POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

How to Measure Game Testing: a Survey of Coverage Metrics and an Implementation on the iv4XR Framework

Politecnico di Torino

Supervisors

prof. Riccardo Coppola

prof. Francesco Strada

dr. Tommaso Fulcini

Candidate

Serenella Manzi

Academic year 2023-2024

Summary

With the increase in their popularity, video games have evolved to become highly complex products that require ever larger funds and huge, diverse teams to be developed. Due to the growing complexity of the software, however, video games are highly prone to bugs, often appearing on day one and sometimes in such large quantities that they cause release delays and negatively impact the product's economic success. To mitigate this issue, it is essential to conduct a structured testing phase to assess and evaluate the software's quality.
In the video game industry, automated testing using autonomous agents holds particular potential. These tools are trained with artificial intelligence to explore the game environment and test various gameplay modes and situations by dynamically changing their strategies.
However, the literature includes few studies on the application of automated testing to video games, and there is a complete lack of attempts to classify and standardize specific coverage metrics for video games. This gap makes very difficult to assess the quality of testing and compare results.
Hence the motivation for this work, which consists of two objectives: a first phase of literature review aimed at identifying and classifying specific coverage metrics for video games, providing a foundation for building testing models that address the core aspects of a video game; and a second implementation phase, during which several gameplay coverage metrics were implemented using iv4XR, a promising open-source framework that employs autonomous agents for testing Extended Reality systems.
The results of the first phase is a taxonomy of 26 specific metrics employed in video game testing, grouped into six categories depending to the domain to which they relate.
In the second phase, some of these metrics were implemented in iv4XR using their designated testing game, LabRecruits, and evaluated on both newly written test cases and demo tests provided by the developers.
From the results of this second phase, it becomes clear that currently it is

not yet possible to build a general coverage model that is applicable to every video game, due to limitations arising from the specificities that differentiate video games and from existing testing frameworks, which are often closely tied to the specific game or require, at the very least, the development of a specific interface between the framework and the game to be tested (as in the case of iv4XR).
Future work could focus on reducing these limitations by using automated testing to efficiently test as many video games as possible, leveraging and expanding the proposed taxonomy of metrics, developing tools that natively implement said taxonomy and test environments that are increasingly independent from specific games.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, video games have gained popularity, resulting in the growth of the industry from niche markets to mainstream. In 2022 the game industry has generated 182.9 billion US dollars in worldwide revenue, with an estimated growth of +2.9% by 2025 [1]. Nowadays video games are incredibly complex products, requiring the combination of aspects such as captivating design, increasingly realistic graphics, and non-deterministic behavior to be entertaining across multiple platforms and devices.

However, with this growing complexity, it is almost inevitable to have a good amount of bugs. An emblematic example in this sense is the case of Cyberpunk 2077 [2]: released after years of development and delays, it received heavy criticism for multiple and significant bugs, to the extent that it forced the producer to issue refunds and release immediate corrective patches.
To launch a successful game its quality is essential, and to assure quality, testing is crucial. Accurate testing can prevent the game from being released in a buggy state that would make it a commercial flop [3].

In traditional software development, tests and their automation are considered a crucial part. This is not yet true for video game development, though, where there is a lack of comprehensive testing, often limited only to manual playtesting. In recent years, a good number of testing tools and frameworks has been created, based on a variety of approaches such as machine learning models, reinforcement learning, autonomous agents, exploration. However, it seems that most researchers are more interested in the performance of testing tools rather than focusing on testing itself; also, game developers still rely mostly on manual testing, remaining skeptical about other approaches [4].

The lack of a clear and standard testing method in video games development is evident from scientific literature. The aim of this work is to build a testing model which can provide an overall coverage of the main aspects to be tested in a video game. This model is useful as a structured basis to later build a game-specific and more articulate testing process during the game development. After conducting a systematic literature review, 26 metrics specifically employed in video game testing were identified and grouped into five main categories according to the area of relevance. The categories and metrics definitions are discussed in chapter 3.

To highlight both the potential and the limitations of the model thus constructed, part of it has been implemented in one of the most promising framework in the current video game testing landscape: iv4xr [35]. The implementation and the results are discussed in chapters 4, 5.

Chapter 2

Background

2.1 Techniques for software testing

Considering all the advances in software development industry, undetected faults become increasingly more expensive. Software testing is the main solution to mitigate this, by assessing and evaluating the quality of software. It is also a complex and expensive phase of software development, with an estimated 50% of the total cost [4].
Essentially, testing purpose is to ensure that the software under examination, so called System Under Test (SUT), perform as intended and, if present, to identify faults in order to facilitate their correction.

Software testing is usually classified according to the level at which the tests are conducted, and a visual classification of these levels is given by the so called testing pyramid, shown in Figure 2.1.

The testing pyramid is a concept introduced by Mike Cohn in his 2009 book *"Succeeding with Agile: Software Development Using Scrum"*, and essentially outlines the different kinds of tests that developers and testers should include in their collection of automated tests, setting an order of priority of execution and size.
As you move down the pyramid, the tests become more numerous but simpler and easier to execute, while as you move towards the top, the tests become more complex and computationally expensive, but also fewer in number.

