An Agent-based Approach to Automated Game Testing: an Experience Report

I. S. W. B. Prasetya
Utrecht University
the Netherlands
S.W.B.Prasetya@uu.nl

Fernando Pastor Ricós
Universitat Politècnica de València
Spain
fpastor@pros.upv.es

Fitsum Kifetew
Davide Prandi
Fondazione Bruno Kessler
Italy
{kifetew,prandi}@fbk.eu

Samira Shirzadeh-hajimahmood
Utrecht University
the Netherlands
S.shirzadehhajimahmood@uu.nl

Tanja E. J. Vos
Open Universiteit and Universitat Politècnica de València
The Netherlands and Spain
tanja.vos@ou.nl,tvos@vrain.upv.es

Premysl Paska
Karel Hovorska
GoodAI
Czechia
karel.hovorka@goodai.com

Raihana Ferdous
Angelo Susi
Fondazione Bruno Kessler
Italy
{rferdous,susi}@fbk.eu

Joseph Davidson
GoodAI
Czechia
joseph.davidson@goodai.com

ABSTRACT

Computer games are very challenging to handle for traditional automated testing algorithms. In this paper we will look at intelligent agents as a solution. Agents are suitable for testing games, since they are reactive and able to reason about their environment to decide the action they want to take. This paper presents the experience of using an agent-based automated testing framework called iv4xr to test computer games. Three games will be discussed, including a sophisticated 3D game called Space Engineers. We will show how the framework can be used in different ways, either directly to drive a test agent, or as an intelligent functionality that can be driven by a traditional automated testing algorithm such as a random algorithm or a model based testing algorithm.

CCS CONCEPTS

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

KEYWORDS

automated game testing, agent-based testing, model-based game testing, 3D game testing

This is a preprint of a paper with the same title. It is published in the 13th Workshop on Automating TEST case Design, Selection and Evaluation (ATEST), 2022. The finalprint is published by ACM and can be found here: https://doi.org/10.1145/3548659.3561305

1 INTRODUCTION

Computer games are notoriously hard to test automatically. Imagine we want to test some specific state in a computer game. To do this, the tester may need to guide an in-game character through thousands of fine grained interactions to arrive in the state of interest; only then the tester can check one or more assertions on that state. In other types of interactive systems, such as web or mobile applications, testers can use a record and replay technology to automate the execution of tests. A tester would record manual sessions where he/she interacts with the application under test. The recorded sequence of interactions are then used as test cases by replaying them. Unfortunately this works poorly, and even more so in the game setup as recorded game plays are very fragile. E.g. non-determinism would break recorded game plays, which is problematical because this is prevalent in computer games, e.g. due to their randomized logic or presence of concurrent components. Furthermore, if the game designer changes the layout of the game world, or the placement of some game items just a little, which happens frequently during the development, these also break recorded tests. So, to robustly automate the execution of test cases, we need a solution that possess reactivity (ability to react to unexpected changes) and some "intelligence" to autonomously re-plan the test sequence if needed, e.g. if the world layout has changed. There is one programming paradigm that allows these to be programmed naturally, namely agent programming. As such this makes an agent programming framework a good candidate to be used as the base to build an automated game testing solution – iv4xr is such a framework.

The iv4xr framework provides a Java implementation of test agents [13]. A test agent can be connected to a game under test through an interface, and then used to autonomously drive a player character to do automated play testing. The framework provides concepts such as 'tactic' and 'goal structure' to abstractly program reactive behavior and complex testing tasks. The framework also provides automated path finding and terrain exploration [12] to enable a test agent to autonomously find a target game object it wants to test, regardless the layout of the game world.

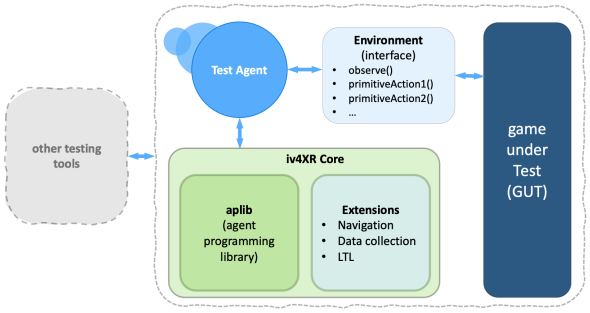

Figure 1: A high-level architecture of iv4xr framework.

This paper presents our experience of using iv4xr framework to test three different games: a Nethack-like 2D game called MiniDungeon, a 3D game called Lab Recruits, and a commercial 3D game called Space Engineers. Each will show a different use case of iv4xr.

This paper is structured as follows. Section 2 gives a brief overview of iv4xr and its agent programming. Section 3 gives an overview of the three case studies that we will present; Sections 4, 5, and 6 discuss each in more details. Section 7 gives a brief overview of related work, and finally Section 8 concludes.

