Using an Agent-Based Approach for Robust Automated Testing of Computer Games[*]

Samira Shirzade-hhajimahmood
Utrecht University
the Netherlands
S.shirzadehhajimahmood@uu.nl

I. S. W. B. Prasetya
Utrecht University
the Netherlands

Frank Dignum
Umeå University
Sweden

Mehdi Dastani
Utrecht University
the Netherlands

Gabriele Keller
Utrecht University
the Netherlands

ABSTRACT

Modern computer games typically have a huge interaction spaces and non-deterministic environments. Automation in testing can provide a vital boost in development and it further improves the overall software's reliability and efficiency. Moreover, layout and game logic may regularly change during development or consecutive releases which makes it difficult to test because the usage of the system continuously changes. To deal with the latter, tests also need to be robust. Unfortunately, existing game testing approaches are not capable of maintaining test robustness. To address these challenges, this paper presents an agent-based approach for robust automated testing based on the reasoning type of AI.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Interactive games**.

KEYWORDS

Robust automated testing, agent-based automated testing, automated testing of computer games

ACM Reference Format:
Samira Shirzadehhajimahmood, I. S. W. B. Prasetya, Frank Dignum, Mehdi Dastani, and Gabriele Keller. 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 (A-TEST '21), August 23–24, 2021.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3472672.3473952

1 INTRODUCTION

Software testing, or more generally Quality Assurance (QA), plays an important role in software development for ensuring that a produced system meets different requirements including functional, performance, reliability, and realises a high quality product. In the hugely competitive market, delivering high quality software can attract users and gain loyal clients which can have a dramatic impact on the software success. However, testing is mostly a manual process, time-consuming, and intensive that moreover has to be repeated for every modification to the system, therefore, makes it expensive. A recent industry [1] indicated that the IT budget allocated to QA, including testing, is estimated to be about one-quarter of the total budget. Test automation can yield significant savings in costs, lead time, and improve consistency and performance.

The computer games industry has seen the emergence of advanced 3D games. These are frequently complex software systems due to their high level interactivity and increased realism. Interactive entities of a 3D game are often very hard to test in isolation (unit test). For example, suppose new interactive entities have been added into the game and we want to verify that they interact correctly with the world of the game. Entities' behavior is often context dependent; the context is often hard for testers to recreate without the help of the visualization provided by the game world itself. Therefore, system-level testing is a critical phase in game testing because it is then where we would test the overall interaction of components (how the components interact with one another and with the system as a whole).

The huge interaction space makes automated testing for such games very challenging. Additionally, changes during development-time makes testing even harder. For example, the world layout and the game logic may be altered regularly during the development or between consecutive releases. This complicates testing because the usage of the system continuously changes. Therefore, in addition to automation, 'test robustness', that is, how flexible a test is in coping with constant development-time changes, is another key measure for the engineering of modern games. In previous work we have addressed test automation in game testing [28]. This paper will focus on the robustness problem.

In this work, we build on the BDI agent-based testing framework iv4XR [28]. The choice for an agent-based approach is appropriate to deal with the high-level of interactivity and complexity of computer games. Properties such as autonomy and reactivity [9] allow agents to perform actions independently in an environment over which they have control and observability. Goal-based behavior and the possibility to do autonomous planning and react to environmental changes make the approach capable of dealing with highly interactive systems. Autonomous planning is achieved through reasoning, e.g. by implementing the so-called Belief-Desire-Intention (BDI) model [31]; which is a model of rational agents. In this model, the agent has their own motivation (goals, desire in the model) and a set of actions, to compose a plan (intention) towards the goals.

[*]This work is supported by EU Horizon 2020 grant 856716 project iv4XR (Intelligent Verification/Validation for Extended Reality Based Systems).

This paper performs a two-stage study addressing the robustness of agent-based automated game testing against typical development-time changes. In the first stage, we conduct an empirical study towards the robustness of agent-based game testing. Robustness is conjectured by [26], but it was not further studied or experimented. In this paper, we show that the approach is indeed **robust** against location and layout changes, though not against logic changes. At the second stage, we extend the iv4XR's previously static goal structure with new constructs allowing agents to dynamically calculate and add new sub-goals as they go about, trying to solve a given task. This leads to higher test robustness, allowing them to deal with logic changes. This makes the task more flexible in the presence of development changes.

Paper structure. This paper is organized as follows. Section 2 briefly introduces iv4XR agent-based testing approach. Section 3 discusses how our agents can facilitate robust testing. Section 4 explains how we extend the iv4XR framework to achive a higher level of robustness by adding dynamic goals. Section 5 discusses experiments we conducted to evaluate the robustness and effectiveness of our approach. Section 6 and 7 cover related and future work, respectively.

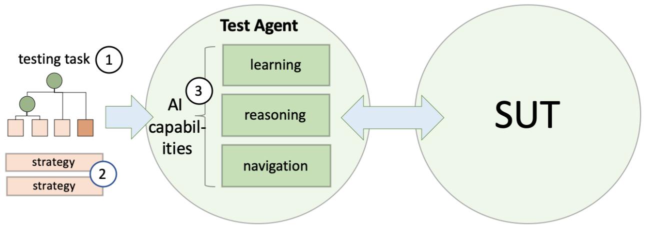

Figure 1: Architecture of typical iv4XR agents deployment.

2 PRELIMINARY: AGENT-BASED TESTING

We use iv4XR[1] a Java multi-agent programming framework for game testing. Figure 1 illustrates its architecture. Every iv4XR agent is typically used to control one in-game entity. For example, an agent can take the role of a player character in the game which can react to the game. The framework is inspired by the popular BDI concept of agency [15], where agents have their belief which represents information the agent has about its current system under test (SUT) and their own goals representing their desire. The agent can decide which current goal to pursue ('intention' in BDI) and which action to perform on the basis of its current goal. To achieve a goal, the agent can perform various actions. An *action* is typically a primitive function provided by the game to change its state. For example, an action could be moving a player character in a certain direction for some small distance.

