Model Checking Reinforcement Learning Agents with SPIN: Project Report

Matteo De Sanctis
desanctis.1937858@studenti.uniroma1.it

September 23, 2025

Abstract

This project demonstrates how to apply formal model checking (SPIN/Promela) to policies learned by tabular reinforcement learning agents in small deterministic OpenAI Gym / Gymnasium environments. I convert deterministic policies (derived from Q-tables) into Promela models and use SPIN/PAN to search for counterexamples that violate safety properties (e.g., "never fall into cliff", "never bust"). This report describes the method, implementation, key results on several environments (CliffWalking, FrozenLake, Blackjack, Taxi), limitations, and reproducibility instructions.

1 Introduction

Reinforcement Learning (RL) produces policies that perform well empirically but rarely come with formal safety guarantees. This project investigates whether classical model checking tools (SPIN) can verify simple, deterministic policies produced by tabular RL and produce human-interpretable counterexamples when a safety property is violated.

Goals:

- Train tabular Q-learning agents on small deterministic Gym environments.
- Export deterministic policy (argmax) to Promela.
- Express safety properties in Promela (assertions or LTL) and run SPIN/PAN.
- Replay and visualize counterexamples in the original Gym environment for debugging and presentation.

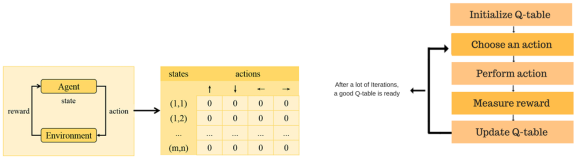
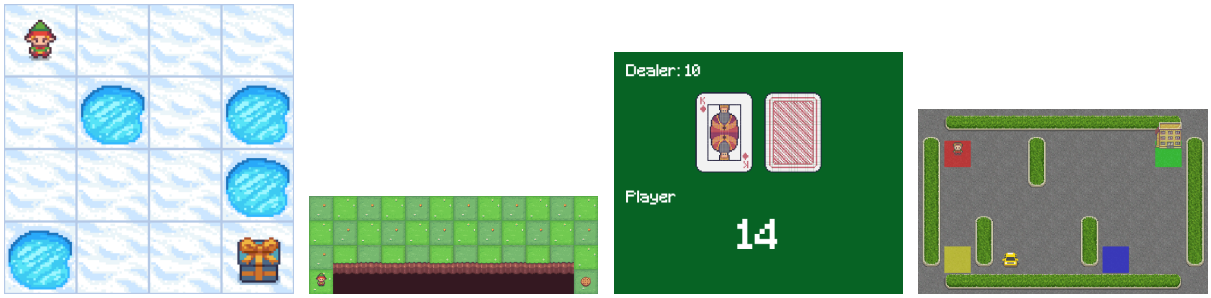


Figure 1: The Q_table and its training.

2 Background

2.1 Reinforcement learning

We use tabular Q-learning (discrete state and action spaces). After training, the Q-table is converted to a deterministic policy by selecting an action with maximal Q-value (ties broken uniformly or by index).



(a) FrozenLake (b) CliffWalking (c) Blackjack (d) Taxi

Figure 2: Overview of environments used in the study.

3 Case Studies

To evaluate the approach we selected four canonical Gymnasium environments that span simple grid-worlds, a small card game, and a planning task. The set was chosen to exercise different state/action structures and to illustrate strengths and limits of model checking RL policies with SPIN.

3.1 FrozenLake (grid navigation)

FrozenLake is a grid-world where the agent must reach a goal cell from a start cell while avoiding holes. States are discrete grid coordinates; actions are the four cardinal moves. We use the deterministic (non-slippery) variant for verification so that transitions map directly to guarded moves in Promela. The primary verification objective is reachability: show that the learned deterministic policy eventually reaches a goal (LTL: $\Diamond$ goal) and avoid unsafe states (assert on falling into a hole). FrozenLake is small (4×4 or 8×8) so it is suitable for end-to-end training and full automatic model checking.

3.2 CliffWalking (risk-aware navigation)

CliffWalking is a 4×12 grid where stepping on the cliff cell produces a large negative reward and resets the agent to the start. The state space is the grid position and actions are the four directions. We treat cliff cells as *unsafe* and encode an assertion violation on any policy step that moves the agent into a cliff. The environment is deterministic, making it straightforward to translate a Q-derived deterministic policy into a Promela proctype whose guarded commands reflect the chosen action from each grid cell. CliffWalking highlights safety violations arising from reward-maximising but unsafe policies.

3.3 Blackjack (simple card game)

Blackjack has a compact, structured state: the player's current sum, the dealer's visible card, and a boolean for a usable ace. Actions are *hit* or *stick*. The game is stochastic because card draws are nondeterministic; in Promela we model draws as nondeterministic branches (card values 1–10). The property checked is a safety-style claim: "the player never busts while following the policy" (LTL: $\Box$ (player sum $\leq$ 21)). This case shows how nondeterministic environment outcomes are represented in Promela and how SPIN can find counterexamples where a deterministic policy will eventually cause a bust for some draw sequence.