2 THE IV4XR TEST-AGENT FRAMEWORK

The high-level architecture of the iv4xr framework is shown in Figure 1. A test agent can be connected to a game under test (GUT) through an interface called *Environment*. The game developers should implement this interface. It should provide a method observe() to observe the state of the game, and methods implementing primitive actions the agent can do on the game, such as interacting with a nearby game object, or moving to a certain direction for some unit of distance. What actions are to be provided, and how much observation observe() reveals, are up to the developers to decide. However, there are some aspects to consider. For example, allowing the agent to instantly access to the state of all game objects would make testing easier, but this might be computationally excessive as the GUT might then need to send over the states of thousands of objects to the agent. We also lose some realism as actual players can only see objects visible in their screens. An all-seeing agent might take actions that actual users would not do.

In the most basic form, a test agent drives the GUT by invoking its Environment's actions and samples the GUT's state as it goes to check if the GUT is in the correct state. Iv4xr is inspired by so-called BDI (Belief-Desire-Intent) agents [2, 5, 14]. This type of agents has a quite different execution model than a traditional procedure, so it is useful to first explain this. A BDI agent has a set of goals and runs in so-called *deliberation* cycles until its goal set is empty. At every cycle, the agent observes its environment, decides which action to do, and executes the action. It also decides if the current goal is accomplished, or if it should be dropped, and if so, which goal to pursue next. Specific for an Iv4xr agent, it accumulates all observations it gets so far into what can be seen as 'belief': the latest observation is factual, but older observations in the belief may no longer be valid in the actual GUT state. A BDI agent is allowed to act on belief, e.g. if an object o exists in its belief, it can optimistically decide to go to o, believing it still exists in the actual game world. The fact that a BDI agent runs in deliberation cycles also makes it highly *reactive*, as it allows the agent to continuously, or at least frequently, sample the state of the GUT and immediately acts after each sampling, which makes it very suitable for controlling a game.

The basic form of BDI agent programming is to specify which action to select at each deliberation cycle, which can be expressed *declaratively* with guarded actions a la Action System [3]. The snippet below shows an example of how this looks like in iv4xr (some concrete syntax are omitted). B represents the agent's belief; $B \rightarrow expr$ is a lambda expression that, here, represents an action.

$$
\begin{aligned}
\textbf{var}\ \ & tactic_1 \ = \ \text{ANYof}(\\
& \text{action}().\text{do1}(B \rightarrow B.env().moveUp()).\text{on}(g_1)\ , \\
& \text{action}().\text{do1}(B \rightarrow B.env().moveDown()).\text{on}(g_2)\ , \\
& \ldots \\
& \text{action}().\text{do1}(B \rightarrow B.env().useHealKit()).\text{on}(g_k)\ )
\end{aligned}
$$

where $moveUp()$, $moveDown()$, $useHealKit()$, etc are methods we can imagine as provided by the Environment $env()$, and $g_1..g_k$ are guards specifying when the corresponding action is enabled for execution (e.g. g_1 could require that the way upwards is clear). Only enabled actions are executable; if there are more than one, the ANYof combinator will select one randomly. In iv4xr, a system of actions such as the one above is called a *tactic*. Given a tactic, an agent will keep executing it until its current goal is achieved (or it runs out of budget). E.g. this goal could be 'to obtain a key' (e.g. because we want to check its properties).

If we remove all the guards, the tactic above would be how we can program a random test agent. Guards add some intelligence in choosing better actions (than just randomly), e.g. the action $useHealKit()$ can be guarded so that it becomes enabled when the character health drops under a certain critical level.

2.1 Navigation

A basic, but important, task that should be automated is navigation. The previous $tactic_1$ can do it, but not effectively. In fact, navigation in a game wold is usually non-trivial due to its complex layout and presence of dynamic obstacles. A standard solution is to represent walkable parts of the game world, which can be an infinite continuous space, as a finite *navigation graph*, after which a path finding algorithm such as A* [8] can be applied to guide the agent to get to a target location. Iv4xr provides several ways to do this reduction. For example if the GUT can export a so-called *navigation mesh*, iv4xr can convert it to a navigation graph. Figure 2 shows an example of such a mesh in a game engine called UNITY. A mesh is a finite set of connected triangles that cover a walkable surface. As such, it induces a navigation graph. If the GUT does not produce a navigation mesh, iv4xr can also construct a navigation graph on the fly, based on the geometry of the objects an agent sees.

From this, two tactics can be constructed [12]. First, $navigateTo(o)$, which when repeatedly executed would guide the agent to reach the location of an object o, if the location is known. Second, $explore()$ to guide the agent to the closest unexplored area of the game world. So, rather than the previous $tactic_1$ we can now have the following, if the goal is to obtain some object 'key' k:

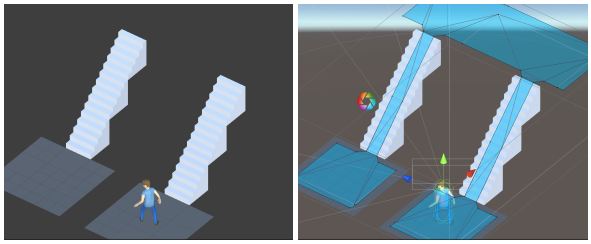

Figure 2: *The picture to the right shows the mesh (blue surface), consisting of triangles (edges colored red), in a UNITY game.*

$$\textbf{var}\ \ tactic_2\ =\ \mathsf{FIRSTof}(\ \mathsf{action}().\mathsf{do1}(B \rightarrow B.env().useHealKit()).\mathsf{on}(g)\ ,\ navigateTo(k)\ ,\ explore())$$

Above we use a priority-based selector FIRSTof rather than the previous random selector ANYof. The sub-tactic $navigateTo(k)$ will take the agent to the key k, if its location is known. Else the tactic is not enabled; FIRSTof will instead choose $explore()$ to explore the world until the agent sees k. Though, if its health drops too low, it will first use a healing kit to fix itself. Note that the last adds *reactivity* to handle an 'emerging situation', namely when the health drops too low. This can be extended to handle more emerging situations, such as approaching enemies, and thus equipping the test agent with some logic to make it more adept in surviving the game (useful, as a dead agent can't perform testing tasks).

