Beyond Value: CHECKLIST for Testing Inferences in Planning-Based RL

Kin-Ho Lam, Delyar Tabatabai, Jed Irvine, Donald Bertucci, Anita Ruangrotsakun, Minsuk Kahng, Alan Fern

Oregon State University
{lamki, tabatase, jed.irvine, bertuccd, ruangroc, minsuk.kahng, alan.fern}@oregonstate.edu

Abstract

Reinforcement learning (RL) agents are commonly evaluated via their expected value over a distribution of test scenarios. Unfortunately, this evaluation approach provides limited evidence for post-deployment generalization beyond the test distribution. In this paper, we address this limitation by extending the recent CHECKLIST testing methodology from natural language processing to planning-based RL. Specifically, we consider testing RL agents that make decisions via online tree search using a learned transition model and value function. The key idea is to improve the assessment of future performance via a CHECKLIST approach for exploring and assessing the agent's inferences during tree search. The approach provides the user with an interface and general query-rule mechanism for identifying potential inference flaws and validating expected inference invariances. We present a user study involving knowledgeable AI researchers using the approach to evaluate an agent trained to play a complex real-time strategy game. The results show the approach is effective in allowing users to identify previously-unknown flaws in the agent's reasoning. In addition, our analysis provides insight into how AI experts use this type of testing approach, which may help improve future instantiations.

1 Introduction

Evaluating reinforcement learning (RL) agents is typically done by estimating a single quantity, the expected value, via Monte-Carlo simulation over a set of validation scenarios. However, this single quantity alone provides little insight into the agent's underlying behavior and reasoning. As a result, this evaluation methodology may not uncover flaws in the agent that hurt generalization to reasonable post-deployment situations. For example, an agent may learn behaviors that maximize reward by abusing details of the simulator used for training and validation (Baker et al. 2020), learn aberrant behaviors unrelated to the general task, or the training reward function may not always relate to task accomplishment after deployment (Clark 2019; Gleave et al. 2021). Further, for planning-based RL agents, a systemic error in the model may go undetected during evaluation, but still manifest in rare, but serious, post-deployment failures.

One approach to improving confidence in RL agents during validation is to produce explanations of agent decisions that can be evaluated by a human (Puiutta and Veith 2020; Heuillet, Couthouis, and Díaz-Rodríguez 2021). There are a wide range of explanation mechanisms for RL, which primarily focus on explaining learned reactive policies. For example, explanations may highlight the most salient parts of an agent's input (e.g. (Greydanus et al. 2018; Mott et al. 2019)), extract interpretable structure of the agent's policy (Koul, Fern, and Greydanus 2019; Verma et al. 2018; Asai and Muise 2020; Madumal et al. 2020), provide a rationale for preferring one action over another (van der Waa et al. 2018; Juozapaitis et al. 2019; Lin, Lam, and Fern 2021), or visualize the internal representations (Wang et al. 2018; Tabatabai et al. 2021; Mishra et al. 2022). Unfortunately, these and most other types of explanations in RL provide no insight into the internal "reasoning steps" that result in action selection. This limits evaluation to the level of overall decisions/actions, for example, noticing that an agent paid attention to seemingly irrelevant information when selecting a particular action.

One way to provide explanations at the level of reasoning steps is to consider planning-based RL agents, which plan using learned models and control knowledge. In concept, this allows for human-validation of an agent's internal reasoning used for action selection. This approach, however, raises at least two challenges.

First, such agents may perform at a superhuman level where the reasoning is too complex for human consumption (e.g. enormous minimax trees). While work on explainable planning (Fox, Long, and Magazzeni 2017; Chakraborti, Sreedharan, and Kambhampati 2020) attempts to mitigate this issue, humans ultimately have limited capacity. In this work, we address this issue by assuming the RL agents use sound planning algorithms (e.g. minimax), which means mistakes are due to inaccuracies in the learned models and/or control knowledge. This implies humans can focus validation effort towards building confidence in the prediction accuracy of learned components, rather than understanding how the planner combines the predictions into an overall decision.

Second, the sheer volume of information to be validated (e.g. thousands to millions of possible actions considered by RL agents) raises the question of how humans can efficiently inspect and analyze them. We address this challenge by taking inspiration from the recent CHECKLIST methodology

for testing learned systems for natural language processing (NLP) (Ribeiro et al. 2020). CHECKLIST allows human experts or developers to create rules that capture invariants or other properties the learned system should never satisfy. In this way, rules provide an empirical and reproducible validation metric for humans to identify flaws in NLP systems.

Our main contribution is an adaptation of CHECKLIST for validation of planning-based RL agents, which we refer to as CHECKLIST *for RL (C4RL)*. While simple in concept, we are unaware of prior work that operationalizes and evaluates such an approach in the context of RL. C4RL allows a human to use their domain and RL-architecture knowledge to create logical assertions, called *query-rules*. A query-rule is a model-agnostic, domain-specific relational-algebra expression that asserts a property the agent's reasoning should never, or very rarely, satisfy. For example, query-rules might specify known invariants or pairwise action-value orderings of a learned value function. We present three general classes of query-rules that can be used for testing reasoning traces produced during validation runs as well as counterfactual situations derived from the traces.

Our second contribution is a formative user study to observe how one might use C4RL to find agent reasoning flaws. Participants were provided with an interface to validate a complex real-time strategy (RTS) game played by a planning-based agent, while creating and evaluating query-rules. They successfully formed a range of query-rules that identify known and previously-unknown flaws in the agent's reasoning.

2 RL Architecture and Game Environment

While the C4RL methodology is domain and model agnostic and can be instantiated for different application domains and types of RL agents, for concreteness, we present C4RL in the context of a game environment and RL architecture developed for our user study. This section describes our real-time strategy game, Tug-of-War, and RL agent used.

