Proceedings of the Nineteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2023)

Automatically Defining Game Action Spaces for Exploration Using Program Analysis

Sasha Volokh, William G.J. Halfond

University of Southern California
volokh@usc.edu, halfond@usc.edu

Abstract

The capability to automatically explore different possible game states and functionality is valuable for the automated test and analysis of computer games. However, automatic exploration requires an exploration agent to be capable of determining and performing the possible actions in game states, for which a model is typically unavailable in games built with traditional game engines. Therefore, existing work on automatic exploration typically either manually defines a game's action space or imprecisely guesses the possible actions. In this paper we propose a program analysis technique compatible with traditional game engines, which automatically analyzes the user input handling logic present in a game to determine a discrete action space corresponding to the possible user inputs, along with the conditions under which the actions are valid, and the relevant user inputs to simulate on the game to perform a chosen action. We implemented a prototype of our approach capable of producing the action spaces of Gym environments for Unity games, then evaluated the exploration performance enabled by our technique for random exploration and exploration via curiosity-driven reinforcement learning agents. Our results show that for most games, our analysis enables exploration performance that matches or exceeds that of manually engineered action spaces, and the analysis is fast enough for real time game play.

1 Introduction

In automatic exploration for games, an exploration agent automatically exercises different game mechanics with the aim of covering various game states or functionalities. This technique has been used for game testing, where different states are explored to reveal potential bugs (Zheng et al. 2019; Gordillo et al. 2021; Liu et al. 2022), for amplifying the coverage of human tester data (Chang, Aytemiz, and Smith 2019), and for building a database of searchable moments (Zhan, Aytemiz, and Smith 2019).

Performing automatic exploration requires the capability for an agent to determine the set of valid actions in its current game state and choose an action to perform, simulating relevant in-game events. A key challenge is that these capabilities are not provided by typical game engines.

Existing work has taken various approaches to determining the game actions. The most common approach is for developers to manually engineer an interface (Zheng et al. 2019; Juliani et al. 2020; Liu et al. 2022) or model (Iftikhar et al. 2015; Schaul 2013), however these approaches require additional developer effort. Another family of techniques guesses the possible game inputs based on the platform or input device (Bellemare et al. 2013; Zhan, Aytemiz, and Smith 2019; Zhang et al. 2018), however this approach can exhibit worse exploration performance due to many actions with invalid inputs having no effect. Finally, another class of techniques uses program analysis to analyze the game code to determine possible user actions (Bethea, Cochran, and Reiter 2011; Volokh and Halfond 2022), however existing work is limited in the types of games and actions supported, and has issues scaling to larger games.

In this paper we propose an approach for automatically determining the possible actions in game states, along with the set of relevant user inputs for each action. Our approach is based on a program analysis of the user input handling logic present in game code, and can scale to larger games by automatically excluding code irrelevant to the analysis. We evaluated the performance of automatic exploration enabled by our technique for 16 Unity games, finding that exploration with the actions determined by our analysis matches or exceeds the performance achieved with the ideal case of manually defined action spaces, and the analysis is fast enough for real time game play.

2 Related Work

Several prior works using automatic exploration invest human effort into modeling the game for the exploration agents. For example, work on reinforcement learning based exploration strategies manually defined the player's possible actions for the agents (Zheng et al. 2019; Gordillo et al. 2021; Liu et al. 2022). Other approaches have manually constructed UML state machine models to generate test cases (Iftikhar et al. 2015; Schaefer, Do, and Slator 2013). In contrast to these prior works, we aim to automatically determine the actions. Moreover, we are not aiming to build a full model of the game, but only to determine the possible actions for the agent's current state.

Another approach taken to determining actions is to guess the possible user inputs based on the platform or input devices. For example, in Atari game playing (Bellemare et al. 2013), agents may assume that all 18 combinations of input

are possible for all games. Zhang et al. (2018) developed an exploration approach for the Super Nintendo platform that utilizes all 4,096 possible combinations of controller buttons. While such an approach does not require additional developer effort per game, it is known to often produce invalid combinations of input with no effect, therefore impacting exploration performance. In our work we aim to provide a precise set of actions based on an analysis of the game code, giving consideration to the game state conditions under which certain input combinations are valid as well.

Some prior work has sampled actions from human demonstrations of game play (Zhan, Aytemiz, and Smith 2019; Chang, Aytemiz, and Smith 2019). In our work we do not expect the presence of human demonstrations, which can be challenging to obtain and can be invalidated upon changes to the game code during development. Our approach instead utilizes the game code to determine the actions.

Bethea, Cochran, and Reiter (2011) developed an approach to detect cheating in online games that is also based on program analysis of the game code. Their approach analyzes the game loop to determine valid user actions, then checks whether a sequence of client packets corresponds to valid user inputs. This approach, like ours, determines valid user inputs, and also uses a similar program analysis approach based on symbolic execution, however is ultimately applied toward a different goal of cheat detection rather than automatic exploration. Therefore it does not address other necessary components we discuss, such as the mechanism for determining valid actions during game play and simulating inputs. It also only considers keyboard inputs, whereas our approach is capable of handling keyboard, joystick, and mouse inputs. To scale to larger games, their work also relies on a time-consuming manual trimming of game code, whereas our approach can scale automatically with an automated approach to irrelevant code exclusion.

Volokh and Halfond (2022) also developed a program analysis approach to determining game actions and considered its application towards automatic exploration. Their analysis is also based on symbolic execution, however does not support mouse actions (only keyboard and joystick actions) and imprecisely models inputs checked over multiple frames, both of which we address in our approach. Their approach also does not scale well to larger games, which we handle in our approach with a technique to automatically exclude irrelevant code.

