




# Scriptless Testing for an Industrial 3D Sandbox Game

Fernando Pastor Ricós<sup>1</sup><sup>a</sup>, Beatriz Marín<sup>1</sup><sup>b</sup>, Tanja Vos<sup>1,2</sup><sup>c</sup>, Joseph Davidson<sup>3</sup> and Karel Hovorka<sup>3</sup>

<sup>1</sup>Universitat Politècnica de València, València, Spain

<sup>2</sup>Open Universiteit, The Netherlands

<sup>3</sup>GoodAI, Prague, Czechia

**Keywords:** Computer Game Testing, Autonomous Agents, Scriptless Testing, Exploratory Testing, Automated Testing.

**Abstract:** Computer games have reached unprecedented importance, exceeding two billion users in the early 2020s. Human game testers bring invaluable expertise to evaluate complex games like 3D sandbox games. However, the sheer scale and diversity of game content constrain their ability to explore all scenarios manually. Recognizing the significance and inherent complexity of game testing, our research aims to investigate new automated testing approaches. To achieve this goal, we have integrated scriptless testing into the industrial game Space Engineers, enabling an automated approach to explore and test sandbox game scenarios. Our approach involves the development of a Space Engineers-plugin, leveraging the Intelligent Verification and Validation for Extended Reality-Based Systems (IV4XR) framework and extending the capabilities of the open-source scriptless testing tool TESTAR. Through this research, we unveil the potential of a scriptless *agent* to explore 3D sandbox game scenarios autonomously. Results demonstrate the effectiveness of an autonomous scriptless *agent* in achieving spatial coverage when exploring and (dis)covering elements within the 3D sandbox game.

## 1 INTRODUCTION

Computer games are dynamic and interactive systems designed to immerse and entertain users in captivating virtual environments. In the early 2020s, the video game industry surpassed 2 billion players worldwide, generating an impressive revenue of 120 billion dollars. This remarkable trend is anticipated to experience substantial growth in the future (Cooper, 2021).


Game testing predominantly relies on game testers, who invest significant manual effort and time verifying that user interactions within virtual scenarios yield the intended outcomes (Politowski et al., 2021). However, as industrial games grow in complexity, companies face the inherent limitations of human game testers' efforts. Automated approaches are needed to support game testers' manual efforts with complementary testing approaches (Pascarella et al., 2018). However, due to the intricate nature of games, which surpasses traditional software in complexity (Santos et al., 2018), the field of game systems lacks standardized frameworks to facilitate test automation.


Sandbox 3D games, such as the industrial game Space Engineers developed by Keen Software House and GoodAI companies, emphasize the freedom and


creativity of users in virtual scenarios. Players are given a wide range of tools and resources to shape the game scenarios according to their preferences and playstyle. The testing team of Space Engineers comprises ten game testers who excel in assessing functionality to create, destroy, modify, or interact with in-game objects, verify visual aspects, and manage game scenarios. Nevertheless, despite the testers are dedicated to performing numerous daily manual tests, the extensive range of in-game elements constrains their time for exploring and testing unforeseen scenarios.

This study evaluates the scriptless testing technique with the industrial sandbox game Space Engineers. *Scriptless testing* automatically generates test sequences at run-time to explore the System Under Test (SUT) by selecting and executing the available actions in the discovered states (Pastor Ricós, 2022). While this approach appears well-suited for 3D sandbox games, existing scriptless testing tools are primarily designed for desktop, web, and mobile applications. Adapting these techniques for game testing requires addressing distinctive 3D game features, such as precise position and orientation data for character movements and properties of interactive elements.

To bridge the gap between scriptless testing tools and technologies capable of discerning a game's states, we leverage the Intelligent Verification/Validation for Extended Reality Based Systems

<sup>a</sup> <https://orcid.org/0000-0002-5790-193X>

<sup>b</sup> <https://orcid.org/0000-0001-8025-0023>

<sup>c</sup> <https://orcid.org/0000-0002-6003-9113>

(IV4XR) framework (Prada et al., 2020).

In this paper, we extend the previous experiences of using the IV4XR framework (Prasetya et al., 2022). The research contributions are:

1. **Advancements in Scriptless Game System Testing.** This research provides insights into the IV4XR framework and scriptless testing tools components, contributing to the landscape of scriptless and game testing methodologies.
2. **Empirical Evaluation with an Industrial Game.** Through empirical evaluation, this study demonstrates the benefits of effective spatial coverage by developing sophisticated decision-making algorithms in autonomous scriptless *agents* to test the industrial Space Engineers game system.

These contributions are valuable for researchers and game development practitioners since they showcase how integrating autonomous exploratory *agents* can enable automated navigation and game testing.

The paper is structured as follows: Section 2 presents related work. Section 3 outlines the Space Engineers game and testing challenges. Section 4 details the integration within the IV4XR framework. Section 5 describes the extension of the scriptless testing tool TESTAR for games. Section 6 presents the empirical evaluation, and Section 7 concludes.

## 2 RELATED WORK

Compared to desktop (Pezze et al., 2018), web (García et al., 2020), and mobile (Kong et al., 2018) applications, for complex 3D game systems, there are no highly adopted automation framework or tools suitable to implement automated testing approaches.

ICARUS framework (Pfau et al., 2017) trains Reinforcement Learning (RL) *agents* to complete a 2D linear adventure game. In (Rani et al., 2023), an RL-BGameTester model was used to detect screen errors in a 2D Atari game. However, 2D games have simpler visuals and mechanics than 3D games. The absence of the third dimension eliminates complexities such as physics intended to simulate real-world scenarios.

Various RL approaches have been researched with diverse demo or gym-training 3D games. In (Gordillo et al., 2021), curiosity-driven RL *agents* move and jump in a self-crafted 3D game map to enhance spatial coverage and detect areas that stuck players. For gym-training semi-realistic games like ViZDoom (Kempka et al., 2016), the study (Ariyurek et al., 2022) use RL to train *agents* that simulate different personas to discover alternative play-style trajectory paths. Simi-

larly, in (Sestini et al., 2022), a curiosity and imitation RL approach is used to train *agents* that explore game areas while uncovering collision bugs and glitches. For Unreal Engine sample games, a pixel-based *agent* called Inspector (Liu et al., 2022) is employed to explore the game space using curiosity-based RL, resulting in the detection of two potential bugs.

Meanwhile, initial studies are using approaches to test open-source Virtual Reality (VR) Unity projects. The VRTest framework (Wang, 2022) streamlines the integration of various testing techniques using rotation, movement, and click trigger events. In (de Andrade et al., 2023), metamorphic tests and RL are used to identify collision and camera faults when moving the game character. Nevertheless, both approaches require further work to support wide types of events and to be evaluated with SUT not based on Unity.

In contrast to the previously mentioned studies, Space Engineers is a complex industrial 3D sandbox game in the market that involves a wide range of functional blocks and items with diverse properties.