To test something the agent must be given a testing task. Abstractly, a *testing task* can be formulated as follow:

$$\underbrace{\phi}_{\text{situation}} \Rightarrow \underbrace{\psi}_{\text{assertion}} \tag{1}$$

where ϕ is a state predicate describing a set of game states considered as a goal situation that the agent should establish to check its correctness, e.g. that it is standing close to a door; and ψ is a state predicate that is expected to hold on all of the states that satisfy ϕ, e.g. that the said door should be open. The term "assertion" refers to a property in a game world that we want to assert on.

We treat a testing task, more specifically the ϕ-part, as a goal that a test agent wants to automatically achieve (and thus providing test automation). An example of a goal is to approach an in-game entity with the given *id* until the latter becomes visible; Fig. 2. Since solving or achieving a goal can be non-trivial for an agent, we also accompany the goal with a 'tactic' as shown in Fig. 2 specified with *toSolve*; we abstract away some technical details.

```
1 IsVisible(id) = goal("This entity is visible")
2     .toSolve((BeliefState B) → check entity is
       visible)
3     .withTactic(
4        FIRSTof(
5           navigateTo(id),
6           explore(),
7           ABORT()))
8     .lift();
```

Figure 2: Defining a parameterized goal to approach an in-game entity with the given *id*, until the entity becomes visible to the agent.

A tactic is a way to hierarchically combine actions, which is more powerful, as a means to achieve a goal. For example, if we consider $T_1, ..., T_n$ as actions or tactics, the composition T = **ANYof**$(T_1, ..., T_n)$ is a tactic that executes one of the enabled T_i randomly. **ANYof** is also called a tactic-combinator, used to impose high-level control over a set of actions/tactics. **FIRSTof**$(T_1, ..., T_n)$ is another tactic combinator, expressing a priority over $T_1, ..., T_n$: it invokes **the first** enabled one in the sequence. For example, the tactic shown in Fig. 2 to solve its goal (line 2) uses this combinator (line 4) to combine other tactics. One of the used tactics is navigateTo(e) (line 5), intended to drive the agent to move towards the given entity e, guided by some path finding heuristic we will explain later.

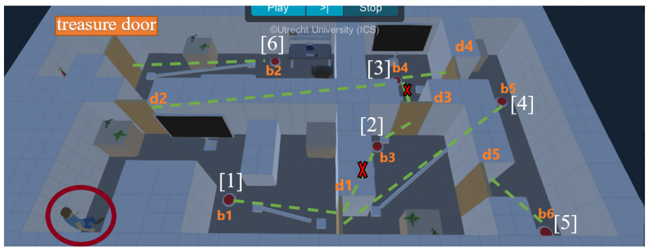

Figure 3: A screenshot of a level in a game called Lab Recruits. The level's objective is to reach a treasure door. In order to achieve this objective, a correct sequence of interactions with buttons should be done to open the right doors; this sequence is indicated by the white numbers. Toggling a button will toggle the state of all doors connected to it; indicated by green lines. The red crosses show that interacting with a button b closes a door d.

A testing task can be very hard for an agent to achieve/solve directly. Consider a simple 'game level' shown in Fig. 3, taken from a maze-like 3D game called Lab Recruits [2]. The buttons and doors in the level form a puzzle. It would be a bug to have a puzzle in the game that can not be finished. Let us then consider:

Example 2.1. Testing task T_1: to verify that the level in Fig. 3 can indeed be finished; so, essentially, verifying that the treasure door can be opened.

[1]https://github.com/iv4XR-project

[2]https://github.com/iv4XR-project/labrecruits

Table 1: Domain specific language of iv4XR framework. Consider $G_1, ..., G_n$ as a goal given to an agent. Using goal combinator SEQ requires the subgoals to be solved sequentially. When given to the goal combinator $FIRSTOF$, the agent will start from the first goal G_1 to achieve it and it will go to the next goal if the current one fails.

goal structure	::=	**SEQ**(goal structure, goal structure,...)
	|	**FIRSTof**(goal structure, goal structure,...)
	|	**WHILEDO**$(predicate, goal\ structure)$
	|	goal.**lift()**
goal	::=	**goal**(name).**toSolve**(predicate).**withTactic**(tactic)
tactic	::=	**SEQ**(tactic,tactic,...)
	|	**FIRSTof**(tactic, tactic,...)
	|	**ANYof**(tactic,tactic,...)
	|	**ABORT()**
	|	action.**lift()**
action	::=	**action**(name).**do**(action expression).**on**(predicate)
	|	**action**$(name)$.**addAfter**$(goalstructure)$

To do this, the agent needs to follow the sequence of steps indicated by white numbers in Fig 3. Without this knowledge, directly solving the testing task will be exponentially expensive. We can however set intermediate goals (subgoals) for helping the agent to solve a given goal. Each subgoal, e.g. approaching a button, is simple enough to be solved automatically. More generally, rather than providing a single goal, an agent can be given a *goal structure* expressing a complex goal, e.g. it can be a set of goals that have to be achieved sequentially. More generally, it is a tree with goal-combinators as nodes and goals as leaves, similar to *Goal-Plan* tree [39]. The goals at the leaves are an ordinary goals that specify a set of desired states. Table 1 shows an overview of iv4XR Java-embedded domain specific language (DSL) [28] for formulating the aforementioned concepts: actions, tactics, goal, and goal structures.

3 ROBUST TESTING

During development-time, changing the design of a game is very common. For example, level designers may change the physical locations of entities in the game, or even the layout of a game. Imagine in Fig. 3, the location of button$_1$ is changed. It is essential to execute related testing tasks again to be sure that the change does not break any correctness assertion. In the traditional record and replay testing [37], the tests would take the form of recorded sequences of primitive actions to get to button$_1$'s old location; when replayed, they would obviously now break. Developers then need to re-construct their tests based on the new design; which means more overall effort and costs.