2.2 Formulating a Testing Task: Goal Structures

An Iv4xr agent can be given multiple goals. Unlike other agent programming languages, iv4xr requires the goals to be structured. A *goal structure* is tree with goals as leaves and control-combinators as nodes, specifying either an order or a priority with which its subgoals are to be solved. For example a sequential testing task to find the key k, to pick it up, and to check that it can be used on door d can be formulated as a goal structure such as the one below:

$$\begin{array}{ll}\mathsf{SEQ}(& \text{"}k\text{ is found"} \quad .\mathsf{withTactic}(T_1(k)),\\ & \text{"}k\text{ is picked up"} \quad .\mathsf{withTactic}(T_2(k)),\\ & \text{"}d\text{ is found"} \quad .\mathsf{withTactic}(T_1(d)),\\ & \text{"}k\text{ is used on }d\text{"} \quad .\mathsf{withTactic}(T_3(k,d)))\end{array}$$

SEQ requires its subgoals to be solved in the order they are given. For more combinators, including conditional and repetition, see [13], with which even a test *algorithm* can be expressed, e.g. when the exact sequence of sub-tasks is not known upfront [15].

2.3 Integration with Other Testing Tools

Another value of the iv4xr framework is to be used as a rich adapter to enable traditional automated testing tools to target computer games (see also the architecture in Fig. 1). For example we used this scheme to allow a GUI-testing tool TESTAR [16] and a model-based testing (MBT) tool to target games in two of our case studies. E.g. TESTAR exploits iv4xr Environment and navigation graph to perform smart monkey testing. The MBT tool can efficiently generate test cases from an EFSM model of a game, which subsequently are translated to goal structures for a test agent to execute. This is a very simple integration scheme. A translator must indeed be written, but this only need to be written once for each game.

3 CASES OVERVIEW

In the coming sections we will discuss our experience in using iv4xr for testing three different games: MiniDungeon, Space Engineers, and Lab Recruits. With each we also want to show a different way of using iv4xr. In the MiniDungeon case we will show a *direct use* of a iv4xr test agent to check a set of correctness properties of the game, such as the reachability of key objects in the game. The game has much randomness in its logic and enemies too. The test agent needs to be able to deal with both to remain robust and survive long enough to complete its testing tasks.

In the Space Engineers case we will show a setup where iv4xr is leveraged to enable another testing tool, in this case the GUI testing tool TESTAR [16], to do automated exploratory game testing.

In the Lab Recruits case shows a setup where iv4xr agents are used as an intelligent executor for model based testing (MBT). This setup allows the behavior of a game under test to be described abstractly using e.g. an extended finite state machine (EFSM), where the concrete layout of the game world can be abstracted away from the model. An MBT algorithm can very efficiently generate abstract test cases from such a model. These abstract cases are fed to the agents that will carry them out. The agents exploit automated navigation and exploration from iv4xr (Section 2.1) to explore the concrete world layout to find the game objects in the test cases.

4 MINIDUNGEON

MiniDungeon is a small 2D, turn-based, Nethack-like game written in Java. Figure 3 shows a screenshot. The game can be played by a one or two players. Figure 3 shows two players (circled blue). The players' goal is to cleanse the shrine (circled white). To do so, a scroll is required (gray icon) which a player must bring to the shrine. Only a holy scroll will cleanse the shrine, but the player does not know which scroll is holy until he/she tries it. If the shrine is cleansed, it becomes a portal that takes the player to the next level. This goes on until the final level; cleansing the shrine there wins the game. Along the way there are monsters (blue figures) that can hurt players, but also potions to help them. A greedy strategy that just collects all scrolls and potions does not work because the player's bag has limited space, which is either 1 or 2.

The case. The source code is about 1.2K lines. There are unit tests providing good coverage for their respective targets. However, in total they only cover 20% of the whole code base because a large part of the game is simply hard to unit test.

Monsters, shrines, scrolls, potions, and even players are so-called *game objects*. As common in game implementations, MiniDungeon has a so-called *game loop*, where at every iteration it executes the entire state update for each new turn; so, executing the players' move and all monsters' move for the turn. Game objects may have multiple properties, but they do not have much behavior on their own. For example, when a monster move, the logic that decides

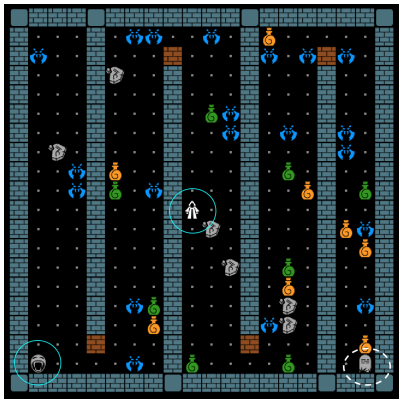

Figure 3: The MiniDungeon game.

where to move resides in the game loop. This cannot be delegated to the monster-object itself, since the latter only knows its own state and does not know which neighbouring squares are empty to move to. So essentially, most of the game logic resides inside the game loop. Unfortunately this game loop is hard to unit-test.