2.1 Experimental Domain: Tug-of-War

Tug-of-War (ToW) is an adversarial two-player RTS game, built on using the StarCraft 2 game engine. ToW is a challenging domain for RL because of the large state and action spaces, complex dynamics, sparse reward, etc. We used the ToW environment we developed in prior work (Lam et al. 2021). Below we provide an overview.

In ToW, two players, Friendly AI (on the left) and Enemy AI (on the right), adversarially play against each other. The map is divided into the top and bottom lanes, where player's bases are placed on opposite ends (Figure 1). The game proceeds in 30 second waves, where before each wave, each player may select either the top or bottom lane and decide which military-unit production buildings to build in the selected lane with their available currency.

At the start of each wave, each production building produces one unit of the specified type. The units automatically walk across the lane toward the enemy base and automatically attack incoming enemy units or the opponent's base if close enough. Each unit has an initial amount of health that

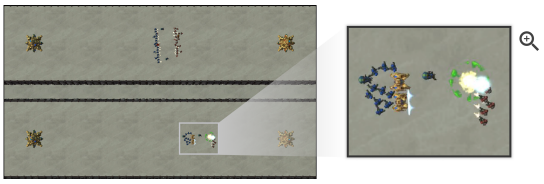


Figure 1: The Tug of War map is divided into two separate lanes, top and bottom. The Friendly AI (left) and Enemy AI (right) own bases (gold star-shaped buildings) on opposite sides of the map. Each round, troops from the opposing player automatically march towards their opponent's side of the map and attack the closest enemy in their lane.

decreases when attacked until no health is remaining and the unit disappears. The three unit types, Marines, Immortals, and Banelings, form a rock-paper-scissors relationship with respect to the amount of damage done when one unit attacks another. The first player to destroy one of the opponent bases wins. If no base is destroyed after 40 waves (20 minutes), the player with the lowest health base loses.

Importantly, players do not control the detailed movement and target selection of individual units as they move across the map, but rather control only the higher-level choice of how to spend resources each wave. Thus, the possible actions/choices available at each wave are the different purchase combinations that can be afforded with the current resources, which can range from 10s to 1000s of possible actions. At any moment of the game there can be 10s to 100s of units on the map, which creates an enormous state space.

Preceding each wave, the RL agents observe the state of the map and select an action that will dictate resource spending at the start of the next round. In concept, the observations provide perfect state information. The game dynamics have randomness due to variations in unit movements, organization, and attacks, making future states and opponent actions difficult to predict. The sparse reward is zero for both players at each decision point until the end of the game where the winning player receives +1 reward and the losing player receives 0. The ToW discrete state space describes the quantity of each troop on the field, the health of the bases, current buildings owned by each player, and available currency to purchase buildings. The discrete action space describes the quantity of each building type a player may purchase.

2.2 Planning-Based Agent Architecture

We used an agent we developed in prior work, and below we overview its architecture (Lam et al. 2021). This planning-based RL architecture makes decisions via tree search using a learned game dynamics model and leaf evaluation function. We use the terminology "planning-based RL" rather than "model-based RL" to emphasize that decisions are made via deliberative planning. This is in contrast to many model-based RL agents, which primarily use the learned model to train a reactive policy with additional simulated experience (e.g. (Sutton 1990; Kaiser et al. 2020)). The search trees produced by the planning-based architecture can serve as a type of explanation for each decision and will be the main artifacts being validated via C4RL.

The agent architecture is similar to AlphaZero (Silver et al. 2018), an RL agent based on game-tree search, except that our agent uses a learned model rather than an exact model for tree construction. Our agent searches over human-interpretable abstract states, each providing information about the health of bases, building counts, the number of friendly/enemy troops of each type in each of four evenly divided grid cells per lane, etc. This abstraction (see Figure 2-C-1) is rich enough for humans to understand the states and make reasonable decisions.

At each decision point, an action is selected by building a minimax search tree using three learned components: 1) *Transition Model*, which predicts the next abstract state (i.e., 30 seconds after) given a current abstract state and both players' actions; 2) *Action Ranker*, which returns a numeric ranking over actions in an abstract state based on their estimated action values; and 3) *State-Value Function*, which returns a value estimate (probability of winning) given an abstract state. The transition model supports building the tree starting at the current abstract state which becomes the root node. The action ranker prunes actions from the tree among many possible friendly and enemy action combinations, to include between 20 to 5 friendly AI's actions and between 10 to 3 enemy's actions depending on the depth of the tree. We specified the depth of the tree as 2, meaning that the tree is a two-step look-ahead. Figure 2-C illustrates a subset of the tree: the leftmost node is the root (C-1); six action combinations are shown next to the root (C-2); the next states are predicted by the agent (C-3); and the rightmost node is a leaf of the tree at depth 2.

3 CHECKLIST for Planning-Based RL

Our work takes inspiration from the recent CHECKLIST methodology for validating natural language processing (NLP) systems. We first overview CHECKLIST for NLP, and then describe our adaptation for planning-based RL agents.

3.1 Background: CHECKLIST for NLP

State-of-the-art in NLP is dominated by machine-learning approaches trained on large data corpuses. Validation of such NLP systems is complicated by both their black-box nature and the vastness of possible natural-language inputs. Recently, Ribeiro et al. (2020) proposed CHECKLIST as a methodology for validating NLP systems and demonstrated its potential. At a high level, CHECKLIST directs NLP developers to use behavioral-testing approaches from software engineering to identify categories of errors (e.g. violating prediction invariance to types of input perturbations). A *domain-specific language (DSL)* is used to define abstract test cases that via rules that can generate a large corpus of ground test cases for validation. This approach allows a developer or tester to quickly produce thousands of valid and reproducible tests for an NLP system using their linguistic, domain, and NLP knowledge.

