Automatic Action Space Specification for Deep Reinforcement Learning in Games via Program Analysis

Sasha Volokh
University of Southern California
Los Angeles, USA
volokh@usc.edu

William G.J. Halfond
University of Southern California
Los Angeles, USA
halfond@usc.edu

***Abstract*—Reinforcement learning has numerous applications for the testing and analysis of video game content. However, deploying it in existing games is a challenging engineering effort, requiring suitable representations of states, actions, and rewards based on the game rules. We propose using the program analysis technique of symbolic execution on the game code to automatically determine a precise action model when deploying reinforcement learning in existing games. Our technique automatically computes appropriate discrete action spaces for games, including action masks indicating the validity of actions depending on the agent's current state. We conduct a comprehensive evaluation of the technique on a varied dataset of seven Unity games with the Proximal Policy Optimization (PPO) and Deep Q Learning (DQN) deep reinforcement learning approaches. The results show that the agents using the analysis significantly out-perform those using generic action spaces covering the input device, and perform on par with those using manually specified action spaces.**

***Index Terms*—reinforcement learning, program analysis, game testing**

I. Introduction

Reinforcement learning has found many applications for game development. It can be used to deploy agents that play through [1] or explore [2]–[5] games, complement script-based testing [6], [7], and evaluate game designs [8], [9].

Given the many use cases of reinforcement learning for games, it is desirable to make it easy for game developers to adopt or experiment with it for their games. However, deploying reinforcement learning in existing games, which may not necessarily have been designed with it in mind, is a difficult technical challenge [6], [10]. Its use requires that a suitable model of states, actions, and rewards is available for the game. Typical game development practices with game engines such as Unity or Unreal Engine do not make such models available by default. Because of this, the deployment of reinforcement learning requires either a non-trivial engineering effort to enable agents to interact with the game, or the use of imprecise generic models that are more difficult to train with or give poor performance.

One of the key requirements of reinforcement learning is a specification of the available actions for agents to take in the game. This is commonly formulated as an *action space*, which defines the set of actions that can be taken by the agents. In typical game engines, the available player actions are implicitly defined by the behavior of the user input handling logic in the game code. Therefore, there is no direct way to query the available actions, and deploying reinforcement learning requires the developer to make a decision about defining the action space. The specification of the action space can be crucial to the performance of reinforcement learning. Prior works on action space shaping [11] and action masking [12] have shown that imprecise or unnecessary actions can result in worse agent performance.

Existing work takes a variety of approaches to defining an appropriate action space for games. A common approach is for the developer to use their expertise with the game and its codebase to manually engineer an interface that the agent interacts with [1]–[4], [6]. This involves implementing a standard environment interface, such as Gymnasium [13], to define an action space and translations from actions into in-game events. While such an approach is effective, it also requires a non-trivial amount of engineering work [6], [10] and knowledge about the game, which represents a barrier to the adoption of reinforcement learning agents in games. An alternative to such manual engineering is to use generic (but imprecise) actions. For example, the use of reinforcement learning with Atari games [14], [15] often involves a generic discrete action space that covers all 18 of the device input combinations. Work on general computer control with reinforcement learning [16] uses a generic action space that covers all common keyboard and mouse events. However, such generic action spaces with a large number of actions are generally more difficult to train with or give worse agent performance.

In this paper, we propose program analysis as a solution for automatically determining a precise action model when deploying reinforcement learning in existing games. With such an approach, we automate part of the involved engineering effort. Furthermore, such an analysis automatically adapts to changes in the game code, which addresses a key challenge of games frequently changing during development [6]. We describe an approach, based on the program analysis technique of symbolic execution, to analyze the user input handling logic

```
1 class PlayerController : MonoBehaviour {
2   ...
3   void Update() {
4     if (Input.GetAxis("Horizontal") > t)
5       Move(Vector2.right);
6     if (isOnGround && Input.GetButton("Jump"))
7       ApplyForce(Vector2.up);
8   }
9   ...
10 }
```

Listing 1. Example of typical user input handling logic in a Unity game

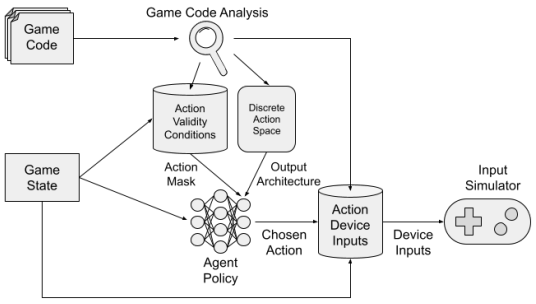


Fig. 1. System diagram of our approach for defining action spaces

in the game code and automatically obtain discrete action spaces for games. We implemented our technique for the Unity game engine and considered the widely used Proximal Policy Optimization (PPO) [17] and Deep Q Learning (DQN) [14] algorithms. In a study with a diverse set of seven Unity games, we found that our approach significantly out-performs generic action spaces, and performs on par with the ideal scenario of manually specified action spaces.