3.4 Taxi (planning / large-state MDP)

Taxi is a larger discrete task (taxi position, passenger location/destination) with a moderate but considerably larger state space. Actions include movement and pick-up/drop-off. Because the full Taxi state-space is much larger, generating and model-checking a complete Promela model for an extracted deterministic policy can be expensive and in many cases impractical with naive encodings. We therefore used Taxi primarily as a discussion case: it demonstrates the scalability limits of exhaustive model checking and motivates abstraction or partial verification techniques (state reduction, symmetry, or compositional checks).

3.5 Rationale and modeling choices

The four environments were chosen to exercise: (i) small deterministic grid navigation (FrozenLake, CliffWalking), (ii) nondeterministic outcomes with a compact structured state (Blackjack), and (iii) a larger combinatorial planning problem (Taxi). For FrozenLake and CliffWalking we used deterministic variants or deterministic policies so the Promela model contains straightforward guarded moves. For Blackjack we preserved stochastic draws as nondeterministic branches. For Taxi we limited the scope or deferred full verification due to state-space explosion; this motivates discussion of collapse/abstraction in the results section.

3.6 Model checking with SPIN

SPIN takes a Promela model and checks properties (assertions or LTL). PAN is the explicit-state search engine. When an assertion fails, SPIN/PAN will (if successful) output a trail file or textual counterexample that can be parsed to extract the sequence of states and actions.

Important SPIN options used in experiments:

- ‘spin -a model.pml‘ to generate ‘pan.c‘.
- ‘gcc -o pan pan.c‘ to compile the model checker.
- ‘./pan -m<maxstates> -w<hashbits>‘ to perform the search; increase memory (‘-m‘) and hash table bits (‘-w‘) when needed.
- Optional compile defines: ‘-DNOREDUCE‘, ‘-DCOLLAPSE‘, ‘-DDBITSTATE‘ — these trade memory vs. soundness/completeness (see Limitations).

4 Implementation

4.1 Overview

For each environment:

1. Train a tabular Q-table using Q-learning (hyperparameters below).
2. Convert Q-table to deterministic mapping state $\mapsto$ action.
3. Generate a Promela model that encodes the environment's deterministic dynamics (no stochasticity in the model) and encodes the deterministic policy as guarded branches.
4. Insert assertions for unsafe states (e.g., cliff or bust) or an LTL property (e.g., ‘[] (player <= 21)‘).
5. Run SPIN/PAN and parse results. If PAN reports an assertion violation or writes a ‘.trail‘ file, extract the sequence of (state,action).
6. Replay that sequence in Gym (render to screen and optionally save GIF) to produce a human-understandable demonstration.

4.2 Q-table extraction and Promela modelling

Q-table extraction. We train tabular Q-learning (episodic, on-policy environment episodes) and store the learned Q-table as a NumPy array. Training uses an ε-greedy behaviour policy (initial ε high, exponentially/ multiplicatively decayed to a small value) and standard one-step Q-learning updates

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\,(r + \gamma \max_{a'} Q(s',a')).$$

After training we export the table to disk (e.g. `np.save("qtable.npy",Q)`). For verification we derive a deterministic policy by taking the $\arg\max_a Q(s,a)$ for each state; ties are broken uniformly at random during extraction to produce a single deterministic mapping.

Promela modelling. Each Gym environment is mapped to a compact Promela model that mirrors the environment dynamics relevant for verification:

- The environment topology (grid/map or simplified state variables) is encoded as static arrays (e.g. `byte map[...]`) or explicit variables (player sum, dealer card, usable ace for Blackjack).
- The deterministic policy is exported as guarded Promela branches of a single `proctype Agent(...)`: for each encoded state we emit a guarded branch `:: (state_guard) -> atomic{ ... }` that records the chosen action and performs the corresponding state update.
- Stochastic or uncontrolled aspects of the environment (card draws, slip outcomes) are modelled as nondeterministic choices in Promela (multiple `::` alternatives) so SPIN explores all possible environment outcomes for a given policy.
- Unsafe behaviours are encoded as assertions. When the model reaches an unsafe configuration (e.g. falling into the cliff or player busts), the Promela model executes `printf("OFF:...");` `assert(0);` so SPIN reports a counterexample trace.
- For trace extraction we keep simple history arrays (`act[m], px[m], py[m]`) and increment an index `m` on each step. The PAN output or `spin -v` verbose trace is parsed to reconstruct the counterexample path.
- Optional LTL properties (e.g. `ltl never_bust { [] (player <= 21) }` or `ltl reach_goal { <> goal_reached }`) are added when useful to illustrate safety or reachability properties.