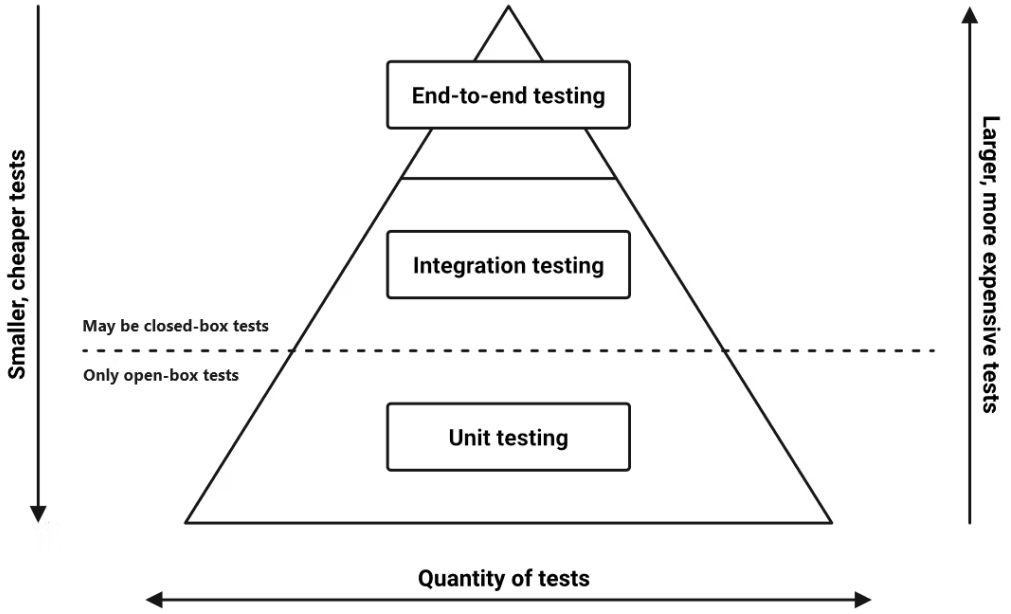

Figure 2.1. Testing pyramid

Image source: circleci.com

Starting from the base of the testing pyramid, there are three types of testing:

- **Unit testing**: unit tests' aim is checking on individual functionalities, focusing on small portions of code, like functions or methods. These unities of code are tested separately from the entire system. This isolation allows developers to catch and correct bugs at an early stage, before the integration of these components into the bigger software system.

  The main objective is to guarantee the correct functionality of each code portion, and by doing so to reduce the time and the costs for future testing. Therefore, Unit tests should be plentiful and generally quick to write and run, so they relies at the base of the testing pyramid.

  To safeguard existing features from being unintentionally disrupted by new code, and to consistently uphold a high level of code quality throughout development, unit tests should be run frequently. Hence, unit testing allows for more frequent releases and also for code refactoring and upgrading.

- **Integration testing**: integration testing focuses on testing multiple functional units as a group.
  It evaluate how these modules interact and exchange information with each other and/or with external modules, so its primary aim is to uncover any faults or issues that surface when these components work together. Integration testing typically takes place after unit testing but before a full system analysis, occupying the middle part of the testing pyramid.

  Since integration tests are more involved, they are run less often than unit tests. Unlike unit tests, which are plentiful and quick to run, integration tests are conducted less frequently due to their complexity and cost.

  They are strategically scheduled for key development milestones, such as after significant features or changes are incorporated into the main code section. So, while unit tests provide a microscopic view with frequent execution, integration tests offer a broader perspective at critical stages, verifying the overall system's functionality and cohesion.

  There are essentially four different approaches to integration testing: big-bang, bottom-up, top-down, and mixed.

  Big-bang approach is the most immediate one and consists of combining all the modules together after individual testing and test the structure thus built. This method is time saving, but it also only applicable for small groups of modules, because with size it also increases the difficulty to localize eventual bugs.

  In the bottom-up testing approach, testing starts with the smallest and low-level components and progresses upwards to test larger, more complex components. The testing sequence continues until all levels of the software hierarchy have been thoroughly tested, in an incremental approach: initially, individual low-level modules are integrated and tested. Once these integrated components pass their tests, they are grouped into the next higher level of modules for further integration testing. This method is particularly effective when most or all modules at each development level are completed. It also enables clear tracking of software development stages and facilitates reporting of testing progress.

On the contrary, top-down approach is useful when low-level modules are not yet integrated: high-level components are tested first and so on until lower-level modules are test, and at the end these are integrated with the complex modules to ensure that the structure functionality is correct.

Finally, the mixed testing approach is a combination of bottom-up and top-down approaches: this technique overcome the limitations of the other two, namely the need to have high-level modules ready (unit-tested) for top-down and the lower-level modules for bottom-up. This mixed approach prove to be very useful in big projects, but it is also significantly more costly and complex to implement than the others.

- **End-to-end or System testing**: system testing aims to evaluate the overall functionality of the complete software system, so to test that the workflow of the system is flawless end-to-end. The starting point are the tested integrated components, which are then evaluated as a whole system in their correspondence to the required functionalities.

  System testing also evaluate the functionalities of the software application user side, checking for example user interfaces and databases interaction: to do so, system test often consists of tools that simulate the end users possible actions.

  This type of testing is on top of the testing pyramid: it is very complex, as it requires a fully designed software system, very time consuming and require high cost and resource consumption to be carried on. Therefore, system tests should be run less frequently and in a fewer number with respect to integration tests, so at milestone points in which is useful to test the workflow of the application to simulate the final one, for example next to a release or before adding significant changes.
  System testing is useful because at this stage will uncover almost every possible bug of failure, hence assuring reliability and quality of the software.

In figure 2.1, there is further separation line between unit testing and integration testing: while unit testing are only open-box tests, from integration testing and above tests may be closed-box.
In fact, in software testing there is also a distinction concerning the access to the source code: these different approaches are called Black box (closed-box),

White box (open-box), and Grey box testing.

In **White box** type of testing, the tester has access to its source code and is so aware of the internal logical structure of the software.
The inner knowledge of code workings is used to design test cases that explore different code paths: testers pick specific inputs to trigger these paths and predict the corresponding outputs.
White box testing is not specific for unit testing as it can be applied also at other levels, but it's most commonly used at the unit level.
This allows testers to examine code flow within a single unit, interactions between units during integration tests, and even communication between subsystems during a system test.

