A Progress-Based Algorithm for Interpretable Reinforcement Learning in Regression Testing

Pablo Gutiérrez-Sánchez, Marco A. Gómez-Martín, Pedro A. González-Calero, and Pedro P. Gómez-Martín

***Abstract*—In video games, the validation of design specifications throughout the development process poses a major challenge as the project grows in complexity and scale and purely manual testing becomes very costly. This article proposes a new approach to design validation regression testing based on a reinforcement learning technique guided by tasks expressed in a formal logic specification language (truncated linear temporal logic) and the progress made in completing these tasks. This requires no prior knowledge of machine learning to train testing bots, is naturally interpretable and debuggable, and produces dense reward functions without the need for reward shaping. We investigate the validity of our strategy by comparing it to an imitation baseline in experiments organized around three use cases of typical scenarios in commercial video games on a 3-D stealth testing environment created in unity. For each scenario, we analyze the agents' reactivity to modifications in common assets to accommodate design needs in other sections of the game, and their ability to report unexpected gameplay variations. Our experiments demonstrate the practicality of our approach for training bots to conduct automated regression testing in complex video game settings.**

***Index Terms*—Automated game testing, game-playing AI, regression testing, reinforcement learning (RL), temporal logics (TLs).**

I. Introduction

VIDEO games are increasingly complex and ambitious software products in terms of extension and the number of elements or assets involved in their creation. In this context, there are a number of quality control tasks related to gameplay validation, i.e., ensuring that the way players interact with the game remains consistent with the designer's preconceived idea of how each challenge should be tackled. Play-testing is a common method of assessing game quality, usually requiring a human tester to answer questions about the degree of difficulty and perceived enjoyment during play, the presence of bugs and glitches, or the feasibility of successfully completing various tasks enforced by the design. The problem with this approach to quality testing is that as levels, shared assets between game scenes, nonplayable character artificial intelligence (AI)s, or functionalities are created or modified throughout development, the answers to these questions may change unexpectedly in an increasing number of scenarios. For instance, modifying the configuration of a shared enemy AI could alter the difficulty of a level, prevent it from being solved using the originally designed strategy, or even enable unexpected new solutions.

In order to maintain a sufficiently thorough coverage of the range of scenarios and checks to be performed within the game, design validations should be performed as frequently as possible, which is not logistically feasible if these tasks are reserved for human testers since they require an unsustainable effort in terms of time and money. By design validations, we will narrow down here to the type of test that receives a specification, typically given by a series of steps or instructions to be performed sequentially by the tester, and reports whether these are feasible in the environment in which they are proposed, together with comments on what makes them invalid in the case where they are deemed nonviable. Having an automated specification validation methodology in the development process does not replace the human tester but allows redirecting their efforts to verify only those sections of the project where it is truly suspected that unexpected changes have occurred, thus increasing efficiency and coverage in QA.

In recent years, different automated playtesting methodologies have been proposed to try to alleviate these incremental costs. In the industry, this is usually done primarily through hand-programmed bots that perform playtesting automatically [1], [2], [3], [4], which is typically cumbersome and experiences issues in adapting to changes in the original environment, often requiring a code rework by a human programmer. Another strategy with increasing traction is the generation of testing agents by means of machine learning methods, such as reinforcement learning (RL) [5], [6], [7], [8], which do not require explicit scripting to learn to play and can be retrained upon level adjustments, thereby alleviating the generalization problem. RL, however, is often sample inefficient and its success depends heavily on the choice of reward function, which often calls for extensive technical expertise, and its integration into a commercial development environment is not straightforward.

In this article, we iterate on our previous work in the field of automated testing in video games, proposing a formal specification-based approach to create agents that are able to perform design validations on a scheduled basis throughout the game development process, as shown in Fig. 1. Our method eliminates the need to specify a manual reward function in the RL loop by automatically constructing a dense function from a staged representation of what is expected to be validated in the

Manuscript received 1 November 2023; revised 3 April 2024 and 20 May 2024; accepted 4 July 2024. Date of publication 11 July 2024; date of current version 17 December 2024. This work was supported by the Spanish Ministry of Science and Innovation under Grant PID2021-123368OB-I00. Recommended by Associate Editor L. Gisslen. *(Corresponding author: Pablo Gutiérrez-Sánchez.)*

The authors are with the Software Engineering, Artificial Intelligence Department, Complutense University of Madrid, 28040 Madrid, Spain.

Digital Object Identifier 10.1109/TG.2024.3426601

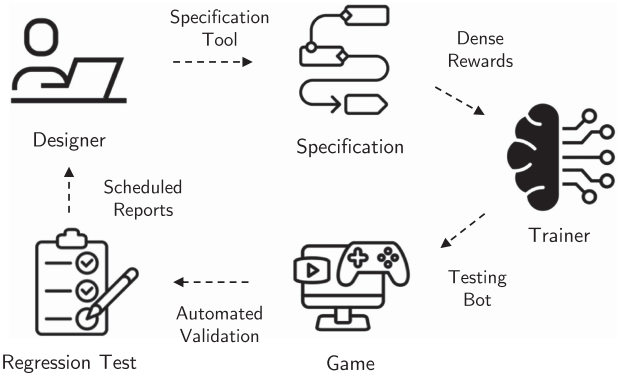

Fig. 1. Our approach allows the designer to provide a plan on how to interact with the level, from which a dense reward function is generated to train a bot to periodically validate the feasibility of the solution throughout development.

level using an improved version of the techniques we previously presented in [9]. More specifically, the algorithm described here builds on the ideas from the domain of robotics in [10] to translate a specification given by a system of temporal logics (TLs) into an automaton that abstracts the temporal structure of the predicate into states with localized tasks. In addition, the generated automaton and reward function are interpretable by design, making it possible to gain a clear idea of the agent's progress in satisfying the specification at all times by examining the reward accumulated during training and the automaton states visited. These agents can be used periodically to perform regression testing on their corresponding in-game validations, allowing the human tester to focus only on reviewing more complex features or sections where the bot alerts they might have undergone significant changes in gameplay. To analyze the feasibility of our approach as an automated testing methodology, we show its applicability in a number of different environments and validation tasks developed in Unity3D [11], training agents for each use case and studying their response to the question of whether it is still possible to satisfy the original design plan when modifications are made in those environments.

The rest of this article is organized as follows. Section II introduces related work in the field of automated game testing. Section III develops the concepts on TLs and their application to RL needed to understand the techniques involved in our contribution, while Section IV exemplifies these notions on a concrete scenario within our testing environment. In Section V, we elaborate on the methodology proposed in this article and the setup used to train our testing agents, while Section VI details the experiments carried out to evaluate this strategy on typical video-game use cases. Finally, Section VII concludes this article.

II. Automated Video Game Testing

Automated video game testing has been gaining traction in both academia and industry in recent years as a consequence of the growing need to maintain adequate coverage of games that are increasingly large in domain and scope [12]. In this section, we will review some of the most relevant strategies proposed in the literature in relation to our contribution.