II. Background

A. Input Handling In Games

In games, the player actions are typically defined through input-handling logic in the code that reads the user's input device. We investigate the automated analysis of such code to define appropriate action spaces for reinforcement learning. An example of input-handling code for the Unity game engine can be found in Listing 1. Here, the `Update()` method defines game logic that is repeatedly executed over time. On line 4, the call to `Input.GetAxis("Horizontal")` checks the state of the user input device that is mapped to the "Horizontal" axis and returns a floating point value. Similarly, on line 6, the call to `Input.GetButton("Jump")` checks the state of the input device mapped to the "Jump" button. However, in this case, this action is only available when the `isOnGround` flag is active.

B. Defining Game Action Spaces

The actions that a reinforcement learning agent can take are typically described with an *action space*. In games, commonly used action spaces include the following [11]:

- Discrete: One action among N actions is chosen by the agent at each step of the game.
- Multi-Discrete: Multiple discrete actions from among M discrete action spaces are chosen at each step of the game.
- Continuous: The action is represented as a real number or vector of real numbers.

Prior work has shown that the specification of the action space affects the performance of reinforcement learning agents [11]. Continuous actions have been shown to be more difficult to learn than discrete ones, and reducing the number of actions can also be crucial to enable learning [11]. Some algorithms can only accept certain kinds of action spaces. For example, Deep Q Learning [14] can only work with discrete action spaces. In games where the set of valid actions varies depending on the game state, the agent performance can be improved with an *action mask* that informs the agent of the invalid actions [11], [12]. Given the challenges of learning continuous actions, we focus on discrete action spaces and discretize any continuous actions.

III. Methodology

Our approach automatically determines an appropriate action space for an existing game via analysis of the game code. Figure 1 gives an overview of our proposed automated action analysis. First, prior to gameplay, the game code analysis (Section III-A) determines a discrete action space for the game (Section III-B), along with the action validity conditions and device inputs needed to perform the actions. The validity conditions are used to generate an action mask (Section III-C) to ensure the agent policy only chooses valid actions. When an action is chosen by the agent policy, the device inputs are resolved based on the current game state (Section III-D) and finally simulated onto the game.

A. Game Code Analysis

First, we use program analysis to exhaustively determine all the player actions available in the game. This includes the game state conditions under which the actions are valid, and the device inputs needed to perform them depending on the game state. This analysis targets the user input handling logic in the game code, such as that shown in Listing 1. Prior work [18]–[20] has demonstrated that the technique of *symbolic execution* provides an effective framework with which to conduct this kind of analysis that is applicable to typical commercial game engines such as Unity. With this approach, a precise set of conditions is determined for the different execution paths through the game loop [18].

The key insight underlying the analysis is that the possible player actions are implied by the execution paths through the user input handling logic of the game code. Each of these execution paths corresponds to a particular combination of user input and game state conditions. With symbolic execution, the possible execution paths of a program can be explored by representing the inputs with symbols. We can then determine *path conditions* for each execution path of the program, which consist of a conjunction of conditions that were true at each

branch point in the program. By representing the game state and user input variables with symbols, we can obtain the conditions under which each action occurs. For example, in Listing 1, a path condition that corresponds to an action of moving right and jumping would be:

$$\texttt{Input.GetAxis("Horizontal")} > t \wedge \texttt{isOnGround} \wedge \texttt{Input.GetButton("Jump")} \quad (1)$$

This path condition consists of user input variables (starting with `Input`...) and game state variables (e.g. `isOnGround`). We define each action as an execution path through the user input handling logic. Under this formulation, an action is valid when the path condition is satisfiable (i.e. the execution path is feasible), and the solution to the path condition gives the device inputs needed to perform the action. Each of these actions take place over a single step of the game, since they correspond to a single iteration of the game loop.

The symbolic execution approach provides a natural method for discretizing continuous actions, which can improve training performance [11] and enable compatibility with algorithms that only support discrete action spaces (such as DQN). Continuous actions are defined by floating point input APIs such as `Input.GetAxis` or `Input.mousePosition`. When such an input is encountered, the execution forks into multiple new states, each having a constraint covering some part of the input range. For example, when `Input.GetAxis` at line 4 is reached, the execution forks into three states having path conditions covering the general cases of interest: negative (`Input.GetAxis("Horizontal")` < 0), zero (`Input.GetAxis("Horizontal")` $= 0$), and positive (`Input.GetAxis("Horizontal")` > 0). These are the general cases that exhibit distinct behaviors in most games, and we use the same strategy for all the games. Similarly, if the execution encounters a mouse position input (`Input.mousePosition`), the execution forks into a grid of states, each bounding the mouse position within a cell of the grid. In our implementation we use a grid size of 4x4, having found that this size is suitable for interacting with the games in our dataset. This value can be easily tuned for games needing more exact mouse movement. Doing this discretization is especially important because continuous inputs can be used in ways for which it is difficult to obtain exact constraints. For example, mouse position inputs are frequently used in ray cast operations implemented in the game engine internals. This approach addresses such cases, producing a generic set of actions covering the general cases of interest, even if exact constraints cannot be obtained.