3 Methodology

The goal of our approach is to determine the possible actions in game states and the relevant user inputs associated with each action. To accomplish this we propose a program analysis based approach. First, in order to achieve scalability to larger games, our analysis identifies the user input handling logic present in the code and excludes from consideration all other code that is irrelevant to handling user inputs (Section 3.1). Then, it analyzes the execution paths through the user input handling code to determine the conditions on user input and game state under which certain behaviors will be observed (Section 3.2). Finally, the results of this analysis are mapped into a discrete action space, for which the agent can determine the valid actions and simulate a chosen action (Section 3.3).

Listing 1: Example of an update function from a game component in a Unity game that is controllable by the user.

```
1  void Update() {
2    if (Input.GetAxis("Horizontal") > t)
3      Move(Vector2.right);
4    else if (Input.GetAxis("Horizontal") < -t)
5      Move(Vector2.left);
6    if (jumpEnabled && IsOnGround()) {
7      bool jump = Input.GetButton(jumpButton);
8      if (jump) {
9        var rb = GetComponent<Rigidbody2D>();
10       rb.AddForce(Vector2.up);
11     }
12   }
13   foreach (Enemy e in NearbyEnemies())
14     e.Alert();
15 }
```

Listing 2: Computed program slice of Listing 1 containing only statements relevant to the handling of user inputs.

```
16 void Update() {
17   if (Input.GetAxis("Horizontal") > t) { }
18   else if (Input.GetAxis("Horizontal") < -t) { }
19   if (jumpEnabled && IsOnGround()) {
20     bool jump = Input.GetButton(jumpButton);
21     if (jump) { }
22   }
23 }
```

3.1 Identifying Input-Handling Logic

Our analysis of game actions targets code that handles user inputs. Typically, only a small portion of the code is responsible for this, since many of the game components are entirely dedicated to defining the game logic and contain no input-handling code. As we show in the evaluation, applying the analysis to the entirety of the code can be prohibitively time-consuming and potentially lead to failure due to timeout or the presence of code difficult for the analysis to handle. Therefore, the first step of our analysis is to determine ahead of time those parts that should be analyzed, excluding from consideration any code that can be determined to be irrelevant to the handling of user inputs.

In traditional game engines, user actions are defined by having parts of the game logic check the state of the user's input device and modifying the behavior of the game's objects accordingly. An example of such input-handling logic for a Unity game can be found in Listing 1. At line 2 the `Input.GetAxis` method returns either a negative, zero, or positive float value, which could, for example, correspond to the position of a joystick on a game controller. At line 7 the `Input.GetButton` method returns a boolean value indicating whether the target button is currently pressed. We can see that within this function, there are program statements irrelevant to the handling of user inputs, such as the loop over game enemies at lines 13-14. Our analysis ex-

Listing 3: Example where user actions depend on the state of input across multiple frames.

```
24 void Update() {
25   if (Input.GetButtonDown("Reload"))
26     ammo = 5;
27   if (ammo > 0 && Input.GetButtonUp("Shoot")) {
28     FireMissile();
29     ammo -= 1;
30   }
31 }
```

Listing 4: Example where axis and mouse position values would be discretized by our approach.

```
32 void Update() {
33   float h = Input.GetAxis("Horizontal");
34   transform.position.x += h * Time.deltaTime;
35   var pt = ScreenToWorldPt(Input.mousePosition);
36   var hit = Raycast(pt);
37   if (hit.collider != null) {
38     Highlight(hit.collider);
39   }
40 }
```

cludes such irrelevant statements from consideration. Prior works either could only exclude entire methods (Volokh and Halfond 2022) or expected manual trimming of irrelevant code (Bethea, Cochran, and Reiter 2011) in order to make the analysis scalable to larger code bases.

Our analysis performs two stages of filtering, first on the method level and then on the program statement level. A key challenge is to ensure that the exclusion of irrelevant code from analysis is done in a safe way, to ensure that all code related to user input handling is considered. Therefore, both approaches to filtering are safe over-approximations. On the method level, the analysis first collects an initial set of known methods that the game engine calls directly to update an object's state, such as `Update()` in the Unity game engine. A call graph is then constructed for the program, with nodes being methods and edges being directed edges from each caller to callee. This initial set is then filtered to only include those update methods that have a path in the call graph from the method's node to the node of an input API (such as those defined by the `Input` namespace in Unity). This gives us a set of update methods that can possibly lead to an input API invocation. Within these methods, our analysis excludes specific program statements that are irrelevant to the handling of user inputs by applying interprocedural program slicing (Horwitz, Reps, and Binkley 1990), which computes a slice of a program where the behavior of a set of desired statements is preserved. In our case, the desired statements are any that either invoke a user input API, or have a data dependency on input API results (such as a branch conditioned on user input). To perform this slicing, our analysis constructs a graph of all the control and data dependencies in the program. The sliced program is then all statements which are transitively control or data dependent on the desired statements.

The resulting sliced program is a reduced version of the original program that only includes those statements relevant to the handling of user inputs. For example, the computed slice of Listing 1 is given in Listing 2. As we will show in the evaluation, the exclusion of the code irrelevant to user input handling enables the analysis to scale to more games.

3.2 Action Analysis

Having identified the input handling code, the next objective is to analyze it to determine the possible combinations of user inputs. There are numerous challenges to such an analysis, which we address in our approach. For one, in general the analysis needs to consider not only individual inputs, but also *combinations* of inputs that result in different behaviors. For example, in Listing 1 it is possible for the player to jump and move right on the same frame if the horizontal axis at line 2 is positive and the jump button at line 7 is pressed. Secondly, some actions may only be valid under specific conditions. For example, the player's jump functionality defined at lines 7-11 is only available if `jumpEnabled` and `IsOnGround()` are true. Therefore such an analysis needs to determine the conditions under which input combinations are valid. Considerations also need to be made for certain input types. Input APIs depending on the state of input over multiple frames, as in Listing 3, need to be modelled accurately to correctly determine the user inputs and action validity. Some inputs, such as axis and mouse position, generate floating point values that can be used in ways that are difficult to analyze such as in Listing 4, requiring approximations. Prior work does not handle mouse inputs (Bethea, Cochran, and Reiter 2011; Volokh and Halfond 2022), and models multi-frame inputs independently of the regular inputs (Volokh and Halfond 2022), which can lead to incorrect results for user inputs and action validity.