One of the simplest automated playtesting approaches involves manually recording traces, in which a human player performs the designated tasks, replaying them from time to time within the game environment to confirm that these actions can continue to meet the original objectives [13]. This methodology, as expected, is only functional in static and/or deterministic contexts, motivating the need to produce intelligent agents that are able to adapt to dynamic environments. Currently in the industry, most of the testing techniques used are based on classical AI, such as manually programmed agents via behavior trees or state machines, model-based approaches [2], [3], [4], or randomized exploration methods to complement manual testing tasks.[1,2] While many of these strategies are applicable to simple environments, most do not scale to more complex, 3-D, or high-dimensional state space games.

Alternatively, several other works have proposed the use of machine learning techniques for quality control in video games. Concretely, strategies that make use of RL are particularly prevalent. Gordillo et al. [5] proposed a solution based on intrinsic rewards to explore complex 3-D environments in order to find in-game issues that are difficult to detect manually. Bergdahl et al. [6] compared traditional strategies based on navigation meshes with reinforcement agents trained to reach arbitrary positions in a complex 3-D setting, showcasing the latter's ability to spot navigation bugs and exploits. RL has also been used to try to model the interactions of different types of players with game environments in order to provide support in the design process. Agarwal et al. [7] generated agents with different behaviors to play levels from a 2-D platform game, *Sonic the Hedgehog 2*, using a genetic deep RL strategy and offer an intuitive methodology for visualizing the trajectories of the bots. Meanwhile, Ariyurek et al. [8] used agents with different personas (explorers, pacifists, etc.) to discover different ways of approaching levels, a strategy conceptually similar to that in [14], where a variation of Monte Carlo tree search is used to demonstrate how generative player models can be leveraged to replicate archetypal play styles across different levels.

While not as prominent as RL in the game testing literature, imitation learning (IL) is starting to gain some popularity in recent years. By IL we refer to the generation of agents capable of mimicking the behavior of an expert demonstrator that provides a set of state-action pair traces as a reference for the bot to be trained. Some of the most widespread strategies are behavioral cloning [15], which makes use of supervised learning techniques from the dataset of demonstrations to predict the most likely actions for a given state, and generative adversarial imitation learning (GAIL) [16], based on an adversarial approach, in which the rewards of the actuator agent are proportional to how much it manages to deceive a second discriminator agent trained in parallel to learn to distinguish between human and synthetic behaviors.

In game testing, Sestini et al. [17] introduced the curiosity-conditioned proximal trajectories algorithm for testing complex

[1][Online]. Available: https://www.gdcvault.com/play/1027049/-Final-Fantasy-VII-Remake

[2][Online]. Available: https://www.gdcvault.com/play/1026366/Automated-Testing-of-Gameplay-Features

environments, alternating between policies similar to and distant from those of a human via a weighting system of signals based on IL, RL and curiosity. Sestini et al. [18] proposed a methodology for trace validation based on the DAGGER algorithm [19], in which the designer interactively demonstrates their desired behavior in real time. A similar approach is used in the Google Research project *Falken*.[3] Tucker et al. [20] used inverse RL to generate rewards from human traces for agents learning to play *Atari* games.

III. TLs for RL

One of the key disadvantages of RL is that the user must manually translate the requirements of the task to be performed by the agent into a numerical reward function, which often requires significant mathematical background, considerable expertise in machine learning and domain knowledge of the target environment, making these strategies inaccessible to the general user. For complex problems, and especially for those involving temporal dependencies (goals that must be fulfilled in a certain order, for example), these functions can be very challenging to design; furthermore, the user is typically confronted with the need to manually shape the rewards to guide the agent during learning. Reward shaping can become very error-prone and is frequently accompanied by tedious trial-and-error iterations to produce reward functions that induce the expected behavior. It is in this context that TLs have attracted increasing interest in the last decade as a means of supporting the generation of reward functions in RL frameworks from requirements formally modeled in these languages, particularly in the field of robotics [21].

By TLs we refer to any system of rules and symbols that allows reasoning about propositions in terms of their evolution over time. TLs play an important role in the formal verification of requirements in both hardware and software [22], with truncated linear temporal logic (TLTL) [23] being one of the most widely adopted examples of these logics. TLTL formulas are defined based on a series of core logical and temporal operators and, more importantly, predicates of the form $f(s) > 0$, where $f : \mathcal{S} \to \mathbb{R}$ represents a function applied to the system's state $s \in \mathcal{S}$. The latter are the ones that are truly responsible for incorporating a certain knowledge base into our specifications, as well as conditions related to the world and context in which we operate, such as "being close to a point" or "perceiving a high temperature." These conditions are highly domain-dependent and consequently, for each game we contemplate, we will inevitably encounter the need to define a specific set of predicates that will determine the expressiveness of our tasks. Having said this, a TLTL specification $\phi \in \Phi$ adheres to the following syntax:

$$\phi := \top \mid f(s) > 0 \mid \neg\phi \mid \phi_A \wedge \phi_B \mid \phi_A \vee \phi_B \mid \phi_A \Rightarrow \phi_B \mid \Diamond\phi \mid \Box\phi \mid \phi_A \,\mathcal{U}\, \phi_B \mid \bigcirc \phi. \tag{1}$$

Here, $\top$ is the True boolean constant, $f(s) > 0$ is a check on a domain function on the system state, $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction) and $\Rightarrow$ (implication) are Boolean connectives, and $\Diamond$ (eventually), $\Box$ (always), $\mathcal{U}$ (until), and $\bigcirc$ (next) are temporal operators.

From a design perspective, $\Diamond$ indicates a requirement that must be met at some point during the game, and is usually referred to as a liveliness condition, whereas $\bigcirc$ mandates that the associated condition be met in the very next simulation time step. For instance, if our predicate is $\bigcirc\,\phi$, we anticipate ϕ to be true in the second time step of the run. Typically, designers would need to use a combination of $\Diamond$ and $\bigcirc$ operators to ensure that the specified condition is met at a later time. This allows for sequences of conditions that must be satisfied one after the other by adding blocks of the form $\phi_A \wedge \bigcirc(\Diamond\,\phi_B)$. For global constraints that must be continuously in effect (which are often called safety conditions), the $\Box$ operator can be employed, followed by the condition we want to persist. This is especially useful for modeling level constraints, such as maintaining a specific item or preventing health points from falling below a certain threshold.

Among TLs, the focus has been mostly on those that allow the formulation of quantitative semantics, commonly known as robustness metrics. As opposed to qualitative semantics, which simply indicate whether or not a system trace satisfies a logical predicate, these robustness measures make it possible to provide a numerical assessment of how well a state trace $\mathcal{T} = (s_1, \ldots, s_n) \in \mathcal{S}^n$ adheres to the given specification through a robustness function $\rho : \mathcal{S}^n \times \Phi \to \mathbb{R}$, which assigns a real-valued number to each predicate ϕ from the predicate space Φ and system trace $\mathcal{T}$.

Once a quantitative semantics is available (and this need not be unique), the obvious strategy to guide learning in a RL setting on the basis of TL specifications is to assign a reward to the agent at the end of each training episode given according to the robustness of the trace generated during the episode, $\rho(\mathcal{T}, \phi)$ [24]. This strategy suffers however from a problem of reward sparsity during training, as reinforcement can only be provided for a whole trajectory, hampering the learning of complex or long-lasting tasks in a significant way.