A key challenge in the practical application of symbolic execution is the exponential growth in the number of paths with respect to the number of branch points in the program [21]. However, for the action analysis only code that is relevant to the handling of user inputs needs to be considered. Prior work has demonstrated that techniques such as call graph analysis, data-flow analysis, and program slicing can be used to automatically determine the parts of code relevant to user input handling [18], [19]. We use the same combination of techniques to enable our analysis to scale to complex games.

B. Action Space

Given the analysis results as a set of path conditions over game state and user input, we can determine an appropriate action space that is suitable for defining the agent's policy network architecture (number of outputs). Given a set of path conditions P, we can obtain a discrete action space A by associating each action $a \in A$ with a path condition $p \in P$.

A key consideration when defining the action space is that the analysis may generate many redundant no-operation (no-op) actions, where all the device inputs (e.g. keyboard and mouse) are in a released state. This is due to path conditions either not having any constraints on user input, or the constraints on user input always giving a zero value under all game states. To address this, we remove all redundant no-op actions produced by the analysis, such that we just have one no-op action as the first action in the discrete action space. Given a path condition $p \in P$ with a set of input variables $I = \{i_1, i_2, \ldots, i_n\}$, we consider p a no-op action if it does not have any input variables ($|I| = 0$) or provably always generates zero values as the solution, such that $p \wedge \neg(i_1 = 0 \wedge i_2 = 0 \wedge \cdots \wedge i_n = 0)$ is determined unsatisfiable by an SMT solver. Aside from no-op actions, the analysis may also produce actions with similar or equivalent inputs, however we have found that in general it is challenging to find automatic action space transformations that consistently retain or improve performance across a variety of games.

The overall outcome is a discrete action space of size N, with a no-op action as the first action and the rest of the actions being associated with path conditions. A deep reinforcement learning agent using this action space would typically then generate a policy network with N outputs. Because the underlying symbolic execution considers all execution paths through the input-handling code, this covers all combinations of device input. Furthermore, as discussed in Section III-A, the discretization of continuous inputs is handled by the symbolic execution as well and generates a set of actions suitable for covering the cases of interest.

C. Action Mask

Certain actions are only available under certain game state conditions. For an action with path condition p to be valid, firstly there there must be at least one active instance of the associated object type. Secondly, the condition p must be satisfiable under the current game state. It is possible that certain actions will never be valid if their associated object type is never active during gameplay. To handle the variability of valid actions during gameplay, we employ *action masking*, where the agent's policy is forced to evaluate invalid actions to low values or probabilities [12].

To compute the valid actions in the agent's current state, first a subset of path conditions $P' \subseteq P$ is computed, where each path condition has at least one active instance of its associated object type. Then, an action associated with path condition

$p \in P'$ is considered valid if p is satisfiable under the agent's current game state. Because there can be hundreds of path conditions, testing satisfiability directly with an SMT solver during gameplay is too inefficient to preserve the real-time performance of games. Therefore, we use an approximation proposed in previous work [19], where each path condition is compiled into an approximate *action validity function* that can be quickly checked. Each path condition p is is split into the parts dependent on game state, p_g, and the parts dependent on user input, p_i. Then, each p_g is compiled to a boolean function that is embedded together with the game code. For the example path condition in Equation 1 in of Section III-A, p_g is `isOnGround`, so p_g would be compiled to a boolean function that reads the field `isOnGround`. During gameplay, the action mask has a `true` value for all actions whose associated action validity function evaluates to true in the agent's current game state, and `false` for all other actions.

D. Action Device Inputs

When the agent chooses an action, the appropriate device events need to be simulated onto the game. Given the associated input condition p_i, these can be determined by adding the concrete value of all game state variables and solving the condition with an SMT solver. For the example in Equation 1 of Section III-A, the input condition depends on the game state variable `t`. Suppose `t` is 0.1 in the agent's current state. Then, the constraint $\texttt{t} = 0.1$ would be added and the condition solved with an SMT solver, giving a solution such as:

$$\begin{aligned} \texttt{Input.GetAxis("Horizontal")} &= 1.0 \\ \texttt{Input.GetButton("Jump")} &= \text{true} \end{aligned} \tag{2}$$

The input variables in this solution would then be looked up in the game's input configuration to map the solution values to the actual input device bindings needed to perform the action.

A common case to consider is that a path condition may not include all the input variables used in the entirety of the game code. This could happen if a method returns early, or if different object types read input from different parts of the input device. For this reason, when simulating the device inputs, all other parts of the input device that are not part of the solution are reset. For the example in Equation 2, all keys not associated with either the "Horizontal" axis or the "Jump" button would be released. This ensures that actions only affecting part of the input device do not remain stuck.

Finally, a key technical challenge is in simulating the device inputs. Some game engines (including Unity) do not offer the capability to simulate device events. To enable this, our implementation automatically substitutes input APIs with a version that allows simulated events.

IV. Evaluation

We evaluated our approach in a variety of Unity games. Firstly, we compared against generic action spaces that cover all common keyboard and mouse actions suitable for interacting with most games. Secondly, we compared against manually specified action spaces, representing the ideal scenario of a developer investing effort to define an action model. Finally, we ran an ablation study to evaluate the impact of the no-op removal (Section III-B) and input resetting (Section III-D) treatments. We considered the following research questions:

1) **RQ1.** How do reinforcement learning agents using the action analysis perform compared to those using generic action spaces?
2) **RQ2.** How do reinforcement learning agents using the action analysis perform compared to those using manually specified action spaces?
3) **RQ3.** What is the effect of the proposed no-op removal and input resetting treatments on the resulting performance of the reinforcement learning agents?