RiverGame framework (Paduraru et al., 2022) uses various AI techniques to test visual, physical, and sound game aspects. The accuracy of visual AI techniques has been validated with demo and open-source games, and the voice testing detection rate approach with an industrial game. In contrast, our research evaluates the effectiveness of exploring an industrial 3D sandbox game while assessing the functional aspects of in-game objects.

Wuji framework (Zheng et al., 2019) uses evolutionary Deep RL to improve state exploration while accomplishing missions in 2D and 3D online combat games. Its effectiveness was demonstrated by detecting real injected bugs from previous versions and uncovering 3 new bugs. However, despite the availability of the Wuji open-source classes, the project lacks the documentation details necessary for seamless integration with other game systems and has had no recent activity since June 8, 2020.

In summary, some studies explored scriptless techniques to train RL *agents* for playtesting demo or gym-training games. Scriptless RL *agents* have been applied to real games in a few cases. Still, there is a lack of standardized open-source frameworks able to effectively identify the states of complex 3D games, which is essential to streamline test automation processes. Thus, our proposal goes beyond the state of the art in two key aspects: (i) We establish a connection to the game environment for robust observation of internal game objects and execution of actions that control the *agent* using game functions by leveraging the IV4XR open-source framework; (ii) We integrate an autonomous scriptless testing *agent* within a real

3D sandbox game that employs intelligent run-time algorithms to simulate the exploration experiences of real players and does not require pre-executing training iterations to learn how to play the game.

### 3 SPACE ENGINEERS

Space Engineers<sup>1</sup> is a sandbox 3D game developed by Keen Software House and GoodAI companies. The game is coded in C#, started its alpha release in 2013, transitioned to beta in 2016, and was officially released in 2019. In 2022, Space Engineers had an average of 5,000 players with peaks of more than 9,000 concurrent players. Between 2013 and mid-2023, the game has evolved over nearly 596 game build changes. The game is available on Steam and console platforms and has sold around 5 million units.

The Space Engineers game simulates realistic open-world 3D scenarios. Due to the nature of the sandbox game, there is no specific objective to finish the game. Users can explore planets in space, portray their idealistic spatial constructions, play challenging scenarios to survive, or collaborate and compete with other players.

#### 3.1 Game Mechanics

In Space Engineers, all game objects reside in a position and orientation of a three-dimensional world and have properties that indicate the object name, velocity, and unique identifier inside the game environment.

The astronaut is a playable game character that allows users to be part of and interact with the game environment. The astronaut has various characteristics such as energy, hydrogen, oxygen, and health, and capabilities like flying using the jet-pack. Due to the game recreating an open space world, the astronaut can move, fly, and rotate in 3D scenarios.

The game has atomic *Block* objects with properties representing attributes like type, integrity, volumetric physics, mass, inertia, and velocity. These blocks can be categorized into *functional* and *structural* blocks. *Functional* blocks have the capability of executing a task. For instance, this functional task can be producing energy for power blocks or restoring the character characteristics for life support blocks such as a medical room block. *Structural* blocks do not execute a task on their own but are used to build constructions. For example, armor structural blocks are used to build the floor and walls of space stations.

Constructions are known as *Grid* objects. A *grid*

can be as simple as a set of structural blocks that constitute the floor of a space station or a complex engine that extends the task capabilities of functional blocks. For example, a medical room that restores the astronaut's health can be connected with an O<sub>2</sub>/H<sub>2</sub> generator to additionally restore the astronaut's oxygen.

To construct blocks or sustain functional tasks, the astronaut needs so-called *Items* like: *Tools* used to interact with blocks and game mines; *Ores* mined from planets or asteroids using drill tools; *Materials* refined from ores into useful ingots; *Components* crafted from materials and required to construct blocks.

Figure 1 shows a Space Engineers scenario with a functional medical room connected to a functional O<sub>2</sub>/H<sub>2</sub> generator via structural conveyor blocks. The O<sub>2</sub>/H<sub>2</sub> generator can refine ice ores and supply oxygen to the connected medical room. This allows the astronaut to restore oxygen when interacting with the medical panel. However, if the integrity of the O<sub>2</sub>/H<sub>2</sub> generator is less than 80%, the ice ores cannot be refined, and the oxygen will not be supplied.

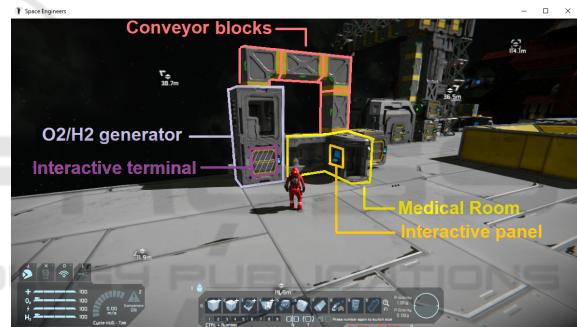


Figure 1: Space Engineers game scenario.

Space Engineers scenarios can be launched in *creative* and *survival* modes. *Creative* mode makes the astronaut invulnerable. His health, oxygen, hydrogen, and energy statistics will not decrease when resources are spent. Moreover, the astronaut can build blocks without the need to have the correct components in the inventory. In contrast, in *survival* mode, oxygen decreases over time, energy reduces with activity, hydrogen is spent when the jet-pack is used, and the astronaut can die if he/she loses its health points.

The variety of blocks and items, the diverse possible constructions to build with them, the scenarios game modes, and the 3D open-world movements make Space Engineers a highly complex game to test.

#### 3.2 Development & Manual Testing Cycle

A typical Space Engineers game development cycle takes about 3 or 4 months, depending on the extent

<sup>1</sup><https://www.spaceengineersgame.com/>

of game changes. This cycle involves two primary teams: developers and testers. The developers design and implement new game features and fix bugs reported by the testers during the release of new game versions. As developers finalize these changes, they open Jira tickets (Fisher et al., 2013) to point testers to the features that require testing.

In turn, testers process developers' Jira tickets to verify the functionality of new or updated game features and validate potential bug fixes. Additionally, they assist the game community in verifying and documenting possible problems that users encounter when crafting specific scenarios. The testers team comprises ten members with different expertise roles that include: *console port* (testing features in console); *scene and world creation* (ensuring scenarios can be created, saved, and loaded); *surround sound* (checking sound volume in scenarios); *player support* (simplifying bug reproduction steps reported by community users to assist developers in resolving issues).

Testers' proficiency in understanding game mechanics from players' perspectives is crucial to accurately reproduce scenarios during sandbox game feature validation. As a result, manual testing remains essential to ensure comprehensive game testing. However, testers lack sufficient time to manually test the extensive and diverse combinations of entity interactions in the game. Manually exploring blocks and items to reach spatial coverage can be prohibitively costly and time-consuming. In light of these challenges, it becomes relevant to investigate scriptless testing as an automated solution.