Below are typical development-time changes; for each we will mention the feature we study and introduce to make tests robust against these changes. The features will be discussed further in the subsections that follow, in Section 4.

Location Change. Entities location may be changed by the developers, such as in the previous example with button$_1$. To prevent tests from breaking, routes to entities are not fixed upfront. Instead, we employ automated world exploration and navigation to entities as tactics.

When a test is re-run the agent first looks at its belief; if a path leading to the current goal entity can be calculated, it will navigate towards the entity. Otherwise, if it cannot find the entity because the location has changed, it will instead *explore* the environment to find a new path to the entity and update its belief. The test will not break as long as the entity remains spatially reachable, by improving the robustness of the test. Subsection 3.1 gives further elaboration of these tactics.

Layout change. This refers to changes to the shape of the game world. Imagine the designers of the level in Fig. 3 decided to add an obstacle between button$_1$ and door$_1$; the latter has to be opened to reach the level's final goal. When executing the testing task T_1 from Example 2.1 again, if after interacting with the button the agent just executes the same sequence of actions as it did before, it will not be able to move toward door$_1$ because of the newly added obstacle. However, since routes to entities are not fixed upfront, the same solution to deal with location change above (i.e, auto navigation and exploration) will allow the agent to find a new route around the obstacle; thereby, adapting to the new layout.

Logic change. The game's logic may also be changed by level designers. For example, a button b in Fig 3 which was connected to a door d can be changed to connect to d' instead. Robustness against such a change cannot be obtained through pure navigation based tactic. We introduce a feature to let the agent dynamically extend its own goal structure. Section 4 will explain how this is employed to deal with logic changes in the system under test.

Additionally, we also want to formulate testing tasks at a high-/abstract level. This makes it easier for the tester, but additionally, from the agent's perspective, a more abstract test depends on less information and hence also imposes less constraint for on underlying tactics in adapting the agent behavior to make the test robust.

The sections below elaborates the above mentioned features which we implemented to improve test robustness.

3.1 Auto-Navigation and Exploration

As pointed out above, the ability to find, and navigating to, an entity in a game, door$_1$ in the task T_1 in Example 2.1 is crucial for test robustness against location and layout changes. However, this could be quite complicated for a test agent because it simulates an actual human user, typically has limited visibility and interactability.

Limited visibility means that information which the test agent receives from the game depends on its visibility range and physical constraints such as that the agent can not see through solid obstacles. Note that this also implies that when an entity's state changes, the agent may not immediately observe the change (and may not even be able to ever observe it). Also, what the agent does can affect the information that it can sense from the game. For example, by closing a door, the agent can no longer see what happens behind the door. So, if the agent needs that information, it would first need to *explore*, e.g. to find an alternate path to get behind that door. Another type of physical constraints is *limited interactibility*: for interacting with an entity the agent typically should be close enough to the entity. It will also not be able to interact with the entity if there is a solid obstacle between them.

We employ tactics to enable agents to cope with these limitations. As an example, imagine a task T_2 where a test agent has to check the state of door$_1$. Although the task is simple, it can not actually be directly solved; the test agent would need to navigate from its initial position to door$_1$. However, the agent cannot initially see where the door is located (its sight is blocked by a wall, see Fig. 3) therefore has no clue how to get there. To solve this, the agent first needs to explore the level to learn the game's spatial layout to find a path to door$_1$. This is done by the tactic 'explore()' in the example in Fig. 2 (line 6). The tactic will locate the nearest reachable navigation node which the test agent has not discovered yet, and drive the agent to go there. In each step, the agent will update its belief with what it observes and it will continue to explore, in principle, until there is no undiscovered (and currently reachable) node.

The tactic navigateTo(id) in line 5 in Fig. 2 handles the situation when the target entity is present in the agent's belief, and furthermore the agent believes that there is a path to the entity. The tactic uses a path finding algorithm (A*) to guide the agent towards the entity. The agent uses this tactic in combination with the aforementioned 'explore()', using the **FIRSTof** combinator (Fig. 2 line 4) to first explore when it cannot immediately navigate, until navigation becomes possible. Note that exploration and navigation should not assume that the position of entities are fixed upfront, since this would make the resulting test fragile with respect to development time changes. So instead, we assume that only the entities' ids are known, and that they are unique.

As mentioned above, the tactic navigateTo(id) needs 'path finding'. Although some games may have an implementation of a path finding algorithm, e.g. to control non-player mobile entities, repurposing this implementation for testing may be difficult as it may lack the needed flexibility; e.g. avoiding obstacles in a fixed pattern could already be hard wired in the game's native path finding. Therefore, to maximize its flexibility, our agents have their own path finding module.

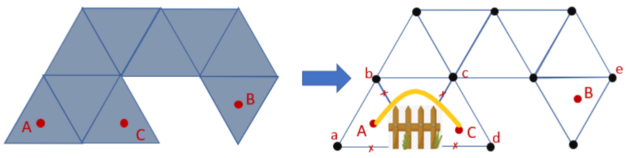

Figure 4: A mesh, and the resulting navigation graph (right)

Some of the basic terms that are frequently used in the description of path finding are *navigation-mesh*, *navigation graph* and *path finder*. A path finder is a chosen graph-based path finding algorithm, e.g. A* [14, 19]. A navigation-mesh represents surfaces in a 3D environment that are walkable by agents. An example is shown in Fig. 4, along with the corresponding navigation graph G obtained from the mesh. To navigate between locations in the walkable part of the world, e.g. from A to B, firstly, vertices in G, e.g. a and e, which are closest to these positions should be found. Then, G.pathfinder(a, e) can be invoked to give an agent a path.