On a high level, our analysis works by analyzing the different possible execution paths through the user input handling code. The key idea is that different paths represent the different possible behaviors the game may exhibit depending on the user input, an insight made by prior work as well (Bethea, Cochran, and Reiter 2011; Volokh and Halfond 2022). As in these prior works, we use the technique of *symbolic execution* (Clarke 1976) in order to execute the game code with symbolic representations of the user input. Whenever the execution encounters a call to an input API, it produces a symbolic user input value. If the execution encounters a global variable (such as `jumpEnabled` in Listing 2), a symbolic game state value is produced for it as well. When the symbolic execution reaches a branch point conditioned on symbolic values, the execution forks into different states, each maintaining a *path condition* comprised of conditions on the symbolic user input and game state. As an example, one of the path conditions produced by running symbolic execution on Listing 2 will be:

$$\texttt{Input.GetAxis("Horizontal")} > t \;\wedge\; \texttt{jumpEnabled} \;\wedge\; \texttt{IsOnGround()} \;\wedge\; \texttt{Input.GetButton(jumpButton)} \quad (1)$$

This path condition corresponds to an execution path

where the player moves right and jumps on the same frame. It consists of conditions on the symbolic user input variables `Input.GetAxis("Horizontal")` and `Input.GetButton(jumpButton)`, and conditions on symbolic game state variables `t`, `jumpEnabled` and `IsOnGround()`. Each of these path conditions gives the conditions under which a particular execution path is taken (i.e. a certain game behavior is observed).

The output of the analysis is a set of path conditions for execution paths through the user input handling code. These conditions are sufficient to determine the different possible combinations of user input for a given game state, since their solution gives the inputs needed to take a desired execution path. Because these path conditions include the game state variables, they can also be used to test the validity of actions under certain game states (this becomes equivalent to checking the satisfiability of path conditions). In the remainder of this section we address the handling of specific types of inputs.

Axis and Mouse Movement Actions Axis inputs (such as those returned by `Input.GetAxis`) and mouse position input (such as `Input.mousePosition`) are core input types used in defining user actions in games. Axis inputs can be used to, for example, read the position of a joystick and are commonly used for controlling player characters. The mouse position refers to the position of the mouse cursor on the screen and is commonly checked in the game logic of games utilizing the mouse, so is therefore crucial to handle correctly for mouse action support.

A key challenge with these types of inputs is that, unlike key/button APIs that return booleans typically used to control a branch, these inputs are represented by floating point values. These values could directly affect the game state without a branch taking place, as demonstrated in Listing 4 at lines 33-34, which would cause them to not appear in the path condition during symbolic execution. Even if used in a branch, the condition itself could still be too complex to relate the branch to the original input API invocation. For example, at lines 35-37 of Listing 4, a ray-cast is performed from the mouse cursor. In this operation there are external functions for projection and ray-casting that may not be symbolically executable, thus losing information about how the mouse position at line 35 relates to the branch at line 37.

In order to address these issues, our analysis discretizes axis and mouse position input values when encountering a method call to such input APIs. When the symbolic execution encounters a call to an axis input API such as `Input.GetAxis`, it forks into three execution states that include assumptions in the path condition for the general cases of interest: in the first state the axis value is assumed to be negative, in the second it is zero, and in the third it is positive. For mouse position APIs such as `Input.mousePosition`, the execution is forked into a grid of mouse positions on the screen such that in each state the mouse position's x and y values are bounded within a cell of the grid (in our experiments we used a grid size of 4x4). With this approach, even if an input value cannot be related to a branch, the resulting action space will still cover a diverse range of behavior as an approximation.

Multi-Frame Input APIs The result of some input APIs depends on both the current and *previous* frame's input state. These cases must be modeled accurately to ensure the agent correctly interacts with the game. For example, in Listing 3, the method `Input.GetButtonDown` at line 25 will only return true if the button's state was released on the previous frame and pressed on the current frame. Conversely, the method `Input.GetButtonUp` at line 27 will only return true if the button's state was pressed on the previous frame, then released on the current frame.

To model these cases accurately, the analysis expands these input APIs into an atomic set of input APIs, treating the previous input state as part of the game state. For example, `Input.GetButtonUp("Shoot")` would be expanded into $\texttt{Input}_{\texttt{prev}}\texttt{.GetButton("Shoot")} \wedge \texttt{!Input.GetButton("Shoot")}$. This same expansion is done for APIs such as `GetKeyDown`, `GetKeyUp`, `GetMouseButtonDown`, or `GetMouseButtonUp`.

3.3 Defining Action Space

With the execution path information from the analysis, an action space for the game can now be automatically defined for the agent to interact with, where each action is associated with an execution path and its path condition. The path condition information can then be used to determine the valid actions for a given game state, and to determine the relevant device inputs to simulate on the game in order to perform a chosen action. The following sections discuss each step.

Mapping Analysis Results Given a set of N path conditions corresponding to execution paths through the input-handling code, a discrete action space can be defined such that there are N actions, each associated with a path condition. This representation is convenient for exploration algorithms that require defining an upper bound on the number of actions in advance (such as the number of outputs of a neural network in deep reinforcement learning). This also simplifies the use of this approach with AI environment frameworks such as Gym (Brockman et al. 2016) that require defining the action space in advance.

In the simplest case, each path condition is associated with a single action. However, an execution path through an update method such as `Update()` is also associated with an instance of a game object. Therefore, it is possible for multiple object instances to be associated with a single path. In this case, statically predicting an upper bound on the number of actions is difficult or impossible, since this would require predicting the maximum number of instances of a type of game object in any game state. To address this issue and remain compatible with techniques that require an upper bound on the number of actions, our prototype implementation associates each execution path with a single action, and if the agent chooses an action associated with multiple object instances, the action is performed with a random object chosen from among these instances.