A. Reinforcement Learning Setup

In our evaluation, we considered the Proximal Policy Optimization (PPO) [17] and Deep Q Learning (DQN) [14] deep reinforcement learning algorithms. PPO has been widely used for video game testing and evaluation [3]–[7], [9]. DQN is a classic approach capable of learning to play video games [14] and has been used for video game testing as well [10].

We used screenshots as inputs to the policy networks for both PPO and DQN, which are both capable of learning with pixel inputs [3], [14]. Screenshots are a generic state representation that can be re-used across all the games. By holding the state representation constant, we ensured that the differences observed are due to our independent variable (the action spaces) and not a particular state representation. The policy networks for both PPO and DQN are convolutional neural networks (CNNs). As in other works [10], [14], the input to the networks is the last four game screenshots stacked, resized to 84x84, and converted to grayscale. Additionally, we implemented action masking [12], such that the network outputs are set to a very low value for the invalid actions.

During training, each chosen action is held for four in-game frames. Each round of training was run for 100,000 steps (400,000 frames), which took 1-2 days of real time due to the overhead of the experiment setup. Each environment episode was limited to 300 steps (1,200 frames), after which the game resets. For DQN, the epsilon value was annealed from 1.0 to 0.05 during the first third of training (33,000 steps), after which it was held at 0.05. All training was repeated ten times. The experiments were run on a Linux cluster, enabling us to run much of the training in parallel.

B. Game Selection

Our evaluation included seven Unity games with various genres, code complexity, and input types. The games were sourced from a combination of open-source projects on GitHub and student projects from a semester-long graduate game development course. All the games are single player games (no networking/online features) with source code available. Games were excluded from consideration if they were not compatible with our action analysis implementation due to the use of unsupported input APIs, or if they were too resource intensive to run on a single node in our cluster setup. We also

TABLE I
GAMES USED IN THE EVALUATION. THE "LOC" COLUMN GIVES THE LINES OF CODE EXCLUDING THIRD-PARTY DEPENDENCIES. THE "RL REWARD FUNCTION" COLUMN GIVES THE REWARD FUNCTION DEFINED FOR THE GAME TO BE MAXIMIZED BY REINFORCEMENT LEARNING.

ID	Source	Genre	Description	LOC	Input Type	RL Reward Function
G1	GitHub	Platformer	Super Mario Bros remake	2107	Keyboard Only	Positive reward for moving right and acquiring powerups. Penalty for damage.
G2	GitHub	Maze	Pac-Man remake	1902	Keyboard Only	Positive reward for gaining score and acquiring powerups. Penalty for dying.
G3	GitHub	Puzzle	2048 remake	699	Keyboard Only	Positive reward for gaining score.
G4	GitHub	Puzzle	Bubble Shooter game	1290	Mouse Only	Positive reward for gaining score.
G5	Student Projects	Maze	Navigate a maze with enemies to reach the goal	4990	Keyboard Only	Positive reward for getting closer to goal and acquiring pickups. Penalty for damage.
G6	Student Projects	Maze	Acquire pickups to unlock doors and reach the goal	9694	Keyboard Only	Positive reward for getting closer to goal, acquiring pickups, and gaining score. Penalty for taking damage or dying.
G7	Student Projects	Platformer	Collect stars and reach the goal by placing platforms.	2240	Keyboard + Mouse	Positive reward for getting closer to goal, acquiring pickups, and gaining score.

excluded games if we were not able to identify an appropriate reward function based on prior work, or if the training could not improve over time for any of the approaches to defining action spaces in our initial experiments (i.e. reinforcement learning was not an appropriate solution for the game).

The complete set of games used in our evaluation is listed in Table I, indicating the source of the game, its genre, description, code complexity in terms of lines of code (excluding third-party dependencies), and type of input device. The last column gives the reward function that we defined for the game to be maximized by the reinforcement learning, which corresponds to effective playing performance in the game. While these games generally have a smaller scale than modern commercial games, they are complete games having a variety of fully implemented mechanics and features, and employ the same input-handling patterns as commercial games.

Running the symbolic execution on each game took an average of 29.8 seconds (minimum 3.57 sec, maximum 100.6 sec), and the overhead incurred at run time for resolving actions was on average 16.3 milliseconds (minimum 11.6 ms, maximum 21.7 ms). We found that this was acceptable for preserving the real-time performance of the games.