For example, the game loop moves the monsters in random directions. Imagine we want to verify that it will never move a monster to an occupied square. To do this with the usual unit-testing setup we will need to create a set of test fixtures in the form of a game level, seeded with at least one monster and different game objects in different neighboring squares. This is combinatorial in nature, so it takes substantial effort to hand-craft the fixtures. Then, to test the monster's move, we run a the game loop for one ore more turns on all the fixtures, and repeated multiple times using different random seeds to account for the randomness in the move's logic. A more practical approach is to implant the check as an assertion inside the game; a snippet of this is shown below:

```
// move the monster to sq:
...
if(Debug.ON) assert(world[sq.x][sq.y]==null) ;
world[sq.x][sq.y]=m;
```

And then we simply 'play' the game several times. The implanted assertion will catch erroneous monster-moves. This does not require manually creating fixtures. We also want to do the play testing automatically. To achieve this we program an automated play testing agent using iv4xr.

Iv4xr solution

We first implement the Environment component in Figure 1 that serves as the interface between the agent and the GUT. Its main APIs is shown below:

```
class MyAgentEnv extends Iv4xrEnvironment{
    WorldModel observe(agentId)
    WorldModel command(agentId,cmd)
}
```

The method command(a, cmd) simulates a key pressed by a player as a command. E.g. the key 'w' and 's' cause the agent a to move up respectively down, whereas 'q' causes the game to end.

The method observe(a) returns what the agent a currently sees. In Fig. 3 we artificially set the view distance to ∞, but normally the view is limited, e.g. only 3 squares away. Observation is represented as an iv4xr datastructure called WorldModel. Essentially, it is a set of 'entities', each representing a game object as a record of a unique ID, timestamp, its physical location, and a list of name-value pairs describing the object's other properties/state. Entities may have sub-entities if needed. As mentioned, the agent automatically accumulates observations into its belief, also represented as a WorldModel. In this belief, when e.g. a monster was observed, the observation is kept even if the monster is no longer visible. It is maintained until a new observation updates the monster's state, or if the new observation says the monster has been destroyed. This belief gives the agent more information/depth for making its decision, rather than just reacting to what it currently sees.

Automated exploration and navigation. We cannot do much automated play testing if the agent is not able to autonomously explore a game level and navigate to game objects. To support we implement the tactics $explore()$ and $navigateTo()$ discussed in Section 2.1. Iv4xr provides the main worker functions that do the calculation over the navigation graph, but the calculation results still need to be translated to calls to actual movement actions as provided by GUT through the Environment. This requires some programming work, but not much (40-50 lines).

Survivable play testing. When the GUT has hazards (such as aggressive monsters), just automated navigation as above is not sufficient. The agent must be smart enough to handle the hazards in order to live long enough to reach the state it is tasked to check. Programming a survivable play testing agent takes more effort.

Imagine a goal $entityInCloseRange(o)$ that is achieved when the agent reaches a square next to the object o. To solve it, rather than simply using $navigateTo(o)$, we use a tactic similar to $tactic_2$ from Section 2.1, extended with more combat sub-tactics, e.g. line 4 to attack a monster that engages the agent and line 3 to quaff a rage potion to increases the agent's power when in combat.

```
1 FIRSTof(
2     useHealingPot().on_(hasHealPot_and_HpLow),
3     useRagePotAction().on_(hasRagePot_and_inCombat),
4     attackMonster().on_(inCombat_and_hpNotCritical),
5     navigateToTac(o),
6     explore(),
7     ABORT())
```

Programming a complete playtest. As an example of a complete playtest we configure MiniDungeon to generate a game world consisting of N=2 levels. We program the test agent to first cleanse level-1's shrine, and then that of level-2, which would then wins the game for the agent. A playtest does not have to be winning though, but a winning playtest usually covers most key features of the game. To cleanse a shrine the agent will have to try different scrolls until it gets the right one. We do not want to explicitly program the sequence of scrolls to try. This would make the test less robust. Instead we use an implementation of the online search algorithm in [15] to let the agent autonomously do the search. The algorithm will require some components to be 'plugged-in', such as the goal

	locs	*cc*
MiniDungeon (GUT)	1196	325
iv4xr-lib (playtest infrastructure)	869	219
Environment implementation	177	
Tactic and goal lib	426	
Utils and other	266	
unit tests	236	129
agent playtests	196	74

Table 1: *The GUT, tests and test-infrastructure size and complexity, given in, respectively, lines of code (locs) and cyclomatic number (cc).*

$entityInCloseRange(o)$ mentioned before, but also a goal to make o interacted (after the agent stands next to it). Once the algorithm is set up, we can use it to automate a task of the form:

$$solver(a, T, o, \phi)$$

This constructs a goal structure for an agent a, that seeks to change the state of object o to a new state satisfying ϕ. In our case, o is a shrine and ϕ is "o is cleansed". It does this by autonomously searching objects of type T and then using them, one at a time, until the aforementioned goal is accomplished.

