On Automating Video Game Regression Testing by Planning and Learning

Tomáš Balyo, G. Michael Youngblood, Filip Dvořák, Lukáš Chrpa, and Roman Barták

Filuta AI, Inc., 199 Water St Fl 34, New York, NY 10038
{tomas, michael, filip, lukas, roman}@filuta.ai

Abstract

In this paper, we propose a method and workflow for automating regression testing of certain video game aspects using automated planning and incremental action model learning techniques. The basic idea is to use detailed game logs and incremental action model learning techniques to maintain a formal model in the planning domain description language (PDDL) of the gameplay mechanics. The workflow enables efficient cooperation of game developers without any experience with PDDL or other formal systems and a person experienced with PDDL modeling but no game development skills. We describe the method and workflow in general and then demonstrate it on a proof-of-concept example — a simple role-playing game provided as one of the tutorial projects in the popular game development engine Unity. This paper presents the first step towards minimizing or even eliminating the need for a modeling expert in the workflow, thus making automated planning accessible to a broader audience.

Introduction

Game testing is a vital yet complex component of video game development, focusing on identifying and resolving bugs, performance issues, and ensuring a high-quality user experience. This multifaceted process involves testing diverse game scenarios, enhancing user interface and playability, and maintaining stability and efficiency. It also includes compliance with standards and regulations, especially for games with online components. Modern game testing combines automated and manual methods, often engaging professional testers and the gaming community, to not only fix issues but also elevate overall player satisfaction and game quality. However, automation of testing methods still has a long way to go, especially for techniques that do not significantly disrupt developer workflow (Politowski, Guéhéneuc, and Petrillo 2022). This paper focuses on a contribution to those automated methods by leveraging planning.

Automated Planning is a less common skill game developers utilize but can be a powerful tool for game testing. One key aspect of that difficulty comes from the authorial burden of writing domain and problem descriptions in PDDL (Planning Domain Definition Language) (Ghallab et al. 1998). Good PDDL modeling is a rare skill that develops over time, so an efficient use of modelers by enabling them to be more effective is another key to adoption. Although there are Knowledge Engineering tools that aim at supporting PDDL modeller effort (Simpson, Kitchin, and McCluskey 2007; Vaquero et al. 2013), the modelling process is still a sort of "black art" (McCluskey, Vaquero, and Vallati 2017).

Our work addresses these critical issues to help game developers utilize planners for generating test scripts driven by PDDL through automation support. We accomplish this by taking the game logs, produced in software development, providing easy-to-implement guidance to augment the logging for action model capture, and performing automated domain synthesis to generate PDDL models easily validated, verified, and adjusted by the game developer. These good PDDL examples also improve the developers' understanding and authorial abilities in PDDL in the flow. Goal templates are then used to formulate domain problems of interest for testing. We demonstrate our approach and process on a simple RPG game in the ubiquitous Unity game engine (Unity Technologies 2023).

Motivation

We will focus on regression testing during game development, i.e., running frequent tests to ensure that recent changes did not introduce unintended breaks and the game still performs as expected. This is a particularly repetitive and dull task for a human tester but crucial for efficient development. It is often the case, that the longer the time difference between the introduction and discovery of the bug, the harder it is to fix.

What can be tested with Planning

The most obvious and important usage is to test whether the game can be completed, i.e., each intended objective can be achieved from the starting state of each scenario. Next, we can generate a large amount of random test scenarios. This approach is beneficial if we can execute and evaluate test scenarios automatically. Furthermore, we can search for dead-ends in the game, i.e., states that can be reached but are impossible to continue from or to win. In modern game design, such states are most undesirable and annoying for the

players. If the PDDL model is detailed enough, one can detect such dead ends by running planners from random reachable states. Last but not least, one can prove that undesired shortcuts are impossible through regular gameplay using optimal planners.

Applicability

The classical planning-based testing approach is most suitable for games with many discreet causal interactions such as role-playing games (RPG), point-and-click adventures, strategy games, visual novels, or puzzle games. It is less suitable for fast-paced action-oriented games such as shooters or driving games. For such games, reinforcement learning-based approaches, that have been studied in recent works, seem to be more suitable (see the Related Work Section of this paper). Overall, we observe that the reinforcement learning and the planning-based approaches can complement each other rather well.