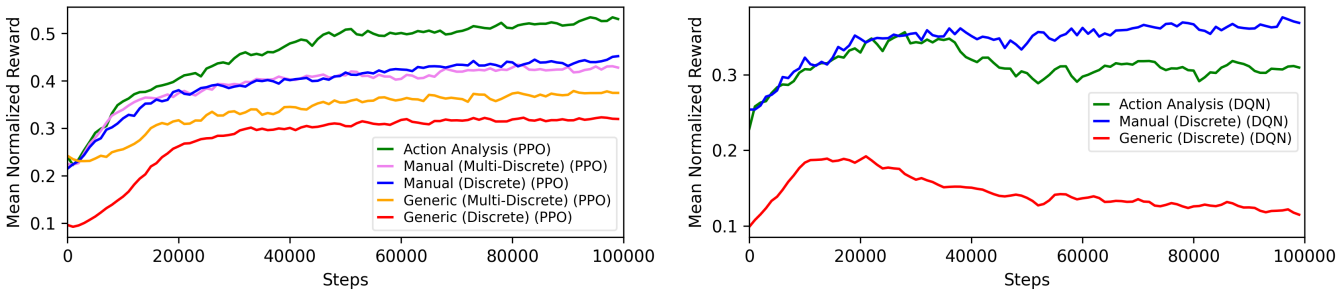
C. Action Spaces

In the evaluation, we compared several approaches to defining the game action spaces:

1) **Action Analysis**: This is a discrete action space automatically determined by our action analysis, where each action corresponds to an execution path through the input-handling code as described in Section III-B. An action mask is generated to indicate the validity of each action as described in Section III-C.
2) **Manual (Multi-Discrete)**: We manually specified the game's action space based on an inspection of the game's code and gameplay. This represents an ideal scenario where the developer has invested effort into implementing an interface. We specify a multi-discrete action space, discretizing any continuous actions (axis/-mouse) to cover all the primary cases. This approach is consistent with the guidelines given in prior work on action space shaping [11]. As is typical for reinforcement learning environments, each action is a basic action taking place over a single step (e.g. move left/right, jump). We also manually specified action masks for actions only available under certain conditions.
3) **Manual (Discrete)**: Some works intentionally use discrete action spaces [3], and some reinforcement learning algorithms only work with discrete action spaces (e.g. DQN). Therefore, we also compare against a discrete variant of the manual specification by automatically converting the multi-discrete space to a discrete one consisting of all combinations of actions.
4) **Generic (Multi-Discrete)**: We defined a multi-discrete action space consisting of common keyboard and mouse actions that is suitable for interacting with most games. This represents a generic action model that covers the entire input device, such as that seen in work on general computer control [16] or Atari game playing [14]. It does not require engineering effort per game, but it is large and imprecise. The first part of the space consists of 101 discrete spaces of size 2 for keyboard keys (excluding some uncommon keys, such as the Print key), each representing the held or released state of the key. Next

TABLE II
ACTION SPACE SIZES FOR OUR AUTOMATED ACTION ANALYSIS AND THE MANUAL SPECIFICATION. COMPLETE ACTION SPACE SIZES ARE ON TOP, AND AVERAGE VALID ACTION COUNTS DURING GAMEPLAY ARE BELOW.

Game	Action Analysis	Manual (MD)	Manual (D)
G1	157 10.54	3, 2, 2 2.85, 1.93, 1.18	12 7
G2	39 27.66	3, 3 3, 3	9 9
G3	17 10.98	3, 3 2.56, 2.13	9 5.96
G4	5 4.73	3, 2 3, 1.54	6 5.05
G5	14 9.30	3, 3, 2, 3 2.77, 2.77, 1.04, 1	54 7.49
G6	11 9	3, 3, 2 3, 3, 1	18 9
G7	46 21.54	3, 3, 2, 3, 34 3, 3, 1.55, 1, 17.34	1836 203.18



(a) Training with Proximal Policy Optimization (PPO)

(b) Training with Deep Q Learning (DQN)

Fig. 2. Mean episode reward over time across all the games. Rewards for each game are normalized to [0, 1] prior to being averaged.

TABLE III
FINAL MEAN EPISODE REWARDS PER GAME FOR PPO. STATISTICALLY SIGNIFICANT DIFFERENCES ARE BOLDED. WHERE THE ACTION ANALYSIS HAD SIGNIFICANTLY BETTER PERFORMANCE, ($\ll$) INDICATES $p < 0.01$ AND ($<$) INDICATES $p < 0.05$.

Game	Action Analysis	Generic (MD)	Generic (D)	Manual (MD)	Manual (D)
G1	15.3 ± 4.28	**5.52 ± 1.48** ($\ll$)	**2.15 ± 0.06** ($\ll$)	**4.57 ± 1.55** ($\ll$)	**4.21 ± 0.6** ($\ll$)
G2	99.4 ± 21.08	**69.9 ± 6.81** ($\ll$)	**75.7 ± 16.18** ($<$)	91.14 ± 28.1	105.25 ± 35.16
G3	183.41 ± 33.2	**122.56 ± 17.39** ($\ll$)	**138.88 ± 18.29** ($\ll$)	**139.01 ± 19.13** ($\ll$)	180.58 ± 29.94
G4	371.2 ± 96.42	**232.45 ± 30.64** ($\ll$)	**229.4 ± 46.03** ($\ll$)	297.25 ± 40.23	335.3 ± 43.26
G5	9.43 ± 0.94	**5.0 ± 1.18** ($\ll$)	**3.5 ± 0.9** ($\ll$)	9.38 ± 1.7	9.29 ± 1.65
G6	26.61 ± 5.26	29.04 ± 3.77	25.24 ± 5.49	20.35 ± 7.17	25.44 ± 3.09
G7	186.84 ± 17.47	167.22 ± 25.42	**133.53 ± 5.6** ($\ll$)	201.28 ± 19.26	177.36 ± 19.41

there are two discrete spaces of size 3 for horizontal (left/center/right) and vertical (bottom/center/top) mouse position. Last is a discrete space of size 2 for the left mouse button state. These actions are always valid.