In response to this problem, Li et al. [25] introduced the concept of Finite State Predicate Automata (FSPA) to abstract away the temporal dependencies of the original predicate and provide dense rewards based on robustness variations between automaton transitions. Essentially, an FSPA can be thought of as a finite state automaton where transitions are associated with TL predicates, such that a transition occurs when the robustness of its linked predicate is both positive and maximal (if multiple edges meet this criterion). This allows the global problem to be converted into small substeps localized in states of the automaton where the progress conditions are well defined. As any given transition in FSPAs comes with an associated robustness variation, these can be used as a step-based reward instead of the episodic one described earlier. In addition, trap states are introduced in these FSPAs to represent task failure.

An additional advantage of this construction is that it eliminates the need to both define and compute the robustness of temporal operators in the predicates of our language, which is often a complex and costly process. In the case of the grammar from (1), here we define the robustness function $\rho_s(\phi) := \rho(s, \phi)$ that

[3][Online]. Available: https://github.com/google-research/falken

outputs $\rho_s(\top) = 1$, $\rho_s(f(s) > 0) = f(s)$, $\rho_s(\neg\phi) = -\rho_s(\phi)$, $\rho_s(\phi_A \wedge \phi_B) = \min(\rho_s(\phi_A), \rho_s(\phi_B))$, and $\rho_s(\phi_A \vee \phi_B) = \max(\rho_s(\phi_A), \rho_s(\phi_B))$, aiming to keep the range of all the domain functions we define in the same interval $f(s) \in [-1, 1]$ so that they are comparable to each other and no conditions outweigh others in composite predicates. Note how here we drop any mention of traces in favor of single states (now $\rho : \mathcal{S} \times \Phi \to \mathbb{R}$), which are the only evaluations that we will need to perform when relying on a FSPA structure.

In the literature, these automata are usually implemented by means of deterministic Rabin automata (DRA) or limit-deterministic Büchi automata (LDBA). External libraries like Rabinizer [26] offer convenient and optimized solutions for automatically carrying out these conversions in the case of DRAs, making the process easily embeddable with game engines and highly practical, which is why in this work we opt for DRAs instead of LDBAs. Whenever we refer to the automata used in our approach, we will be talking about DRAs. Further conversion options are described in [27].

IV. A Game Environment Example: Liquid Snake

To illustrate the concepts from Section III, we will now provide an example in the test environment used in this work of how TLTL can be used to formulate a design specification for one of its levels, together with a possible FSPA generated from the requirements to be used later in a RL framework.

Liquid Snake, described in [28], is a prototype of a 3-D stealth game created for the purpose of conducting AI-driven testing experiments within the Unity3D engine [11]. In this game, the player's objective is to guide the main character through a series of chambers and lead them to the level exit while evading enemy detection. Enemies display typical stealth game behavior by patrolling their assigned routes in search of their target. When they spot the player, they give chase and continue to pursue them until their target is either defeated or manages to evade their view. In the latter scenario, the enemies retrace their steps to the last known location of the player, conduct a brief search, and then return to their original patrol path if they are unable to find them. The main character can jump to overcome obstacles and reach otherwise inaccessible distant platforms, as well as crouch or crawl to take cover behind low walls, allowing them to evade enemy sight or navigate sections with low ceilings.

For the examples presented here, we define $\eta : \mathcal{S} \to [0, 1]$ as the proportion of the player's remaining health, with $\pi := \eta(s) > 0$ representing the predicate "the character is still alive." For the notion of proximity of the character to a target *tar*, we define the function $\gamma_{\text{tar}} : \mathcal{S} \to [-1, 1]$

$$\gamma_{\text{tar}}(s) = \begin{cases} 1 - \frac{\delta_{\text{tar}}(s)}{b}, & 0 \leq \delta_{\text{tar}}(s) < b \\ \frac{\delta_{\text{tar}}(s)^2 - 2B\delta_{\text{tar}}(s) + 2bB - b^2}{(b-B)^2}, & b \leq \delta_{\text{tar}}(s) \leq B \\ -1, & B < \delta_{\text{tar}}(s) \end{cases} \quad (2)$$

where $\delta_{\text{tar}}(s)$ is the Euclidean distance between the character and the target in s, b is the lower bound below, which we consider proximity to be achieved, and B is the upper bound beyond, which we assume the distance to be too large to be considered close. This is nothing more than the second-degree polynomial that evaluates to -1 when the distance between objects is at its maximum ($\delta_{\text{tar}}(s) = B$) and 0 when it equals the proximity threshold ($\delta_{\text{tar}}(s) = b$), which turns into a linear function approaching 1 as the distance gets closer to 0. Empirically, we choose B as the size of the level's bounding box, $b = 1$, and write $\gamma_{\text{tar}} := \gamma_{(\text{tar})}(s) > 0$ to denote the player's proximity predicate to a target in the scene. For instance, if we set $B = 10$ and are currently 5 units away from the target, then $\gamma_{\text{tar}}(s) \approx -0.69$. This function has a steeper gradient as $\delta_{\text{tar}}(s)$ approaches b and is only positive when proximity has been reached.

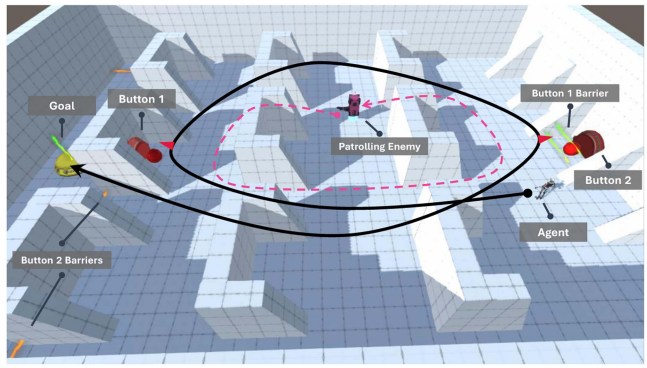

Fig. 2. Screenshot of the "Cover and Hide" environment.

One of the levels included in our experiments, depicted in Fig. 2, introduces an enemy that patrols an environment filled with multiple grid-based obstacles, obstructing visibility and movement for both the enemy and the player. The objective here is to visit and activate the button to the left, s_1 ("Button 1" in the figure), to unlock the barrier guarding the button on the right side of the environment, s_2 ("Button 2"). After that, the player must visit this second button to unlock the barriers protecting the area in the leftmost section of the level, and then reach the goal located there, e. All of this must be done while remaining alive. In terms of TLTL the specification for this level can be expressed as follows:

$$\Diamond(\gamma_{s_1} \wedge \bigcirc(\Diamond(\gamma_{s_2} \wedge \bigcirc\Diamond\gamma_e))) \wedge \Box\,\pi. \quad (3)$$

From the designer's point of view, this translates into listing a series of conditions that must be met sequentially through blocks of "and next eventually" operators as mentioned in Section III, in this case visiting three level points in order, along with a clause with task restrictions ("without dying").