Preliminaries

The planning problem is to find a plan — a sequence of grounded actions that transform the world from an initial state to a goal state, i.e., a state where all goal conditions are satisfied. A planning problem instance consists of a domain definition and a task definition. The domain definition describes possible actions with their preconditions and effects. In STRIPS planning (Fikes and Nilsson 1971), both preconditions and effects are sets of predicates and negated predicates connected by a conjunction. The task definition contains the descriptions of the initial state and the goal conditions.

Planning action model learning (AML) involves synthesizing a domain definition from logs with state sequences. Many algorithms have been published for this task (Wang 1996; Aineto, Celorrio, and Onaindia 2019; Zhuo et al. 2010; Yang, Wu, and Jiang 2007; Juba, Le, and Stern 2021), and a recent overview and new implementations of multiple techniques are provided in the MacQ project (Callanan et al. 2022). For the evaluation in this paper we use our own new action learning method (Balyo et al. 2024).

Related Work

Planning is rooted in search and, as such, has had a long history with games. At the beginning of AI, the focus was on games like chess (Newell, Simon et al. 1972) and checkers (Samuel 1959), which initially relied on the search for solutions. Planning continued to dominate in the 80s and 90s with checkers and chess with Deep Blue (Hsu et al. 1990) and Chinook (Schaeffer et al. 1996). Agent architectures and production systems added value, and soon, planning started to add value in games like bridge (Smith, Nau, and Throop 1998) and the class of Real-Time Strategy (RTS) games (Chung, Buro, and Schaeffer 2005). In (Duarte et al. 2020), they survey the history of planning and learning in games, covering the spectrum as well as diving into the lineage of planning from search, minimax and alpha-beta pruning, hierarchical task networks, and Monte Carlo Tree search, through classical planning, rapidly-exploring random trees, case-based planning, and behavior trees. Most of the work is focused on creating AI-driven opponents (Wurman et al. 2022), which are sometimes used to play both sides for evaluation, AI training, and testing.

Automated testing with AI has been a rising research focus more recently with work that has focused on agent-based approaches that include navigation mesh pathfinding (Shirzadehhajimahmood et al. 2021), reinforcement learning agents for finding design and environmental defects (Ariyurek, Betin-Can, and Surer 2019; Ferdous et al. 2022), reinforcement learning for load testing (Tufano et al. 2022), modeling of user interaction for boundary testing (Owen, Anton, and Baker 2016), search for test case generation (Ferdous et al. 2021b), and search for automated play testing (Ferdous et al. 2021a). Despite planning use in-game AI, we do not see its use in-game testing more broadly beyond back to search. However, as evidenced by Bram Ridder's (AI Programmer for Rebellion) keynote talk at the 2021 AIIDE Conference on "Improved Automated Game Testing Using Domain-Independent AI Planning" (Riddler 2021) and his 2021 GDC AI Summit talk "Automated Game Testing Using a Numeric Domain Independent AI Planner," planning techniques for game testing are beginning to be used in the games industry mixed in with calls for more AI automation of testing (Fray 2023).

In (Volokh and Halfond 2023) the authors proposed an automated approach for determining actions when conducting automated exploration for games. It is based on program analysis (slicing) of the game code. Although they are not using the usual planning formalism (like PDDL), they work with symbolic representation of states and actions and rely on SMT (satisfaiblity module theories) solvers to determine the set of applicable actions in a given state.

The General Workflow

The general workflow is visualized in a flowchart in Figure 1. Each task requires a person (or a group) having one of the following two skill sets:

- A game developer or a game designer (Developer for short). This is someone familiar with the rules of the game and the ability to modify the source code of the game to add log generation functionality. Also, they need to be able to play the game in such a way that all the available game mechanics are utilized. Lastly, they must be capable of (automatically) evaluating whether a given test script is consistent with the game's rules.

- A person with PDDL modeling skills (Modeller for short). This is someone who knows PDDL and has some experience with modeling planning scenarios in this language.