In more detail, CHECKLIST allows human testers to define *test types* that specify classes of inputs (e.g. sentences of a specific form) and desired outputs (e.g. a sentiment classification). The test types are instantiated via human-created *test templates*, that specify general sentence structures with variables for which specified sets of words can be substituted. For example, in sentiment analysis, a template may transform a pre-existing "positive" sentiment test sentence to a negated form, which should result in the system producing a "negative" sentiment classification. This approach demonstrated human testers were able to effectively bring large quantities of previously unknown failures in NLP sentiment-analysis systems to the attention of developers.

3.2 Adapting CHECKLIST to RL

While the problems studied in NLP and RL are quite different, we describe how the high-level idea of CHECKLIST for NLP can be adapted to flaw identification for planning-based RL. Conceptually, CHECKLIST for NLP allows a tester to translate common-sense and domain-specific knowledge into sets of examples (i.e. sentences), each with desired system responses that can be checked. Similarly, an RL engineer "CheckLists" an RL agent by translating their common-sense and domain-specific knowledge to tests on reasoning steps of an RL agent. Thus, rather than regarding an RL agent's performance by the value of its cumulative rewards on hold-out data, C4RL encourages engineers to also judge an agent based on the quality of its reasoning.

More specifically, CHECKLIST for NLP is based on defining *test types* that are instantiated into *test templates* for example generation. C4RL instead uses *query classes* and *query rules*, which can apply to search trees. While we expect the space of query classes to increase as C4RL evolves, in this work, we support the following three classes.

1. **Static State Rules.** Agent search trees contain many abstract states produced by learned knowledge that are also evaluated (e.g. estimating the win probability). Static state rules look for violations of constraints among state variables and value estimates that should always or typically hold. For example, in our ToW domain, there cannot be units of a certain type on the field unless there are production buildings for that unit type.
2. **Transition Rules.** Agent search trees contain state-action-state transitions that should capture the causal dynamics of a domain. Transition rules look for violations of causal properties that should always or typically hold. For example, in ToW, base health can never increase, so transitioning from one state to another should never result in increasing a base's health.
3. **Symmetry Rules.** In many planning domains there will be "common sense" symmetries based on domain knowledge, where correct inferences depend on learning a specific relationship. Similar to metamorphic testing (Barr et al. 2015), symmetry rules validate the agent has learned such logical consistencies. For example, in ToW, an agent's knowledge and reasoning should be invariant to swapping the top and bottom lanes or switching the sides of the two players. Our C4RL system includes native support for defining such rules.

To support the construction of rules for an RL domain, we assume a semantically-meaningful domain schema that defines the relevant entities, attributes, and relations. The

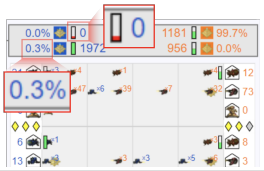
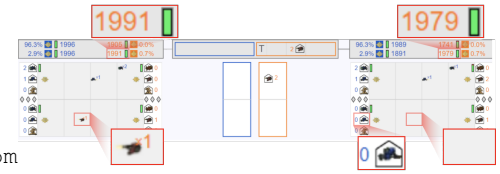
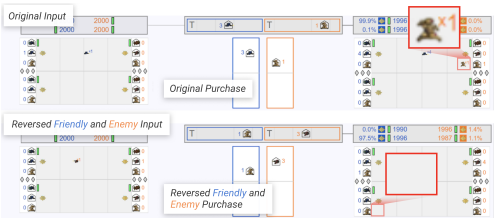
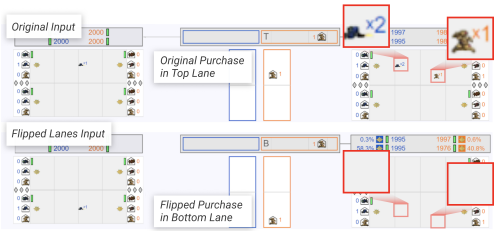
Query Class	Example Query Rule
Static State Rules: Validate a single output state	*Example 1-1.* This outputs state produced by the agent places a troop (immortal) on the field although there are no buildings to produce it. `outputState.friendlyImmortalBldgsTop = 0 AND ( outputState.friendlyImmortalTopGrid1 + outputState.friendlyImmortalTopGrid2 + ... + outputState.friendlyImmortalTopGrid4) > 0` 
	Example 1-2. The agent assigns a non-zero probability of winning the game by destroying the enemy's bottom base despite having already lost. `outputState.friendlyHealthTop = 0 AND (winProb.probabilityOfWinInTopLane + winProb.probabilityOfWinInBottomLane) != 0` 
Transition Rules: Validate the relationships between an output state and its input.	*Example 2-1.* Base health cannot increase due to game rules. In this example, the output state's base health is higher than the input's. See Section 5.1 for more details. `outputState.enemyHealthTop - inputState.enemyHealthTop > 5.0` 
	Example 2-2. The enemy's marine troop disappears although there is no friendly troop to destroy it. Additionally, the enemy's bottom base loses health even though there are no friendly troops to damage it. `(outputState.friendlyMarineBldgsBottom + outputState.friendlyBanelingBldgsBottom + outputState.friendlyImmortalBldgsBottom = 0) AND outputState.enemyHealthBottom < inputState.enemyHealthBottom` 
Symmetry Rules: Validate if the agent has learned the symmetric or player-agnostic properties of the domain.	*Example 3-1.* Reversing the features for the two players where the friendly agent becomes the enemy and the enemy becomes the friendly should produce a perfectly reversed output state. In this example the agent failed to mirror the immortal troop. `outputState.friendlyMarineTopGrid1 > 0 AND outputState.friendlyMarineTopGrid1 != outputStateForReversedInputs.enemyMarineTopGrid4` 
	Example 3-2. Flipping the top and bottom lanes should produce a perfectly flipped output state. In this example, the agent failed to produce a flipped version of the output state (bottom right). `outputState.friendlyMarineTopGrid2 != outputStateForFlippedInputs. friendlyMarineBottomGrid2` 