So now the whole playtest can be written, essentially, just as:

$$\begin{aligned}\text{SEQ}(\ & solver(a, Scroll, shrine_1, \text{"}shrine_1 \text{ is cleansed"}),\\ & interacted(shrine_1),\\ & solver(a, Scroll, shrine_2, \text{"}shrine_2 \text{ is cleansed"}))\end{aligned}$$

The test passes if this goal structure is solved. Additionally, various assertions over the agent's belief are also checked, e.g. that the agent health is expected to eventually drop below its maximum (as there are monsters attacking the player) but it never drops to 0, that the number of items in its bag never exceeds the bag's capacity, that the agent never walks through a wall, and so on.

Experience

Table 2 shows the improvement we get. Without agent, the unit tests only covers 18.8% (U_{cov}) of the total code base . With the agent play tests we can cover 88.4% (all_{cov}), which is a huge improvement. Furthermore the latter found two bugs that were not found by unit testing. These are subtle bugs that are difficult to catch at the unit level unless we specifically were looking for them. For example one of the bugs occurs when the two players are in different levels but the xy-projection of their visibility areas overlap. The bug caused some squares that should be visible to a player to be missed. Such a situation is just hard to anticipate at the unit level.

Table 1 gives some indication on the investment and effort needed to do agent playtesting. The library that provides a implementation of *Environment* along with smart tactics and goal-structures is about 850 lines large, which substantial relative to the size of the GUT. Fortunately, it is less complex (see the *cc* number). This part is a one-off investment. The playtests themselves are smaller and less complex than all the unit tests together. Given the huge gain in the test coverage, and the convenience with which we can subsequently write automated playtests, the extra investment in the infrastructure is arguably well spent.

	C	i	cc	U_{cov}	PT_{cov}	all_{cov}	U_{bug}	PT_{bug}
Entity	11	360	21	81%	97.8%	100%	2	
Maze	1	320	18	89.9%	95%	95%	1	
MiniDungeon	2	2282	197	14.2%	84.4%	84.5%		2
MDApp	1	1228	89	0%	92.3%	92.3%		
All	15	4790	325	18.8%	88.2%	88.4%	3	2

Table 2: *The table shows some statistics of four classes that made the game MiniDungeon, and how well the tests cover them. C : the number of classes that a top-level class has; i : the number of instructions; cc : cyclomatic complexity; U_{cov}, PT_{cov}: instruction coverage of respectively unit tests and play tests with an agent; all_{cov} : the coverage of combined tests; U_{bug}, PT_{bug} : the number of bugs found by respectively unit tests and play testing with an agent.*

5 SPACE ENGINEERS: TESTAR-IV4XR

Space Engineers (SE) is a complex open-world game developed by Keen Software House and GoodAI (GA). It offers users the simulation of a realistic 3D environment with volumetric physics and objects with mass, inertia, and velocity. SE users can use multiple types of blocks and tools to build any structure like bases or spaceships. Figure 4 shows a construction in the space of the game. The testing process of SE consists of a team of testers who manually test the functionality and visuals of aspects of the game. Given the game's complexity, more than 10,000 manual tests are performed for each major release.

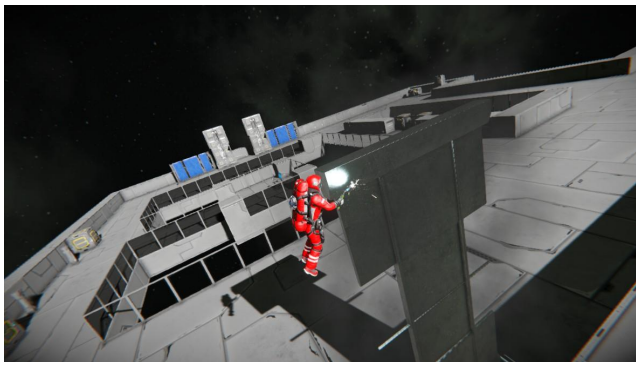

Figure 4: *An example of a Space Engineers level.*

TESTAR is an open-source tool for scriptless GUI testing that automatically generates test sequences of (state, action)-pairs at run-time. This scriptless tool does not follow previously created or recorded scripts or models to select which action to execute, but instead follows an action selection mechanism (ASM) to make decisions on the fly. The main advantage of this is that it is really a push and go approach to complement manual testing.

A computer game like SE is however not a GUI application in the traditional sense, as it does not expose a widget tree that a typical GUI testing tool can target. To be able to target SE we exploit iv4xr, in particular its Environment and Core components. Similar to the MiniDungeon case (Section 4), the Environment's observe() constructs a WorldModel that keeps track of active game objects. This WorldModel has the same role as a widget tree in a GUI and

allows TESTAR to target the game objects it tracks. We integrated TESTAR with iv4xr and use it as an exploratory test agent on SE [9]. The logical flow of TESTAR consists of: connecting with the SE system, realizing an observation to obtain information about all existing virtual entities in a specific range, deriving the possible actions to execute, and selecting one to transit to a new state. There are two types of actions: basic commands and compound tactics. A basic command action is the most basic event we can execute in SE, e.g., move or rotate one step, equip a tool and start or stop using a tool. A compound tactic action contains several basic commands that simulate user decisions. For example, to interact with a block, we need to rotate the agent to aim at the block, move to reach the block, equip a tool and then start using this tool.