Dealing with dynamic obstacles. In many computer games, there are also obstacles that may dynamically change the navigability of the environment, e.g. a dynamic obstacle such as a fence between two locations A and C, as shown in Fig. 4. Closing this fence would block some paths from A to C. We propagate the state change by marking the edges that intersect with the fence; in Fig. 4, these are shown by the red crosses. These edges are thus blocked. A subgraph G' can be (dynamically) obtained by removing the marked edges. We then can invoke G'.pathfinder() to obtain an unblocked path. The test agent should always check the state of dynamic obstacles to update the G' [26].

3.2 High-Level Testing

Our agent-based approach provides an implementation of a testing task at a high level, where testers do not need to be aware of the underlying programming of the tactics. Below we will explain how our approach enables high level formulation of testing tasks.

In a traditional setup, e.g. using a game testing framework like the Unity Test-Framework[3], the test agent would have to be guided to the goal entity step by step, which means much programming effort and time. In contrast, in iv4XR, we have navigation and exploration tactics (Subsection 3.1) to *automatically* guide the test agent to travel from any position A to B. Programming such details are thus detached from the testers' concern, who then can focus on formulating such testing tasks in terms of goals. In particular, tactics are hidden from the testers' concern.

More complicated testing tasks can be formulated purely at the goal level. To show this, consider again the task T_1 in Example 2.1. This task can be formulated as shown in Fig 5. The agent cannot directly solve T_1, so we break this task into subgoals. This decomposition can be directly translated into iv4XR testing task as shown in Figure 5. Although in this example we do give the agent the solution for solving the task, notice that the testers do not have to manually program the underlying 3D navigation; the latter approach is not only time consuming, but also very fragile.

We have shown that testers can formulate testing tasks in terms of goals. Tactics are hidden from them, but indeed someone needs to provide the tactics, with which goals will be solved. Tactics are likely to be quite game specific, so we can't provide a library that would work for all. However, once defined, the tactics can be used over and over again for solving goals and goal structures.

```
1 var testingTask = SEQ(
2   isInteracted("button1"),
3   isChecked("door1"),
4   isInteracted("button3"),
5   isChecked("door3"),
6   ...
7   isChecked("treasureDoor"),
8   isOpen("treasureDoor")
9 );
```

These are subgoals to solve the ϕ part in Eq.(1).

assertion part in Eq.(1).

Figure 5: A testing task to verify that the level in Fig 3 can be finished. isInteracted($button_1$) **constructs a goal structure that would be solved if the agent manages to interact with the** button$_1$**; indeed, it first needs to move to the button. Line 3 checks the state of the door, whether it is open.** isOpen **checks the assertion that the treasure door should be open.**

4 DYNAMIC GOAL STRUCTURE

In the previous section we discussed how to deal with location and layout changes to obtain robust automated testing. We will now show an extension that would enable our agents to handle logical change during development-time as well. As an example, suppose a door d in the level in Fig. 3, which was connected to a button b, is now connected to another button in the new version of the level. Consequently, the logic of the level also changes, as

[3]https://docs.unity3d.com/Packages/com.unity.test-framework@1.1/manual/index.html

interacting with b will not open d anymore. Suppose the change is intentional (so, not an error). To prevent the original test from breaking we will extend it with a fragment of adaptive logic, e.g. through appropriate (parameterized) goal structure: if interacting with a preset button b does not open the corresponding door, the agent should now look around to find another button b' that can open the door. This b' is not known upfront though, so this involves dynamically introducing new goal structures to search for it.

In the initial setup of iv4XR, an agent can dynamically insert a new goal structure to the given testing task by invoking the action addAfter. In our example, this new goal would be a new target button that the agent should try. Note that if this new button still can not open the door d, the addition of the new goal needs to be repeated. To do this, a new goal-combinator is introduced to iv4XR's DSL called WHILEDO; Fig 1.

Algorithm 1 Dynamic Search Strategy

```
1: enhance(b, $\phi_d$) =
2:   FIRSTof(
3:     SEQ(isInteracted(b), $\phi_d$),
4:     WHILEDO(there is an untouched interactable $b'$,
5:       SEQ(
6:         FIRSTof(
7:           addAfter the goal "isInteracted($b'$)" ) ,
8:         $\phi_d$ )
9:     ))
```

In the testing task shown in Fig 5, if interacting with the associated button can not open the door d, the test fails. However, to enhance robustness, the goal combinator WHILEDO is invoked in Algorithm.1 (line 4). Inserting a new goal will be repeated until the agent can open the door d, or until it has tried all currently reachable buttons. The number of goals to be added is not known upfront. Hence, the first subgoal of this combinator is a predicate to check whether the agent is allowed to continue adding a new goal. The predicate is checked in each iteration of this combinator.

To get some insight on the algorithm, consider again the above mentioned example. To open door d, button b first needs to be toggled; this is specified by the sequencing combinator SEQ in line 3 in Algorithm 1, where ϕ_d is here the goal that the door d is open. If that fails, the WHILEDO combinator is invoked with a predicate to check buttons' status to know is there any untouched button to interact with; line 4. Based on the predicate's result, it decides to insert, dynamically, a new goal structure; or to terminate the repetition; lines 5-8. The action addAfter adds a new goal to look for a new button b' to interact; line 7. Until achieving the goal ϕ_d, or until all buttons have been touched the sequence will be repeated. The algorithm may indeed tolerate a logical change that is actually unintended (error). However, since it is an extension of the original testing task, it would still know if the logical change breaks the base task and can mention this as a warning.

5 EXPERIMENT

To evaluate robust automated testing in the proposed framework, a set of experiments are required. We use the Lab Recruits game as the case study. Fig. 3 showed a screenshot of this game; as mentioned, it is a maze-like 3D game[4]. The game is also an AI-gym (in our case we want to evaluate testing AI), comparable to OpenAI Gym or Unity Obstacle Tower gym.The latter two are more suitable for vision-based AI, whereas our approach is based on structural observation on the game world, which they do not provide. Lab Recruits allows game-levels to be custom made, allowing us to create controllable environments for experiments.