On the contrary, in **Black box** type of testing the tester doesn't have access to the source code and only uses the SUT specifications to generate test cases.
In other words, testers are only aware of the functionalities that the application should meet, but not of how they are implemented. In this case developers design test cases based on specified requirements which can simply verify that for a given input, the system produce a certain output (or functionality). The correctness of the output is often checked using an oracle or a precedent value that for sure is correct.

Grey box testing is a combination of White box and Black box: developers have a partial knowledge of the inner workings of software and algorithms implementation, and they also have knowledge of high-level requirements, using these informations to define test cases executed at Black box level, but combined with the code-targeted type of tests of White box.

Software testing can be further divided into two more ways of testing: Manual or Automated.

2.1.1 Manual testing

Manual testing involves human testers to run and examinate test cases. A test case is an important part of testing activity and consist of two components: a description of the input data and a description of the correct output for that specific input data, a so called oracle. By comparing the outputs produced by the SUT with the oracle, it is determined whether the test case has passed or failed.

Manual testing is an essential step during software development, but on a large scale, it's a very costly method as it requires someone to manually set up an environment and run the tests, which can be prone to human errors such as typos or missing steps in the test script.

2.1.2 Automated testing

Automated testing is the logical answer to reduce cost and complexity, by improving efficiency and reducing human errors. The most common method of automated testing is re-running test scenarios quickly and repeatedly while executing test scripts, but it's not the only one.

In general, automated testing is "the use of special software (separate from the SUT) to control the execution of tests and the comparison of actual outcomes with predicted outcomes" [4]. It is useful to automate necessary but repetitive tests or execute test difficult to perform in a manual way, avoiding the need of a human tester. Automated testing is a growing field and the discussion on which parts of testing process is more convenient to automate is in progress [5]. Automation is also facilitated by advancements in other sectors that yield applicable solutions, such as the latest machine learning techniques [6].

Automated testing approaches

There are many different approaches to automated testing, some of the most employed are model-based testing, capture/replay, random testing, fuzz testing, scripted based testing.

- **Model-based testing** is essentially an application of model-based design method to software testing: an abstract model is created to represent partially the required behaviour of the software application, and

then used by testing tools that create test cases automatically or semi-automatically based on the model. The real outputs of these test case are then confronted with the expected ones. This approach is usually useful for complex systems which may be in multiple states or behaviours. The model can either be static or dynamic, when static models are usually employed for GUI testing (Graphical User Interface) and dynamic ones for API testing. Combining a model-based approach with automation is beneficial since does not require to manually write test cases and allows to create a variety of accurate test cases.

- **Capture and replay** approach is widely employed to test principally GUI and other features of web applications [48]. It consists of recording each interaction of the user with the software application interface and then using these actions to produce test scripts which will be executed several times to automatically reproduce those interactions. Capture and replay test case are easy to produce and don't require particular skills since they are generated over a sequence of recorded interactions, but they can be difficult to maintain because even a small change in the GUI potentially invalidate and require to re-write the test cases.

- **Random testing** is a black-box approach that consists of generating random inputs, feed them to the software system and then check whether the test is successful or failed by comparing the final output with the expected one described in features specifics. Each phase can be automated, from the generation of the random data, to the execution of test cases, to their final confrontation [49]. Random testing is a relatively cheap type of testing that can lead to discover eventual inconsistencies between the requirements and the actual application behaviour.

- **Fuzz testing** can be considered a sub-group of random testing in which the input data are specifically generated to be invalid or incorrect and that can lead to crashes of failures of the system. Fuzz testing process usually starts by generating a set of inputs data, then mutating these inputs again and again and giving them to the system, and finally evaluating if those data produce the desired behaviour, recording what happened [50]. The final goal is always to produce some sort of bug or crash. Hence, fuzz testing is useful to reveal any sort of bugs or critical security problems in software applications.

- **Scripted testing** approach is opposed to scriptless or exploratory testing approach, and consists in writing scripts (so sequential procedures)

for each test case, either manually or using automated tools or approaches such as Capture and replay. The scripts are generated before the execution of test cases, and can later be used to automate their execution, while in scriptless type of testing the tester or any automated tool does not generate scripts, but the testing sequences are created on the run during test execution [51]. Two famous tools employed for testing web application are Selenium [52], for the scripted approach, and Testar [53], which follows a scriptless approach.

2.2 Video game testing

Nowadays, video game testing is a multi-stage process which involves both developers and final players, and can be summarized into five key stages, reported in figure 2.2:

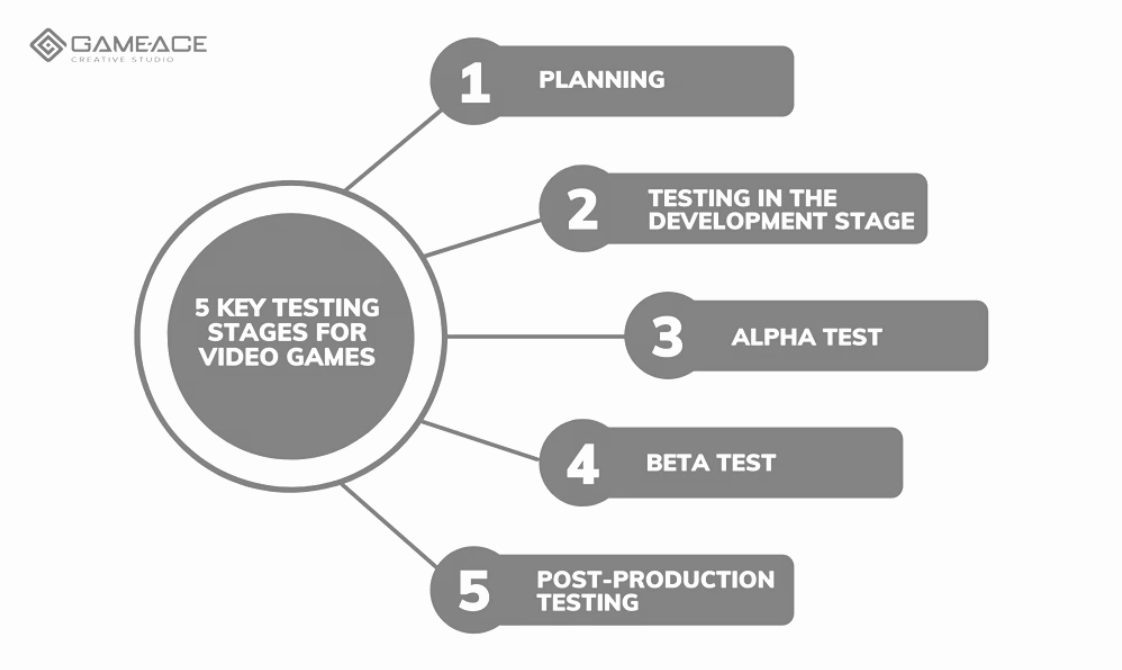

Figure 2.2. Video game testing stages

Image source: game-ace.com

1. Planning: Before testing, a detailed testing strategy based on the specified requirements is created to guide the future process.

2. Early Testing (during development stage): Developers and testers work together to identify and fix bugs early in the development cycle.

3. Alpha Test: after the core functionalities are implemented on a basic level, a dedicated QA (Quality assurance) team conducts in-depth testing of the game's core functionalities.

4. Beta Test: in this phase, the game should be almost ready to be released. Actual gamers provide feedback on the game, helping identify final corrections.

5. Post-Production Testing: Testing continues even after release to address, because end players will encounter issues and provide further feedback.

But while in traditional software development testing is essential, there is currently no exhaustive and systematic report available on the testing practices (or lack of them) employed by game developers, especially regarding automated testing.
This is primarily due to the inherent characteristics of a video game, that make the system under test different from most other software applications, presenting specificities that need to be considered when creating an efficient testing process and making it challenging to adapt some common software testing techniques [7].

Generalizing, we can consider video game testing a subset of software testing, as the main software testing techniques are essentially applicable to video game testing as well. However, going into more detail, it becomes evident that a video game is a complex product made up of a set of different components that interact with each other [24]. These modules and their presence can vary greatly from game to game, but in general, we can have:

- Scripts and code modules: these define game's functionality, so they represent the true core of the game, encompassing everything related to the gameplay and game logic. These scripts define game events from the given inputs, such as character and enemy behaviors, game mechanics, win/lose and scoring mechanisms, and user interfaces.

- AI: an Artificial Intelligence module is often used to manage NPCs (Non-player character) and enemies behavior, creating the illusion of intelligence as if controlled by a human player, and thus generating more engaging challenges.

- Graphics/Multimedia: these modules are in charge of everything related to rendering graphic elements and audio processing in the game, often delegated to a game engine (Unity, Unreal).
- Physics engine: essential in modern 3D video games, the physics engine ensures that all elements of the game world behave according to physics laws, increasing realism. This applies, for example, to throwable or destructible objects, or to the movement of natural elements such as fire, water, wind.
- Inputs: these modules manage all commands from the player and thus has to deal with different input devices such as keyboards, mice, gamepads, joysticks, VR sensors, or more special purpose ones such as steering wheels for driving games.
- Networking: another important aspect is the handling of multiplayer, in which many users interact with each other in the game world, requiring communication protocols and reliability of game servers.

All these elements focus on visual quality and productivity, but games by definition aim to provide an engaging and enjoyable experience rather than purely technical performance.

This is the first big difference: traditional software development focus is to provide an useful service, while video games's primary purpose is entertainment, aiming to deliver an immersive and enjoyable experience, not just productivity. This aspect is particularly hard to test and at the moment relies essentially on human testers, even though some approaches based on a quantitative description of certain emotional states are emerging [33] [34].

An aspect hard to test, especially in an automated way, is also the coupling between the user interface (UI) and the game mechanics, which can be very complex depending on the case [54].

In relation to code, in general, there is little or no code reuse in video game development compared to traditional software development. In traditional software development, there are many similarities between products, while video games strive to deliver unique experiences, and each component of a video game is often written specifically for that particular product [55].

Also, video games are software products that require frequent changes and

releases, for example, in case of mobile games, they are made to be continuously upgraded over time. This makes difficult to automate the testing process, because if code and/or design is constantly changing the process becomes obsolete very fast and requires new implementation [7].

Another testing difficulty is related to the large set of possible game states, which grow along with the game complexity, and ideally require to cover a huge quantity of possible game paths [54].
The set of possible game states increases even more if we think about the randomness that is a main attribute of any game [55]: the non-determinism is a desired features, but the lack of a pattern or predictability makes testing really challenging, as it requires to reproduce some of that randomness to eventually recreate and discover bugs.

For all these reasons, nowadays the primary testing technique used is manual game-play testing, also called play-testing, performed by developers or specific quality teams.

Despite the difficulties, automated tests in game development can be highly beneficial: they are resistant to human errors, are executed in less time, are easier to organize and reproduce and in general they ensure fewer bugs. Hence, there is a necessity for fresh initiatives to explore alternative methods of automating game testing, methods that do not solely depend on human play-testers.