5) **Generic (Discrete)**: We also defined a discrete variant of the Generic action space. The set of all combinations of the generic multi-discrete space is intractably large, so we instead defined a smaller variant that exercises input events individually. The first action is a no-op action that releases all keyboard and mouse buttons. Next there are 101 actions for holding down a single key, then nine actions for mouse cursor movement within a 3x3 grid. Lastly, there are another nine actions for moving the mouse cursor together with the left mouse button held. Overall this gives 120 actions that are always valid.

In Table II we give the action space sizes and average valid action counts during gameplay (branching factor). For the action analysis we can see that, although the complete sizes may be large, during gameplay the actual number of valid actions is often much smaller.

D. Experiment Results

In Figure 2, we visualize the overall trends with aggregate graphs of mean episode rewards across all games. The rewards were normalized with inter-algorithm normalization [15], where for each game we computed the minimum episode reward r_{min} and maximum episode reward r_{max} obtained by any of the approaches, then rescaled the range of values to $[0, 1]$. In Tables III and IV we present the average and standard deviation of the final mean episode reward for each game across ten repetitions of training. A two-tailed Mann-Whitney U test is performed between the samples from the action analysis and the other approaches being compared against. We do not employ a mathematical multiple-comparison correction, but rather we present the outcome of all the tests conducted and examine the frequency with which significant results were found. We highlight those results where statistical significance was established at either $p < 0.01$ or $p < 0.05$. We expect that frequently occurring outcomes would generalize well.

TABLE IV
FINAL MEAN EPISODE REWARDS PER GAME FOR DQN. WHERE THE ACTION ANALYSIS HAD SIGNIFICANTLY BETTER PERFORMANCE, ($\ll$) INDICATES $p < 0.01$. WHERE IT PERFORMED SIGNIFICANTLY WORSE, ($\gg$) INDICATES $p < 0.01$ AND ($>$) INDICATES $p < 0.05$.

Game	Action Analysis	Generic (D)	Manual (D)
G1	1.59 ± 0.93	**0.05 ± 0.06** ($\ll$)	3.05 ± 1.69
G2	84.69 ± 22.06	**29.69 ± 13.36** ($\ll$)	**181.76 ± 31.55** ($\gg$)
G3	86.84 ± 10.44	**2.07 ± 0.67** ($\ll$)	95.68 ± 7.03
G4	235.6 ± 76.47	**24.95 ± 11.75** ($\ll$)	**296.25 ± 61.1** ($>$)
G5	3.98 ± 1.61	**1.18 ± 0.55** ($\ll$)	3.85 ± 1.81
G6	22.37 ± 12.19	17.11 ± 10.2	27.47 ± 9.28
G7	69.91 ± 22.44	**12.38 ± 9.53** ($\ll$)	**82.39 ± 9.97** ($>$)

1) RQ1: Action Analysis vs Generic: For the vast majority of games, the action analysis gives superior performance to the generic action spaces for both PPO and DQN. In the aggregate graphs of Figure 2 we can see the action analysis achieve superior performance compared to the generic action spaces. In Tables III and IV we see that for the vast majority of games, the action analysis achieved significantly better performance compared to the generic action spaces. Overall, we conclude that the action analysis is a superior option compared to the use of generic input device action spaces, illustrating the value

TABLE V
RESULTS OF THE ABLATION STUDY. "NR" REFERS TO NO-OP REMOVAL AND "IR" REFERS TO INPUT RESETTING.

Game	All Off (PPO)	Only IR (PPO)	Both NR and IR (PPO)	All Off (DQN)	Only IR (DQN)	Both NR and IR (DQN)
G1	15.84 ± 5.05	17.3 ± 2.98	15.3 ± 4.28	1.62 ± 0.9	1.87 ± 1.22	1.59 ± 0.93
G2	87.2 ± 16.07	93.74 ± 11.78	99.4 ± 21.08	84.26 ± 20.19	76.21 ± 19.75	84.69 ± 22.06
G3	134.88 ± 16.77	**180.14 ± 41.22** (>)	**183.41 ± 33.2** (≫)	44.87 ± 7.85	**76.94 ± 12.25** (≫)	**86.84 ± 10.44** (≫)
G4	312.65 ± 62.84	394.1 ± 103.67	371.2 ± 96.42	103.1 ± 62.31	165.15 ± 93.82	**235.6 ± 76.47** (≫)
G5	8.43 ± 1.9	9.53 ± 1.87	9.43 ± 0.94	4.4 ± 1.24	3.55 ± 1.43	3.98 ± 1.61
G6	26.39 ± 6.47	23.97 ± 7.85	26.61 ± 5.26	24.21 ± 4.62	18.84 ± 11.08	22.37 ± 12.19
G7	179.88 ± 25.84	194.2 ± 19.03	186.84 ± 17.47	65.22 ± 15.81	76.01 ± 16.49	69.91 ± 22.44

of having precise action models.