This predicate can then be translated into a DRA using the procedures listed in Section III, outputting an FSPA like the one in Fig. 3. In this figure, the connections between states are labeled with the predicates that must be satisfied to trigger the transition. These automata are constructed so that there is always an edge equipped with a true formula, meaning that there will always be a transition to take at any given step. Furthermore, each of the states is labeled with a predicate representing what part of the specification remains to be fulfilled at that node, with the initial state holding the complete original formula, and subsequent states containing increasingly smaller formulas. We will discuss

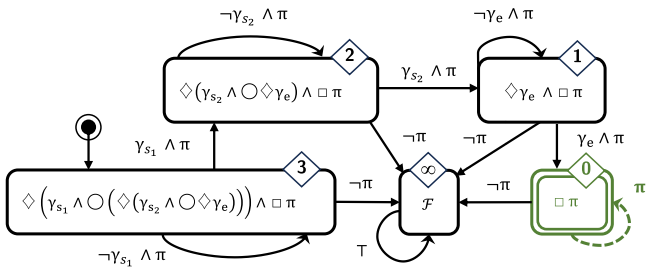

Fig. 3. FSPA translation of the predicate from formula 3.

later how this property can be of help in supplementing the information provided by regression tests. Also note here the trap state that can be reached from any point of the automaton by violating the safety condition (when the player loses all health points), labeled with a false predicate indicating that it is no longer possible to satisfy the specification from it. The final state, on the other hand, is still accompanied by the original safety condition and outlined in green to denote that the automaton will accept any trace that ends in this state. Note that although this example produces a fairly simple automaton for the sake of readability, the FSPAs generated can be arbitrarily complex, especially when conditional clauses are introduced and multiple nested temporal operators are combined.

V. Proposed Method

We propose an approach based on RL with reward functions automatically generated from design specifications in TLTL to conduct automated regression tests throughout the development of the game. These specifications provide a framework to express a course of action whose validity the designer wishes to verify over a given level on a regular basis. In this section, we first detail our setup and training algorithm, and then describe why we decide to adopt this methodology.

A. Reward Generation Algorithm

Our approach to the reward allocation strategy, which is formally stated in Algorithm 1, is based on the concept of tracking progress within the task specification. We heavily rely on the structure of the reward automaton to measure this progress. In simple terms, at each step of the simulation, we calculate a value between 0 and 1, representing the proportion of the specification that has been satisfied up to that point. The agent is then rewarded based on how much it has improved compared to the best value of this metric during the episode.

We start with a design specification provided via a TLTL predicate and translate it into an equivalent DRA-based FSPA. For each state as in the automaton, we then determine the minimum number of transitions required to reach a goal state. This metric is denoted by $d_g(as)$. This also allows us to identify "trap" states from which it is impossible to reach a goal (i.e., $d_g(as) = \infty$). This is valuable for determining when we can end the episode due to a specification violation. In the example from Fig. 3, $d_g(as)$ is indicated by a diamond in the upper right corner of each state. Note how this is ∞ for the trap state and 0 for the single goal state outlined in green. With these distances, we can define SpecSize, as $\max_{as}(d_g(as))$ (line 4). This helps us reason about our progress based on an upper limit. In our example, SpecSize $= 3$.

With this information, during each agent step t (not necessarily linked to the game's update step), we apply the action chosen by the agent's current policy to the game state $gs^t \xrightarrow{\text{Action}} gs^{t+1}$ and update the automaton's state $as^t \xrightarrow{gs^{t+1}} as^{t+1}$. To do that, we evaluate the robustness of each outgoing edge ϕ_e from as^t, $\rho_{gs}(\phi_e)$, and take the transition with the highest value. We then estimate the base progress in the specification from as^{t+1} as $\frac{\text{SpecSize} - d_g(as^{t+1})}{\text{SpecSize}}$. This is stated in lines 7–11 and tells us how many steps we have taken toward a goal state relative to the maximum number of steps required to reach it. Continuing with our example, the state as with label $\diamondsuit\gamma_e \wedge \square\,\pi$ has $d_g(as) = 1$, and so we can estimate that we have completed at least $2/3$ of the task and initialize the progress at that value. This way of estimating progress assumes that all subtasks are equal in size, which might be an oversimplification depending on the context.

Now, if the transition would keep us in the current state, the reward is adjusted based on the best robustness that would improve the distance (lines 12–14). This indicates how much progress has been made in the current task. For instance, if we are at the same as from before, the only edge that would bring us closer to the goal would be the one with predicate $\phi_e = \gamma_e \wedge \pi$. If at the current time-step we are at full health and $d_e = \frac{t+T}{2}$ (half of the maximum distance to the level target), then we would have that $\rho_{gs}(\gamma_e) = -0.75$ [see (2)], $\rho_{gs}(\pi) = 1$ and $\rho_{gs}(\phi_e) = \min(-0.75, 1) = -0.75$. As the base progress was 2/3 this means that our actual progress at the current step is $\frac{2}{3} + \frac{1-0.75}{\text{SpecSize}}$, which is approximately 0.75. If this value is higher than the last maximum recorded up to that point, the difference is added to the step reward (lines 25–28).

However, it could be the case that no transition would allow us to improve the distance to the goal. This means we are either in a trap state (line 19) from which we can never reach a goal or are located in a goal state from which we cannot improve further because it is already as good as possible (line 16). Typically, when we reach a trap state, it is common to end the episode directly. In an acceptance state, it might sometimes be interesting to let the episode continue beyond the original goal for the agent to learn how to stay in it and improve the specification's quality. Nevertheless, for now, we are only interested in tasks that end after reaching a goal state, so we will not delve into the optimization details in that case.

Considering this structure, it is easy to see that the total accumulated reward throughout the episode always falls between 0 and 1. This implies that there is no difference between completing the specification sooner or later, as long as it eventually gets fulfilled. Because of this, it is often useful to introduce a penalty in each simulation step to encourage the agent to optimize the time required to complete the tasks (line 29). This also provides an initial estimate of how long it might take to fulfill the design intent. To maintain the interpretability of the results, we choose the value of the existential penalty in

Algorithm 1: Algorithm for Reward Generation-Rabin FSMA.

Initialize game state as gs^0
Initialize automaton state as as^0
Initialize episode progress as 0
SpecSize $\leftarrow \max_s(d_g(as))$
while $t < t_{\max}$ **do**
  Initialize step reward as 0
  Action $\leftarrow$ policy(gs^t, as^t)
  Step Action in game state $gs^t \xrightarrow{Action} gs^{t+1}$
  Use gs^{t+1} and as^t to tick the Automaton, computing $\rho_{gs^{t+1}}(\phi_e)$ for all predicates ϕ_e in edges attached to as^t
  Select edge from as^t with the highest $\rho_{gs^{t+1}}(\phi_e)$ and transit Automaton using that edge $e : as^t \rightarrow as^{t+1}$
  step progress $\leftarrow \frac{\text{SpecSize} - d_g(as^{t+1})}{\text{SpecSize}}$
  if $(as^t = as^{t+1})$ **then**
    if other e' would improve distance to goal **then**
      step progress += $\frac{1+\max_{e' \in impEdges}(\rho_{gs^{t+1}}(\phi_{e'}))}{\text{SpecSize}}$
    else
      if e leads to a goal state **then**
        step progress $= 1$
        terminate the episode (success)
      else
        step reward -= terminal penalty
        terminate the episode (failure)
      end if
    end if
  end if
  if step progress $>$ episode progress **then**
    step reward += step progress - episode progress
    episode progress $=$ step progress
  end if
  step reward += existential penalty
  grant step reward to agent
end while

such a way that if the agent fails to complete the specification within the provided time, the total penalty amounts to -1. This changes the total reward range to [-1, 1]. With this decision, a negative value indicates a failure to meet the specification, with values closer to 0 representing progress towards task completion, while a positive value corresponds to successful specification completion, with higher values indicating "faster" policies. If existential penalties are applied, it is often advisable to apply a terminal penalty upon reaching a trap state, so as to prevent the agent from attempting to break the specification on purpose (for instance, by letting themselves die) in order to avoid further negative rewards.