2.2.1 Autonomous agents

Autonomous agents are software tools designed to perform a series of complex tasks in order to achieve a goal or objective function [36]. Training these agents with artificial intelligence and machine learning techniques, they can work undirected and dinamically change their strategies.

Their applications range from personal assistants to autonomous vehicles, to healthcare, finance and so on, but, thanks to their features, they also have great potential in the field of game testing.
They can replicate various player actions such as moving, attacking, solving puzzles, and interacting with objects in the game world, testing how the game responds to different types of behaviors.

Thanks to artificial intelligence and machine learning, they can adapt and learn from the game environment, which allows them to test a broader range of scenarios compared to a human tester. Also, they can run tests continuously and execute repetitive tasks quickly, with a considerable saving of time. Lastly, by mimicking human interactions within the game, they can test and improve user experience.

For all these reasons, autonomous agents have been employed in various works to test multiple aspects of video games.

In [8], autonomous agents built on the belief concept within iv4XR framework [35] are employed to evaluate robustness and consistency between changes in design and development of video games, considering a change of in-game entities locations, changes in the game world and changes in the game logic.

The approach described in [10] makes use of autonomous agents to highlight uncorrect functionalities by operating on two types of goals: synthetic ones based solely on game scenario, and human-like ones trained from human data with reinforcement learning. These two types of agents are employed for bug finding; and finally, a comparison between human-like agents and human playtesters is discussed.

In [15] agents built on a model of memory regarding in-game entities and decision making based on them are employed in PathOS, a tool created as a Unity extension with the aim to facilitate the automation of testing at

development stage.

Dynamic game balancing refers to the adaptability of the game to the evolving level of the player, maintaining a certain degree of balance and believability. In [17], adaptive agents trained with reinforcement learning are employed to address dynamic balancing.

ICARUS [26] is framework focused on testing the core functionality and riddles of adventure games in a bug finding way, and this research utilizes automated players programmed to perform "speedruns" with the purpose to identify and report any crashes, freezes, or situations that prevents progress.

An agent-based approach for automated UX (user experience) testing is proposed in [34]: agents have some problem-solving capacities and a core affect model based on two dimensions, valence (which increases when the agent is able to solve his goal) and arousal (which increase when the agent find new interactable items), leading to model a simple artificial state of emotions.

2.3 Coverage models

The efficiency of test cases is evaluate through coverage metrics, which are quantitative measures of how much of a component or aspect of system under test has been executed during the running of test cases.

Two of the most used coverage metrics in software testing are code coverage and test coverage [29].

Cove coverage metrics essentially measure in several ways how much source code has been executed during test cases. Hence, it is a white-box type of metric, requiring knowledge of code, and performed principally at unit testing level.
So, code coverage is performed by developers, that, by doing this, can assess whether the number of tests executed is adequate to effectively test the software application or whether there is a need to expand the test suite.
Code coverage is also useful to assure a certain standard of code correctness to be maintained after changes or new releases.
Furthermore, measuring the code coverage means discover areas of dead or unused code, that can be eliminated to assure efficiency.

Code coverage can be performed at several levels. The most common criteria includes:

- Function coverage: measures how many of the functions or sub-modules are been called during the execution of test cases.
- Statement coverage: measures the percentage of statement (definitions, declarations, boundary cases...) executed during test cases.
- Line coverage: how many lines of code have been executed during tests.
- Condition coverage: measures how many of the conditional statements (boolean expressions) have been executed.
- Branch coverage: measures the percentage of branches of each control structures (such as if-else and do-while) executed during test cases. For example, for a "if" statement, both the "true" and "false" branch should be covered.
- Modified condition/decision coverage (MC/DC): this is a combination of function and branch coverage, and cover the invocation of every point of entry and exit and all possible outcomes based on decisions.
- Path coverage: measures how many of the possible paths have been executed during tests.
- Loop coverage: measures the coverage of every loop in the program execution.

On the contrary, Test coverage follows a black-box approach as it measures how many tests have been executed, aiming to evaluate how well the software application has been tested.
Hence, it can be performed by quality assurance teams at any level of testing, from unit testing, to integration testing and finally at system/functional level. It is useful to improve the quality of test cases and evaluate if there is a need to expand the test coverage, to identify earlier defects and eliminate eventually redundant tests.
There are several types of test coverage:

- Features coverage: measures how many of the software application features are covered.

- Risk coverage: address the risks related to the application and based on requirement documentation, considering the probability of them to happen in a real scenario.

- Requirement coverage: how many of the requirements described in the application specifications are met during tests.

- Compatibility coverage: verifies the compatibility of the software application across different platform, browser or devices, with different configurations or interaction with other software.

- Boundary value coverage: selecting test cases that fall near or between boundary values is useful to uncover any issues related to boundary constraints and data.

- Number of issues: it's important a system level to report the quantity of eventual bugs, crashes, faults or in general every issue emerged during testing.

The choice on implementing code coverage or test coverage and in what measure has to be considered on the specific applications needs and requirements, but usually a balanced approach between the two is preferable.

For an effective and measurable process of software testing, considering a standardized taxonomy of coverage metric is crucial.

A taxonomy is a scheme of classification in the context of a knowledge field. Taxonomies can have different structures, but their role is to set a uniform and common interpretation of knowledge in a specific area of interest [37]. This is clearly useful in terms of providing comparable results and setting a standard reference.
In the field of software testing, there are many studies focused on taxonomies of requirement coverage metrics, some better organized than others: this disorder is explained by the lack of shared and clear definitions of coverage metrics [38].