### 3.3 Scriptless Testing

Traditional testing approaches, such as manual testing, focus on evaluating specific scenarios by interacting with and verifying the expected behaviors of SUT elements. However, these approaches may not capture the full range of interactions and emergent behaviors that can emerge during gameplay. Scriptless testing techniques do not rely on explicit test case instructions. Instead, they employ algorithms to dynamically generate non-sequential actions during run-time, enabling them to explore and discover SUT objects autonomously. This introduces randomness and variability, which helps to complement traditional testing by uncovering unexpected issues and performing unanticipated combinations of interactions (Jansen et al., 2022; Bons et al., 2023).

Existing scriptless testing tools rely on different Graphical User Interface (GUI) technologies to detect the interactable elements. For example, TESTAR (Vos et al., 2021; Jansen et al., 2022) uses UIAutomata-

tion, WebDriver, Java Access Bridge, and Appium; Murphy (Aho et al., 2014) relies on UIAutomation and image recognition; GUI Driver (Aho et al., 2011) uses Jemmy Java library, Crawljax/ATUSA (Mesbah and Van Deursen, 2009) and Webmate (Dallmeier et al., 2012) use WebDriver; GUITAR (Nguyen et al., 2014) uses Java Accessibility, WebDriver, and UNO Accessibility; AUGUSTO (Mariani et al., 2018) and AutoBlackTest (Mariani et al., 2011) use IBM Functional Tester and Selenium.

For traditional GUI applications, the aforementioned technologies suffice with the identification of GUI elements that can be interacted with through keyboard and mouse inputs. However, these technologies fall short of meeting the necessary requirements for games. Game environments demand additional information to accurately identify their respective states, such as positional or orientation vectors for movements or properties associated with the objects being interacted with. Consequently, prior to employing scriptless testing for games, it becomes imperative to establish a connection between this testing approach and technologies capable of detecting the interactive elements within a game. We used the iv4XR framework described below to establish this connection.

## 4 SPACE ENGINEERS IN THE iv4XR FRAMEWORK

iv4XR is a Java framework with a plugin architecture that provides a set of interfaces that can be implemented to connect, get information, and interact with game objects. The *Entity* interface represents the existing game objects and their properties. The *Environment* interface allows connecting with game scenarios, observing the defined entities, and defining the actions that can be executed in the game. The *agent* can be any automated software testing tool that uses the environment to connect with the game and takes the role of a playable entity to observe the game entities, execute game actions, and apply test oracles.

These iv4XR interfaces streamline the development of game plugins. The development of the iv4XR Space Engineers-plugin<sup>2</sup> enables access to the internal data and functions of the game. In Space Engineers, the *agent* takes the role of the astronaut.

The Space Engineers-plugin consists of server and client components. The server-side is implemented in C#, has 8462 lines of code (LOC), and allows the connection with the game by defining the properties and controller functions of game objects. The client-side

<sup>2</sup><https://github.com/iv4xr-project/iv4xr-se-plugin>



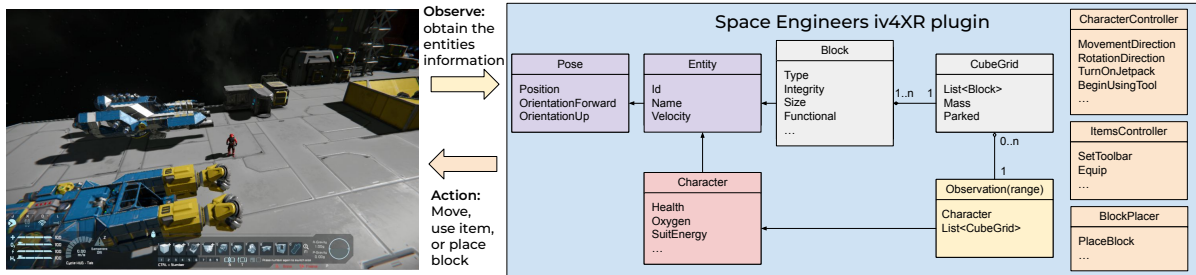


Figure 2: Space Engineers-plugin overview.

is written in Kotlin to ensure interoperability between the C# game and the Java IV4XR framework, has a size of 17671 LOCs, and provides classes that grant access to game data and logical functions like navigation for the *agents*. Figure 2 shows an overview of the plugin classes, which are discussed below.

#### 4.1 Space Engineers-Plugin Entities

In the Space Engineers environment, each game object is represented as an *Entity* that stands in a specific *Pose*. The *Pose* denote their position and orientation within the game, and the *Entity* properties indicate each object’s identifier, name, and velocity.

*Block* extends *Entity* with properties that indicate the block’s type, integrity, and size, together with an attribute that indicates if the block is of the functional category (e.g., power block or medical room). *CubeGrid* contains the list of blocks that compose a grid (e.g., a spaceship grid is composed of a cockpit, thruster, and power blocks), and properties representing the grid mass and if the grid is parked (e.g., a spaceship is parked or is being controlled). The *Character* entity extends *Entity* properties with the astronaut’s characteristics of health, oxygen, energy, etc.

#### 4.2 Space Engineers-Plugin Observation

The *agent* connects with the *Character* through the Space Engineers-plugin to *Observe* the Space Engineers environment. The *Character* is always present. The existence of *CubeGrid* and *Block* entities depends on a configurable observation range of the *agent* and its distance from the game objects. Figure 3 shows how the observation range, a 3D sphere, works in the Space Engineer’s environment. The *agent* observes itself, the main platform grid, and one spaceship grid. As we have explained, the grids are composed of a set of block entities. The spaceship grid is composed of a cockpit block, thruster block, power block, etc. The grid platform is composed of a group of structural

*ArmorBlock* representing the floor of the scenario, together with functional *MedicalBlock* and *PowerBlock*.

#### 4.3 Space Engineers-Plugin Actions

The Space Engineers-plugin allows the *agent* to control the *Character* to interact with the game. The plugin controllers call internal game functions to move or rotate the character, turn on/off the jet-pack, equip/unequip an item, place a block, etc. To invoke these controls, the *agent* executes *commands*.

Observing the entity’s data within the game environment and executing *commands* by invoking game controls make the IV4XR Space Engineers-plugin a more robust approach compared to the usage of keyboard and mouse inputs or visual recognition tools that lack access to the internal game data. However, *commands* are insufficient even to accomplish simple tasks such as grinding a block. It is necessary to group sequences of *commands* in *actions*. For example, an *action* to grind a block is composed by the *commands*: Find the grinder tool, equip the grinder, aim the block, start using the grinder, and stop using the grinder.

#### 4.4 Space Engineers-Plugin Navigation

Moving the *agent* within the game is a complex task, as it requires the *agent* to perceive which positions are obstructed/walkable to determine a path of positions to reach the desired entity. In the IV4XR framework, this functionality is known as *navigation*.