2) RQ2: Action Analysis vs Manual: For PPO, we observe that the action analysis performs on par with or better than the manually specified action spaces. In Figure 2a, we see that on average the action analysis surpasses both manual variants. In Table III, we see that for the vast majority of games there is no significant difference between the action analysis and manual specifications, indicating similar performance. An outlier was the game G1, where the action analysis achieved significantly better performance. We conclude that for the PPO algorithm our approach is effective, performing on par with or better than typical manual specifications. Given the widespread use of PPO and that our approach is fully automated, this finding highlights the strengths of our program analysis solution.

For DQN, we find that for some games, the manual specification has an advantage over the action analysis. In Figure 2b we see the Manual action space achieve better performance on average than the action analysis, and in Table IV we see several games where Manual achieved significantly better performance (G2, G4, and G7). An observation here is that the game where Manual had the biggest advantage (G2) was the one where the average number of Manual actions was about three times smaller than that of the action analysis (see Table II). However, evidently action count is also not the only factor, since for G4 and G7 the action analysis had a similar or smaller size but still performed worse. Nevertheless, for over half the games (4/7) we also observe that there is no significant difference between the action analysis and manual specification performance, indicating similar performance. We conclude that for DQN, the action analysis gives performance competitive with manual specification, but some games may still benefit from hand-crafted action spaces.

3) RQ3: Action Analysis Treatments: In this work we proposed two treatments to the action analysis to make it more amenable for reinforcement learning. First, we proposed the redundant no-op removal in Section III-B, which identifies and combines all no-op actions into a single one. Across the seven games, we found that this treatment reduces the action space size by an average of 19.7% and the number of valid actions per state by 11.3%. Secondly, in Section III-D, we proposed identifying and resetting all inputs that are not present in the path condition of the action being performed to address the issue of unintended sticky actions. We examined the impact of these two treatments by conducting an ablation study, repeating the evaluation procedure with variants of the analysis having some or all of these treatments disabled.

Table V gives the final mean episode rewards and statistical tests for the variants. The no-op removal alone did not lead to any significant difference in performance, so its column is omitted. However, we can see that combining it with input resetting gave the strongest improvement in performance. Although these improvements only manifest themselves in a few games, we observe only positive changes among the significant differences in performance, so we conclude that including both analysis treatments is valuable when deploying reinforcement learning with the action analysis.

V. RELATED WORK

A few prior works have applied program analysis techniques for determining game action models. Bethea et al. [20] proposed a program analysis technique for cheat detection. Their approach also uses symbolic execution, but they generate different kinds of constraints that are not suitable for determining the validity and device inputs of actions as we do. Feldmeier et al. [22] proposed a test generation technique for Scratch games that evolves neural networks. In their work, they perform pattern matching on syntax trees to identify Scratch blocks that listen for user input events in order to automatically specify the network outputs. However, such pattern matching is insufficient for commercial game engines such as Unity, where action validity conditions and input API parameters can be arbitrary dynamic expressions expressed in code, necessitating the symbolic execution approach that we propose. Nevertheless, their work illustrates another application of automated action space analysis for video games.

Volokh et al. [18], [19] proposed the symbolic execution approach that we build on in this work, however their focus was on maximizing the state and code coverage of automatic exploration approaches for video games, which does not necessarily correspond to better game playing performance. In our work we proposed novel techniques and transformations for determining action spaces specifically with the aim of maximizing the game playing performance (episodic reward) of reinforcement learning agents.

More broadly, the research area of Automated Reinforcement Learning (AutoRL) [23] aims to automatically determine effective reinforcement learning configurations, including state and action representations, reward functions, and network architectures. However, existing work in this field typically employs learning or search-based approaches. Our use of

program analysis to identify appropriate action spaces is a novel direction based on software engineering methodologies.

Several works have considered the effect of action space formulation on the performance of reinforcement learning agents. Kanervisto et al. [11] considered different approaches to shrinking and transforming action spaces, such as by removing or discretizing actions. They found that such transformations can be crucial for enabling reinforcement learning agents to learn. Huang et al. [12] demonstrate the importance of invalid action masking as the number of invalid actions increases. Both of these findings motivate the need for precise action models in reinforcement learning.

VI. Conclusion

In this work we have proposed the use of automated program analysis for determining precise action models when deploying reinforcement learning in games. We implemented our technique for the Unity game engine and evaluated its performance across a variety of different existing games with the PPO and DQN reinforcement learning algorithms. We found that agents using the resulting action spaces achieved significantly better performance than those using generic action spaces, and perform on par with the ideal scenario of manually specified ones. We conclude that such an approach is practical and effective for automatically specifying action spaces. Our contribution takes a step towards simplifying the deployment of reinforcement learning in existing games, an increasingly important endeavor as it finds new applications for game development, testing, and analysis.

Acknowledgments

The authors acknowledge the Center for Advanced Research Computing (CARC) at the University of Southern California for providing computing resources that have contributed to the research results reported within this publication.