[4]https://github.com/iv4XR-project/labrecruits

Research Question. Can our agent-based approach be used to improve the test robustness of computer games?

Towards answering this RQ, we created Lab Recruits levels of various complexity (Table 2, we will explain them later) and set our agent to test them. The testing task's objective posed for all these levels is to check whether a door marked as the treasure door can be opened and reached. Similar to the task $T1$ in Example 2.1, reaching the treasure door requires the agent to do a certain sequence of interaction with buttons and checking doors. We first assume the solving sequence is known to the developers. Later, when the levels are mutated, the agent will have to find the new solution by itself.

We consider two different types of automated tests;

In $Test_{\text{base}}$ we turn the solving sequence into the corresponding testing task. This is similar to the code in Fig 5. So, $Test_{\text{base}}$ would contain subgoals for checking the state of relevant doors. In each, the agent first interacts with a preset button known to be connected to a door d. Then it checks d's state. If it is open the agent proceeds to the next subgoal, which is opening the next door by interacting with the associated button. Otherwise, it fails.

$Test_{\text{dynamic}}$ extends $Test_{\text{base}}$ by incorporating extra robustness. In section 4, we discussed how dynamically adding a new goal structure can help an agent to cope with logical changes.

Levels. Three different levels of the game Lab Recruit are created, with the same underlying design; Fig. 2. The underlying design has two rooms, connected with doors in between. Each level has different complexity for an agent towards solving the corresponding testing task. $Level_2$ and $Level_3$ have more buttons than in $Level_1$; more effort in finding a new solution when executing $Test_{\text{dynamic}}$ would be spent by the agent. Also, they have a backtracking scenario: after entering the second room the agent needs to back to the first room for the purpose of interacting with a button connected to the treasure door. $Level_1$ has a move forward scenario: the agent can open the doors by interacting with the buttons *in the same room*. We also say that $Level_2$ has 'backtrack depth' 1, whereas $Level_1$ has depth 0. In $Level_3$, the developers incorporate some obstacles so that some buttons may not be immediately visible to the agent which affect the level's complexity.

Evaluating robustness. To evaluate the robustness of our tests, a mutation test [13] is applied against the aforementioned $Test_{\text{base}}$ and $Test_{\text{dynamic}}$. Mutations are applied to the above mentioned three levels of Lab Recruits, simulating three different types of development changes discussed in Section 3. After that, we re-execute the aforementioned tests on the generated mutants and inspect the results of the tests. Since we want to evaluate robustness, these mutants represent intended changes. Importantly, note that they do not represent bugs.

The robustness of a test will be assessed by the number of mutants that the test can 'survive' (that is, it manages to complete without failing). Since developers can change multiple things in a

Table 2: The features of each level. Each contributes to the level's complexity. BD is the backtracking depth of the corresponding level.

level	room-size	button	door	obstacle	BD
$Level_1$	10×10	8	4	-	0
$Level_2$	10×10	12	4	-	1
$Level_3$	10×10	12	4	4	1

Table 3: Mutation test result, showing how many mutants in 50 mutated samples can survive in each level, and the p-value of this result.

Mutation Type	Level	$Test_{\text{base}}$ success (p-val)	$Test_{\text{dynamic}}$ success (p-val)
Location change	$Level_1$	50/50 (0.0003)	50/50 (0.0003)
	$Level_2$	49/50 (0.0029)	49/50 (0.0029)
	$Level_3$	48/50 (0.0142)	48/50 (0.0142)
Location and logic change	$Level_1$	-	50/50 (0.0003)
	$Level_2$	-	50/50 (0.0003)
	$Level_3$	-	50/50 (0.0003)

single commit, we let each mutant to have multiple mutations. The first group of mutants only applies location changes to the buttons and doors in the target levels. The second group additionally applies logic changes. We do not explicitly do layout mutation. However, $Level_3$ has obstacles. So, when the location of e.g. a button is mutated, then relative to the button the layout appears to change. E.g. if before the button could be directly reached, after the mutation the agent may have to get around one or more obstacles to reach it.

Evaluating performance. The time needed by a test to solve a testing task is a measure for computing performance. Preferably, we want testing tasks to be solved within reasonable time. To gain some perspective on the relative performance of our approach, we compare our $Test_{\text{dynamic}}$ strategy with two other search strategies: Genetic Algorithm [20], Q-learning; and a random-testing tool; T3 [29]. In order to investigate these three approaches in our case study, a model of the levels is constructed to simulate them at the logical level (geometric details are abstracted away). After that, the non-agent algorithms are run on the simulator. As a simulator, it runs much faster than the actual Lab Recruits. Note that in this setup, the non-agent algorithms are set to search at the logical/subgoal-level. So, we allow them to operate at a high level of search rather than to literally search over the interaction space of Lab Recruits. In contrast, $Test_{\text{dynamic}}$ does not use such a model.

All experiments were run on a machine equipped with an Intel(R) Core i7-8565U processor and 32GB of RAM.

5.1 Result on Robustness

For each level (Table 2) mutants are randomly created. To achieve statistical significance 50 mutants are created per level and per type of mutation. We then exercise our automated tests (so, $Test_{\text{base}}$ and $Test_{\text{dynamic}}$) to these mutants. Due to engineering factors, it is inherently difficult to deterministically control a game under test. Because of that, we executed each test on each mutant three times.

The outcome of a test on a mutant is a "success" if it manages to complete on all its three runs on the mutant, and else it is a "failure". This can be modeled by a binomial distribution. In this experiment, the hypothesis is 85%, that is to say, the probability of individual test success is H_0:85%. $P{-}val(x)$ predicts the probability that the test would succeed on x mutants out of a total of n mutants with respect to H_0 hypothesis. A rejection level is considered as 0.05 (a traditional level used) which confirms the hypothesis if the $P{-}val(x)$ outcome is less than 0.05. Table 3 shows the result.