Some game engine platforms, such as Unity, can automatically build a *navigation* mesh<sup>3</sup> that contains the walkable positions of the game by using the virtual objects geometry. Pathfinding algorithms can then optimize the traversal of these navigable mesh nodes to reach a desired position (Cui and Shi, 2011).

The IV4XR framework provides an A\* pathfinding algorithm to efficiently find the best path between two nodes within a *navigation* mesh. Nonetheless, the initial version of the Space Engineers-plugin did

<sup>3</sup><https://docs.unity3d.com/es/Manual/Navigation.html>

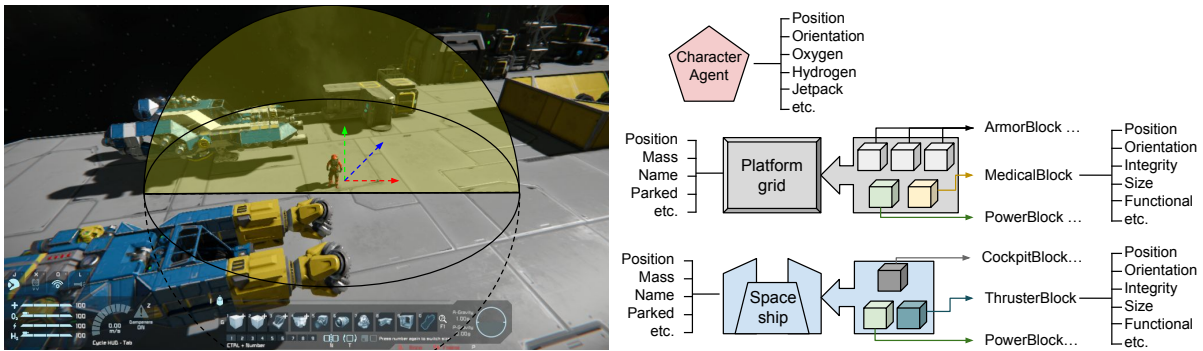


Figure 3: Space Engineers observed environment by the agent.

not have the capability to create a default *navigation* mesh (Prasetya et al., 2022). Instead, it constructed a *navigation* graph on-the-fly using the geometry information of observed entities. This approach has drawbacks, as it incurs a significant time cost after each game exploration movement and is not robust in three-dimensional space, where the *agent* could dynamically change its orientation.

To enhance game *navigation*, the Space Engineers team introduced automatic graph calculation for each *CubeGrid* entity. This calculation generates a list of positions the *agent* could reach without obstructions. In Figure 4, the *agent* observes the navigable positions, allowing him to create actions with a path of command movements to reach the interactive entities.

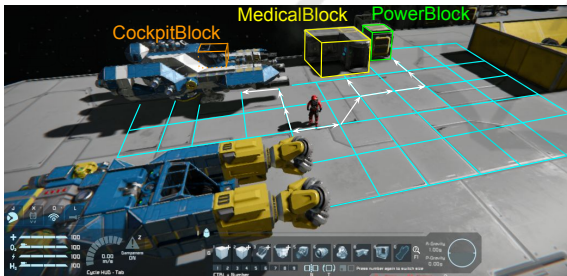


Figure 4: Navigable actions to reach interactive entities.

## 5 TESTAR FOR GAME TESTING

Once the IV4XR Space Engineers-plugin resolves the technical prerequisites necessary for detecting the state, it is necessary to integrate this capability into a scriptless automation tool. We select the TESTAR tool (Vos et al., 2021) since it already initiated the integration process with the IV4XR framework (Pastor Ricós, 2022; Prasetya et al., 2022), making it a fitting choice to continue the integration efforts.

TESTAR is an open-source tool for scriptless GUI testing that automatically obtains the state of desktop, web, and mobile applications, derives and

executes GUI interactions such as click, type, or drag, and applies oracles to check if the system responds correctly. To be able to interact with the Space Engineers game, TESTAR has been extended to integrate the IV4XR Space Engineers-plugin from Fig. 2.

TESTAR launches the Space Engineers game as a Windows executable, then connects to it using the Space Engineers-plugin and loads the desired scenario. Subsequently, it starts a cyclic flow that can generate multiple test sequences of various actions until a STOP condition is met (e.g., perform a maximum number of actions). The operational flow steps of the TESTAR *agent* are shown in Figure 5:

- **First**, it observes the entities that constitute the game *state*.
- **Second**, it derives all the available actions that can be executed for each entity.
- **Third**, based on the available derived actions, it decides *what to do next?* by selecting one of the derived actions using an Action Selection Mechanism (ASM).
- **Fourth**, it executes the selected action and applies a series of oracles to verify the robustness of the system and the functional aspects of the game entities.

TESTAR has a Java class called protocol that contains the methods corresponding to each of these four steps. A tester can, for example, change the ASM by plugging a different one into that Java protocol.

### 5.1 TESTAR Agent: Game State

The TESTAR *agent* employs the Space Engineers-plugin to actively *Observe* all the game entities that reside in the observation range area. Each game entity contains a set of aforementioned properties, such as position and orientation for all entities; health and oxygen for the *Character* entity; and type and integrity for *Block* entities. Together, these observed entities and their properties constitute the game *state*.