References

[1] J. Pfau, J. D. Smeddinck, and R. Malaka, "Automated game testing with icarus: Intelligent completion of adventure riddles via unsupervised solving," in *Extended Abstracts Publication of the Annual Symposium on Computer-Human Interaction in Play*. New York, NY, USA: Association for Computing Machinery, 2017, p. 153–164.

[2] Y. Zheng, X. Xie, T. Su, L. Ma, J. Hao, Z. Meng, Y. Liu, R. Shen, Y. Chen, and C. Fan, "Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 772–784.

[3] G. Liu, M. Cai, L. Zhao, T. Qin, A. Brown, J. Bischoff, and T.-Y. Liu, "Inspector: Pixel-based automated game testing via exploration, detection, and investigation," in *Proceedings of the IEEE Conference on Games (CoG)*, 2022, pp. 237–244.

[4] C. Gordillo, J. Bergdahl, K. Tollmar, and L. Gisslén, "Improving playtesting coverage via curiosity driven reinforcement learning agents," in *Proceedings of the IEEE Conference on Games (CoG)*, 2021, pp. 1–8.

[5] J. Bergdahl, C. Gordillo, K. Tollmar, and L. Gisslén, "Augmenting automated game testing with deep reinforcement learning," in *Proceedings of the IEEE Conference on Games (CoG)*, 2020, pp. 600–603.

[6] J. Gillberg, J. Bergdahl, A. Sestini, A. Eakins, and L. Gisslén, "Technical challenges of deploying reinforcement learning agents for game testing in aaa games," in *Proceedings of the IEEE Conference on Games (CoG)*, 2023, pp. 1–8.

[7] V. Mastain and F. Petrillo, "A behavior-driven development and reinforcement learning approach for videogame automated testing," in *Proceedings of the ACM/IEEE 8th International Workshop on Games and Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2024, p. 1–8.

[8] O. Keehl and A. M. Smith, "Monster carlo 2: Integrating learning and tree search for machine playtesting," in *Proceedings of the IEEE Conference on Games (CoG)*, 2019, pp. 1–8.

[9] J. Jara Gonzalez, S. Cooper, and M. Guzdial, "Mechanic maker 2.0: Reinforcement learning for evaluating generated rules," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 19, no. 1, Oct. 2023, pp. 266–275.

[10] R. Tufano, S. Scalabrino, L. Pascarella, E. Aghajani, R. Oliveto, and G. Bavota, "Using reinforcement learning for load testing of video games," in *Proceedings of the 44th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2022, p. 2303–2314.

[11] A. Kanervisto, C. Scheller, and V. Hautamäki, "Action space shaping in deep reinforcement learning," in *Proceedings of the IEEE Conference on Games (CoG)*, 2020, pp. 479–486.

[12] S. Huang and S. Ontañón, "A closer look at invalid action masking in policy gradient algorithms," in *Proceedings of the Thirty-Fifth International Florida Artificial Intelligence Research Society Conference, Hutchinson Island, Jensen Beach, Florida, USA, May 15-18, 2022*, R. Barták, F. Keshtkar, and M. Franklin, Eds., 2022.

[13] M. Towers, A. Kwiatkowski, J. Terry, J. U. Balis, G. D. Cola, T. Deleu, M. Goulão, A. Kallinteris, M. Krimmel, A. KG, R. Perez-Vicente, A. Pierré, S. Schulhoff, J. J. Tai, H. Tan, and O. G. Younis, "Gymnasium: A standard interface for reinforcement learning environments," 2024.

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, 2015.

[15] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *J. Artif. Int. Res.*, vol. 47, no. 1, p. 253–279, May 2013.

[16] P. C. Humphreys, D. Raposo, T. Pohlen, G. Thornton, R. Chhaparia, A. Muldal, J. Abramson, P. Georgiev, A. Santoro, and T. Lillicrap, "A data-driven approach for learning to control computers," in *Proceedings of the 39th International Conference on Machine Learning*, vol. 162. PMLR, 17–23 Jul 2022, pp. 9466–9482.

[17] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017.

[18] S. Volokh and W. G. Halfond, "Automatically defining game action spaces for exploration using program analysis," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 19, no. 1, Oct. 2023, pp. 145–154.

[19] ——, "Static analysis for automated identification of valid game actions during exploration," in *Proceedings of the 17th International Conference on the Foundations of Digital Games*. New York, NY, USA: Association for Computing Machinery, 2022.

[20] D. Bethea, R. A. Cochran, and M. K. Reiter, "Server-side verification of client behavior in online games," *ACM Trans. Inf. Syst. Secur.*, vol. 14, no. 4, Dec. 2008.

[21] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in *Proceedings of the 33rd International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2011, p. 1066–1071.

[22] P. Feldmeier and G. Fraser, "Neuroevolution-based generation of tests and oracles for games," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2023.

[23] J. Parker-Holder, R. Rajan, X. Song, A. Biedenkapp, Y. Miao, T. Eimer, B. Zhang, V. Nguyen, R. Calandra, A. Faust, F. Hutter, and M. Lindauer, "Automated reinforcement learning (autorl): A survey and open problems," *J. Artif. Int. Res.*, vol. 74, Sep. 2022.