We start by writing an informal but structured description of all the legal player moves/actions. For each player action, we need to provide a name and list of preconditions (what conditions need to be valid in order to allow the player to play this action) and effects (what changes after the action is played out) written in natural language. This task should be
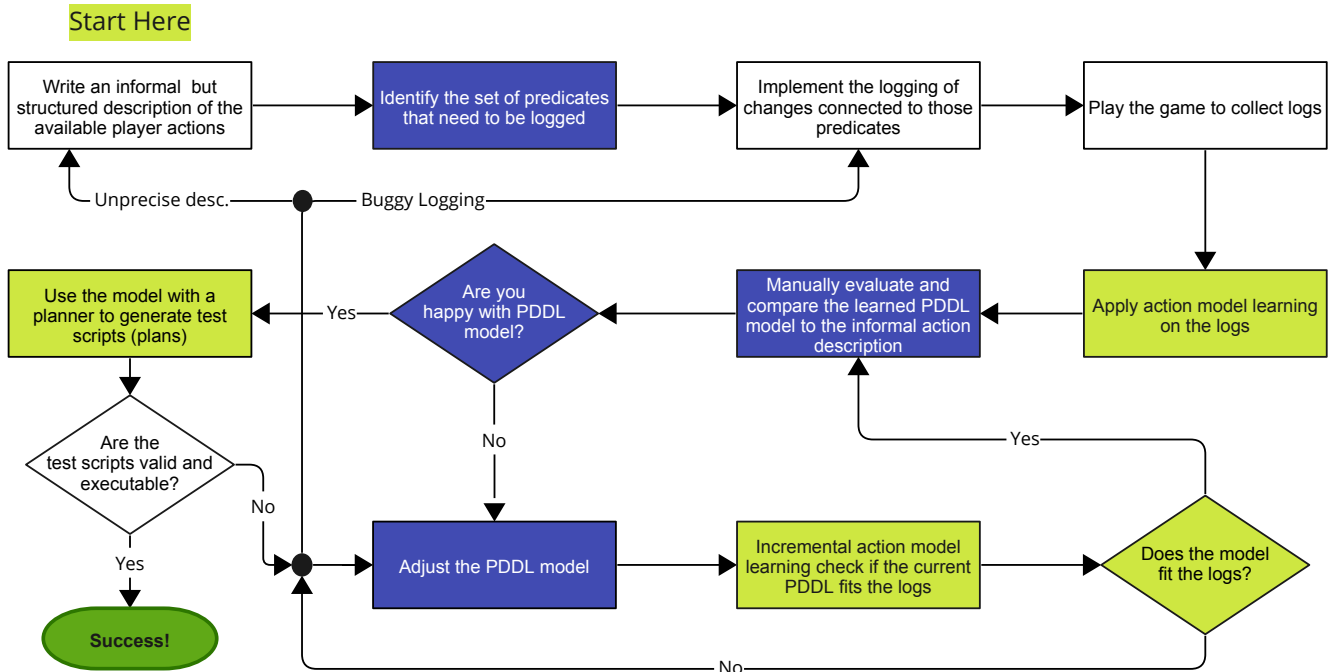
Figure 1: The overview of the workflow for acquiring PDDL domain models based on logs from the game execution. Tasks with a white background should be done by game developer(s). The tasks and decisions with a blue background should be done by the PDDL modeller. The light green tasks can be done automatically.

```
 1  ...
 2  2023-08-15 19:31:20 697 ;NEXT-STATE
 3  2023-08-15 19:31:21 916 (not (questState apples-quest ready));start_quest
 4  2023-08-15 19:31:21 916 (questState apples-quest started);start_quest
 5  2023-08-15 19:31:21 917 ;NEXT-STATE
 6  2023-08-15 19:31:24 315 (not (location player 8,-4));move
 7  2023-08-15 19:31:24 315 (location player 9,-4);move
 8  2023-08-15 19:31:24 316 ;NEXT-STATE
 9  2023-08-15 19:31:24 574 (not (location player 9,-4));move
10  2023-08-15 19:31:24 574 (location player 10,-4);move
11  2023-08-15 19:31:24 574 ;NEXT-STATE
12  2023-08-15 19:31:24 598 (not (collected apples 0));pickup
13  2023-08-15 19:31:24 598 (collected apples 1);pickup
14  2023-08-15 19:31:24 598 (not (location golden-apple0 10,-4));pickup
15  2023-08-15 19:31:24 598 (location golden-apple0 none);pickup
16  2023-08-15 19:31:24 598 ;NEXT-STATE
17  ...
18  2023-08-15 19:31:29 816 ;NEXT-STATE
19  2023-08-15 19:31:30 055 (not (questState apples-quest started));complete_quest
20  2023-08-15 19:31:30 056 (questState apples-quest complete);complete_quest
21  2023-08-15 19:31:30 056 ;NEXT-STATE
22  ...
```

Figure 2: A fragment of the log file created by playing the game and used for action model acquisition.

easily doable by the developer. It requires no PDDL modeling experience, even though the required structure of the action is very similar to that of an STRIPS action.