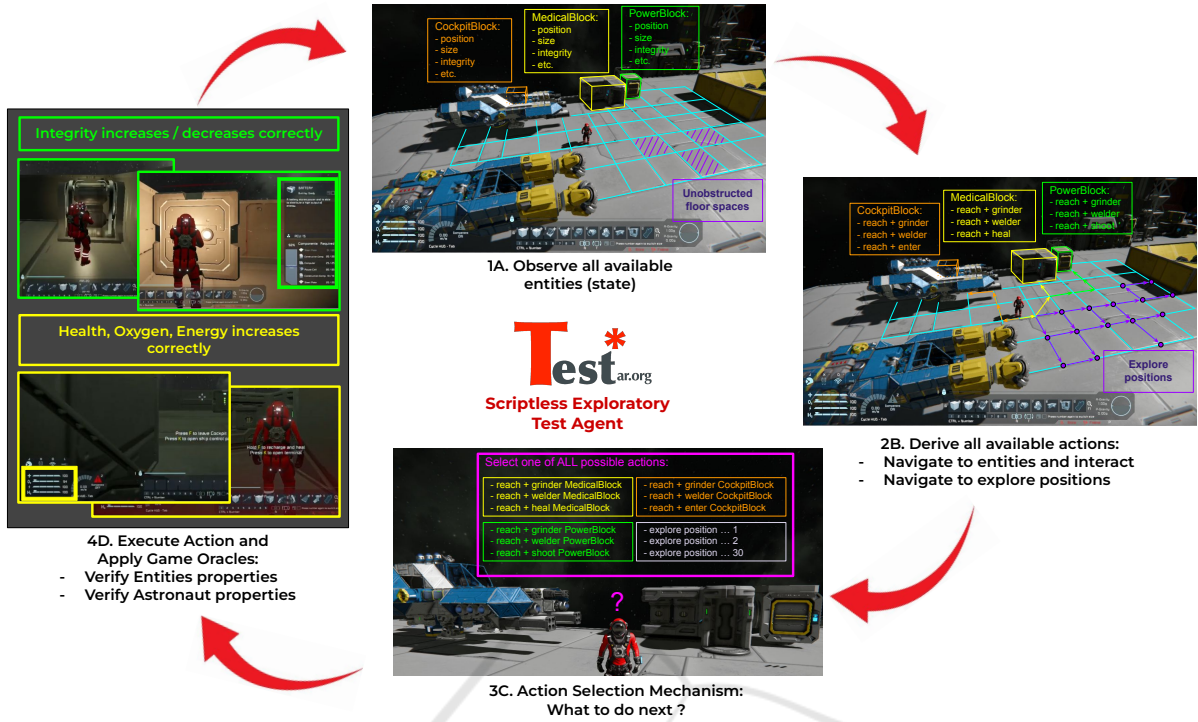


Figure 5: TESTAR operational flow with Space Engineers.

## 5.2 TESTAR Agent: Derived Actions and Navigation

Depending on the type and other entity properties, the TESTAR agent derives all the available actions that can be executed for each entity. For example, to grinder or welder all non-structural *ArmorBlock* entities, or to interact with *MedicalBlock* or *CockpitBlock* functional blocks to restore oxygen.

A distinctive characteristic between testing traditional GUI software and games is the need to reach the desired entity to interact, as well as explore through states where no functional blocks exist in the observed area to potentially discover new game entities. While the Space Engineers-plugin's *navigation* graph and the A\* pathfinding algorithm provided by the IV4XR framework facilitate finding the optimal movement path between initial and destination nodes, there is a necessary decision-making step at the top level to determine which action to derive and select during the exploration process.

To reach the desired *Block* to interact, the TESTAR agent exploits the *navigation* capabilities of the Space Engineers-plugin to observe the unobstructed floor spaces and calculates if there is a *navigable* path of positions that can be followed to reach the *Block*. If so, TESTAR derives an *action* that *navigates* the path, rotates to aim the *Block*, and interacts with the

*Block* using a *Tool*. However, if the *Block* is not reachable, TESTAR does not even try to derive an interaction with the *Block action*.

To potentially discover new game entities, the TESTAR agent not only considers deriving actions that interact with observed reachable *Blocks* but also derives actions that explore unobstructed positions. To do this, TESTAR's protocol has been extended so that after deriving all available interaction actions with unobstructed *Blocks*, it also derives all available exploration actions to unobstructed positions.

## 5.3 TESTAR Agent: Action Selection Mechanism

After deriving all available actions, the TESTAR agent uses, by default, a random ASM to decide which action to execute next. Although random ASMs have proven practical for traditional software (Vos et al., 2021), for exploring 3D sandbox games, it is necessary to research on more sophisticated ASMs.

Let us consider the example in Figure 5. First, the TESTAR agent observes 3 functional blocks (Cockpit, Medical, and Power) and derives 3 different actions for each block to navigate and interact with. This computes a total of 9 navigate and interact actions with functional blocks. Second, because there are 30 unobstructed positions in the observation area



(e.g., imagine there are 30 purple dots), the TESTAR *agent* derives other 30 available exploration actions.

A random ASM will have less than 25% probability of selecting one of the 9 available interaction actions from the 39 total actions. This increases the chance of selecting an exploration action to more than 75%. Moreover, within the set of available exploration actions, selecting remote areas that remain unexplored can potentially allow the TESTAR *agent* the discovery of new entities. To enhance the exploration of unexplored areas, we have developed the so-called Interactive Explorer ASM depicted in Algorithm 1.

Algorithm 1: Interactive Explorer ASM.

---

<b>Data:</b> <i>interacted</i>	▷ List the interacted entities
<b>Data:</b> <i>explored</i>	▷ Area of explored position
<b>Data:</b> <i>actions</i>	▷ All available state-actions

---

```

1 if actions contains entities that were not interacted then
2   nearEntity ← nearestEntity(actions) ;
3   a ← select to navigate and interact with the
   nearEntity ;
4   save nearEntity as interacted ;
5 else if actions contains positions that were not explored
   then
6   remotePos ← remotePosition(actions) ;
7   a ← select to navigate to explore the remotePos ;
8   save remotePos as explored ;
9 else
10  a ← random selection from all actions ;
11 end
12 return a                                ▷ Return the selected action

```

---

The Interactive Explorer ASM tracks a list of *interacted* entities and an area containing the *explored* positions. First, the ASM checks whether the set of available *actions* contains an action that interacts with a non-interacted entity (line 1). In that case, because there can be several non-interacted entities, it prioritizes choosing the nearest entity (*nearEntity*) to the *agent* (line 2). Thus, the ASM selects the action that navigates and interacts with the *nearEntity* (line 3), saves this *nearEntity* as interacted to not to be prioritized in the next iterations (line 4), and finally, returns the selected action (line 12).

Second, the ASM checks whether the set of available *actions* contains an action that explores a position out of the *explored* area (line 5). If so, because there can be several unexplored positions, it prioritizes choosing the remote position (*remotePos*) to the *agent* position (line 6). Consequently, the ASM selects the action that navigates and explores the *remotePos* (line 7) and includes the position in the *explored* area to enhance selecting other unexplored positions in the next iterations (line 8). Finally, the ASM returns the selected action (line 12). In case the actions do not contain a non-interacted entity or non-

explored position (line 9), the ASM selects (line 10) and returns an action randomly (line 12).

Different ASMs can be configured in the Java protocol of TESTAR. This way, the game testers can adjust the decision-making of the *agent* based on the requirements of different Space Engineers scenarios or testing objectives.

## 5.4 TESTAR Agent: Oracles

TESTAR integrates generic oracles intended to verify the robustness of the SUT: detect if the process has crashed or hung or if the state elements, or debugging logs, contain suspicious exception messages. Although these generic oracles are a good way to start with automated scriptless testing, for Space Engineers, it is of paramount importance to test also the functional aspects of the game entities.

Examples of oracles can be to check that the integrity of all blocks decreases after grinding or shooting or increases after welding; that the *agent's* health, oxygen, hydrogen, and energy are restored when interacting with medical rooms or cockpits; or that the jet-pack and the dampeners are not enabled automatically without player activation after entering a cockpit, medical room, or interacting a ladder.

These oracles have been studied in (Prasetya et al., 2022). In this paper, we apply oracles that validate the integrity of blocks, but we mainly emphasize evaluating the effectiveness of ASMs exploration.