Determining Valid Actions In general, only a subset of the whole action space may be available in a given game

state. Therefore, during game play the path conditions produced from the analysis are used to inform the agent of the actions that are valid in its current game state, so that it does not choose the invalid ones. While an SMT solver could be used to determine validity, in practice invoking a solver on every path condition is too slow for game play in real time.

Our approach therefore relies on an approximation for quickly testing action validity proposed in previous work (Volokh and Halfond 2022). Every path condition p can be split into conditions that are only on game state variables, p_g, and conditions that include user input variables, p_i, such that $p = p_g \wedge p_i$. It then follows that if p_g is false, then the path condition is unsatisfiable since p must be false. For example, consider the path condition in equation 1 of Section 3.2. In this example, $p_g =$ `jumpEnabled` $\wedge$ `IsOnGround()`, which gives us the conditions under which jumping is available. In order to enable quickly testing the action's validity, our approach is to compile p_g into an expression in code to test the validity of the action (such as a C# function for Unity). Therefore, in addition to the path conditions, one of the outputs of the analysis is also code generated for each of these compiled action validity functions. The agent then tests the compiled p_g for every action to determine which actions are valid in its current game state. More details about this compiling process can be found in Section A.

Performing Chosen Action When the agent chooses an action to perform, the specific concrete inputs need to be determined and simulated. To determine these inputs, the associated path condition is solved with an SMT solver. For example, suppose we wish to determine the inputs to simulate for the path condition in equation 1 of Section 3.2. We would add the concrete values of the game state variables, for example `t` $= 0.2$, `jumpEnabled` $=$ true, `IsOnGround()` $=$ true. The solver can then produce a solution such as:

$$\begin{aligned} &\texttt{Input.GetAxis("Horizontal")} = 0.3 \\ &\texttt{Input.GetButton(jumpButton)} = \text{true} \end{aligned} \tag{2}$$

At this point the parameters to the input APIs are looked up in the game's input configuration to determine the device inputs to simulate (key codes, joysticks, mouse buttons, etc.). Note that in this example the button for jumping is not a string constant, it refers to a global variable `jumpButton`, so for this case the symbolic parameter would first be evaluated under the current game state to determine its string value.

A key technical challenge is simulating the device inputs, since some game engines do not provide this capability. Prior work has simulated inputs on the operating system level (Volokh and Halfond 2022), however such an approach makes it difficult to run multiple games in parallel. Therefore, our approach is to instrument the game's user input APIs such that they are substituted with a version that allows for simulating inputs directly. This not only enables running parallel instances but also simplifies simulating more devices such as game controllers or mouse cursors.

4 Evaluation and Results

We address the following research questions about our approach by running experiments that measure automatic exploration performance for a variety of games:

1. RQ1: How does the exploration performance of agents using our analysis for defining the action space compare to existing approaches?
2. RQ2: What is the run time of the different parts of our analysis (execution path analysis, determining valid actions, performing a chosen action)?
3. RQ3: How is analysis success rate and exploration performance affected by the irrelevant code exclusion and mouse action components of our analysis?

4.1 Experiment Setup

In each of our experiments we ran automatic exploration with a given approach to determining game actions. In this section we detail the metrics used for measuring exploration performance, the two exploration strategies, and the games used in our evaluation.

Exploration Performance Metrics We measured exploration performance with two widely used metrics of *state coverage* and *code coverage*. We defined state coverage as the number of distinct states reached by an agent, as in prior work (Zheng et al. 2019), where it was found that higher state coverage enables finding more bugs. In order to make states countable for state coverage, we used a state abstraction that reads and abstracts the game's scene hierarchy, removing detailed attributes such as coordinates or colors, giving a more compact state representation. For code coverage we measured the percentage of program statements executed (statement coverage).

When aggregating coverage results, we first applied normalization to the state coverage values because they represent the number of distinct states visited and therefore their magnitude differs among games. This is similar to the problem of normalizing game scores when aggregating game playing performance across games (Bellemare et al. 2013). We used inter-algorithm normalization, such that we computed the minimum and maximum state coverage s_{min} and s_{max} achieved by any approach for a given game, then normalized the range $[s_{min}, s_{max}]$ to $[0, 1]$.

Exploration Strategies We evaluated the impact of our approach for two popular exploration strategies. The first exploration strategy we used is random action selection, which has been commonly compared against as a baseline for automatic exploration of games (Liu et al. 2022; Gordillo et al. 2021; Zheng et al. 2019; Zhan, Aytemiz, and Smith 2019) and mobile application testing (Choudhary, Gorla, and Orso 2015). In our evaluation, we used a random exploration policy that chooses uniformly at random from all valid actions in a given game state. The second exploration strategy we used is exploration with curiosity-driven reinforcement learning agents, where agents are given a curiosity reward that rewards the agent for entering novel states. We employed a count-based approach to curiosity based on previous work (Tang et al. 2017; Gordillo et al. 2021), where

the agent is given a reward inverse to the number of times it has visited a state previously. We made the state space countable with the same scene hierarchy abstraction used for measuring state coverage. For the reinforcement learning algorithm, we used deep Q learning (DQN) (Mnih et al. 2015) with the same convolutional neural network architecture and image processing pipeline as in the original work. Because not all actions in our action space may be valid, we additionally employed invalid action masking (Huang and Ontañón 2022) to avoid choosing invalid actions. After the network produces the action values, invalid actions are assigned a very low value such that they are guaranteed not to be chosen.