Next, the informal action descriptions are handed to a modeler. The modeler's task at this point is only to identify a set of predicates required to formally describe all the properties mentioned in the preconditions and effects sections of the action descriptions.

Then, the set of predicates is given to a developer who needs to extend the game's source code to facilitate the logging of changes to those predicates. Now, we are ready to play the game while collecting logs. The logs are passed to an action model learning (AML) tool, and a PDDL model is automatically generated.

Many games already produce logs, so why cannot we just use existing logs instead of modifying the code and creating new ones? In principle, we could do that. However, in our experience, the existing logs (if they are available at all) usually only contain method calls and error messages and do not record changes in state variables. Those kinds of logs unfortunately cannot be used to learn action models because the required information (state changes) is not present in them.

Now, it is the modeler's turn again, as they need to examine and evaluate the generated PDDL model manually. Due to the imprecise nature of AML algorithms, the generated model often needs minor corrections and adjustments. After such adjustments, it is possible to verify automatically whether the new PDDL model is still consistent with the logs. Note that the first model coming from the AML is always consistent with the logs (this is a property of the used action model learning algorithm (Balyo et al. 2024)). If the model is consistent with the logs, the modeler can either continue editing or decide that it is good enough and can be passed to the planning step. On the other hand, if the model is inconsistent with the logs, we need to either change the PDDL model or the logs. Changing the logs may be required because they are inconsistent due to bugs in the implementation of the logging process or because we are not logging the proper predicates. In the latter case, we may need to go back all the way to the beginning of the process and adjust the informal description, update the set of required predicates, and repeat all the follow-up steps (see Figure 1).

If the logs are consistent with the current domain model and the modeler believes it is accurate, we can use a planner to create a collection of plans. These plans are then handed over to the developers, who can check whether they are consistent with the game's rules following some simple goal templates. This process can be done manually or automatically if the game can execute test scripts. If no problems are discovered with the generated plans/test scripts, we are finished, and we have obtained a PDDL model representing the game's rules. If we discover any problems, we handle the situation the same way as in the case of having a domain inconsistent with the logs (see Figure 1).

Proof of concept: A Simple RPG Game

We will demonstrate our workflow on the tutorial project of "Creator Kit: RPG" (Unity Technologies 2019), available in

Move hero	
Hero is at the start tile	Hero is at the goal tile
Start and goal tiles are neighbours	
Pick up quest item	
Hero is at the same location as the item	The item is gone
The quest related to this item is active	The hero has one more of this kind of item
Start quest	
The quest is ready	The quest is active
Hero is at the same location as the quest giver	
Complete quest	
The quest is active	The quest is done
The hero has enough items of the required type	
Hero is at the same location as the quest giver	

Table 1: Structured informal description of player actions. The left column is the preconditions, right is the effects.

the popular video game engine Unity (Unity Technologies 2023) versions 2019.1 – 2021.3.

It is a basic single-player RPG with a 2D environment viewed from above. The player controls a hero character using the arrow keys. The hero can talk to non-player characters (NPCs) on the map. Some NPCs can provide a "quest" (a task for the hero to complete) when spoken to. The quests all follow the pattern of: "collect n items of type T". After a quest is activated, the relevant items appear on the map. To pick up an item, the hero only needs to walk near it. When the required amount of items is collected, the player must visit the NPC who provided the quest to complete it. There are two quests in the demo: "collect 3 golden apples" and "collect 10 chickens".

Following the workflow defined above (in Figure 1), we need to start by providing an informal but structured description of the possible actions. For our simple example, we only require four actions described in Table 1.

In this example, the modeller decided to use a grid-based (i.e., via tiles) abstraction of locations[1] and came up with the following list of properties that should be logged:

- location of the hero (tile),
- locations of each item (tile or none),
- the state of each quest (ready, active, done),
- the location of each quest-giving NPC (tile),
- number of items of each kind the hero has.