One of the main challenges in SE was to allow agents to calculate the navigation to a position to reach blocks. Because, unlike other game systems, SE does not have the functionality to produce a default navigation mesh. To deal with this, the iv4xr framework allows using the geometry information of the observed entities to construct a navigation graph on the fly, and calculate if the agent can follow a path of nodes to reach a position. TESTAR agent uses this feature to derive compound tactical actions, containing navigation, over the 3D space adjacent to an observed block. Figure 5 shows a representation of how the geometry of an SE level is used to calculate the navigable space.

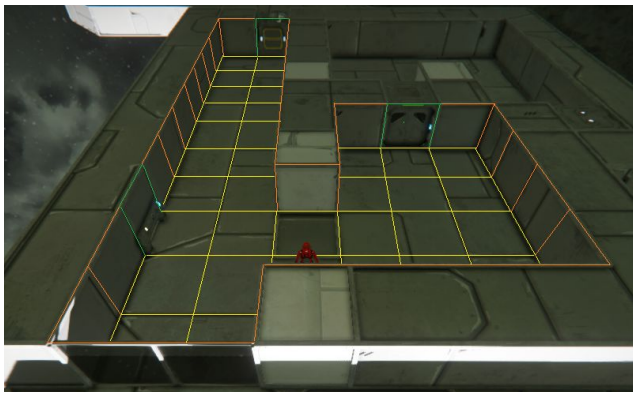

Figure 5: *An example of the agent observation and use of block geometry to compute navigable space in Space Engineers. The floor contains 2D navigable nodes (yellow lines). The walls require the use of a jet-pack to fly over them (orange lines). The agent can observe a series of interesting blocks to interact (green lines).*

Experience

A computer game like SE is very challenging for any automated testing tool to target, as the tool will also have to deal complex calculation to control terrain navigation and body motion, which most GUI testing tools are not equipped with. Using the above setup with iv4xr, TESTAR can now do this. We then use the setup to do automated exploratory testing on SE. As the ASM policy we simply use guided random, where priority is given to interacting with game objects that have not been tried before. The objective of the exploratory agent is to navigate to the interesting blocks, interact with them, and automatically validate their physics such as material integrity. We use TESTAR's feature for specifying oracles to strengthen the test by adding generic oracles to test the robustness of SE systems, e.g., to detect if the process crashes or hangs or to find exception messages in the application log. Additionally, to test part of the game's functional aspects, custom oracles are also added, such as validating that the jet-pack settings are correct after interacting with functional blocks that move the agent.

We apply this exploratory test on various levels of SE to verify the TESTAR agent functionality regarding navigable actions and oracles. A video of one of these tests can be seen here[1] where TESTAR explored a small level for about three minutes and found a jet-pack bug.

6 LAB RECRUITS: MBT-IV4XR

In this example, we exploit iv4xr to enable model based testing (MBT) of computer games. MBT is widely used in industrial engineering, but applications in gaming are quite limited [6, 7]. A major burden is the complex layout of the game world, which is very difficult to describe in the usual behavioral models (e.g. FSM) used in MBT. The iv4xr framework adds an intermediate abstraction level that provides navigation primitives and goal structures, allowing MBT to focus on the behavioural aspects of the game.

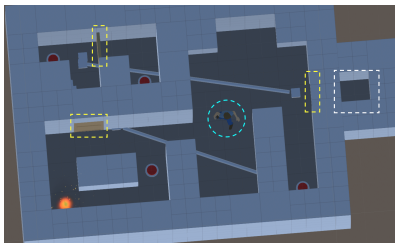

Figure 6: *A small level in the Lab Recruits game. It has three rooms guarded by doors (marked yellow). There are four buttons (red), each can toggle the state of zero or more doors. The player is marked by the blue circle. The goal of this level is to reach the white marked room.*

Here, we apply MBT through iv4xr on Lab Recruits[2], a 3D maze game. The game allows players to explore a level, that typically consists of rooms guarded by doors, which can be opened by toggling the right buttons. The button-door connections are many-to-many, defining non-trivial paths that a player must discover to arrive at a given room. Other game objects include fire hazards and goal flags that give points and heal the player. A small Lab Recruits level is shown in Fig. 6. Game levels are defined as CSV files, which include the layout of the world and the (initial) placements of game objects. We exploit this to write a level generator [6], capable of creating very large levels (which otherwise would be very labour intensive to craft by hand), along with an Extended Finite State Machine (EFSM) [4] that models the logic of the levels.