Table 1: Three classes of C4RL query rules with examples

schema should be rich enough to capture the the states and actions of the agent's search trees. Given a domain schema, query rules can be constructed using a domain-specific language (DSL), which may be application specific or more general purpose. As detailed in Section 4, we leverage database technology for this purpose, where query rules are converted to SQL for efficient processing.

Note that by assuming a domain schema, we are restricting the proposed C4RL framework to RL agents that build search trees over interpretable states and actions (e.g. AlphaGo searches over explicit Go positions). This rules out agents that build search trees over learned latent state representations, which are not directly human interpretable (e.g. MuZero (Schrittwieser et al. 2020)). Such a restriction is currently necessary whenever humans want to interact with an agent's knowledge and reasoning, since the problem of interpreting learned latent representations is in very early stages of research (e.g. (Voskuil et al. 2021)). Indeed, there are no current approaches that can reliably map such latent representations to human-interpretable schemas over which C4RL query rules could be defined.

4 C4RL Instantiation for Tug-of-War

We describe the C4RL instantiation for ToW used for our evaluation in Section 5. While some elements of the instantiation are necessarily specialized to ToW, the general structure can serve as a schema for C4RL in other domains.

4.1 Schema & Query Language

Given a collection of game episodes, we store the agent's search trees (one per decision point) into a relational database and transform users' query-rule expressions into SQL queries to retrieve matched records from the databases. In this work, we consider game episodes played against opponents added to the opponent pool during training. Our relational schema consists of the following tables:

- **`Episodes`** (`episodeId, isWin, ...`)
- **`States`** (`id, episodeId, decisionIdx, isRoot, friendlyMarineBldgsTop, friendlyMarineBldgsBottom, ...`)
- **`Actions`** (`id, parentStateId, numOfMarineBldgsPurchasedByFriendly, ...`)
- **`WinProbabilityOfState`** (`id, parentStateId, probabilityOfDestroyingEnemyTopBase, ...`)

This schema allows for representing the agent's tree at each decision point (`decisionIdx`) of each episode (`episodeID`), by encoding each tree path as chains of states and parent actions. The state schema has attributes for describing the agent's state abstraction, including the number of each type of building, current base health, etc. The action attributes encode the action parameters of both the friendly and enemy agent at a decision point, which specify the exact purchases made by the friendly and enemy agent. Importantly, recall that the trees are constructed based on learned transition models. Thus, some state attributes in the trees are produced via predictions from those models. In addition, the agent also makes learned predictions about the probability of each win/loss condition at each state, captured by `WinProbabilityOfState`. Using this schema, our three rule types, for validating the learned predictions, can be captured as follows

Static State Rules apply to records in the `States` table and have a general relational algebra form: $\sigma_{\text{user-specified rule}}\mathbf{States}$, where *user-specified rule* is a boolean expression over the numeric state attributes. Table 1 gives two examples of such user defined rules.

Transition Rules apply to predicted state transitions in the search trees, where we refer to the parent state of a predicted transition as the *input state* and the child state as the *output state*. Transition rules have the following general relational algebra form: $\sigma_{\text{user-specified rule}}\mathbf{States} \bowtie \mathbf{Actions} \bowtie \mathbf{States}$ where the first `States` represents the input state, the second `States` represents the output state, and $\bowtie$ is the *join* operator. Again, *user-specified rule* is a boolean expression over the numeric attributes of the input/output states and action. Table 1 gives two examples.

Symmetry Rules differ from the above classes in that they involve testing of the agent's prediction on counterfactual states that were not in the original episodes. An example is to check if the win probabilities of the agent change significantly if the top and bottom lanes were swapped. While there exist many ways to support this, we create additional database tables that store attributes of counterfactual states and actions resulting from symmetry transformations of the original states and actions. Specifically, we perturb all static states and their corresponding action pairs, producing flipped or reversed versions of the originals. We then run inference on all flipped or reversed static states and actions, and include a reference in the schema linking the inference result of a flipped or reversed state and action to its original result. The relational algebra form of the queries is like that of transitions, except the join involves an original state-action-state and corresponding transformed state-action-state. The *user-specified rule* can then be any boolean expression that compares information between the original and transformed transition data. Table 1 gives two examples.

4.2 Visualization of Search Trees

To help users examine the results from C4RL queries, we developed a visualization of the agent's search trees by adapting our earlier work (Khanna et al. 2022; Tabatabai et al. 2021). A tree starts with the root node (Figure 2 C-1), which represents the state at the selected decision point. Each state node has a compact view (smaller box with 4 health points for each base) and a larger state view (graphical representation with thumbnail of the game map) that can be expanded by the user. Nodes connected to the root node in the tree show combinations of friendly (blue) and enemy (orange) actions (at C-2) and states the friendly agent predicts next (at C-3). Figure 2-C shows 6 children of the root node, where one of them is expanded, while the remaining five are in the compact view. For the expanded example (right of C-1), the friendly agent's action is buying 2 baneling buildings and 1 immortal building in the bottom lane and the enemy agent's action is buying 1 marine building in the top lane. The agent's prediction of the state at the next decision point (30 seconds after action) is displayed to the right of the corresponding action (C-3). Each predicted state includes the predicted win probabilities at the top of the state. The four percentages give the probability that the friendly/enemy agent will win in the top/bottom lanes. For example, at C-3 the predicted state shows the agent is 95.4% confident it will end up winning the game from the top lane. This state node is followed by another pair of actions, and the predicted grandchild state (leaf of the search tree).