But while interest in standardizing metrics in various fields of software testing has increased in recent years, work regarding video game testing is progressing slowly.
Currently, there are no clear definitions of specific coverage metrics for video

game testing, as each study either defines new ones or adopts internal measurements without providing precise definitions. Hence the need to provide a coverage model that serves as a basis for future work.

Chapter 3

A Taxonomy of Coverage Metrics for Game Testing

3.1 Methodology

The objective of the first phase of this work is to identify existing or definable coverage metrics specific for video games and create a taxonomy useful as a standardized base to be expanded and improved in future.
Hence, the methodology followed essentially consists of two phases: a systematic literature review and the taxonomy formulation.

3.1.1 Systematic Literature Review

A Systematic literature review is a rigorous and unbiased approach used to identify, evaluate, and synthesize all existing research relevant to a particular topic. The literature review has been conducted in a systematic way by using a search string on a group of scientific literature repositories, based on the guidelines defined by Kitchenham in [56].
Additionally, also Grey literature has been considered, performing in fact a Multivocal systematic literature review as defined in [57]. Grey literature consists of non-published studies, such as reports, blog posts, theses, working papers and so on, opposed to White literature, which is traditional academic literature.
Including grey literature is useful to have a more a comprehensive understanding of the topic, encompassing both the state of practice and the academic literature, and thus to generally avoid to miss important aspects of the topic.

The multivocal literature review has been conducted following essentially the guidelines described in [58].
Quality assessment techniques and snowballing are not reported because their application did not lead to significant improvements.

After a planning phase to further define the objective of the literature review, the next step is to select the primary repositories providing white and grey literature to be searched.

Selected white literature repositories

- Google scholar
- ACM Digital library
- Science direct
- Springer Link
- IEEE Xplore
- Research Gate

Considered grey literature sources

- Google search
- Game Developer
- Browser Stack

Search string

Next step of the literature review is to formulate search strings, which is usually an incremental search process, where the first searches reveal more effective search strings. In order to efficiently scan the literature, the search string was defined to be as inclusive as possible, containing words like "Game" (or "Video Game"), "Test" or "Testing", "Metric", "Coverage"; and finally adapted to work with the specific syntax of each repository.

Inclusion/Exclusion criteria for Source selection

To filter the results obtained from applying the search string and only consider the relevant ones, the following inclusion/exclusion criteria were defined:

- IC1: the source is directly related to the topic of video game testing.
- IC2: the source explicitly defines or employ metrics or measures to evaluate and track the test execution process.
- IC3: the source is written in a language the author understand (English or Italian).
- IC4: the source can be either a formally published article (white literature) with accessible full text, or a publicly available report or document (grey literature) published between 2012 and 2023.

Exclusion criteria are not defined because they can be formulated essentially as opposite to inclusion criteria.

The initial search results after applying the search string identified 65 sources. After filtering for duplicates and applying the inclusion criteria defined above, they were reduced to a final set of 25 relevant sources, including 22 academic publications and 3 grey literature sources.

3.1.2 Taxonomy definition through Open Coding

After conducting the systematic literature review, the results can be used to develop a structured classification system for metrics used to measure how thoroughly game testing covers different aspects of the game.
To define this taxonomy, Grounded theory approach was followed, based on the guidelines described in [59].

Grounded theory is a general and systematic research methodology based on inductive type of reasoning, in which theories are generalized and defined from a collection of observations. This approach is well-suited for developing taxonomies because emphasizes data-driven insights and reduces confirmation bias.

The sites selected for the research are the sources resulting from the multivocal literature review performed before, and the data collection strategy is based on technical observations.

Open Coding

The process of categorizing and labeling observations is called "coding". The approach followed to perform coding is Straussian technique of Open coding [60]: an analytic process of capturing concepts from observations during data analysis.

Open coding consists of breaking down the data into meaningful units and assigning them short descriptive labels, often consisting of a single word or a brief phrase; this allows to identify key concepts within the data.
The application of Open coding led to formulate the low-level concepts, or codes, of the taxonomy, based on the key aspects identified.

After that, a set of standardized and common definitions has been developed, and used to classify each individual metric identified or derived in the literature sources.
The codes of the taxonomy are considered mutually exclusive, meaning there is a 1-N kind of relationship between codes and metrics (one metric can be assigned to one code only).

To ensure the taxonomy captured the full range of metrics, it was built in an incremental way: if a new metric defined from the literature sources didn't fit into any existing code, a new code was added specifically for that metric, and so on.

Metrics inclusion criteria

To decide whether to include or not a metric in the taxonomy, the following criteria were followed.

First of all, the metric has to be explicitly used for video game testing. Common coverage definitions usually employed in software unit testing, such as branch, condition or line coverage, were not included in the set of selected metrics because they are generic across software domains and can implicitly be applied also to video game testing.

Moreover, the metric has to be measurable and effectively be used to evaluate a defined aspect.
Finally, duplicate metrics were avoided by merging metrics that were practically identical at a high level, differing only in minor details or in the name defined in that specific work.

Axial Coding

Following the completion of the Open Coding phase and the definition of low-level codes, the Axial Coding procedure was applied in two passes to the newly formulated codes.

Axial Coding, as defined in [60], is a process in grounded theory with the aim to relate codes to each other, putting back data from Open coding process by making connections of themes between them.
The application of Axial Coding thus is essential to define a structured model and identify different levels or categories in the taxonomy.

Results

After the data analysis of the sources collection derived from the systematic literature review and the application of Open coding procedure, a set of 26 codes (metrics definitions) has been formulated.

The following application of Axial coding process resulted in the definition of 6 high-level categories of metrics: **Functionality**, which includes **User interface (UI)** and **Gameplay**, **Multimedia**, **Operability and User experience (UX)**, **Performance** and **Reliability**.
Next, a description of these six categories and the aspects of testing they cover is provided.