## 6 SCRIPTLESS GAME TESTING EVALUATION

In order to assess the efficacy of scriptless testing for exploring the Space Engineers game, we evaluate the potential benefits of investing time and effort in developing ASMs for more sophisticated exploration techniques. To accomplish this, we quantitatively measure the spatial coverage of discovered and interacted entities and navigated positions within a randomly generated scenario. To guide our study, we have formulated a research question and null hypothesis:

RQ: *How effective is spatial exploration in the Space Engineers game when using different TESTAR ASMs?*  
*H*<sub>0</sub>: *The Interactive Explorer ASM is not more effective than a random ASM.*

We designed a controlled experiment based on Wohlin's guidelines (Wohlin et al., 2012) and a methodological framework specifically built to evaluate software testing techniques (Vos et al., 2012).

The experiment consists of running the random default ASM and the more intelligent decision-



making Interactive Explorer ASM from Algorithm 1 that prioritizes the interaction with newly discovered blocks and the exploration of remotely unexplored areas. Each trial measures spatial coverage of discovered Space Engineers blocks and floor positions.

## 6.1 Space Engineers Generated Scenario

A randomly generated Space Engineers scenario was used to ensure that the evaluated ASMs were not biased. The scenario consists of a 100x100 map with 8157 navigable positions and obstructive walls that TESTAR must navigate around to reach interactive blocks. The number of interactive blocks is randomly placed in a uniform distribution in various reachable parts of the map. From the 62 blocks specified by the company as fundamental for manual testing, we chose 16 types of 1x1 blocks that were allowed to be placed in the random scenario creation. Gravity blocks are also included to simulate gravity, resulting in 313 functional blocks of 17 different types.

## 6.2 Independent Variables

To focus on the exploratory capabilities of the ASMs and prevent the *agent* from dying, we load the generated scenario in creative mode.

The TESTAR *agent* can use diverse tools and weapons to test the integrity of blocks. However, since this study focuses on spatial exploration, we applied the *blocking principle* (Wohlin et al., 2012) to limit the TESTAR *agent* interactions to a grinder tool that verifies that the integrity of functional blocks decreases and shooting one bullet at gravity blocks to reduce their integrity without destroying their gravitational functionality. We also limited the observation range through the Space Engineers-plugin to encourage exploring and discovering new blocks.

## 6.3 Dependent Variables

To answer our research question, we measured the number of discovered and interacted blocks, and the observed and walked positions. The Space Engineers game stores the scenario data in local XML files. This data contains information about the floor positions and existing blocks, and we compare it with real-time observations during the exploration process to obtain spatial coverage. Using this data, we can generate a 2D map highlighting the covered space.

Figure 6 shows an example of one exploratory sequence of 500 actions in the experimental map. Regarding navigable positions, yellow squares represent

floor positions, red squares obstructive walls, blue circles observed areas, and green dots walked positions. Then, about interactive blocks, magenta dots represent not observed blocks, pink dots observed but not interacted blocks, and orange dots interacted blocks.

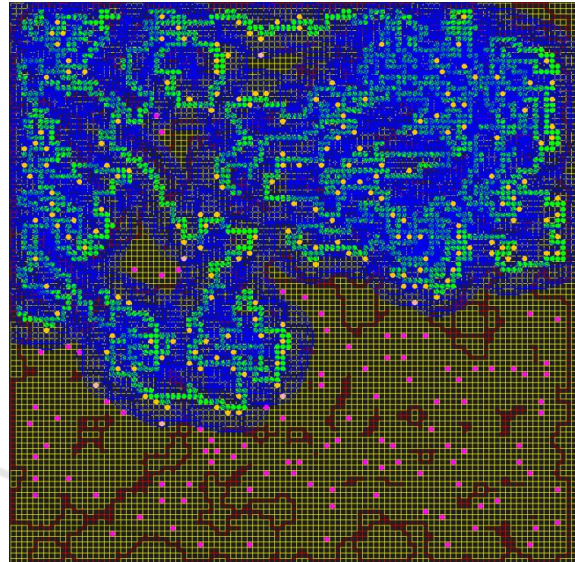


Figure 6: Space Engineers spatial coverage map.

## 6.4 Design of the Experiment

We evaluate the random and the Interactive Explorer ASMs by executing an exploration of 500 actions on the generated scenario. We repeated the exploration process 30 times for each ASM. For each new execution, we reload the same initial conditions in the same Windows machine with 8 CPU cores and 16 GB RAM. We obtain independent spatial coverage metrics for each execution and accumulative spatial coverage metrics for the 30 executions of the different random and Interactive Explorer ASMs.

## 6.5 Results

We first present the spatial coverage achieved in the 30 independent runs. Next, we compare the accumulative spatial coverage of the two ASMs. Finally, we use the Wilcoxon test to determine whether there is a significant difference between the ASMs. The experiments were performed in Space Engineers v201.14. The replication package can be found here <sup>4</sup>.

Figure 7 shows the results for the observed and interacted blocks. Each line represents one of the 30 independent runs. The random ASM achieved a coverage ranging from 8% to 30% for observed blocks and

<sup>4</sup><https://doi.org/10.5281/zenodo.10683676>

3% to 8% for total interacted blocks. In contrast, the Interactive Explorer ASM achieved coverage ranging from 54% to 82% for observed blocks and 52% to 77% for total interacted blocks.

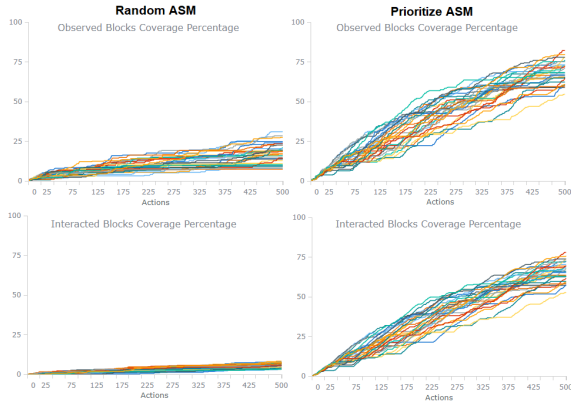


Figure 7: Observed and interacted blocks coverage.

Figure 8 shows the results for the observed and walked floor positions. The random ASM achieved a coverage ranging from 7% to 30% for observed positions and 4% to 12% for walked positions. In comparison, the Interactive Explorer ASM achieved a coverage ranging from 54% to 77% for observed positions and 21% to 24% for walked positions.

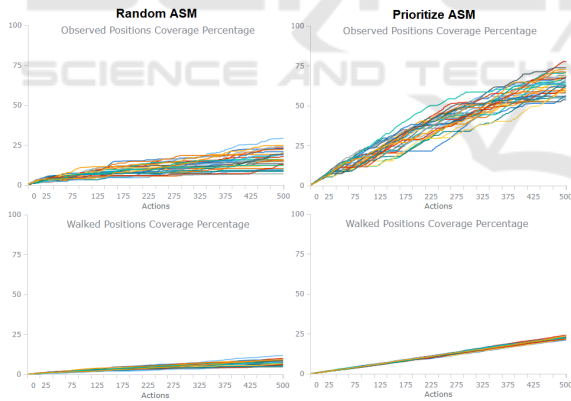


Figure 8: Observed and walked floor positions coverage.