4.3 C4RL Interface

We design and build an interactive C4RL user interface for specifying query rules and visualizing violations. For purposes of the user study, the current interface is optimized to explore a single selected game at a time, though it is straightforward to support multiple game repositories. The interface (shown in Figure 2) consists of three sections: **A.** Query Rule Specification, **B.** Flaw Count over Decisions, and **C.** Search Tree Visualization with Flaws Highlighted. Figure 2 illustrates a case of checking whether the agent's transition predictions violate the *monotonicity* constraint that the Health Points (HP) of a base can never increase.

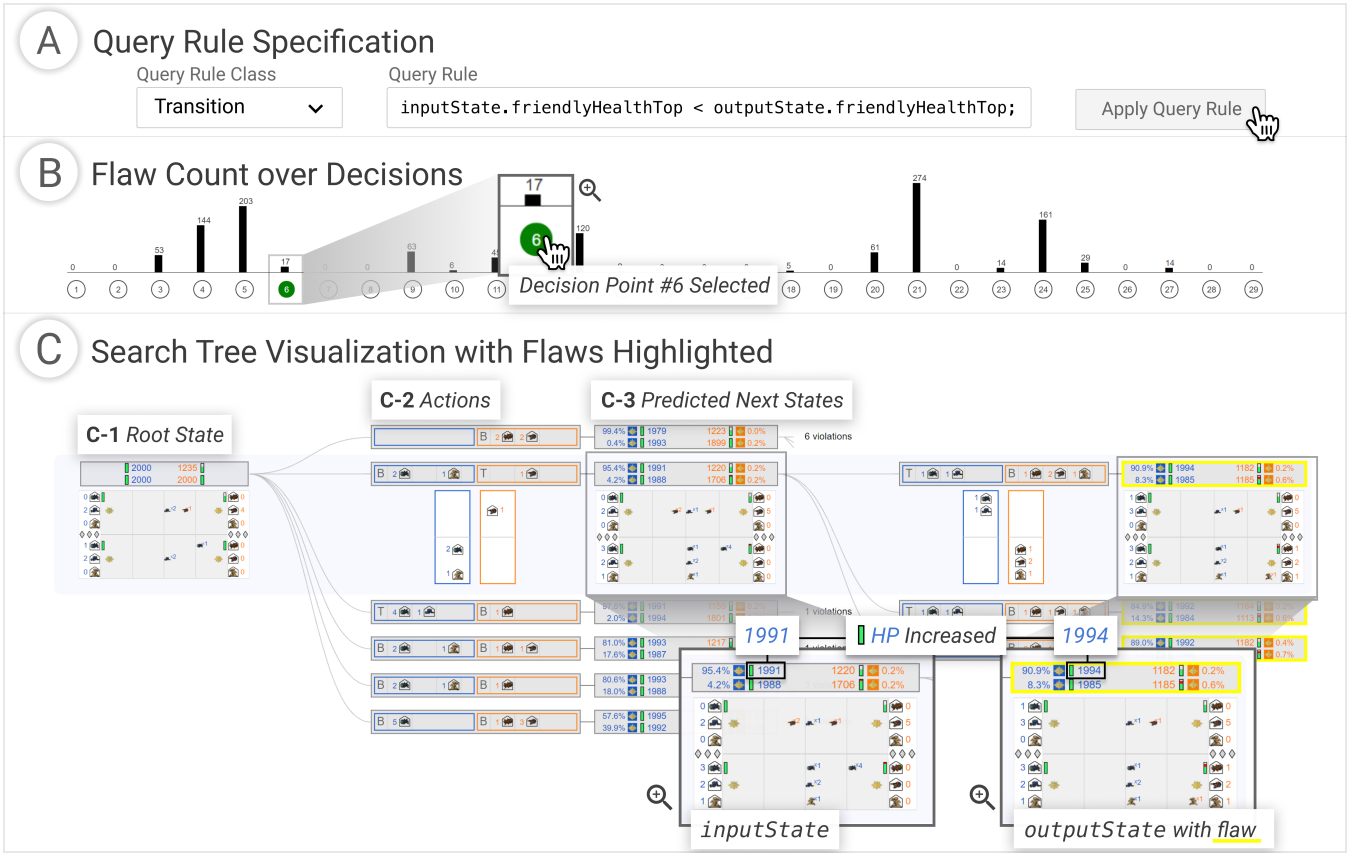


Figure 2: C4RL user interface displaying results of a query rule. Above, a user constructs a query rule stating the agent's base Health Points (HP) cannot increase (section A). After applying this rule, the user selects decision point #6 which contains 17 flaws (section B). This displays a visualization of the matching subset of paths from the full search tree (section C). The root state (C-1) depicts the state at the selected decision point. Nodes connected to this state represent Friendly (blue) and Enemy (orange) actions (C-2) and subsequent predicted next states (C-3). Nodes can be expanded to show the full state features. Nodes matching this query rule (thus containing a flaw) are highlighted in yellow.

A. Query Rule Specification. The user can specify the monotonicity-checking rule using the first section (at Figure 2-A). They specify the rule class using the dropdown menu, (i.e., Transition) and the rule using a SQL-like statement: `inputState.friendlyHealthTop < outputState.friendlyHealthTop`. The interface provides a visual guide for users to simply click components to auto-complete their attribute names.