The taxonomy defined in this way is reported in Tables 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, along with references to existing literature for each metric. Also, a graphical representation of the taxonomy is reported in Figure 3.1.

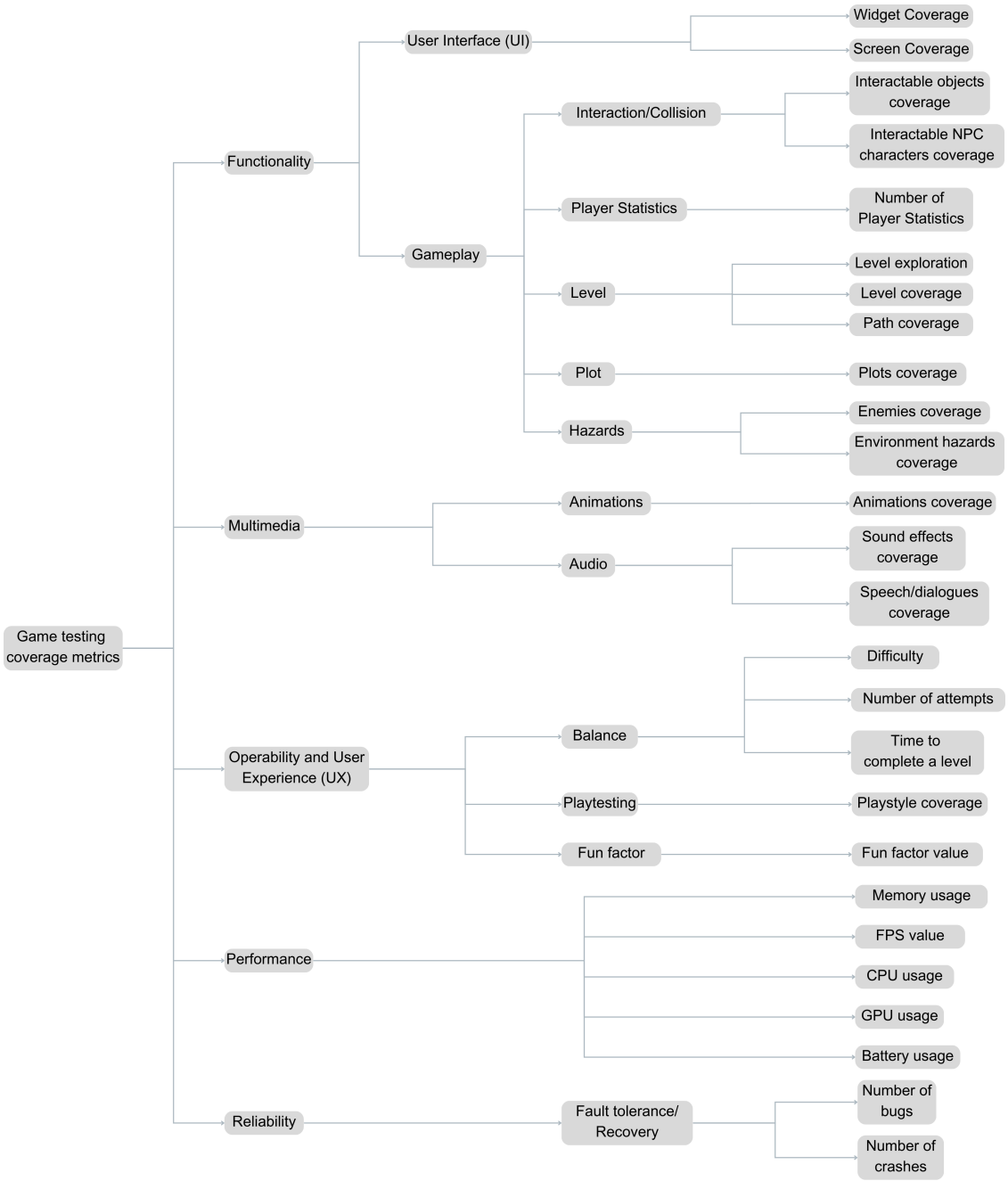

Figure 3.1. Taxonomy of video game testing coverage metrics

3.2 Categories

3.2.1 Functionality

This category refers to functional testing, whose definition is very broad and potentially encompasses many different types of tests [39]. Testing the functionality of a software system means testing whether the application meets the specifications and functional requirements provided by the original design: the behavior of the system must be consistent with the expected features.
Usually, functional testing is black-box since the aim is not to test the code but rather the functional characteristics.

In video game testing there is no single definition of functional testing. In fact, many works arbitrarily consider as functionality testing evaluations of graphical elements and realism, technical performance, game mechanics, and user experience [40] [41].
In this work, the Functionality category considers the elements that make up the backbone of the video game, those without which the video game cannot be defined as such: **User interface (UI)**, so checking that the layout and widgets are meeting the design and functionality requirements, and **Gameplay**, concerning functional aspects of game mechanics and players' interactions with the game.

User interface (UI)

User interface can be considered as anything the end-user will interact with when using some software [42]. In particular, GUI (Graphical User interface) testing includes checking all the graphical components and widgets, such as toolbars, menus, text boxes, buttons, checkboxes; and their interactions.

In game applications that present a GUI, it is necessary to test that all graphical components such as statistics, buttons, menus, and maps are displayed and updated correctly, and that the interactive ones respond to the functional specifications.
Additionally, it is necessary to test the various game screens, as they can vary greatly depending on the application, such as loading, settings, pause, game over, menu, or gameplay screens.