B. Training Setup

Once the reward generation method has been established, we can opt for any RL algorithm that suits the characteristics of our problem to train the testing agents. In our case, we decided to use the proximal policy optimization (PPO) algorithm [29] due to its relative generality and lack of over-reliance on hyperparameter tuning.

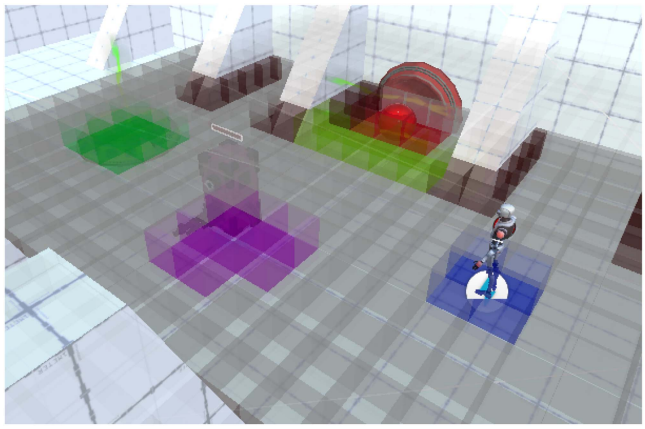

Fig. 4. Semantic map in our environment with a single vertical dimension.

On a different note, in an effort to generalize the approach to as many game environments as possible, we define the state space based on a typical structure of 3-D spaces and actions, which tend to be prevalent in a good part of modern titles.

To setup the training of a new testing agent, developers define the specification on which the test is to be performed by means of a predicate in TLTL. Depending on what is to be described, this process may require the declaration of new atomic predicates containing domain knowledge for the game (e.g., that a switch is activated, that the player does not run out of ammunition, etc). From this specification, it is possible to compile a DRA-based FSPA as described in Section V-A, which will be used to generate the rewards for the RL algorithm. The user can also specify which entities and variables from the scene should be taken into account by the agent for sensing the environment creating a vector v_{user}. In our case, we include information, such as end or intermediate targets, enemies, switch activation states, character health points, whether the player can jump or crouch, or any other piece of information deemed useful to satisfy the specification. We currently allow the user to enter references to integer or float values (normalized by a range of expected values), boolean variables, or relative positions of the agent to entities in the scene. More specifically, for each of the objects in the scene whose position is regarded as relevant, the vector between it and the agent is calculated and its length is normalized to the game area, given by a bounding box. Finally, and following the approaches in [17] and [18], we use a solution based on a semantic map given by a discrete categorization of the space and the elements around the agent to give it a certain level of spatial perception. This is defined by a 3-D grid of cube-shaped regions centered on the agent, with each of these regions describing the type of the object included in it by means of an integer value. The size of the regions, as well as the number of regions per dimension (s_x, s_y, s_z) and the entities detectable by the map can be configured by the developer. An example in our environment for a semantic map with a single vertical dimension can be seen in Fig. 4. Each cube-shaped region represents a

discrete categorization of the entity contained in it (e.g., enemies, switches, goals, etc).

As for the architecture of the neural networks used, we deploy structures with two fully connected layers, each with 512 hidden units. The first segment of the network's input is given by a one-hot-encoded vector $v_{\text{Automata}} \in \mathbb{R}^n$ with a length equal to the number of states n in the automaton, with all values set to 0 except for the one corresponding to the currently active state. Second, the entities and variables declared as relevant by the user v_{user} are appended to the input vector. The last part of the input of these layers is given by the state of the semantic map $M \in \mathbb{R}^{s_x \times s_y \times s_z}$, previously fed in parallel to a convolutional layer with output $x_M \in \mathbb{R}^d$.

VI. Experiments

We applied the proposed approach to multiple levels in the test environment described in Section IV, on each of which a design specification was formulated in TLTL. To evaluate the performance of our approach, we define these specifications to encompass several typical settings that designers may be interested in validating in their work, with use cases addressing a variety of challenges that can arise when training testing agents. In this environment, agents have a set of five discrete actions: move in the *X*-axis (none, left, and right), move in the *Y*-axis (none, backward, and forward), jump, crouch, and sprint.

A. Experimental Setup

For the experiments conducted in this work, we made use of the ML Agents library [30] with the PPO algorithm [29] to train agents and the reward function generated from the specification as detailed in Section 1 (learning rate $= 0.0003$, $\beta = 0.01$, $\epsilon = 0.2$, $\lambda = 0.95$, and $\gamma = 0.99$). The only parameters that were changed between levels of the experiment were the number of steps per episode, which is inherently dependent on the estimated length of each task, and the maximum number of steps per training session, typically linked to the complexity of the specification. It is worth noting that the number of steps per episode has a direct impact on the scale of the accumulated episodic reward due to the nature of the existential punishment in the algorithm. Setting a step number too close to the optimal value (which is unknown to the practitioner except for maybe some empirically recorded estimate) often results in reward values very close to 0, whereas increasing this margin allows for results closer to 1. Therefore, our primary concern is to achieve consistent positive values for the reward, and the magnitude of the reward beyond this threshold is not as crucial.

We compare our approach to an imitation baseline in which a human tester provides a set of demonstrations (at least $10 - 20$ complete episodes per level) satisfying the specification for each of the levels, which are then used to train an agent to learn to mimic the expert's behavior based on this dataset of traces. More specifically, we bootstrap a neural network with an architecture identical to the one of the TL model using behavioral cloning (first 5% of training steps), and then run training using a GAIL-generated reward signal (learning rate $= 0.0003$, encoding size $= 128, \gamma = 0.99$) and adding a terminal reward at the end of each episode of +1 if the specification was successfully completed and -1 otherwise. For each of the methods, we are interested in

1) the agent's success rate in satisfying the specification in the environment, calculated as the proportion of simulations, in which it achieves the level objective over 25 runs,
2) the agent's average progress in the specification's predicate, i.e, at which point in the specification it tends to be when the time allocated to the task runs out or when a safety condition is violated, and
3) the average reward assigned by the FSPA during the simulated episodes, which can be seen as a metric that aggregates the progress value with the "quality" of the generated traces in terms of speed and compliance with the safety conditions.

Note that these metrics are always computable even if they are not used to guide the training of an agent by RL. In fact, for each of the levels, we keep a record of the values of these metrics during demonstrations by human experts in order to provide a reference of the values that would be expected in a real player. In both approaches, training stops when it is detected that the model starts to satisfy the specification consistently (i.e., pass rate $= 1$ for the duration of a fixed number of episodes) or when a training time of 20 h is exceeded without this condition being met, after which the best model based on the pass rate metric is saved.