Games Evaluated The games used in our evaluation are Unity games with source code available. We excluded games if they use online networking, virtual/augmented reality, or are too resource intensive for our experiment setup, which requires running multiple instances of games over a long period of time. We included 5/6 games from GitHub evaluated in previous work (Volokh and Halfond 2022), with one excluded due to being too resource intensive. Because the prior work did not support mouse actions, we introduced 3 additional games from GitHub that use the mouse. We obtained the remainder of our games from student projects in a graduate game development course. Each project is a Unity game developed over the course of a semester. There were 19 projects, our analysis implementation worked for 12/19 projects. Four projects were incompatible due to the use of alternative input APIs (Unity Input System and Unity UI), which are not supported by our prototype. The analysis of two projects failed due to symbolic execution issues, and one game was incompatible with our helper scripts for communicating with agents. Among the 12 projects, four were excluded due to being too resource intensive for our experiment setup.

Overall, we had 8 games from GitHub and 8 student projects used in our evaluation. Section B gives more details about each game and the number of actions. For each game we also defined an initialization script to bring the game through any initial menus to the main game play, and a done condition to determine when the game is over.

4.2 RQ1: Exploration Performance

In this experiment, we measured the performance of exploration agents using the action spaces produced by our analysis. We ran our action analysis on each game and integrated the resulting discrete action space into a Gym (Brockman et al. 2016) environment that the exploration agents interact with. We then ran exploration for each game for 150,000 steps with 3 game instances in parallel, such that each step holds the selected action for 10 frames. The exploration strategy was varied, such that for each game we ran with both exploration strategies of random exploration and curiosity-driven reinforcement learning. During each game's execution, we recorded the state and code coverage achieved during exploration.

To provide a frame of reference for the performance of our approach, we additionally ran the same experiments with

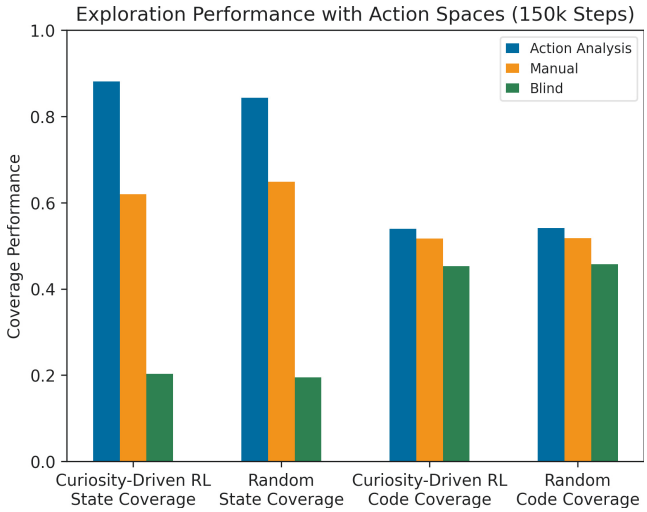

Figure 1: Average exploration performance across all 16 games with the different action space approaches. State coverage values are normalized to $[0, 1]$ prior to being averaged.

two other approaches discussed in the related work for providing the game actions to agents. First, we implemented the ideal case where we manually define each game's possible actions with a discrete action space, as done in prior work (Liu et al. 2022; Zheng et al. 2019), such that the action space covers all the functionality of the game as determined by a manual inspection of the game and its source code. Secondly, we define an action space which samples the common inputs on the platform's input device, as considered in previous work (Bellemare et al. 2013; Zhan, Aytemiz, and Smith 2019). We refer to this as a *blind* action space and define it for the keyboard and mouse in a way suitable for interacting with most games: we have an action for pressing down and an action for releasing 66 commonly used keys on the keyboard and the three mouse buttons, as well as 16 actions for moving the mouse cursor within cells on a 4x4 grid on the screen. For some games there are undesirable actions that we blacklist for all approaches, such as pausing/quitting/restarting the game.

Results The average coverage results achieved for the 16 games with the different action space approaches are shown in Figure 1. We can see that on average, both exploration strategies achieved the highest state and code coverage when using the action spaces produced by our analysis, with the manually engineered action spaces giving second-best performance and the blind sampling approach giving the lowest.

Exploration coverage results per game are given in Section C. We found that compared to the blind sampling approach, our approach consistently gave higher state coverage, and either the same or higher code coverage. We then compared our approach against the manually engineered action spaces. We found that with the random exploration strategy, our approach matched or exceeded state coverage in 12/16 games and code coverage in 13/16 games. With the reinforcement learning based strategy, our approach matched

or exceeded state coverage in 14/16 games and code coverage in 15/16 games. We found there were two games for which the manually engineered action spaces gave consistently better exploration performance than our analysis, and two others where this was only the case with the random exploration strategy. Investigating these cases revealed that for these games, the manual action space had a greater bias towards actions that could lead to entering new states. For example, it was more likely for the agent to randomly perform an action that created a new game object (such as firing a bullet or placing a puzzle piece). This resulted in a new state according to our state definition, and could also lead to more code coverage (for example due to defeating more enemies due to the presence of more bullets).

Discussion Overall we see these results as positive for our approach. For the majority of games, agents using the actions from our analysis achieved exploration performance that matches or exceeds that of the ideal case of manually engineered action spaces, on average achieving better performance. The manual action spaces require developer effort, whereas our analysis is fully automated.

4.3 RQ2: Run Time

We measured the run time of the different parts of our approach as the exploration experiments were running. The average time taken for the execution path analysis, which was only performed once prior to game play, was 29.7 seconds (minimum 2.1 seconds, maximum 110.6 seconds). During game play, the total time spent determining and performing actions each step was on average 23.75 ms (minimum 18.11 ms, maximum 42.64 ms). The times per game can be found in Section C. The slowest part during game play is performing the actions, which is due to the invocation to the SMT solver. Nevertheless, the average time taken for an action step is still fast enough to observe game exploration in real time. It is also on average within the 40 ms action selection time limit used in some game playing competitions (Perez-Liebana et al. 2015).

4.4 RQ3: Impact of Slicing and Mouse Support

In this experiment we examined the impact of two key components of our action analysis: the automatic irrelevant code exclusion via program slicing and the mouse action support. We ran the same exploration experiments as for RQ1 with variants of the analysis having these components enabled or disabled.