Additionally, we will need to know the following static properties of the environment and quests:

---

[1]Using the grid abstraction turned out to be rather inconvenient. The reasons are that it is not very robust if the items are not aligned to the grid and it is hard to determine which tiles are connected. In a game with free movement such as our example, it would be better to use a navigation-mesh-based abstraction (Shirzadehhajimahmood et al. 2021).

```
1   (define (domain rpg)
2       (:requirements :strips :typing :negative-preconditions)
3       (:types
4           number state location locatable typename - object
5           quest item hero - locatable
6       )
7       (:constants
8         active ready done - state
9       )
10      (:predicates
11          (next ?n1 - number ?n2 - number)
12          (connected ?l1 - location ?l2 - location)
13          (questState ?q - quest ?s - state)
14          (itemType ?i - item ?t - typename)
15          (questItemType ?q - quest ?t - typename)
16          (location ?l - locatable ?loc - location)
17          (collected ?t - typename ?n - number)
18          (questItemCount ?q - quest ?n - number)
19      )
20      (:action start_quest
21          :parameters(?quest - quest ?hero - hero ?location - location)
22          :precondition (and
23              (questState ?quest ready)
24              (location ?hero ?location)
25              (location ?quest ?location)
26          )
27          :effect (and
28              (not (questState ?quest ready))
29              (questState ?quest active)
30          )
31      )
32      (:action complete_quest
33          :parameters(?quest - quest ?hero - hero ?typename - typename
34                      ?number - number ?location - location)
35          :precondition (and
36              (questState ?quest active)
37              (location ?hero ?location)
38              (location ?quest ?location)
39              (collected ?typename ?number)
40              (questItemType ?quest ?typename)
41              (questItemCount ?quest ?number)
42          )
43          :effect (and
44              (not (questState ?quest active))
45              (questState ?quest done)
46          )
47      )
```

Figure 3: First half of the RPG domain PDDL description.

```
1    (:action pickup
2        :parameters(?hero - hero ?item - item ?location - location
3                    ?number-1 ?number-2 - number ?typename - typename ?quest - quest)
4        :precondition (and
5            (collected ?typename ?number-1)
6            (location ?item ?location)
7            (location ?hero ?location)
8            (itemType ?item ?typename)
9            (questItemType ?quest ?typename)
10           (questState ?quest active)
11           (next ?number-1 ?number-2)
12       )
13       :effect (and
14           (not (collected ?typename ?number-1))
15           (not (location ?item ?location))
16           (collected ?typename ?number-2)
17       )
18   )
19   (:action move
20       :parameters(?hero - hero ?location-1 ?location-2 - location)
21       :precondition (and
22           (location ?hero ?location-1)
23           (connected ?location-1 ?location-2)
24       )
25       :effect (and
26           (not (location ?hero ?location-1))
27           (location ?hero ?location-2)
28       )
29   )
30 )
```

Figure 4: Second half of the RPG domain PDDL description.

- initial locations of each item (tile),
- type of each item (apple, chicken, ...),
- type of item required for a quest (apple, chicken, ...),
- number of items required to fulfill the quest,
- which tiles are neighbours.

To see the concrete predicate declarations corresponding to these properties see Figure 3.

The developers then implemented logging and provided logs for learning (see Figure 2). The modeller refined the PDDL domain synthesized from the logs (see Figures 3 and 4) and used a planner to generate a plan to win the game (see Figure 6). We used the FF planner (Hoffmann and Nebel 2001) which worked very well for our problem and found a plan in 0.01 seconds.

In order to test whether a generated plan (test script) is valid[2], we implemented automatic test script execution. In this case, it was very easy, since the game is controlled just by moving the hero. Quests are activated/completed automatically when their requirements are fulfilled and the hero comes near their NPC. Also, items are picked up automatically just by walking up to them. Therefore, to execute the test scripts we only need to execute the Move actions. We did that by simulating the pushing of the arrow buttons in

the directions of the next goal tile until we are close enough to the center of that tile[3].

Usage During Development

The process we described should be applied at the beginning of the development process when the game is simple and only contains a few game mechanics like our RPG demo. As the game is developed and new features are added, the PDDL model should be developed together with the game. Using incremental action model learning can aid the developers in maintaining the PDDL model.

Challenges with Fully Automated Logging for Learning

Deciding which properties/predicates should be logged and then implementing the logging into the game code can be complex and time-consuming. Therefore, automating this part of the workflow would be very beneficial. This section will discuss some examples of issues and challenges that need to be solved to achieve full automation.