This approach keeps the verification models small and focused on the safety property: the learned policy is fixed (no learning inside Promela), while SPIN explores nondeterministic environment outcomes and returns concrete counterexample traces when assertions fail.

4.3 Key code modules

- `train_q_table(env,...)` — tabular Q-learning trainer (saves to `.npy`).
- `qtable_to_policy(Q, env)` — extract deterministic policy for each discrete state.
- `generate_promela_from_policy(...)` — environment-specific Promela generator (CliffWalking, FrozenLake, Blackjack).
- `run_spin(model.pml, pan_mem)` — wrapper invoking 'spin -a', 'gcc', and './pan' and returning outputs.

- `extract_offender_and_path_from_text(pan_out, spin_verbose)` — parse textual verbose trace or 'OFF:' prints to reconstruct the counterexample path.
- `replay_path_in_env(path, env)` — replay actions in Gym and render them for visualization.

5 Experiments and Results

5.0.1 Training hyperparameters

All experiments used tabular Q-learning with one-step updates. The main hyperparameters were:

- **Learning rate (α)**: 0.5 — moderate step size to balance stability and speed of learning.
- **Discount factor (γ)**: 0.99 — to favour long-term returns where reaching the goal gives delayed reward.
- **Exploration (ε)**: start = 1.0, end = 0.02, multiplicative decay (e.g. 0.99998) — full exploration early, gradual shift to exploitation.
- **Episodes / max steps per episode**: Number of epochs/simulations that make up the training: CliffWalking: 60,000; FrozenLake: 20,000; Blackjack: 100,000; Taxi: (not verified). Max steps per episode = 200 — chosen to give episodes enough length for convergence while bounding runtime.
- **Random seed**: fixed (e.g. 0 or 42) for reproducibility of training runs and exported Q-tables.

Why these values? The episode counts reflect a trade-off between convergence of a tabular method and available computation time; $\alpha = 0.5$ and high γ are standard for episodic tasks where later rewards matter; the ε-decay schedule ensures broad initial exploration so the derived deterministic policy (argmax of the Q-table) is representative of the learned behaviour.

5.1 Properties checked

- **CliffWalking:** never step on cliff cell (assert on stepping into cliff) and eventually reach goal (LTL '<> goal').
- **FrozenLake:** eventually reach goal (LTL '<> goal'), and "never fall into hole".
- **Blackjack:** '[] (player <= 21)' (never bust). This is strict: blackjack cannot guarantee never busting if you keep hitting; we instead expect SPIN to find counterexamples for policies that hit recklessly.
- **Taxi:** omitted from full verification (state explosion) and I suspect incorrect modeling of Promela.

5.2 Representative outcomes

Environment	SPIN found CE?	Notes
CliffWalking	Yes	PAN reported `OFF:` and verbose trace; replay shows agent walking off the cliff.
FrozenLake (deterministic)	Yes	Unsafe moves (holes) detected for poorly trained policies; safe policies pass.
Blackjack	Sometimes	For aggressive "hit" policies, PAN quickly finds assertion violations; for trained policies the search may be partial (SPIN aborted).
Taxi	No	State space too large to complete search; PAN aborted before full exploration.

Example SPIN summary (interpreting output)

- 'State-vector 36 byte, depth reached 70, errors: 1' — PAN used 36 bytes per state vector, found an assertion violation at search depth 70; 'errors: 1' confirms one error found.
- '12 states, stored; 1 states, matched' — number of stored and matched states during exploration.
- 'collapse counts: [0:14 2:11 3:5 4:1]' — statistics about template collapse (technical).

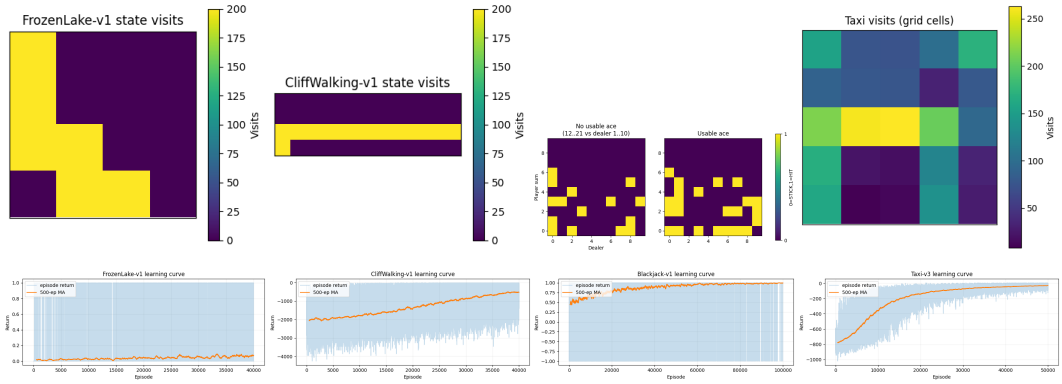


Figure 3: Comparison of training results across environments. Top row: policy state visits heatmaps. Bottom row: training dynamics.