B. Violations over Decisions. Upon clicking the apply button, the system internally transforms the rule into an SQL query and runs it over the database. A bar chart is shown to denote the number of query-rule violations in the tree at each decision point during the game (depicted at Figure 2-B). The user can then select decision points for a deeper investigation of violations in a particular search tree.

C. Search Tree Examinations. Upon selecting a decision point, the interface displays a subset of the search tree where query-rule matching nodes are highlighted in yellow. In Figure 2-C, grandchildren nodes on the right are highlighted because the base's health (HP = 1994) is greater than that of the parent state (HP = 1991) which match the query rule specified by the user. In other words, this is a flaw where health monotonicity is violated.

Users may perform further analysis by expanding the collapsed representation of the nodes (or the sub-trees), to examine the details of the other highlighted violations or other decision points. The end result of this exploration might be a report that gives concrete examples of violations, which AI engineers can use to judge the severity of violations, locate the cause of faults, and plan for potential fixes (e.g. additional training in specific scenarios)(Nushi, Kamar, and Horvitz 2018; Khanna et al. 2022).

5 Qualitative Experimental Study

We present two qualitative investigations for C4RL in the context of Tug-of-War. We describe a case study where agent developers used C4RL to discover agent flaws and a user study we conducted with 7 human-subject participants.

5.1 Agent Developer Case Study

A subset of the authors who developed the agent "CheckList" it using C4RL. For a learned agent which is able to

win 80% to 90% of ToW games against different model-free agents, we selected one game replay where the agent lost. This game consists of 29 decision points and 103,520 transition inferences, where a single search tree has a maximum of 1,227 transitions, of which contain 3,680 static states. Using C4RL, developers investigated how the agent's reasoning flaws at the inference level contributed to its defeat.

No buildings but units on the field (Example 1-1). Example 1-1 in Table 1 depicts an abstract state produced by the agent where it expects to have an immortal unit on the field; an impossible inference because the agent has not built any buildings to produce this unit. This query rule finds a total of 663 unique instances of this static state flaw in the entire game from 3 of 29 decision points. There are 611 static state flaws occured at decision point 29 (the last decision point in the game before the agent loses) out of 3,680 static states. This type of error could completely change the evaluation of a static state and its upstream transitions. The developers hypothesize the agent has not experienced enough situations with immortals as the agent rarely uses immortals or has not learned the unit/building association. Modifying this query rule to check whether this flaw occurs with another troop, banelings, yields only two static state violations in the entire game, all occurring at decision point 2. This shows the agent is missing some common-sense regarding production of baneling troops too, but the static state violations involving the immortal troops are more prevalent. This could be due to the fact that the agent has more training experience with banelings. Banelings are cheaper to purchase and are bought more frequently than immortals.

Monotonicity violation (Example 2-1). In Tug-of-War, the health of a base can never increase. Example 2-1 in Table 1 checks for non-trivial violations of this monotonicity property for the enemy's top base. It found 26,540 violations out of all 103,520 predicted transitions in the game trees. The violations occur at almost all decision points, except for the first two. Interestingly, in the specific flaw instance shown in Example 2-1, the agent's bottom base also violates the monotonicity property. Modifying this prompt to instead check monotonicity for the agent's top base finds 2,403 violations dispersed across 5 decision points. This is a serious error that clearly demonstrates the the agent has not fully learned some key constraints of the game. The developers noticed that the flaws occur most frequently in states where the agent's base has lower health, which suggests more agent training is needed in such situations.

Phantom Health Decrease (Example 2.2). If an agent does not have any buildings, the opponent's base should have full health (2000), because no troops are present to damage it. This fact is expressed by the query rule shown in Example 2-2 in Table 1 which returns 6,311 violations, all occurring in the first five decision points. This is another indicator that the agent has not fully captured common sense constraints about the health dynamics of bases.

5.2 User Study

We conducted an in-person qualitative user study to investigate how researchers and engineers with AI expertise, but no prior experience with the agent or domain, would use C4RL.

Participants and Protocol Recruited participants included 7 graduate students at our university (2 M.S. and 5 Ph.D), who had taken a course on Reinforcement Learning and did not have previous experience with the task, domain, or the C4RL interface.

Participants were invited to a two hour session in a controlled laboratory environment and were compensated with a $20 gift card at the conclusion of the study. All sessions began with a 30 minute tutorial covering the ToW domain, agent architecture basics, and one query-rule example for each query-rule class. Participants were then asked to use the C4RL interface to construct as many query-rules as they could for the remainder of the session. To help formulate query-rules, participants were provided a cheatsheet with examples from the tutorial. Participants recorded their query-rules and described the intent of their rules in a spreadsheet.

Analysis User-study result analysis was conducted based on the following five research questions.

Q1. Are participants able to produce correct rules? We consider three cases of rule correctness. *Sound rules* are logical consequences of the domain constraints. Violation of a *sound rule* imply said violation is true a reasoning flaw. *Suspicion rules* correspond to common-sense game conditions that will usually hold, thus a violation should be investigated. Finally, *unsound rules* are invalid rules that indicate the participant formed an incorrect assumption; violations of aforementioned do not reflect agent flaws.

Participants constructed a total of 126 query-rules. The research team analyzed their correctness and found 16 unsound, 98 sound, and 12 suspicion rules. This indicates participants, given a short tutorial, were able to "checklist" the agent.

Q2. What types of rules did participants specify? Researchers grouped the 110 sound or suspicion rules into multiple categories, through *affinity diagramming*, a well-known technique used in user study analysis. Two of the authors went through the 110 rules independently, assigned categories based on the type of RL violations they would detect, and discussed standard category names to resolve disagreed rule categorizations. The following four high-level rule categories were identified.