B. Regression Test Use-Case Evaluation

To evaluate the viability of our approach, we define a series of use cases for testing agents that address some of the typical challenges that can be encountered in modern video game scenarios when training policies that satisfy design specifications. On the one hand, we wish to verify that our methodology is able to generate policies in a reasonable time to validate the specification on the original environment. On the other hand, since the main purpose of these agents is to be used for regression testing, we focus on analyzing how each strategy reacts to common development scenarios that induce unexpected changes in previous levels. Thus, for each of the environments presented, we consider cases in which a designer modifies a shared asset in different scenes of the game to accommodate the design of later levels, and analyze the ability of each testing agent to adapt to the potential unexpected effects of these changes in their corresponding levels.

Where these edits do not completely invalidate the original specification, we expect agents to continue to be able to satisfy their predicates, possibly informing of meaningful variations in their performance. That is, if the agent is still able to validate the specification, but its reward metrics are noticeably altered, we can issue a warning that there is a chance that the gameplay experience of the level has been compromised. Predictably, there will be situations where the agent is no longer able to complete the task after the change, in which case we would like the test to be as informative as possible as to what may have caused this failure to meet the objective. Our intention will therefore be to ensure exhaustive coverage of the game design specifications, running periodic automated checks and reporting suspicious

TABLE I
QUANTITATIVE RESULTS OF OUR EXPERIMENTS

	Case 1									
	Pass Rate			Specification Progress			Automaton Reward			Time
	v_0	v_1	v_2	v_0	v_1	v_2	v_0	v_1	v_2	
Human	1.00	1.00	0.00	1.00 ± 0.00	1.00 ± 0.00	0.50 ± 0.01	0.63 ± 0.01	0.65 ± 0.02	-0.50 ± 0.01	-
Imitation	0.92	0.32	0.00	0.98 ± 0.05	0.79 ± 0.22	0.47 ± 0.07	0.50 ± 0.19	-0.09 ± 0.37	-0.53 ± 0.08	20h
Ours	1.00	0.90	0.00	1.00 ± 0.00	0.93 ± 0.21	0.49 ± 0.05	0.64 ± 0.01	0.39 ± 0.37	-0.51 ± 0.05	3.5h
	Case 2									
	Pass Rate			Specification Progress			Automaton Reward			Time
	v_0	v_1	v_2	v_0	v_1	v_2	v_0	v_1	v_2	
Human	1.00	1.00	0.80	1.00 ± 0.00	1.00 ± 0.00	0.93 ± 0.14	0.73 ± 0.01	0.74 ± 0.01	0.41 ± 0.40	-
Imitation	0.05	0.00	0.00	0.74 ± 0.27	0.49 ± 0.26	0.41 ± 0.22	-0.25 ± 0.27	-0.50 ± 0.27	-0.58 ± 0.22	20h
Ours	1.00	0.50	0.20	1.00 ± 0.00	0.85 ± 0.20	0.70 ± 0.25	0.74 ± 0.03	0.21 ± 0.55	-0.14 ± 0.51	2.7h
	Case 3									
	Pass Rate			Specification Progress			Automaton Reward			Time
	v_0	v_1	v_2	v_0	v_1	v_2	v_0	v_1	v_2	
Human	1.00	1.00	1.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	0.76 ± 0.01	0.73 ± 0.02	0.74 ± 0.08	-
Imitation	0.40	0.24	0.10	0.82 ± 0.37	0.80 ± 0.36	0.71 ± 0.24	0.01 ± 0.51	-0.11 ± 0.45	-0.24 ± 0.33	20h
Ours	1.00	0.93	0.47	1.00 ± 0.00	0.96 ± 0.17	0.89 ± 0.23	0.66 ± 0.01	0.64 ± 0.36	0.27 ± 0.46	6.8h

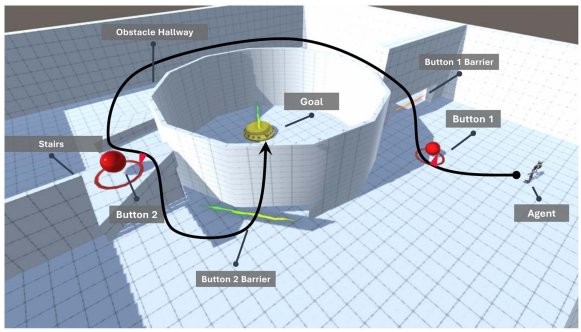

Fig. 5. Screenshot of the "Navigation" environment.

scenarios, these being the only ones that need to be manually reviewed by a human tester. Note that in the testing process the same agent is always used without retraining until a significant change is found.

Case 1—Validating navigation: One of the most common testing use cases is to check that the critical path of a level remains feasible from a navigability point of view. The classic way to manage navigation in modern games involves prebaking a navigation mesh and applying classical pathfinding algorithms on it, but this strategy is usually best suited for enemy movement in constrained environments, requires recalculation every time the level structure is modified, and is often too costly to adapt to environments with a high degree of freedom of movement. For this use case, we consider a first level that serves as a tutorial in our test environment, shown in Fig. 5, introducing players to the basic mechanics of the game. Here, the character must activate the switch on the right side of the image s_1 ("Button 1") to open the barrier on the right wall, which then allows passage through the gap by entering a crouched position. After going through an obstacle section and up a set of stairs, players must perform a careful jump to reach the switch on the left side of the image s_2 ("Button 2"), which in turn unlocks the exit e accessible by purposely falling down from the upper platform. In TL, this can be expressed via the following predicate:

$$\Diamond(\gamma_{s_1} \wedge \bigcirc(\Diamond(\gamma_{s_2} \wedge \bigcirc\Diamond\gamma_e))) \tag{4}$$

following the notation from (3).

Table I summarizes the results for this use case and the ones described hereafter. For each of the metrics mentioned earlier, we report the values produced with each training model in 25 simulations on the level in its original state (v_0) and after applying a subtle and a significant modification on assets shared with other levels (v_1 and v_2, respectively), together with the time needed to train a successful agent. Values are given as averages with their corresponding standard deviation.

