps) - With only 2 optimal steps required, this is an extremely trivial environment. The unverified model's chaotic policy accidentally achieved near-optimal performance through random luck, while the verified model's learned systematic exploration was unnecessarily complex for such a simple case.

These outliers highlight an important insight: for extremely simple environments, the systematic policies learned from verified training may introduce unnecessary complexity. However, these cases represent only 15% of the test set, and the overall 9.3× improvement demonstrates that verified training produces more robust and generalizable policies for realistic environment complexity.
These results validate our core hypothesis: formal verification not only ensures environment correctness but also produces higher-quality training data that leads to significantly more efficient learned policies, particularly for environments of meaningful complexity.

6 Conclusions

This project presents a framework to perform formal validation of 2D digital environments for Reinforcement Learning, showing the PROMELA implementation and its practical effectiveness. The model successfully validates environment properties through LTL verification, while the state-space analysis reveals important insights about the relationship between environment size, structure, and verification complexity.

The 9.3× performance improvement achieved by the agent trained on verified versus unverified environments provides the evidence for the actual value of formal environment verification in improving the quality of RL training.

The BMC-based verification system, combined with intelligent dataset generation that filters out trivial and unsolvable environments, creates a robust pipeline for producing high-quality training environments. The integration of SPIN verification with RL frameworks through Gymnasium demonstrates the feasibility of incorporating formal methods into practical machine learning workflows and also suggests applicability to other domains where environment quality critically impacts learning performance.