Results Figure 2 shows the average exploration performance for each of these variants. Note that the configuration with both program slicing and mouse support disabled is a close approximation of previous work in this area by Volokh and Halfond (2022). We observed that without program slicing, the analysis would fail for three of the games due to non-termination (30 minute time-out) and fail for one of the games due to a symbolic execution failure encountered with code normally excluded by the program slicing. Therefore our aggregate plot does not include these four games in the comparison. For the 6 games in our evaluation with mouse actions, we observed that support for mouse actions gave

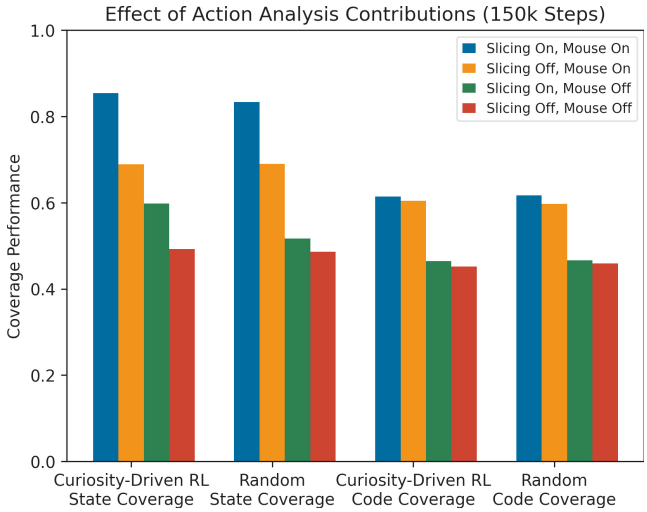

Figure 2: Exploration performance with the different action analysis variants having the program slicing and mouse action components enabled or disabled.

improved state and code coverage for 4/6 games, with an especially large increase for 3/6 games which exclusively relied on mouse inputs. The remaining 2/6 games, where non-mouse inputs were sufficient for many of the game functions, had approximately the same exploration performance.

Discussion Our results show that both the mouse action support and program slicing components improved the average exploration performance, with the best performance achieved with both enabled. Besides exploration performance, with program slicing enabled we also observe a higher analysis success rate (without slicing, the analysis did not work for four games). Although we did find that with program slicing the analysis time takes longer (on average about three times as long) due to the construction and processing of the dependence graphs, we believe the improved success rate and exploration performance justifies this cost.

5 Discussion and Future Work

Our evaluation shows the capability of our automated analysis to match or exceed the performance of the ideal case of manually engineered action spaces for the majority of games in our data set. While the manual annotation of the actions does aim to cover all the user functionality of the game, a key advantage of the action analysis is its capability to exhaustively consider all possible execution paths through the user input handling code, therefore often identifying more combinations of valid inputs than the human annotation. For example, for the platformers in our evaluation, the action analysis determines all possible combinations of functionalities such as movement, jumping, dashing, or firing, which enables the agents to exhibit more complex behaviors and therefore cover more states.

While in this paper our implementation focuses on the Unity game engine, conceptually the symbolic path analysis and code exclusion of our approach could apply to any

game engine with a standard input API checked within the game loop. For instance, the related work on cheat detection by Bethea, Cochran, and Reiter (2011) also applied symbolic execution to the game loop for determining valid user actions of games written in C. Our approach does leverage several non-trivial analyses. For one, our analysis uses data and control dependency analysis to determine relevant statements and perform the program slicing, for which we rely on safe over-approximations to ensure all relevant code is included. Secondly, implementing a symbolic execution engine can be a difficult engineering challenge due to the many types of program instructions and values to handle and the exponential growth in the number of program paths (Cadar et al. 2011). Because our analysis only targets code involved in handling user input, this growth is reduced both through the techniques for code exclusion described in Section 3.1 and by avoiding method evaluation when possible, summarizing methods with a symbolic return value if they are not involved in handling user input or are not symbolically executable, such as native game engine or system APIs.

As future work, the development of a variant of the action analysis capable of generating continuous action spaces could be valuable, as these are currently discretized by our approach, thus limiting the reachability of some states. Furthermore, even with our automated approach to identifying valid actions and their relevant device inputs, the exploration of large game state spaces remains difficult. The development of refinements and heuristics for our analysis or its application could be useful towards this end as well.

6 Conclusion

In this paper we have proposed an automated approach for determining actions when conducting automated exploration for games. Our program analysis based approach automatically determines a game's possible actions and relevant user inputs, and can scale to larger games by automatically excluding code irrelevant to the analysis. The evaluation of our approach shows the action spaces determined by our automated approach match or exceed the performance of the ideal case of manually defined ones for most games. With the increasing importance of automated testing and analysis techniques for games, we believe our approach can provide a valuable component to simplify the use of next generation tools based on intelligent agents.

A Path Condition Compiling

In order to provide a mechanism for quickly approximating the validity of game actions in a given state, our analysis generates boolean functions for each action that are then invoked by the agent during game play. After the symbolic execution completes, our analysis iterates over every path condition p, splitting it into conditions that are only on game state variables, p_g, and conditions on user input variables, p_i, such that $p = p_g \wedge p_i$. For example, the path condition in Equation 1 of Section 3.2 would give $p_g =$ `jumpEnabled` $\wedge$ `IsOnGround()`. The analysis then translates these conditions into boolean functions in the target language of the game engine (for example, C# for Unity), which are compiled in together with the game code. An example of the compiled p_g is given in Listing 5. The function takes as input an instance of a game object's component because each Update method is associated with a game object instance. It accesses fields and methods via reflection to address the presence of access protections. This code generation only occurs once prior to game play, and the resulting functions are then invoked repeatedly during game play by the agents to determine action validity. As shown in Table 3, our experimental times show this mechanism is fast enough for real time usage (on average 5 ms, minimum 0.3 ms, maximum 26.2 ms).