In this environment, the first modification involves a slight alteration of the shape of the assets corresponding to the obstacles in the central corridor. This does not affect gameplay significantly, but it does require a minor adjustment to the route taken. The second variation represents an extensive tuning of the obstacle, stretching it so that it starts obstructing the path entirely, thus preventing the level from being completed. As shown in the table, after the first change the imitation model perceives a significant alteration in the level, with a sharply reduced performance after applying the change, while our model, despite showing a small drop in performance, remains at relatively stable values, with the test yielding a consistently positive result. This is a desirable property in testing: given subtle modifications that do not alter the gameplay of the level, a good testing agent should not warn against potential alterations (i.e., it should not throw false positives). On the other hand, both the human and the two models are unable to complete the level after the second change, with both models exhibiting similar results and reporting true positives. However, in the latter case, our model provides us with an additional tool to yield a more informative test result: since the FSPA logs are available during the executions, we are able to identify which state the agent was in before becoming locked in the environment, and how far away it was from moving on to the next one. As mentioned before, each state of an FSPA is labeled with the predicate representing what part of the specification remains to be fulfilled, so we can easily gain a preliminary but intuitive idea of what might have gone wrong and direct our attention to that point in the level during human review. Hence, another desirable property is for the agent's progress to be as close as possible to that of the human tester within the same

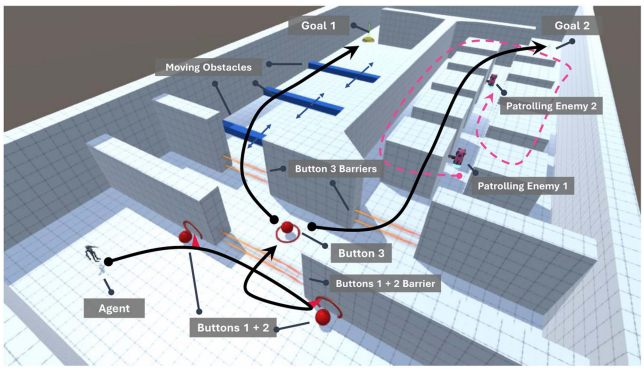

Fig. 6. Screenshot of the "*Branching Tasks*" environment.

situation, so that the result provides information that is as close as possible to a real test.

Case 2—Validating multistage environments: Another typical situation can be seen in environments, such as the one used as an example in Fig. 2, where the design specification requires a series of steps to be followed sequentially, potentially leading to stages where it may be necessary to take very different actions depending on which part of the test the player is in. Returning to the prior example with specification given in (3), the player is required to traverse the same spaces moving in opposite directions based on which switches they have already activated, while looking for ways to avoid detection by the patrolling enemy. Such scenarios can be problematic in machine learning, where it is often difficult for the agent to efficiently learn to discriminate between the stimuli needed to distinguish phases of the sequence where its behavior is likely to be significantly varied.

Here, both modifications are applied to the behavior tree that defines the AI of the enemy. The first situation involves only parameter adjustments to improve the responsiveness of the enemy at later levels, while the second adds an additional behavioral branch in an attempt to make the sections involving tight spaces more compelling. From the results in the table we can see that, although our approach suffers a drop in performance, it keeps being able to regularly complete the level after the first change, indicating that the specification is still valid, whilst requiring human review to verify that the user experience is intact. The second, seemingly innocuous change to the level that motivates the asset modification ends up making it noticeably harder: the human no longer has a perfect success rate, and our agent suffers an even more substantial drop in performance, but continues to indicate that the specification is satisfactory, albeit in need of manual review. Moreover, despite the low success rate, progress remains high and close to that of the human tester. The agent trained by imitation does not manage to complete the specification in any of these cases, making it unfit for testing.

Case 3—Validating branching environments: The third and final level of the experiments, depicted in Fig. 6, provides an example of a complex environment where the agent is encouraged to follow very different behavior flows depending on random conditions. In this scenario, the design specification requires the player to start by sequentially activating the switches in the left room of the level (s_1 and s_2) to disable the first laser barrier. Subsequently, the switch placed behind this barrier s_3 must be activated, which randomly unlocks one of the barriers on its sides. Depending on which section is unlocked at this point, the character must address a different problem: if access to the left section is granted (σ_{left}), the player must navigate a corridor with moving obstacles, timing their jumps correctly to make progress; conversely, if the right section is unlocked (σ_{right}), the player must face an area with grid-based obstacles patrolled by two enemies thoroughly exploring the intersection points. In both cases, the goal is to reach the exits placed at the end of their respective corridors ($\gamma_{e_{\text{left}}}$ and $\gamma_{e_{\text{right}}}$). In terms of TL, this is modeled by the following specification:

$$\Diamond(\gamma_{s_1} \wedge \bigcirc(\Diamond(\gamma_{s_2} \wedge \bigcirc\Diamond(\gamma_{s_3} \wedge \bigcirc\Diamond((\sigma_{\text{right}} \Rightarrow \gamma_{e_{\text{right}}})) \wedge (\sigma_{\text{left}} \Rightarrow \gamma_{e_{\text{left}}}))))) \wedge \Box\pi. \quad (5)$$

The first modification here involves slightly tweaking the path of the moving obstacles to include some vertical elevation, as well as the same changes to the enemy AI that were applied in the first variation in case 2; this does not induce substantial variations in gameplay from a human perspective, but it does force subtle adjustments to the trajectories and jumps to be taken. The second modification applies the same addition to the behavior tree described for the previous case to the enemy AI; this ends up requiring a slightly more cautious strategy to traverse the right section of the level, as enemies are more exhaustive in exploring tight spaces. Looking at the results in the table, we can see that our model only perceives a subtle change in v_1, as expected, while the second change results in a substantial drop in agent performance. This is actually expected and positive, as although the difficulty of the level has not been significantly increased, the way in which it must be dealt with does change, requiring human review to ensure that the design continues to be adequate. Meanwhile, the imitation model has very limited success rates from the outset and progressively lower success rates with each asset update, rendering it once again unreliable for testing.

Overall, these experiments highlight the preliminary potential of our strategy to generate agents with sound regression test properties, with manageable training times especially considering that the ultimate goal is to maintain continuous validation throughout development rather than interactive in-situ validation, and more informative than nonspecification-based agents. The fact that our agents do not always maintain the original levels of performance in the face of subtle changes can be viewed as a virtue as evidenced in case 3, where potentially unexpected variations in the way the level was approached could have gone unnoticed by different human testers playing it with no awareness of how it was handled in its original version. Lastly, an advantage of RL-based systems is that once the presence of a substantial change is established, it is possible to automatically launch a new training session to fine-tune the behavior to the change, so that when several parts of the game are affected, it is no longer necessary for human testers to rerecord traces for all conflicting scenarios.

VII. Conclusions and Future Work

In this article, we introduce a RL approach to perform automated regression tests in the context of the validation of design specifications throughout the development of a game. We propose a new algorithm for generating dense rewards from descriptions of the tasks to be performed expressed through a TL language (TLTL), addressing the issue of reward shaping and only requiring the designer to provide a specification of what they would like to validate within an environment in order to produce a reward function that guides the agent to a behavior that satisfies it. This algorithm is based on the idea of specification progress and is naturally interpretable by a human practitioner. To investigate the performance of our strategy, we design a series of experiments aimed to produce agents capable of satisfying different specifications in three design validation use cases in a 3-D stealth game environment. For each of these cases, we introduce modifications on common project assets that could potentially affect gameplay of the original levels and analyze the ability of the produced agents to both cope with and report possible unexpected consequences on player experience and design soundness. Our experiments offer preliminary results on how this type of approach can be more effective than other techniques based on IL when running periodic regression tests during development.

Our future work focuses on developing regression test analysis methods to improve the information provided by agents, aiding quality control. Using formal language and specification-based models, we aim to generate meaningful descriptions of gameplay variations detected by testing agents. In addition, we plan to study the usability of our system to ensure smooth integration into team workflows. This includes evaluating how easily developers can implement these methods (considering here conceptual complexity and intrusiveness of our framework's ideas, and specially the potentially challenging adoption of TLTL as a design specification language), as well as new ways to deal with computational overhead and deployment logistics, which are often problematic side-effects of using RL-based approaches in real environments.