In general, logging must be detailed enough to capture all the game mechanics precisely; therefore, it is necessary to log the values and changes of all variables, arrays, and other data structures of selected game-play-related classes in the

---

[2]A test script is valid if it is executable in a bug-free implementation of the game. If a test script is valid, but not executable, then a bug in the game is indicated

---

[3]A recording of the plan execution within the game is available here: https://www.youtube.com/watch?v=BASKvQAbG04

```
1   (define (problem level1)
2       (:domain rpg)
3       (:objects
4           player - hero
5           ready - state
6           apples chicken - typename
7           apples-quest chicken-quest - quest
8           chicken0 ... chicken9 - item
9           golden-apple0 ... golden-apple2 - item
10          n-1x-7 ... n9x2 - location
11          n0 ... n9 - number
12      )
13      (:init
14          (collected apples n0)
15          (collected chicken n0)
16          (questItemCount apples-quest n3)
17          (questItemCount chicken-quest n10)
18          (questItemType apples-quest apples)
19          (questItemType chicken-quest chicken)
20          (questState apples-quest ready)
21          (questState chicken-quest ready)
22          (location player n4x10)
23          (location apples-quest n8x-4)
24          (location chicken-quest n-6x7)
25          (connected n-1x-7 n-2x-7)
26          ...
27          (itemType chicken0 chicken)
28          (location chicken0 n-1x3)
29          ...
30          (itemType golden-apple0 apples)
31          (location golden-apple0 n10x-4)
32          ...
33          (next n0 n1)
34          ...
35      )
36      (:goal (and
37          (questState apples-quest done)
38          (questState chicken-quest done)
39      ))
40  )
```

Figure 5: The problem file for the RPG demo. Some of the objects and initial state predicates are redacted to shorten the listing.

```
1   ...
2   22: MOVE PLAYER N10X-4 N9X-4
3   23: MOVE PLAYER N9X-4 N8X-4
4   24: START_QUEST APPLES-QUEST PLAYER N8X-4
5   25: MOVE PLAYER N8X-4 N9X-4
6   26: MOVE PLAYER N9X-4 N10X-4
7   27: PICKUP PLAYER GOLDEN-APPLE0 N10X-4 N0 N1 APPLES APPLES-QUEST
8   28: MOVE PLAYER N10X-4 N10X-3
9   29: MOVE PLAYER N10X-3 N10X-2
10  30: MOVE PLAYER N10X-2 N10X-1
11  31: MOVE PLAYER N10X-1 N10X0
12  32: PICKUP PLAYER GOLDEN-APPLE1 N10X0 N1 N2 APPLES APPLES-QUEST
13  33: MOVE PLAYER N10X0 N11X0
14  34: MOVE PLAYER N11X0 N11X-1
15  35: PICKUP PLAYER GOLDEN-APPLE2 N11X-1 N2 N3 APPLES APPLES-QUEST
16  36: MOVE PLAYER N11X-1 N11X-2
17  37: MOVE PLAYER N11X-2 N10X-2
18  ...
```

Figure 6: A fragment of the plan to solve the demo level of the game.

code. On the other hand, for automated planning to be efficient, we must work with an abstraction of the game rules. We need high-level predicates describing world states for this. The level of abstraction also cannot be too high; otherwise, the learned domain would be useless since it could only generate nonspecific and broad test scenarios. Those would be very difficult to execute automatically.

Precisely, we must map all numeric variables, arrays, strings, and custom data structures to predicates. This step is required because we cannot automatically decide what is necessary to describe all game-play rules fully. Interestingly, mapping all available variables is both too detailed and not necessarily sufficient at the same time. To completely model the behavior of a general computer program, we would also need to represent the "hidden" data like stack traces, the instruction pointer, the variables in the game engine, the operating system, and properties that are implied by the level design or interactions in the physics engine that are not represented in the code. For example, it is possible to jump from one location to another.

Furthermore, we would like to (1-to-1) map planning actions to certain significant functions and methods in the code that represent player actions. The reason is that we want to call these methods from the plans obtained by the learned model. This step is problematic since STRIPS actions have a very limited structure. On the other hand, functions in code are usually much more complex. They contain branching, loops, other function calls, recursion, and other elements a STRIPS action cannot do. Therefore, mapping most functions from the program code one to one is impossible with the planning actions. One solution would be to break up game-code functions into simpler elements that can be mapped to planning actions. We would also need to use additional helper predicates to express the stack traces, program pointers, etc. The disadvantage is that the generated plans would then contain these elementary actions instead of the high-level ones we desired.