Figure 9 compares the accumulative spatial coverage achieved by both ASMs over the 30 exploratory runs. The Interactive Explorer ASM achieves over 95% of observed and interacted blocks and observed floor positions around the 2000 executed actions, corresponding to the combination of 4 independent executions. At the end of the 30 exploratory runs, it reached 88% of walked positions. In contrast, the random ASM requires over 5000 actions to observe 50% of the existing blocks and floor positions, over 12000 actions to interact with 50% of blocks, and achieves

only about 54% of walked floor positions at the end of the 30 runs. Due to the random uniform distribution of blocks when creating the experimental map, we can appreciate that the observed blocks and positions curves grew similarly during the exploration.

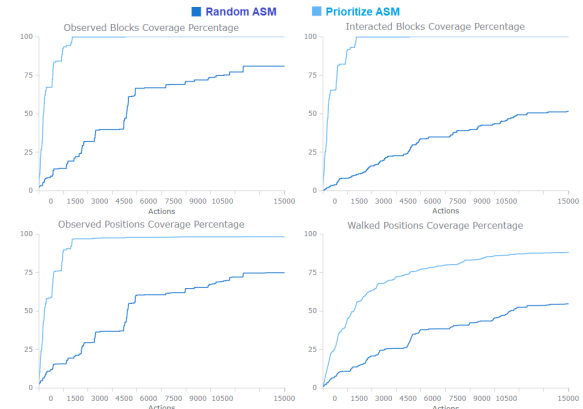


Figure 9: Accumulative spatial coverage comparison.

The Interactive Explorer ASM outperforms the random ASM by prioritizing interacting with newly observed blocks and calculating efficient routes to unexplored floor areas. Table 1 shows Wilcoxon test results to verify a significant difference between the two ASMs. We extracted values from the 30 different runs when executing 100, 300, and 500 actions. This means we calculate the significant difference in 3 different moments of the exploratory process. For the observed and interacted blocks and the observed and walked positions, the Wilcoxon test results show a p-value of less than 0.05, indicating that the Interactive Explorer ASM is statistically superior to the random ASM. This allows us to reject  $H_0$  and confirm that investing time and effort in developing intelligent ASMs benefits TESTAR exploration effectiveness.

Table 1: Wilcoxon p-value significant difference.

Wilcoxon test p-values results		
Executed actions	Observed Blocks	Interacted Blocks
100 actions	p=1.730e-06	p=1.718e-06
300 actions	p=1.732e-06	p=1.734e-06
500 actions	p=1.730e-06	p=1.729e-06
Executed actions	Observed Positions	Walked Positions
100 actions	p=1.734e-06	p=1.733e-06
300 actions	p=1.734e-06	p=1.732e-06
500 actions	p=1.734e-06	p=1.734e-06

## 6.6 Threats to Validity

We discuss some threats to validity according to (Wohlin et al., 2012; Ralph and Tempero, 2018).

**Construct Validity.** For the exploratory evaluation, we use the information from the Space Engineers scenario to design the concept of spatial coverage. Then, we use this data to measure the effectiveness of the random and Interactive Explorer ASMs. Although this spatial coverage is a self-design benchmark, the metrics come from the Space Engineers game's data.

**Content Validity.** For the exploratory evaluation, the spatial coverage measures the existing blocks and floor positions over a 2D scenario space. Still, there are various types of blocks, and the game environment allows 3D motions. Although more sophisticated spatial coverage metrics can be researched in the future, the obtained 2D metrics allow us to measure the effectiveness of the exploratory ASMs. Moreover, while we did not encounter any bugs related to the integrity of the functional blocks used, it's important to emphasize that our solution effectively covered the scenario space. The lack of failure detection may be attributed to the random distribution of the test scenario or the absence of issues in the types of blocks utilized.

**Internal Validity.** For the exploratory evaluation, we launched the Space Engineers scenario in creative mode to avoid the astronaut dying and provide enough ammo items to realize the shoot gun actions.

**External Validity.** The empirical study has been realized with the highly complex Space Engineers game. Even though we demonstrated that the TESTAR *agent* has exploratory capabilities to navigate and test Space Engineers automatically, we consider this to be a first step regarding game scriptless test automation. Moreover, to facilitate the generalization of our results, we use the architectural analogy (Wieringa and Daneva, 2015) since we carefully describe the components of the case and the corresponding interactions, such as the game system and the scriptless tool with the corresponding configuration.

**Conclusion Validity.** Due to the degree of randomness in the action selection of the exploratory ASMs, we cannot assume normal distribution in the experiments (Arcuri and Briand, 2011). To address this, we repeated the exploration 30 times and used Wilcoxon statistical non-parametric tests on the results.

## 7 CONCLUSION

Computer 3D sandbox games, such as Space Engineers, are complex games characterized by a multitude of in-game features and entities. Manual testing of these games poses challenges due to time and resource constraints, especially when exploring and testing unforeseen scenarios or large combinations of gameplay interactions.

In this paper, we have showcased the automated scriptless exploration of an industrial 3D sandbox game using TESTAR and iv4XR. This work shows the value of implementing TESTAR's ASMs as reusable artifacts at a high abstraction level. These ASMs prove to be effective in enhancing game navigation and testing capabilities by guiding the *agents* toward specific areas of the game. Our research showcases the advantages of using an intelligent ASM as a powerful tool for optimizing spatial coverage. Implementing different ASMs allows directing the *agents* toward specific parts of the game to achieve more comprehensive coverage and uncover potential issues that might have been overlooked otherwise.

This paper demonstrates that with a dedicated *navigation* layer, an autonomous scriptless *agent* can effectively reach and test game entities, and it is possible to exercise automated exploration of scenarios without training the *agent* to play the game.

We consider the integration of the TESTAR *agent* as a first step in the inclusion of intelligent scriptless testing *agents* for games. Future research is planned to use high-level artifacts like ASMs to promote the reusability and maintainability of the testing framework. We plan to study if ASMs can be adapted to different scenarios, reducing the effort required to configure the testing environment. Finally, we will continue the research with future experiments to extrapolate our results to other games in the market.

## ACKNOWLEDGEMENTS

This work has been partially funded by the iv4XR H2020 project and the ENACTEST project.