Listing 5: Example of a function generated from a path condition for testing action validity.

```
41 bool p_g(MonoBehaviour instance) {
42   return (bool)f_jumpEnabled.GetValue(instance)
43       && (bool)m_IsOnGround.Invoke(instance,
44                                    new object[0]);
45 }
```

B Games Evaluated

Table 1 gives information about the Unity games (sourced from GitHub and student projects) used in the evaluation. The LOC column gives the number of lines of C# source code, excluding third-party dependencies and our analysis helper code. The Inputs columns describe the types of inputs that are present in the games. The Analysis Action Count column gives the number of actions (i.e. execution paths through the user input handling code) determined by the analysis. The Analysis Avg Valid Actions column shows the average number of valid actions across the states encountered by the exploration agents, which gives an estimate of the actual number of actions that agents have to choose from. The Manual columns give these values for the manually engineered action spaces compared against in the evaluation.

C Per Game Results

The exploration results per game measured for RQ1 in Section 4.2 are given in Table 2. The bold cells in this table indicate the action identification approach that achieved the highest coverage with a given exploration strategy. The run time results per game measured for RQ2 in Section 4.3 are shown in Table 3, which gives the average times both for the initial execution path analysis, and the time to determine valid actions and perform them during game play.

Acknowledgments

The authors acknowledge the Center for Advanced Research Computing (CARC) at the University of Southern California (USC) for providing computing resources that have contributed to the research results reported within this publication. The authors acknowledge Scott Easley from the USC Games program for facilitating access to the student projects used in the experiments. This work was partially supported by U.S. National Science Foundation grant no. 2211454.

Identifier	Genre	LOC	Button Inputs	Axis Inputs	Mouse Inputs	Analysis Action Count	Analysis Avg Valid Actions	Manual Action Count	Manual Avg Valid Actions
GH1	Platformer	2,107	✓			432	17	6	5
GH2	Platformer	3,283	✓	✓		140	51	8	7
GH3	Maze	1,902	✓	✓		45	31	7	7
GH4	Puzzle	1,493	✓			600	18	7	5
GH5	Puzzle	699	✓			22	10	6	5
GH6	Physics	133	✓		✓	116	60	19	13
GH7	Tile-Matching	1,290	✓	✓	✓	8	6	6	5
GH8	Physics	96	✓		✓	38	9	19	5
SP1	Platformer	2,241	✓	✓		77	25	13	7
SP2	Platformer	7,670	✓	✓		87	28	16	7
SP3	Maze	4,990	✓	✓		18	10	13	4
SP4	Maze	4,293	✓	✓		164	5	23	4
SP5	Maze	9,694		✓	✓	13	10	9	7
SP6	Platformer	2,240	✓	✓	✓	56	23	31	19
SP7	Shoot 'em up	7,378	✓	✓	✓	219	24	24	10
SP8	Platformer	5,218	✓	✓		2,560	11	19	6

Table 1: Information about the Unity games used in the evaluation. Games whose identifiers start with "GH" are sourced from GitHub, and those with "SP" are sourced from student projects.

Game	State Coverage						Code Coverage					
	Action Analysis		Manual		Blind		Action Analysis		Manual		Blind	
	RL	Rand	RL	Rand	RL	Rand	RL	Rand	RL	Rand	RL	Rand
GH1	**59**	**62**	29	44	6	9	**90.33%**	90.25%	87.11%	**90.41%**	73.35%	85.53%
GH2	**106**	**104**	35	55	6	10	**48.58%**	**50.99%**	42.75%	43.58%	36.18%	38.76%
GH3	**24,993**	**26,455**	21,076	21,357	1,182	902	**70.01%**	**70.01%**	**70.01%**	68.76%	**70.01%**	**70.01%**
GH4	**199**	83	73	**85**	25	15	**91.51%**	**91.51%**	**91.51%**	**91.51%**	81.25%	81.25%
GH5	**15**	**15**	**15**	**15**	12	9	**90.02%**	**90.02%**	**90.02%**	**90.02%**	86.51%	86.51%
GH6	**1,255**	**1,380**	630	287	132	49	**92.68%**	**92.68%**	**92.68%**	**92.68%**	**92.68%**	**92.68%**
GH7	**78**	**101**	77	99	30	40	**82.56%**	**82.56%**	**82.56%**	**82.56%**	**82.56%**	**82.56%**
GH8	**13**	**18**	6	6	1	1	**100.0%**	**100.0%**	77.05%	77.05%	22.95%	22.95%
SP1	**70**	**57**	28	40	11	11	**45.52%**	**45.52%**	41.49%	41.49%	41.72%	41.72%
SP2	2,499	1,228	**3,930**	**2,552**	575	662	**17.9%**	17.87%	**17.9%**	**17.9%**	17.83%	17.87%
SP3	**16**	**17**	12	11	10	10	**16.68%**	**16.68%**	**16.68%**	**16.68%**	15.77%	**16.68%**
SP4	**15**	**18**	10	14	7	7	**15.19%**	**15.19%**	**15.19%**	**15.19%**	12.99%	12.99%
SP5	**184**	184	168	**226**	63	61	**4.32%**	**4.32%**	**4.32%**	**4.32%**	**4.32%**	**4.32%**
SP6	**17**	**18**	14	15	7	7	**34.73%**	**34.73%**	**34.73%**	**34.73%**	29.44%	29.44%
SP7	72,492	76,416	**82,063**	**87,210**	44,818	47,361	27.66%	27.57%	**28.46%**	**27.6%**	27.25%	27.22%
SP8	**57**	**21**	43	15	4	3	**35.71%**	**35.83%**	35.2%	34.64%	29.29%	21.2%

Table 2: Results for exploration coverage with the curiosity-driven reinforcement learning and random exploration strategies. Bold cells represent the highest coverage achieved with a given exploration strategy by any of the three approaches to identifying game actions.