6 Limitations and practical advice

- **State explosion:** PAN is explicit-state; high branching factor or many state variables quickly blow up memory/time. Taxi is an example.
- **Partial search:** Warnings like 'Warning: Search not completed' indicate PAN stopped early (limits). Use larger '-m' and '-w' or state space reduction.
- **Hash compression modes:** 'DBITSTATE' uses compressed bitstate hashing (very memory-efficient, but can miss errors due to collisions). 'DCOLLAPSE'/'DNOREDUCE' influence reduction and completeness; 'DCOLLAPSE' can reduce memory usage by collapsing symmetric templates but changes search behavior. I avoided bitstate hashing, and when necessary used DCOLLAPSE which uses hash buckets.
- **Model fidelity:** We encode deterministic dynamics in Promela (no stochastic transitions) to keep the model small and amenable to exhaustive search. This simplifies the mapping but omits stochastic behaviors.

Figure 4: SPIN verification on models/policies (TOP). SPIN verifcation traces (BOTTOM).

7 Reproducibility (how to run)

7.1 Software requirements

- Python 3.8+ with: `gymnasium` (or `gym`), `numpy`, `matplotlib` (optional for visualization).
- SPIN (tested with Spin 6.5.2), GNU `gcc`.
- For GIF saving, `imageio` or `Pillow`.

7.2 Example commands

```
# generate pan.c from Promela
spin -a model.pml

# compile pan
gcc -o pan pan.c

# run pan (increase -m and -w as needed)
./pan -m50000000 -w28

# verbose trace for human-readable output
spin -v model.pml
```

8 Conclusions and Future Work

This project shows SPIN can be an effective debugging and formal-checking tool for deterministic policies from tabular RL in small environments. SPIN produces concrete counterexamples that can be replayed in the original environment for demonstration and debugging.

Future improvements:

- Automate Promela abstraction for larger environments (symbolic abstractions).

- Integrate automated policy repair suggestions (e.g., synthesize a minimal patch to avoid the unsafe action).
- Combine SPIN with state abstraction or predicate abstraction to scale to larger problems (e.g., Taxi).
- Follow the stochastic nature of some of these policies and environments and experiment with probabilistic model checkers, like PRISM.

Appendix A: Useful Promela snippet

```
/* Example guard for CliffWalking cell (r,c) -> action a */
:: (!goal_reached && x==2 && y==3) ->
    atomic {
        if
        :: ( /* out-of-bounds */ ) -> /* ... */
        :: ( /* next cell is cliff */ ) -> printf("OFF:%d:%d:%d\n", x, y, a); assert(0)
        :: else -> /* move */ x = x+1; y = y;
        fi
    }
```

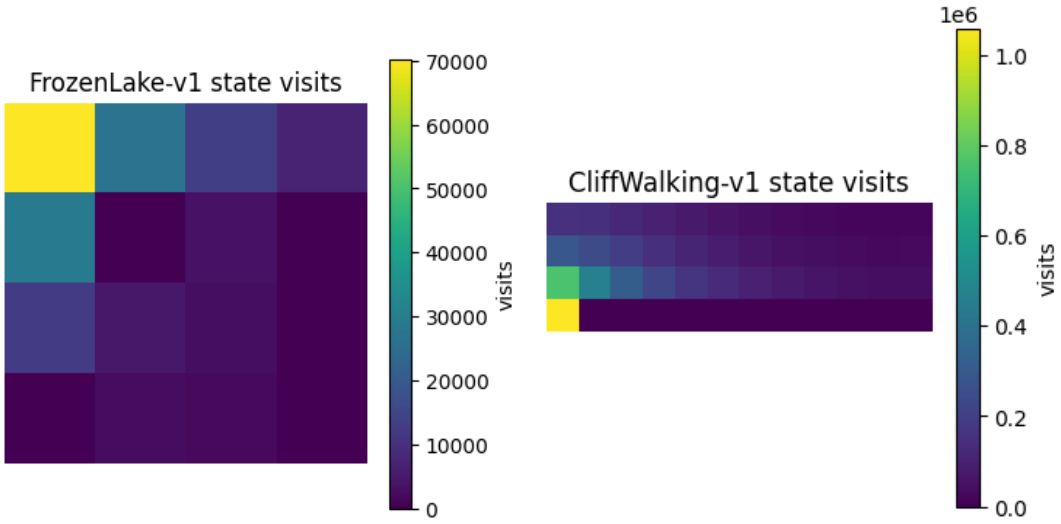


Figure 5: State visits during training exploration for FrozenLake and CliffWalking envs.