Fig. 7 shows the EFSM that models the level in Fig. 6. Note that the EFSM does not contain information about the physical layout of the world, but only the reachability relation between neighboring entities (doors and buttons). With the model at hand, a test case is

[1] https://www.youtube.com/watch?v=ho1EMVtr8C4
[2] https://github.com/iv4xr-project/labrecruits

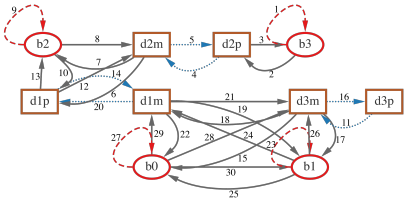

Figure 7: *An EFSM modelling the behavior of the level in Fig. 6. Red circle nodes are in-game buttons; squares are doors. Each door is represented by a pair of nodes, representing the two sides of the door. The agent starts at b_0. Solid black transitions represent travel between two game objects. Blue transitions represent travel through a door (only possible if it is open). A red transition represents the toggling of the associated button. The extended state of the EFSM consists of the state of the doors, which is either open or closed (not shown). All doors are initially closed. Toggling a button will toggle the state of associated doors (not shown).*

	States	Transitions	Variables
L1	144	558	40
L2	155	646	40
L3	225	1439	40

Table 3: *Characteristics of the EFSM models corresponding to three randomly generated Lab Recruits levels.*

generated as a sequence of transitions from the initial state. This step happens offline (i.e., without executing on the GUT), greatly speeding up the generation time (minutes, rather than hours). However, the produced test suites cannot be directly executed on the GUT (Lab Recruits), since the information on how to navigate through the game world is missing. For instance, a test case could require going from b_0 to b_1, but the model does not have information on how to navigate in Lab Recruits. The iv4xr framework fills this gap by providing the notion of goal structure. In particular, each transition of a test case generated from the EFSM is translated into a goal along with the tactic to solve it. The whole test case, which is a sequence of transitions, is translated to corresponding SEQ goal structure similar to the example in Section 2.2 that can be executed on Lab Recruits.

Experience

An empirical study of this setup has been presented in [6]. Here, we will show a small example to discuss the experience. We automatically generate three large levels, L1, L2, and L3, and the corresponding EFSM models; Table 3 shows their main features. For each model, we take advantage of the search-based test generator tool EvoMBT[3] to assess the performances of three test suite generation strategies: pure Random generation, classic evolutionary strategy algorithm Mu+Lambda, and many objective algorithm MOSA. For each model, each generation strategy runs 30 times for 300s. EFSM transition coverage observed is reported in Figure 8.

The EvoMBT tool supports execution on Lab Recruits of a test suite generated on an EFSM model of a level. EvoMBT exploits the iv4xr framework to translate an EFSM transition to a goal structure

[3]https://github.com/iv4xr-project/iv4xr-mbt

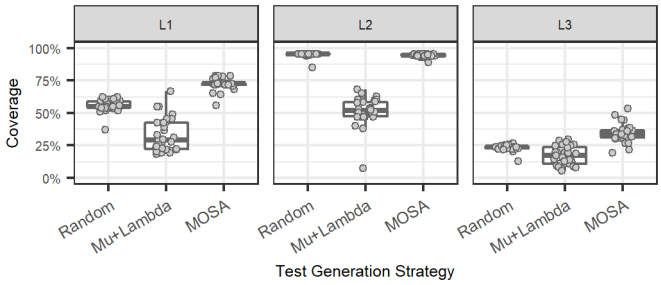

Figure 8: *EFSM transition coverage achieved by different search-based test generation strategies. For each model (L1, L2, and L3), the plot report the boxplot of the transition coverage as well as the coverage (gray dots) of each replica.*

and to execute it on Lab Recruits. For each level and each generation strategy, we selected a test suite generated from the EFSM model and executed it on Lab Recruits. As can be seen from Table 4, the mean execution time of a test suite on Lab Recruits is more than two hours and a half, while the generation on the model required only five minutes. Clearly, if test generation were done directly on Lab Recruits the time required would not be feasible. Another interesting point is the high number of failed tests when run on Lab Recruits(mean value 50%). The execution of a test case fails because the iv4xr agent does not complete a goal within a specified time budget or because the agent cannot reach a specific position in the current Lab Recruits level. The first problem can be solved by increasing the agent's time budget, thus making the execution more time consuming. The second type of failure highlights a problem in the Lab Recruits autonomous navigation support that we are investigating.

Level	Strategy	Time(h)	n Tests	n Fails
L1	Random	2.79	108	6
L1	Mu+Lambda	1.31	60	52
L1	MOSA	2.62	160	118
L2	Random	3.54	252	179
L2	Mu+Lambda	0.46	33	33
L2	MOSA	4.15	302	219
L3	Random	1.04	67	9
L3	Mu+Lambda	6.05	133	9
L3	MOSA	1.87	95	23

Table 4: *Exectution of the test suites generated from the EFSM models on Lab Recruits.*

The integration of the iv4xr framework with MBT enables fast test suite generation while providing a natural and general notion of coverage based on EFSM models. Here, we presented our experience with Lab Recruits a real 3D maze game, and we showed that the combination of model base generation and the framework iv4xr is effective in identifying potential issues in the game under test. Future work includes supporting other game entities (e.g., fire hazards) in the models, as well as the extension to multi-agent scenarios.