	GH1	GH2	GH3	GH4	GH5	GH6	GH7	GH8	SP1	SP2	SP3	SP4	SP5	SP6	SP7	SP8
Analysis (sec)	15.1	9.8	5.3	37.4	23.6	3.9	2.9	2.4	8.0	58.3	97.5	10.1	2.1	13.2	74.5	110.6
Validity (ms)	12.2	3.9	3.3	17.2	1.5	1.7	0.8	0.3	2.0	2.0	1.3	1.6	2.5	1.5	1.9	26.2
Perform (ms)	17.8	17.5	15.9	16.3	20.6	17.2	17.4	18.5	19.6	16.1	19.4	23.1	19.4	23.9	21.2	16.4

Table 3: Analysis run times per game. The first row gives the time for the execution path analysis (average of 10 runs). The second and third rows give the time to determine valid actions and perform a chosen action during game play (averaged over all agent steps).

References

Bellemare, M. G.; Naddaf, Y.; Veness, J.; and Bowling, M. 2013. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, 47: 253–279.

Bethea, D.; Cochran, R. A.; and Reiter, M. K. 2011. Server-side verification of client behavior in online games. *ACM Transactions on Information and System Security*, 14(4): 1–27.

Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; and Zaremba, W. 2016. OpenAI Gym. arXiv:1606.01540.

Cadar, C.; Godefroid, P.; Khurshid, S.; Păsăreanu, C. S.; Sen, K.; Tillmann, N.; and Visser, W. 2011. Symbolic Execution for Software Testing in Practice: Preliminary Assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, 1066–1071. New York, NY, USA: Association for Computing Machinery. ISBN 9781450304450.

Chang, K.; Aytemiz, B.; and Smith, A. M. 2019. Reveal-more: Amplifying human effort in quality assurance testing using automated exploration. In *2019 IEEE Conference on Games (CoG)*, 1–8. IEEE.

Choudhary, S. R.; Gorla, A.; and Orso, A. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 429–440.

Clarke, L. 1976. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering*, SE-2(3): 215–222.

Gordillo, C.; Bergdahl, J.; Tollmar, K.; and Gisslén, L. 2021. Improving Playtesting Coverage via Curiosity Driven Reinforcement Learning Agents. In *2021 IEEE Conference on Games (CoG)*, 1–8.

Horwitz, S.; Reps, T.; and Binkley, D. 1990. Interprocedural Slicing Using Dependence Graphs. *ACM Trans. Program. Lang. Syst.*, 12(1): 26–60.

Huang, S.; and Ontañón, S. 2022. A Closer Look at Invalid Action Masking in Policy Gradient Algorithms. In Barták, R.; Keshtkar, F.; and Franklin, M., eds., *Proceedings of the Thirty-Fifth International Florida Artificial Intelligence Research Society Conference, FLAIRS 2022, Hutchinson Island, Jensen Beach, Florida, USA, May 15-18, 2022*.

Iftikhar, S.; Iqbal, M. Z.; Khan, M. U.; and Mahmood, W. 2015. An automated model based testing approach for platform games. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 426–435.

Juliani, A.; Berges, V.-P.; Teng, E.; Cohen, A.; Harper, J.; Elion, C.; Goy, C.; Gao, Y.; Henry, H.; Mattar, M.; and Lange, D. 2020. Unity: A General Platform for Intelligent Agents. arXiv:1809.02627.

Liu, G.; Cai, M.; Zhao, L.; Qin, T.; Brown, A.; Bischoff, J.; and Liu, T.-Y. 2022. Inspector: Pixel-Based Automated Game Testing via Exploration, Detection, and Investigation. In *2022 IEEE Conference on Games (CoG)*, 237–244.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M. A.; Fidjeland, A.; Ostrovski, G.; Petersen, S.; Beattie, C.; Sadik, A.; Antonoglou, I.; King, H.; Kumaran, D.; Wierstra, D.; Legg, S.; and Hassabis, D. 2015. Human-level control through deep reinforcement learning. *Nat.*, 518(7540): 529–533.

Perez-Liebana, D.; Samothrakis, S.; Togelius, J.; Schaul, T.; Lucas, S. M.; Couëtoux, A.; Lee, J.; Lim, C.-U.; and Thompson, T. 2015. The 2014 general video game playing competition. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(3): 229–243.

Schaefer, C.; Do, H.; and Slator, B. M. 2013. Crushinator: A framework towards game-independent testing. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 726–729.

Schaul, T. 2013. A video game description language for model-based or interactive learning. In *2013 IEEE Conference on Computational Intelligence in Games (CIG)*, 1–8.

Tang, H.; Houthooft, R.; Foote, D.; Stooke, A.; Chen, X.; Duan, Y.; Schulman, J.; De Turck, F.; and Abbeel, P. 2017. #Exploration: A Study of Count-Based Exploration for Deep Reinforcement Learning. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, 2750–2759. Red Hook, NY, USA: Curran Associates Inc. ISBN 9781510860964.

Volokh, S.; and Halfond, W. G. 2022. Static Analysis for Automated Identification of Valid Game Actions During Exploration. In *Proceedings of the 17th International Conference on the Foundations of Digital Games*, FDG '22, 1–10. Association for Computing Machinery. ISBN 978-1-4503-9795-7.

Zhan, Z.; Aytemiz, B.; and Smith, A. M. 2019. Taking the Scenic Route: Automatic Exploration for Videogames. In *Proceedings of the 2nd Workshop on Knowledge Extraction from Games co-located with 33rd AAAI Conference on Artificial Intelligence, KEG@AAAI 2019, Honolulu, Hawaii, January 27th, 2019*.

Zhang, X.; Zhan, Z.; Holtz, M.; and Smith, A. M. 2018. Crawling, Indexing, and Retrieving Moments in Videogames. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, FDG '18. New York, NY, USA: Association for Computing Machinery. ISBN 9781450365710.

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