## REFERENCES

- Aho, P., Menz, N., Rätty, T., and Schieferdecker, I. (2011). Automated java gui modeling for model-based testing purposes. In *2011 8th ITNG*, pages 268–273. IEEE.
- Aho, P., Suarez, M., Kanstrén, T., and Memon, A. (2014). Murphy tools: Utilizing extracted gui models for industrial software testing. In *IEEE 7th ICST Workshops*, pages 343–348.
- Arcuri, A. and Briand, L. (2011). A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *33rd ICSE*, page 1–10. ACM.
- Ariyurek, S., Surer, E., and Betin-Can, A. (2022). Playtesting: What is beyond personas. *IEEE Transactions on Games*, pages 1–1.
- Bons, A., Marín, B., Aho, P., and Vos, T. E. (2023). Scripted and scriptless gui testing for web applications: An industrial case. *Information and Software Technology*, 158:107172.



- Cooper, K. M. (2021). *Software Engineering Perspectives in Computer Game Development*. CRC Press.
- Cui, X. and Shi, H. (2011). A\*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, 11(1):125–130.
- Dallmeier, V., Burger, M., Orth, T., and Zeller, A. (2012). Webmate: a tool for testing web 2.0 applications. In *Workshop on JavaScript Tools*, pages 11–15.
- de Andrade, S. A., Nunes, F. L., and Delamaro, M. E. (2023). Exploiting deep reinforcement learning and metamorphic testing to automatically test virtual reality applications. *STVR*, 33(8):e1863.
- Fisher, J., Koning, D., and Ludwigsen, A. (2013). Utilizing atlassian jira for large-scale software development management. Technical report, LLNL, Livermore, CA (United States).
- García, B., Gallego, M., Gortázar, F., and Munoz-Organero, M. (2020). A survey of the selenium ecosystem. *Electronics*, 9(7):1067.
- Gordillo, C., Bergdahl, J., Tollmar, K., and Gisslén, L. (2021). Improving playtesting coverage via curiosity driven reinforcement learning agents. In *Conference on Games (CoG)*, pages 1–8. IEEE.
- Jansen, T., Ricós, F. P., Luo, Y., van der Vlist, K., van Dalen, R., Aho, P., and Vos, T. E. (2022). Scriptless gui testing on mobile applications. In *22nd QRS*, pages 1103–1112. IEEE.
- Kempka, M., Wydmuch, M., Runc, G., Toczek, J., and Jaśkowski, W. (2016). Vizdoom: A doom-based ai research platform for visual reinforcement learning. In *Symposium on computational intelligence and games (CIG)*, pages 1–8. IEEE.
- Kong, P., Li, L., Gao, J., Liu, K., Bissyandé, T. F., and Klein, J. (2018). Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability*, 68(1):45–66.
- Liu, G., Cai, M., Zhao, L., Qin, T., Brown, A., Bischoff, J., and Liu, T.-Y. (2022). Inspector: Pixel-based automated game testing via exploration, detection, and investigation. In *CoG*, pages 237–244. IEEE.
- Mariani, L., Pezzè, M., Riganelli, O., and Santoro, M. (2011). Autoblacktest: A tool for automatic black-box testing. In *33rd ICSE*, pages 1013–1015. ACM.
- Mariani, L., Pezzè, M., and Zuddas, D. (2018). Augusto: Exploiting popular functionalities for the generation of semantic GUI tests with oracles. In *40th ICSE*, page 280–290. ACM.
- Mesbah, A. and Van Deursen, A. (2009). Invariant-based automatic testing of ajax user interfaces. In *31st ICSE*, pages 210–220. IEEE.
- Nguyen, B. N., Robbins, B., Banerjee, I., and Memon, A. (2014). Guitar: an innovative tool for automated testing of gui-driven software. *Automated software engineering*, 21:65–105.
- Paduraru, C., Paduraru, M., and Stefanescu, A. (2022). Rivergame-a game testing tool using artificial intelligence. In *15th ICST*, pages 422–432. IEEE.
- Pascarella, L., Palomba, F., Di Penta, M., and Bacchelli, A. (2018). How is video game development different from software development in open source? In *15th MSR*, pages 392–402.
- Pastor Ricós, F. (2022). Scriptless testing for extended reality systems. In *16th RCIS*, pages 786–794. Springer.
- Pezze, M., Rondena, P., and Zuddas, D. (2018). Automatic gui testing of desktop applications: an empirical assessment of the state of the art. In *ISSTA/ECOP 2018 Workshops*, pages 54–62.
- Pfau, J., Smeddinck, J. D., and Malaka, R. (2017). Automated game testing with icarus: Intelligent completion of adventure riddles via unsupervised solving. In *CHI PLAY'17 Extended Abstracts*, pages 153–164.
- Politowski, C., Petrillo, F., and Guéhéneuc, Y.-G. (2021). A survey of video game testing. In *2nd AST*, pages 90–99. IEEE.
- Prada, R., Prasetya, I., Kifetew, F., Dignum, F., Vos, T. E., Lander, J., Donnart, J.-y., Kazmierowski, A., Davidson, J., and Fernandes, P. M. (2020). Agent-based testing of extended reality systems. In *13th ICST*, pages 414–417. IEEE.
- Prasetya, I., Pastor Ricós, F., Kifetew, F. M., Prandi, D., Shirzadehhajimahmood, S., Vos, T. E., Paska, P., Hovorka, K., Ferdous, R., Susi, A., et al. (2022). An agent-based approach to automated game testing: an experience report. In *13th A-TEST Workshop*, pages 1–8.
- Ralph, P. and Tempero, E. (2018). Construct validity in software engineering research and software metrics. In *22nd EASE*, pages 13–23. ACM.
- Rani, G., Pandey, U., Wagde, A. A., and Dhaka, V. S. (2023). A deep reinforcement learning technique for bug detection in video games. *International Journal of Information Technology*, 15(1):355–367.
- Santos, R. E., Magalhães, C. V., Capretz, L. F., Correia-Neto, J. S., da Silva, F. Q., and Saheer, A. (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 ESEM*, pages 1–10. ACM/IEEE.
- Sestini, A., Gisslén, L., Bergdahl, J., Tollmar, K., and Bagdanov, A. D. (2022). Automated gameplay testing and validation with curiosity-conditioned proximal trajectories. *IEEE Transactions on Games*.
- Vos, T., Aho, P., Pastor Ricos, F., Rodriguez-Valdes, O., and Mulders, A. (2021). testar—scriptless testing through graphical user interface. *STVR*, 31(3):e1771.
- Vos, T. E., Marín, B., Escalona, M. J., and Marchetto, A. (2012). A methodological framework for evaluating software testing techniques and tools. In *12th QSIC*, pages 230–239. IEEE.
- Wang, X. (2022). Vrtest: an extensible framework for automatic testing of virtual reality scenes. In *ACM/IEEE 44th ICSE Companion*, pages 232–236.
- Wieringa, R. and Daneva, M. (2015). Six strategies for generalizing software engineering theories. *Science of computer programming*, 101:136–152.
- Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in software engineering*. Springer.
- Zheng, Y., Xie, X., Su, T., Ma, L., Hao, J., Meng, Z., Liu, Y., Shen, R., Chen, Y., and Fan, C. (2019). Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In *34th ASE*, pages 772–784. IEEE.