Alternatively, we could achieve a "one to many" mapping where one function from the program code gets mapped to a couple of planning actions, which altogether represent the whole original function. This step can be done easily and automatically, but from our experience, many very specific planning actions are learned, and the model does not generalize well. It is somewhat similar to the situation known as *overfitting* in machine learning.

We could mitigate these problems by supporting additional PDDL features like conditional effects and universal quantifiers so that more code functions can be mapped to a single (or at least fewer than before) planning action(s). This process is complicated since most action model learning methods only support STRIPS-like action definitions.

Conclusion

In this paper, we presented a workflow that applies research from the areas of automated planning and action model learning in order to help with the automation of regression testing during video game development. We demonstrated it with a proof-of-concept project and discussed what challenges need to be addressed to further automate its steps.

Future Work

We plan to evaluate and refine the workflow on larger and more complex games. We are currently experimenting by applying it to a real-time strategy game. We also want to run proper user studies to quantify the usefulness of the proposed workflow compared to just creating the domains manually. Furthermore, we want to address and resolve the challenges related to further automating the synthesis of PDDL domains mentioned in the previous section.

References

Aineto, D.; Celorrio, S. J.; and Onaindia, E. 2019. Learning action models with minimal observability. *Artificial Intelligence*, 275: 104–137.

Ariyurek, S.; Betin-Can, A.; and Surer, E. 2019. Automated video game testing using synthetic and humanlike agents. *IEEE Transactions on Games*, 13(1): 50–67.

Balyo, T.; Suda, M.; Chrpa, L.; Šafránek, D.; Dvořák, F.; Barták, R.; and Youngblood, G. M. 2024. Learning Planning Action Models from State Traces. arXiv:2402.10726.

Callanan, E.; De Venezia, R.; Armstrong, V.; Paredes, A.; Chakraborti, T.; and Muise, C. 2022. MACQ: a holistic view of model acquisition techniques. *arXiv preprint arXiv:2206.06530*.

Chung, M.; Buro, M.; and Schaeffer, J. 2005. Monte Carlo Planning in RTS Games. In *Proceedings of the 2005 IEEE Symposium on Computational Intelligence and Games (CIG05)*. IEEE.

Duarte, F. F.; Lau, N.; Pereira, A.; and Reis, L. P. 2020. A survey of planning and learning in games. *Applied Sciences*, 10(13): 4529.

Ferdous, R.; Kifetew, F.; Prandi, D.; Prasetya, I.; Shirzadehhajimahmood, S.; and Susi, A. 2021a. Search-based automated play testing of computer games: A model-based approach. In *International Symposium on Search Based Software Engineering*, 56–71. Springer.

Ferdous, R.; Kifetew, F.; Prandi, D.; Prasetya, I. S. W. B.; Shirzadehhajimahmood, S.; and Susi, A. 2021b. Search-Based Automated Play Testing of Computer Games: A Model-Based Approach. In *Search-Based Software Engineering: 13th International Symposium, SSBSE 2021, Bari, Italy, October 11–12, 2021, Proceedings*, 56–71. Berlin, Heidelberg: Springer-Verlag. ISBN 978-3-030-88105-4.

Ferdous, R.; Kifetew, F.; Prandi, D.; and Susi, A. 2022. Towards Agent-Based Testing of 3D Games Using Reinforcement Learning. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 1–8.

Fikes, R. E.; and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4): 189–208.

Fray, A. 2023. Automated Testing Roundtables GDC 2023. https://autotestingroundtable.com/. (Accessed on 12/12/2023).

Ghallab, M.; Howe, A.; Knoblock, C.; Mcdermott, D.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL—The Planning Domain Definition Language.

Hoffmann, J.; and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14: 253–302.

Hsu, F.-h.; Anantharaman, T. S.; Campbell, M. S.; and Nowatzyk, A. 1990. Deep thought. In *Computers, Chess, and Cognition*, 55–78. Springer.

Juba, B.; Le, H. S.; and Stern, R. 2021. Safe Learning of Lifted Action Models. In *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning*, 379–389.