The table shows that $Test_{\text{base}}$ can survive almost all location changes on all levels in the experiment, indicating that it is indeed robust against such changes. Furthermore, the levels' complexity has little influence on the robustness. We do not execute $Test_{\text{base}}$ on the mutants that involve logic changes; it will not survive such a change as its scenario is already fixed. The result for $Test_{\text{dynamic}}$ shows that it is just as robust, and additionally can survive the introduced logical changes in all levels. We also investigated the failure-cases, and observed that the fails is caused by the agent's failure to see some object that is subtly positioned behind another.

The experiment confirms the hypothesis that the tests have 'robustness' of at least 85%, suggesting that we can reduce the developer's manual effort that would otherwise have to be spent for inspecting and fixing broken tests by the same amount. We also want to note that if we consider $Test_{\text{dynamic}}$ as a general algorithm, rather than as three separate instances for each of the levels in the experiment, then its failure rate is 3 out of 300 samples, which corresponds to robustness level of 95% with the p-value of 0.0001.

Indeed there are some mutants that the tests fail to survive. Upon closer investigation this is caused by the agent's failure to see a button because it is hidden behind another button or behind a corner. On the other hand, the agent has seen all navigation vertices. So it reasons that the button must thus be absent. This can be solved by implementing a heuristic, which we did not do, arguing that this might be a usability issue that should be reported to the developer.

Effectiveness. Table 3 also shows the effectiveness of our automated test. The effectiveness of a test will be assessed by the number of different "scenarios" that a test can automatically solve within reasonable time. This only makes sense for $Test_{\text{dynamic}}$ since $Test_{\text{base}}$ is fixed towards the particular scenario it is meant for.

To automatically solve a testing task T, the more challenging part is finding an execution that can solve its scenario part[5]. Recall that the set testing task T for each level in our experiment is to check if the treasure door can be opened. The mutants generated during the mutation test also randomise the scenario part of T (in the case of logical mutation), or maintain the same scenario but the required solving executions are different (when only location mutations are applied). So, the results in Table 3 also give us insight about the effectiveness of the used tests.

As remarked before, $Test_{\text{base}}$ follows a fixed scenario; it will not be able to solve other scenarios. However, $Test_{\text{dynamic}}$ can, because it incorporates a search strategy to deal with the situation that the logic connecting buttons to doors is changed. Notice that the results in Table 3 also implies that $Test_{\text{dynamic}}$ can thus solve all but a few of the randomly generated alternate scenarios on all three levels. So, $Test_{\text{dynamic}}$ is not only robust, but also effective; whereas, $Test_{\text{base}}$ is only limitedly robust. Some effort must be expended to write the

[5]To compare this with unit testing a function, to automatically test an assertion within the function the hard part is not the checking of the assertion itself, but rather, finding inputs that would trigger an execution that exposes the assertion.

Table 4: The performance of various algorithms to solve a $Level_1$ mutant. $Test_{\text{dynamic}}$ is run on-line on the Lab Recruits. The others are run off-line on a model. #attemps is the total number of subgoals tried until $Level_1$'s testing task is solved. The run time is given in seconds; those marked with * are the projected run time if the corresponding algorithm would run on-line on Lab Recruits. This is obtained by multiplying #attemps with the average execution time per subgoal of $Test_{\text{dynamic}}$.

Algorithm	Minimum		Maximum		Average	
	#attemps	time	#attemps	time	#attemps	time
$Test_{\text{dynamic}}$	6	8	54	75	30	42
Genetic algorithm	214	300*	50292	70409*	6872	9621*
Q-learning	90	126*	32198	45077*	5641	7897*
T3	52	73*	53790	75306*	44925	62895*

search strategy used by $Test_{\text{dynamic}}$, but a strategy only needs to be written once, and then it can be used in for different testing tasks.

5.2 Result of Performance

To get some insight on the relative performance of our automated test, we compare $Test_{\text{dynamic}}$ with two other algorithms: genetic algorithm, Q-learning, and T3. For the genetic algorithm, the used fitness function incorporates information about the state of the buttons and doors (e.g. opening a door increases the fitness). Comparison with T3 is included because it is a testing tool [27]. Its algorithm is mainly random-based.

Table 4 shows their performance comparison on a randomly chosen logic-mutant of $Level_1$. Given the magnitude of the difference in their performance, the choice of the mutant or level does not matter much. As mentioned before we run the other algorithms on a logical model created based on $Level_1$. This allows them to run much faster (else their run time would be prohibitively long) and to solve the testing task at the higher level, whereas $Test_{\text{dynamic}}$ has to solve it on-line on Lab Recruits, without a model.

The table shows the minimum, maximum, and average of the total number of subgoals each algorithm tries, until it solves the given testing task. A 'subgoal' here is for example 'interact with button *b*' or 'go to door *d*'. The needed time to solve the testing task, for non-$Test_{\text{dynamic}}$ algorithms, is the projected time if they would be executed on-line on Lab Recruits.

As shown in Table 4, it is evident that $Test_{\text{dynamic}}$ has much better run time compared to others. Q-learning algorithm has the best average compared to the other non-$Test_{\text{dynamic}}$ algorithms used, but its projected run time is still in the order of several hours compared to less than a minute of $Test_{\text{dynamic}}$.

Result of scaling up. To evaluate how the size of a game level affects the test performance, we progressively increase the rooms' size. Table 5 shows the result of the agent's performance on $Level_1$. As it is shown, the time needed and the number of tried subgoals increases, roughly, linearly with respect to the rooms' size. It indicates that increasing the size of a game environment (in our case levels) would not unproportionally degrade the test performance using our dynamic strategy.