- **Monotonicity Violation.** While the values of certain features must behave monotonically, the agent's output (e.g. predicted state) may not follow this property. For example, base health, a feature for a state, cannot increase.
- **Value Out Of Range Violation.** The values of some features must be within certain ranges defined by domain constraints. For example, base health must always be within a range of 0 to 2000.
- **Causal Violation.** The values of some features can only be changed when other features are changed. An example of causal violation includes when the agent produces a state where there are troops on the field, but there is no buildings that can produce them.
- **Symmetry Violation.** This includes all cases for the *symmetry rules* class. When inputs are flipped, nearly perfectly flipped outputs are expected.

Category	Sub-categories (domain-specific)	Count
Monotonicity	Base health (increasing)	4
	Building count	3
Value Out of Range	Base health	12
	Win probability	7
	Number of units	1
Causal Violation	Not supposed to win/lose	26
	No building but troops	19
	No building but health change	7
	Unit positioning	5
	No troops but health change	3
	Rock-paper-scissors	1
Symmetries	Base health	9
	Win probability	8
	Unit count	4
	Building count	1

Table 2: Violation Categories Table: Four categories of high-level RL violations and 15 sub-categories according to the domain. The rightmost column displays total rule counts constructed by participants.

Table 2 further decomposes the high-level categories into sub-categories that are Tug-of-War specific and provides the count for each. The most common rule type was *causal violations* (61 rules), followed by *symmetries* (22 rules), *value-out-of-range* (20 rules), and *monotonicity* (7 rules). There were 6 sub-categories of causal violations, some of which were surprising. Notably, participants identified a number of rules that captured situations when the agent erroneously predicted a win/loss (i.e., "*not supposed to win/lose*"). For example, participant #1 specified a causal relationship between the chance of winning and the agent's base health as "*[If the base health of friendly AI in the top lane is equal to zero, then the chance of winning for friendly AI in the top lane should be zero.]*" The fact that the agent violated such a rule was surprising and clearly a point of concern.

Q3. Can participants creatively construct new rules? Participants were given two types of guidance: a handout of example rules and screenshots of trees with flaws. We analyzed how many of the participant rules were variations of the guidance versus original constructions. Each rule was judged whether it was ideated from: 1) handout, 2) a screenshot, or 3) none of these (i.e. original). Of the 110 sound or suspicion rules, only 5 were direct adaptations of rules explicitly given in the handout and 30 were constructed based on flaws in the tree examples provided. The remaining 76 rules (68%) appeared to be original and functionally distinct from the provided examples. This included a number of rule forms that the research team had not previously considered. For example, participant #1 reported a suspicion rule regarding the "Rock-Paper-Scissors" relationship among the three types of troops as "*If the friendly AI has some Banelings and the enemy AI did not purchase any Immortal buildings, it should be guaranteed that the enemy base health is going to be reduced.*" Participants also expressed rules in English that the current version of the C4RL syntax does not support. For example, participant #3 wrote, "... a violation type to combine player swap and lane swap. ... When both players and lanes are swapped, friendly top base health become enemy bottom base health."

Q4. How diverse are the rules formed by each participant? Overall there were a diversity of rules among the 4 categories and 15 sub-categories. However, this leaves the question of the diversity of rules for individual participants. To address this, we counted the number of unique sub-categories generated by each participant. On average participants form rules from 6.7 categories and all participants constructed at least four unique sub-categories of rules. This shows that participants were naturally considering different types of relationships throughout their investigation. We observed that the participants differed in their quantity and diversity of query rules and the strategies to find rules. For example, participant #2 constructed 41 sound or suspicion rules, the highest quantity among all participants. They focused exhaustively on checking all bounds for a narrow set of features (e.g., out of range violations). On the other hand, although participant #6 constructed only 13 rules (which is less than the average), they successfully created rules from 10 different sub-categories. This suggests further study into the types of strategies taken by C4RL users and how they might be guided toward the most effective strategies.

Q5. Are there any patterns in the participants' exploration? Lastly, we wondered if there exist any interesting patterns in the list of query-rules which participants constructed. One interesting observation is that some participants formed query-rules based on what they previously constructed. For example, after participant #2 formed a rule as "*If friendly AI dies in the top lane, then enemy wins,*" they constructed 7 query-rules that have almost the same structure, such as "*If friendly AI dies in the **bottom** lane, then enemy wins*," "*If **enemy** dies in the top lane, then **friendly AI** wins*," and so on. This implies that some of the rules have many variations that can easily be populated systematically. An interesting future direction would be to create templates for the rules, so that the system automatically creates or suggests combinations for rules so that the users do not need to construct variations of the same rule manually.

6 Summary

We present CheckList for Reinforcement Learning (C4RL), a behavioral testing methodology adapted for RL agents; allowing researchers to evaluate an agent beyond some measurement of expected value. We formalized three classes of query-rules and developed a C4RL system in the context of a planning-based agent for a real-time strategy game. Our qualitative studies demonstrates users can use C4RL to identify a diverse range of flaws in the agent's reasoning. Overall, C4RL is a straightforward yet effective approach for validating RL agents, and there are several possible variations to be explored.

Acknowledgements

This work was supported in part by DARPA (N66001-17-2-4030), Google (GCP19980904), and NAVER AI Lab.

References

Asai, M.; and Muise, C. 2020. Learning Neural-Symbolic Descriptive Planning Models via Cube-Space Priors: The Voyage Home (to STRIPS). In *29th International Joint Conference on Artificial Intelligence*.

Baker, B.; Kanitscheider, I.; Markov, T.; Wu, Y.; Powell, G.; McGrew, B.; and Mordatch, I. 2020. Emergent Tool Use From Multi-Agent Autocurricula. arXiv:1909.07528.