References

[1] S. Stahlke, A. Nova, and P. Mirza-Babaei, "Artificial players in the design process: Developing an automated testing tool for game level and world design," in *Proc. Annu. Symp. Comput.-Hum. Interact. Play*, 2020, pp. 267–280, doi: 10.1145/3410404.3414249.

[2] G. Lovreto, A. T. Endo, P. Nardi, and V. H. S. Durelli, "Automated tests for mobile games: An experience report," in *Proc. 17th Braz. Symp. Comput. Games Digit. Entertainment*, 2018, pp. 48–488.

[3] S. Iftikhar, M. Z. Iqbal, M. U. Khan, and W. Mahmood, "An automated model based testing approach for platform games," in *Proc. ACM/IEEE 18th Int. Conf. Model Driven Eng. Lang. Syst.*, 2015, pp. 426–435.

[4] S. Stahlke, A. Nova, and P. Mirza-Babaei, "Artificial playfulness: A tool for automated agent-based playtesting," in *Proc. Extended Abstr. CHI Conf. Hum. Factors Comput. Syst.*, 2019, pp. 1–6.

[5] C. Gordillo, J. Bergdahl, K. Tollmar, and L. Gisslen, "Improving playtesting coverage via curiosity driven reinforcement learning agents," in *Proc. IEEE Conf. Games*, 2021, pp. 1–8. [Online]. Available: https://ieeexplore.ieee.org/document/9619048/

[6] J. Bergdahl, C. Gordillo, K. Tollmar, and L. Gisslen, "Augmenting automated game testing with deep reinforcement learning," in *Proc. IEEE Conf. Games*, Aug. 2020, pp. 600–603. [Online]. Available: https://ieeexplore.ieee.org/document/9231552/

[7] S. Agarwal, C. Herrmann, G. Wallner, and F. Beck, "Visualizing AI playtesting data of 2D side-scrolling games," in *Proc. IEEE Conf. Games*, 2020, pp. 572–575. [Online]. Available: https://ieeexplore.ieee.org/document/9231915/

[8] S. Ariyurek, A. Betin-Can, and E. Surer, "Automated video game testing using synthetic and humanlike agents," *IEEE Trans. Games*, vol. 13, no. 1, pp. 50–67, Mar. 2021. [Online]. Available: https://ieeexplore.ieee.org/document/8869824/

[9] P. Gutierrez-Sanchez, M. A. Gomez-Martin, P. A. Gonzalez-Calero, and P. P. Gomez-Martin, "Reinforcement learning with temporal logic specifications for regression testing NPCs in video games," in *Proc. IEEE Conf. Computatonal Intell. Games*, 2023, pp. 1–8.

[10] X. Zhao and M. Campos, "Reinforcement learning agent training with goals for real world tasks," 2021, *arXiv: 2107.10390*.

[11] Unity-Technologies, "Unity real-time development platform 3D, 2D VR & AR engine," 2022. [Online]. Available: https://unity.com/

[12] C. Politowski, Y.-G. Guéhéneuc, and F. Petrillo, "Towards automated video game testing: Still a long way to go," in *Proc. 6th Int. ICSE Workshop Games Softw. Eng.: Eng. Fun, Inspiration, Motivation*, May 2022, pp. 37–43, doi: 10.1145/3524494.3527627.

[13] M. Ostrowski and S. Aroudj, "Automated regression testing within video game development," *GSTF J. Comput.*, vol. 3, no. 2, Jul. 2013, Art. no. 10, doi: 10.7603/s40601-013-0010-4.

[14] C. Holmgard, M. C. Green, A. Liapis, and J. Togelius, "Automated playtesting with procedural personas through MCTS with evolved heuristics," *IEEE Trans. Games*, vol. 11, no. 4, pp. 352–362, Dec. 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8295256/

[15] M. Bain and C. Sammut, "A framework for behavioural cloning," in *Proc. Mach. Intell.*, 1995, pp. 103–129.

[16] J. Ho and S. Ermon, "Generative adversarial imitation learning," in *Proc. 30th Int. Conf. Neural Inf. Process. Syst.*, 2016, pp. 4572–4580.

[17] A. Sestini, L. Gisslen, J. Bergdahl, K. Tollmar, and A. D. Bagdanov, "Automated gameplay testing and validation with curiosity-conditioned proximal trajectories," *IEEE Trans. Games*, vol. 16, no. 1, pp. 113–126, Mar. 2024. [Online]. Available: https://ieeexplore.ieee.org/document/9970382/

[18] A. Sestini, J. Bergdahl, K. Tollmar, A. D. Bagdanov, and L. Gisslén, "Towards informed design and validation assistance in computer games using imitation learning," in *Proc. IEEE Conf. Games*, 2023, pp. 1–8.

[19] S. Ross, G. J. Gordon, and J. A. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," in *Proc. 14th Int. Conf. Artif. Intell. Stat.*, 2011, pp. 627–635.

[20] A. Tucker, A. Gleave, and S. Russell, "Inverse reinforcement learning for video games," Oct. 2018 https://arxiv.org/abs/1810.10593

[21] H.-C. Liao, "A survey of reinforcement learning with temporal logic rewards," 2020. [Online]. Available: https://mediatum.ub.tum.de/doc/1579215/document.pdf

[22] I. Buzhinsky, "Formalization of natural language requirements into temporal logics: A survey," in *Proc. IEEE 17th Int. Conf. Ind. Informat.*, 2019, pp. 400–406.

[23] X. Li, C.-I. Vasile, and C. Belta, "Reinforcement learning with temporal logic rewards," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2017, pp. 3834–3839. [Online]. Available: https://ieeexplore.ieee.org/document/8206234/

[24] Y. V. Pant, H. Abbas, and R. Mangharam, "Smooth operator: Control using the smooth robustness of temporal logic," in *Proc. IEEE Conf. Control Technol. Appl.*, 2017, pp. 1235–1240.

[25] X. Li, Z. Serlin, G. Yang, and C. Belta, "A formal methods approach to interpretable reinforcement learning for robotic planning," *Sci. Robot.*, vol. 4, no. 37, Dec. 2019, Art. no. eaay6276, doi: 10.1126/scirobotics.aay6276.

[26] Z. Komárková and J. Křetínský, "Rabinizer 3: Safraless translation of LTL to small deterministic automata," in *Proc. Int. Symp. Automated Technol. Verification Anal.*, 2014, pp. 235–241.

[27] K. Y. Rozier, "Explicit or symbolic translation of linear temporal logic to automata," Ph.D. dissertation, Rice Univ., Houston, TX, USA, Jul. 2013. [Online]. Available: https://scholarship.rice.edu/handle/1911/71687

[28] P. Gutiérrez-Sánchez, M. A. Gómez-Martín, P. A. González-Calero, and P. P. Gómez-Martín, "Liquid snake: A test environment for video game testing agents," in *Proc. Actas del I Congreso Español de Videojuegos*, Madrid, Spain, 2022, pp. 1–12. [Online]. Available: https://ceur-ws.org/Vol-3305/paper7.pdf

[29] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017, *arXiv:1707.06347*.

[30] A. Juliani et al., "Unity: A general platform for intelligent agents," 2020, *arXiv: 1809.02627*.