6 RELATED WORK

In the last decades, various automated testing techniques have been developed and extensively studied [3, 10, 16, 23, 32]. Search-based testing (SBT) [2, 11, 12, 18] has shown its effectiveness. However, SBT lacks the ability to reason, hence severely hampering its ability

Table 5: The performance of $Level_1$ when the number of rooms is increased exponentially.

	2-rooms	4-rooms	8-rooms	16-rooms	32-rooms
time	18	69	81	298	539
attemps	289	1007	1465	4835	7344

to handle the vastness and the complexity of typical games' interaction space. Another alternative approach is model-based testing [10, 35, 36]. The approach uses models of software systems to derive test cases from the models. In order to achieve accurate results, high-quality models are extremely important. Iftikhar et al. [17] developed a UML-based model to support automated system-level game testing of platform games. However, models often have to be manually constructed. which requires a lot of efforts.

Recently, testing has become an increasingly important instrument to improving the quality of computer games [33, 34]. Research has provided various methods [17, 21] towards automated game testing, but they still require substantial manual work, e.g. to prepare models [17] or to redesign and re-record test sequences when the game is changed. Hence, researchers have been investigating ways to combine automated testing and the application of techniques from machine learning [4, 40, 41] in the context of game testing. E.g. Pfau et al. [25] developed ICARUS to test and detect bugs in an adventure game. However, this type of AI also requires much training, which could make it impractical to be deployed during the development time where SUT would undergo frequent changes. The approach proposed in this paper is different in that it relies on agent-based AI (e.g. reasoning) instead of machine learning. So, it does not require training. It does mean that some programming effort will be needed, but we would argue that this can be done at the high level, and results in tests that are robust.

Agent-based approach was thought to be better in reducing the complexity of systems under the test but the idea is not well investigated. E.g. researchers in [6, 24, 30] developed an agent-based approach for testing services and web applications. Askarunisa et al. [5] proposed a multi-agent framework for automating the testing process of a database application. The common idea behind these works is to use agent properties mentioned before to handle complicated situations, but neither work actually investigates how to exploit agent reasoning. In our work, agent reasoning is utilized to make an intelligent test agent that can perform more effectively to deal with the complexity of the game as well as improving the robustness of the tests that it does.

Recently, researchers have also been trying to combine artificial intelligence techniques into existing automated testing techniques to improve the latter. E.g. researchers in [7, 22], apply Reinforcement Learning for user interface level testing and automated testing of Android applications. Approaches discussed in [8, 41] extend automated approach with Reinforcement Learning. Zheng et al. [41] developed an on-the-fly framework called Wuji for automated game testing using a combination of evolutionary algorithm, Deep Reinforcement Learning, and multi-objective optimization. Vuong et al. [38] apply Q-learning to leverage both random and model-based testing. Although these approaches are not agent-based, they shed some light on how they could be applied to agents. Having test-agents that are able to learn would reduce the amount of work required to program them and might even improve their robustness.

7 CONCLUSION

This paper focused on the problem of automated testing in modern computer games as a significant enhancement for quality assurance. In this regard, we proposed an agent-based approach based on the iv4XR framework to produce robust automated tests. The approach is suitable for testing games that are played in a 2D/3D virtual world. Our framework provides an ability to program tests in high-level, regardless of the underlying details such as 3D navigation and geometric reasoning. Path finding is applied to provide auto-navigation and exploration skills to test agents that detaches testers' concern, when programming tests, from such details.

To evaluate our approach we considered different types of development changes that might be applied by level designers. It was observed that our tests could survive the changes. This implies that we can reduce the developer's manual effort that otherwise would have to be spent to edit tests according to the last changes. We also investigated our test performance by comparing the proposed automated test strategy with two other search strategies and a random testing tool, T3. We found that our automated testing strategy has superior run time to find a solution to solve a goal.

In the future, we would like to study how to apply learning techniques to our agent-based framework in order to improve our level of automation. Also, we would like to investigate our approach on more case studies.

REFERENCES

[1] 2019-2020. World Quality Report 2019-2020. https://www.capgemini.com/research/world-quality-report-2019/

[2] Tomás Ahumada and Alexandre Bergel. 2020. Reproducing Bugs in Video Games using Genetic Algorithms. In *2020 IEEE Games, Multimedia, Animation and Multiple Realities Conference (GMAX)*. IEEE, 1–6.

[3] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, et al. 2013. An orchestrated survey of methodologies for automated software test case generation. *Jour. of Sys. and Software* 86, 8 (2013).

[4] Sinan Ariyurek, Aysu Betin-Can, and Elif Surer. 2019. Automated Video Game Testing Using Synthetic and Human-Like Agents. *IEEE Trans. on Games* (2019).

[5] A Askarunisa, P Prameela, and N Ramraj. 2009. A proposed agent based framework for testing data-centric applications. *International Journal of Computational Intelligence Research* 5, 4 (2009), 429–453.

[6] Xiaoying Bai, Bin Chen, Bo Ma, and Yunzhan Gong. 2011. Design of intelligent agents for collaborative testing of service-based systems. In *Proceedings of the 6th International Workshop on Automation of Software Test*. 22–28.

[7] Sebastian Bauersfeld and Tanja EJ Vos. 2014. User interface level testing with TESTAR; what about more sophisticated action specification and selection?. In *SATToSE*. 60–78.

[8] Joakim Bergdahl, Camilo Gordillo, Konrad Tollmar, and Linus Gisslén. 2020. Augmenting automated game testing with deep reinforcement learning. In *2020 IEEE Conference on Games (CoG)*. IEEE, 600–603.

[9] John-Jules Ch. Meyer. 2007. Agent Technology. *Wiley Encyclopedia of Computer Science and Engineering* (2007), 1–8.

[10] Arilo C Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H Travassos. 2007. A survey on model-based testing approaches: a systematic review. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies*.