McCluskey, T. L.; Vaquero, T. S.; and Vallati, M. 2017. Engineering Knowledge for Automated Planning: Towards a Notion of Quality. In Corcho, Ó.; Janowicz, K.; Rizzo, G.; Tiddi, I.; and Garijo, D., eds., *Proceedings of the Knowledge Capture Conference, K-CAP 2017, Austin, TX, USA, December 4-6, 2017*, 14:1–14:8. ACM.

Newell, A.; Simon, H. A.; et al. 1972. *Human problem solving*, volume 104:9. Prentice-hall Englewood Cliffs, NJ.

Owen, V. E.; Anton, G.; and Baker, R. 2016. Modeling user exploration and boundary testing in digital learning games. In *Proceedings of the 2016 conference on user modeling adaptation and personalization*, 301–302.

Politowski, C.; Guéhéneuc, Y.-G.; and Petrillo, F. 2022. Towards automated video game testing: still a long way to go. In *Proceedings of the 6th International ICSE Workshop on Games and Software Engineering: Engineering Fun, Inspiration, and Motivation*, 37–43.

Riddler, B. 2021. Improve Automated Game Testing Using Domain Independent AI Planning - YouTube. https://www.youtube.com/watch?v=2KXmxuCjjCw. (Accessed on 12/12/2023).

Samuel, A. L. 1959. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3): 210–229.

Schaeffer, J.; Lake, R.; Lu, P.; and Bryant, M. 1996. Chinook the world man-machine checkers champion. *AI magazine*, 17(1): 21–21.

Shirzadehhajimahmood, S.; Prasetya, I.; Dignum, F.; Dastani, M.; and Keller, G. 2021. Using an agent-based approach for robust automated testing of computer games. In *Proceedings of the 12th International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 1–8.

Simpson, R. M.; Kitchin, D. E.; and McCluskey, T. L. 2007. Planning domain definition using GIPO. *Knowl. Eng. Rev.*, 22(2): 117–134.

Smith, S. J.; Nau, D.; and Throop, T. 1998. Computer bridge: A big win for AI planning. *AI magazine*, 19(2): 93–93.

Tufano, R.; Scalabrino, S.; Pascarella, L.; Aghajani, E.; Oliveto, R.; and Bavota, G. 2022. Using reinforcement learning for load testing of video games. In *Proceedings of the 44th International Conference on Software Engineering*, 2303–2314.

Unity Technologies. 2019. Creator Kit: RPG - Unity Learn. https://learn.unity.com/project/creator-kit-rpg. (Accessed on 12/09/2023).

Unity Technologies. 2023. Unity Real-Time Development Platform — 3D, 2D, VR & AR Engine. https://unity.com/. (Accessed on 12/09/2023).

Vaquero, T. S.; Silva, J. R.; Tonidandel, F.; and Beck, J. C. 2013. itSIMPLE: towards an integrated design system for real planning applications. *Knowl. Eng. Rev.*, 28(2): 215–230.

Volokh, S.; and Halfond, W. G. 2023. Automatically Defining Game Action Spaces for Exploration Using Program Analysis. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 19(1): 145–154.

Wang, X. 1996. *Learning planning operators by observation and practice*. Ph.D. thesis, Citeseer.

Wurman, P. R.; Barrett, S.; Kawamoto, K.; MacGlashan, J.; Subramanian, K.; Walsh, T. J.; Capobianco, R.; Devlic, A.; Eckert, F.; Fuchs, F.; Gilpin, L.; Khandelwal, P.; Kompella, V.; Lin, H.; MacAlpine, P.; Oller, D.; Seno, T.; Sherstan, C.; Thomure, M. D.; Aghabozorgi, H.; Barrett, L.; Douglas, R.; Whitehead, D.; Dürr, P.; Stone, P.; Spranger, M.; and Kitano, H. 2022. Outracing champion Gran Turismo drivers with deep reinforcement learning. *Nat.*, 602(7896): 223–228.

Yang, Q.; Wu, K.; and Jiang, Y. 2007. Learning action models from plan examples using weighted MAX-SAT. *Artificial Intelligence*, 171(2-3): 107–143.

Zhuo, H. H.; Yang, Q.; Hu, D. H.; and Li, L. 2010. Learning complex action models with quantifiers and logical implications. *Artificial Intelligence*, 174(18): 1540–1569.