7 RELATED WORK

Until today the game industry heavily relies on manual play testing to test games. Automated testing is rarely done. The challenges range from process related, where testing is not rigorously implanted in development cycles, to engineering, e.g. the lack of testing tools that can target games out of the box. Politowski et al. give a good overview on issues and challenges of game testing in the industry [11].

In terms of research, there are indeed work in automated play testing, though in much less volume than in other areas of automated testing. For example the use of model based testing (MBT) was investigated by Iftikhar et al. [7] and later by Ferdous et al. [6]. However, recent work in automated play testing seem to focus more on the use of machine learning, in particular reinforcement learning (RL). E.g. Pfau et al. use RL to train an automated test agent for an adventure game [10]. Zheng et al. use evolutionary deep RL to do the same for an action game [17] (which has much more dynamics than an adventure game). Ariyurek et al. use RL to train an agent to play with different styles, e.g. killer or explorer [1]. Gordillo et al. use curiosity driven RL to improve coverage.

Despite advances in RL, its scalabilty for game testing is still an open discussion. RL requires a huge amount of training, which all must be executed on the actual game where actions are relatively much slower to execute. So, the overall computation cost might be prohibitive e.g. for smaller companies. Moreover, if the game logic is changed, or the world layout is changed, which happen very often during the development, the agent may have to be retrained.

Obviously programming a game play is much harder than, for example, scripting a test sequence for a web application. For this reason the fascination towards RL is understandable. However we can also reduce the investment cost for building automation by providing a proper programming language, or at least a framework, that offers the right concepts and abstraction so that the effort for programming play testing becomes manageable. The iv4xr framework tries to fill this role. The advantage of a more programming-based approach is that we have much more control on the test agent behavior, and we have also shown that a BDI test agent is robust against development time changes [15].

8 CONCLUSION

We have discussed our experience of using iv4xr to do automated testing on three different games, ranging from a turn-based 2D game to a complex commercial 3D game. In all three cases the use of iv4xr has successfully introduced automation and contributed in finding bugs and issues. Unlike e.g. a machine learning based approach, iv4xr is a programming approach that allows automated testing to be programmed at a high level. The approach gives developers more control on the behavior of the test agent while keeping the agent versatile and robust. Moreover, we have demonstrated that iv4xr can be used as a rich interface to enable more traditional testing tools to target computer games. Building an interface between the game under test and iv4xr along with a library of basic tactics and goals does require some effort, but this is one off investment, after which developers will benefit from powerful test automation.

ACKNOWLEDGMENTS

This work is supported by the EU ICT-2018-3 H2020 Programme grant nr. 856716.

REFERENCES

[1] Sinan Ariyurek, Aysu Betin-Can, and Elif Surer. 2019. Automated Video Game Testing Using Synthetic and Human-Like Agents. *IEEE Transactions on Games* (2019).
[2] Rafael H Bordini, Jomi Fred Hübner, and Michael Wooldridge. 2007. *Programming multi-agent systems in AgentSpeak using Jason*. Vol. 8. John Wiley & Sons.
[3] K Mani Chandy and Jayadev Misra. 1988. *Parallel Program Design: A Foundation*. Addison-Wesley.
[4] Kwang-Ting Cheng and Avinash S Krishnakumar. 1993. Automatic functional test generation using the extended finite state machine model. In *30th ACM/IEEE Design Automation Conference*. IEEE, 86–91.
[5] Mehdi Dastani. 2008. 2APL: a practical agent programming language. *Autonomous agents and multi-agent systems* 16, 3 (2008).
[6] Raihana Ferdous, Fitsum Kifetew, Davide Prandi, ISWB Prasetya, Samira Shirzadehhajimahmood, and Angelo Susi. 2021. Search-Based Automated Play Testing of Computer Games: A Model-Based Approach. In *International Symposium on Search Based Software Engineering*. Springer, 56–71.
[7] 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.
[8] Ian Millington and John Funge. 2019. *Artificial intelligence for games, 3rd edition*. CRC Press.
[9] Fernando Pastor Ricós. 2022. Scriptless Testing for Extended Reality Systems. In *International Conference on Research Challenges in Information Science*. Springer.
[10] 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.
[11] Cristiano Politowski, Fabio Petrillo, and Yann-Gaël Guéhéneuc. 2021. A survey of video game testing. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*. IEEE.
[12] 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.
[13] I. S. W. B. Prasetya, Mehdi Dastani, Rui Prada, Tanja EJ Vos, Frank Dignum, and Fitsum Kifetew. 2020. Aplib: Tactical agents for testing computer games. In *International Workshop on Engineering Multi-Agent Systems*. Springer, 21–41.
[14] Anand S Rao and Michael P Georgeff. 1991. Modeling rational agents within a BDI-architecture. In *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*. 473–484.
[15] Samira Shirzadehhajimahmood, ISWB 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*. 1–8.
[16] Tanja E. J. Vos, Pekka Aho, Fernando Pastor Ricos, Olivia Rodriguez-Valdes, and Ad Mulders. 2021. TESTAR – scriptless testing through graphical user interface. *Software Testing, Verification and Reliability* 31, 3 (2021). https://doi.org/10.1002/stvr.1771
[17] 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 International Conference on Automated Software Engineering (ASE)*.