[11] Alexander Elyasov, I. S. W. B. Prasetya, and Jurriaan Hage. 2018. Search-Based Test Data Generation for JavaScript Functions that Interact with the DOM. In *29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE.

[12] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[13] Gordon Fraser and Andreas Zeller. 2011. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering* 38, 2 (2011), 278–292.

[14] Peter E Hart, Nils J Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107.

[15] Andreas Herzig, Emiliano Lorini, Laurent Perrussel, and Zhanhao Xiao. 2017. BDI logics for BDI architectures: old problems, new perspectives. *KI-Künstliche Intelligenz* 31, 1 (2017), 73–83.

[16] John Hughes. 2009. Software testing with QuickCheck. In *Central European Functional Programming School*. Springer, 183–223.

[17] Sidra Iftikhar, Muhammad Zohaib Iqbal, Muhammad Uzair Khan, and Wardah Mahmood. 2015. An automated model based testing approach for platform games. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 426–435.

[18] Phil McMinn. 2004. Search-based software test data generation: a survey. *Software testing, Verification and reliability* 14, 2 (2004), 105–156.

[19] Ian Millington and John Funge. 2019. *Artificial intelligence for games, 3rd edition.* CRC Press.

[20] Seyedali Mirjalili. 2019. Genetic algorithm. In *Evolutionary algorithms and neural networks*. Springer, 43–55.

[21] Michail Ostrowski and Samir Aroudj. 2013. Automated Regression Testing within Video Game Development. *GSTF Journal on Computing* 3, 2 (2013).

[22] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement learning based curiosity-driven testing of Android applications. In *29th ACM SIGSOFT International Symposium on Software Testing and Analysis.*

[23] Corina S Păsăreanu and Willem Visser. 2009. A survey of new trends in symbolic execution for software testing and analysis. *International journal on software tools for technology transfer* 11, 4 (2009), 339.

[24] Samad Paydar and Mohsen Kahani. 2011. An agent-based framework for automated testing of web-based systems. *Journal of Softw. Eng. and Applications* 4, 02 (2011).

[25] Johannes Pfau, Jan David Smeddinck, and Rainer Malaka. 2017. Automated game testing with icarus: Intelligent completion of adventure riddles via unsupervised solving. In *Extended Abstracts Publication of the Annual Symposium on Computer-Human Interaction in Play*. 153–164.

[26] ISWB Prasetya, Maurin Voshol, Tom Tanis, Adam Smits, Bram Smit, Jacco van Mourik, Menno Klunder, Frank Hoogmoed, Stijn Hinlopen, August van Casteren, et al. 2020. Navigation and exploration in 3D-game automated play testing. In *Proceedings of the 11th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. 3–9.

[27] Ignatius SWB Prasetya. 2016. Budget-aware random testing with T3: benchmarking at the SBST2016 testing tool contest. In *2016 IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 29–32.

[28] I. S. W. B. Prasetya, Mehdi Dastani, Rui Prada, Tanja E. J. Vos, Frank Dignum, and Fitsum Kifetew. 2020. Aplib: Tactical Agents for Testing Computer Games. *8th International Workshop on Engineering Multi-Agent Systems (EMAS)* (2020).

[29] IS Wishnu B Prasetya. 2013. T3, a combinator-based random testing tool for java: benchmarking. In *International Workshop on Future Internet Testing*. Springer.

[30] Yu Qi, David Kung, and Eric Wong. 2005. An agent-based testing approach for Web applications. In *29th Int. Comp. Software and Applications Conf.*, Vol. 2. IEEE.

[31] Anand S Rao, Michael P Georgeff, et al. 1995. BDI agents: From theory to practice.. In *ICMAS*, Vol. 95. 312–319.

[32] Urko Rueda, Tanja E. J. Vos, and I. S. W. B. Prasetya. 2015. Unit testing tool competition –round three. In *Int. Worksh. on Search-Based Software Testing*. IEEE.

[33] Ronnie ES Santos, Cleyton VC Magalhães, Luiz Fernando Capretz, Jorge S Correia-Neto, Fabio QB da Silva, and Abdelrahman Saher. 2018. Computer games are serious business and so is their quality: particularities of software testing in game development from the perspective of practitioners. In *12th ACM/IEEE Int. Symposium on Empirical Software Engineering and Measurement*. 1–10.

[34] Charles P Schultz and Robert Denton Bryant. 2016. *Game testing: All in one.* Stylus Publishing, LLC.

[35] Tom Tervoort and I. S. W. B. Prasetya. 2017. APSL: A light weight testing tool for protocols with complex messages. In *Haifa Verification Conference*. Springer.

[36] GJ Tretmans and Hendrik Brinksma. 2003. Torx: Automated model-based testing. In *First European Conference on Model-Driven Software Engineering*. 31–43.

[37] J Tuovenen, Mourad Oussalah, and Panos Kostakos. 2019. MAuto: Automatic Mobile Game Testing Tool Using Image-Matching Based Approach. *The Computer Games Journal* 8, 3 (2019), 215–239.

[38] Thi Anh Tuyet Vuong and Shingo Takada. 2018. A reinforcement learning based approach to automated testing of android applications. In *9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation.*

[39] Yuan Yao, Lavindra de Silva, and Brian Logan. 2016. Reasoning about the executability of goal-plan trees. In *International Workshop on Engineering Multi-Agent Systems*. Springer, 176–191.

[40] Imants Zarembo. 2019. Analysis of Artificial Intelligence Applications for Automated Testing of Video Games. In *Proceedings of the 12th International Scientific and Practical Conference. Volume II*, Vol. 170. 174.

[41] Yan Zheng, Xiaofei Xie, Ting Su, Lei Ma, Jianye Hao, Zhaopeng Meng, Yang Liu, Ruimin Shen, Yingfeng Chen, and Changjie Fan. 2019. Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 772–784.