Barr, E. T.; Harman, M.; McMinn, P.; Shahbaz, M.; and Yoo, S. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering*, 41(5).

Chakraborti, T.; Sreedharan, S.; and Kambhampati, S. 2020. The Emerging Landscape of Explainable Automated Planning & Decision Making. In *IJCAI*, 4803–4811.

Clark, J. 2019. Faulty reward functions in the wild.

Fox, M.; Long, D.; and Magazzeni, D. 2017. Explainable planning. *arXiv preprint arXiv:1709.10256*.

Gleave, A.; Dennis, M.; Wild, C.; Kant, N.; Levine, S.; and Russell, S. 2021. Adversarial Policies: Attacking Deep Reinforcement Learning. arXiv:1905.10615.

Greydanus, S.; Koul, A.; Dodge, J.; and Fern, A. 2018. Visualizing and Understanding Atari Agents. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80, 1792–1801. PMLR.

Heuillet, A.; Couthouis, F.; and Díaz-Rodríguez, N. 2021. Explainability in deep reinforcement learning. *Knowledge-Based Systems*, 214: 106685.

Juozapaitis, Z.; Koul, A.; Fern, A.; Erwig, M.; and Doshi-Velez, F. 2019. Explainable reinforcement learning via reward decomposition. In *Proceedings of the IJCAI 2019 Workshop on Explainable Artificial Intelligence*, 47–53.

Kaiser, L.; Babaeizadeh, M.; Milos, P.; Osiński, B.; Campbell, R.; Czechowski, K.; Erhan, D.; Finn, C.; Kozakowski, P.; Levine, S.; Mohiuddin, A.; Sepassi, R.; Tucker, G.; and Michalewski, H. 2020. Model Based Reinforcement Learning for Atari. In *International Conference on Learning Representations*.

Khanna, R.; Dodge, J.; Anderson, A.; Dikkala, R.; Irvine, J.; Shureih, Z.; Lam, K.-h.; Matthews, C.; Lin, Z.; Kahng, M.; Fern, A.; and Burnett, M. 2022. Finding AI's Faults with AAR/AI: An Empirical Study. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 12(1).

Koul, A.; Fern, A.; and Greydanus, S. 2019. Learning Finite State Representations of Recurrent Policy Networks. In *International Conference on Learning Representations*.

Lam, K.-H.; Lin, Z.; Irvine, J.; Dodge, J.; Shureih, Z. T.; Khanna, R.; Kahng, M.; and Fern, A. 2021. Identifying Reasoning Flaws in Planning-Based RL Using Tree Explanations. arXiv:2109.13978.

Lin, Z.; Lam, K.-H.; and Fern, A. 2021. Contrastive Explanations for Reinforcement Learning via Embedded Self Predictions. In *International Conference on Learning Representations*.

Madumal, P.; Miller, T.; Sonenberg, L.; and Vetere, F. 2020. Explainable reinforcement learning through a causal lens. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, 2493–2500.

Mishra, A.; Soni, U.; Huang, J.; and Bryan, C. 2022. Why? why not? when? visual explanations of agent behavior in reinforcement learning. In *Proceedings of Pacific Visualization Symposium (PacificVis)*.

Mott, A.; Zoran, D.; Chrzanowski, M.; Wierstra, D.; and Jimenez Rezende, D. 2019. Towards Interpretable Reinforcement Learning Using Attention Augmented Agents. In *Advances in Neural Information Processing Systems*.

Nushi, B.; Kamar, E.; and Horvitz, E. 2018. Towards Accountable AI: Hybrid Human-Machine Analyses for Characterizing System Failure. *CoRR*, abs/1809.07424.

Puiutta, E.; and Veith, E. M. 2020. Explainable reinforcement learning: A survey. In *International Cross-Domain Conference for Machine Learning and Knowledge Extraction*, 77–95. Springer.

Ribeiro, M. T.; Wu, T.; Guestrin, C.; and Singh, S. 2020. Beyond Accuracy: Behavioral Testing of NLP Models with CheckList. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*.

Schrittwieser, J.; Antonoglou, I.; Hubert, T.; Simonyan, K.; Sifre, L.; Schmitt, S.; Guez, A.; Lockhart, E.; Hassabis, D.; Graepel, T.; et al. 2020. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839): 604–609.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; et al. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419): 1140–1144.

Sutton, R. S. 1990. Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming. In *International Conference on Machine Learning*, 216–224.

Tabatabai, D.; Ruangrotsakun, A.; Irvine, J.; Dodge, J.; Shureih, Z.; Lam, K.-H.; Burnett, M.; Fern, A.; and Kahng, M. 2021. "Why did my AI agent lose?": Visual Analytics for Scaling Up After-Action Review. In *2021 IEEE Visualization Conference (VIS)*. IEEE.

van der Waa, J.; van Diggelen, J.; van den Bosch, K.; and Neerincx, M. A. 2018. Contrastive Explanations for Reinforcement Learning in terms of Expected Consequences. *ArXiv*, abs/1807.08706.

Verma, A.; Murali, V.; Singh, R.; Kohli, P.; and Chaudhuri, S. 2018. Programmatically interpretable reinforcement learning. In *International Conference on Machine Learning*, 5045–5054. PMLR.

Voskuil, K.; Moerland, T. M.; Plaat, A.; et al. 2021. Visualizing MuZero Models. In *ICML 2021 Workshop on Unsupervised Reinforcement Learning*.

Wang, J.; Gou, L.; Shen, H.-W.; and Yang, H. 2018. Dqnviz: A visual analytics approach to understand deep q-networks. *IEEE Transactions on Visualization and Computer Graphics*, 25(1).