GUI testing techniques can be manual or automated, and range from employing the exact coordinates of components to using artificial intelligence algorithms to test the correctness of visual screens [38]. For example, in [12] a convolutional neural network is trained to detect buttons in UI images and perform an automated exploration to test as many UIs as possible.

Gameplay

This category considers the core functionality of the game and all gameplay mechanics. The aspects to be tested can vary greatly depending on the type of game, its structure, and the presence or absence of certain gameplay mechanics. If there is a playable character, it's necessary to test that movements/actions work correctly. If it's a level-based game, it's essential to check that all levels are functioning and that all intended paths are accessible.

Modern video games offer a vast 3D world, often open world, which contains a variety of interactable sections or objects, such as colletibles (objects which can be taken to achieve achievements or unlock content), goal objects (objective items to win a level or continue the plot) or environmental hazards (game world objects which can damage the player). [15]

This aspect also includes any NPCs (Non-player character), which are characters in a game that are not controlled by the player but with which he can interact for plot purposes: they can be enemies, allies, traders of equipment or items. Testing should include verifying for example that the player does not get stuck in specific parts of the game world and that he can interact correctly with each game entity.

Many modern video games offer a dynamic storyline which changes according to the player's decisions through the game, for example by establishing a system of good/bad actions around which the game world is built [31], so the narrative must be tested as well.

Gameplay is potentially the broadest category to test, since the aspects to be tested increase along with the complexity of the video game, and also the one that can vary the most depending on the game, making it difficult to generalize.

Table 3.1. Functionality - UI metrics

Sub-category	Metric	Definition	Refs
Screen coverage	Screen coverage	Number of covered screens of the game over total number of screens	[12] [11]
Widget coverage	Widget coverage	Number of covered widgets in the game over total number of widgets	[12] [11]

Table 3.2. Functionality - Gameplay metrics

Sub-category	Metric	Definition	Refs
Interaction/ Collision	Interactable objects coverage	Interactable objects triggered by test cases over the total number	[13] [8] [14] [15]
Interaction/ Collision	Interactable NPC characters coverage	Interactable NPC (Non-Playing Characters) triggered by test cases over the total number	[15]
Player statistics	Number of player statistics	Player statistics exercised or modified by the test cases over the total number of player statistics available	[19] [18]
Level	Level exploration	Explored percentage of levels in terms of a specified in-game measure (e.g., meters or square meters)	[8] [9]
Level	Level coverage	Number of levels completed over the total number of available levels in the game	[10]
Level	Path coverage	Number of paths covered in a single level over the total number of possible paths in the level	[10]
Plot	Plots coverage	Number of tested dynamic plots (depending on the players' choices) over the total number of available plots	[31]
Hazards	Enemies coverage	Number of interacted enemies over the total number of enemies present in a level	[15]
Hazards	Environment hazards coverage	Number of the triggered environment hazards over the total number of hazards present in a level	[15]

3.2.2 Multimedia

Another fundamental element of a video game is multimedia, including a variety of animations for characters, objects and environments which should be as fluid as possible. As the complexity of the video game increases, so does the technical quality of animations and graphic detail (textures, 3D models).

In modern 3D games it's important to test the goodness of the physics engine, in order to make the game realistic and immersive. For example, when the player interacts with destructible or throwable items the collision system should respond correctly in terms of realism. [43]
Testing the physics requires very specific knowledge of the element to be tested, which can be the movement of natural elements, of humans, animals, of vehicles, weapons or other objects, and each of them has its own difficulties.

Lastly, another broad multimedia element regard audio testing, which can include background music and ambient sounds, that varies depending on the section/point of the game; sound effects, which could be triggered with specific actions, such as an explosion, the firing of a weapon, a door being opened, footsteps etc.; and dialogues between characters [11].

Table 3.3. Multimedia metrics

Sub-category	Metric	Definition	Refs
Animations	Animations coverage	Number of tested animations over the total number of animations present in a level/scene	[11]
Audio	Sound effects coverage	Number of played sounds over the total number of sounds present in a level/scene	[11]
Audio	Speech/dialogues coverage	Number of played speeches or dialogues over the total number of speeches and dialogues present in a level/scene	[11]

3.2.3 Operability and user experience (UX)

User experience is an important parameter in the testing of any software application. Testing the usability means testing the user-friendliness of the application, in terms of ease of use and understanding, flexibility and smoothness of the navigation flow, clarity of the content; finally, also the attractiveness of the design [44].

These aspects should also be evaluated in game testing, considering, for example, the learnability of gameplay mechanics, how understandable the various interfaces and statistics are, and the clarity of tutorials. This category also includes accessibility testing: options for players with disabilities, such as subtitles, colour configurations and customizable controls.

Two other aspects, however, are exclusive to game testing: balance and the so called fun factor. Many modern video games are designed to increase in difficulty in parallel with the player's progress: it's important to verify the fairness of the game and the balance of the game's parameters also in terms of level difficulty. But since the ultimate goal of a video game is entertainment, it becomes primarily important to understand how enjoyable and immersive it actually is to play, in terms of different emotions such as arousal, tension, boredom.

End users may also have very different styles of gameplay ("personas"): for example, some seek to complete as many objectives as possible, some want to explore the game world more extensively, and others are more interested in combat.

By their nature, testing regarding balance and fun factor is difficult to automate and relies essentially on human playtesting; but recent works about automation make use of autonomous agents [8] [14] [15] [34].

Table 3.4. Operability and UX metrics

Sub-category	Metric	Definition	Refs
Balance	Difficulty	Evaluation of the difficulty of a level or scene, in terms of time to complete the level, number of failed attempts, size of the level, etc.	[16] [17]
Balance	Number of attempts	Minimum, maximum, average number of attempts required by test agents to complete a level	[8]
Balance	Time to complete a level	Minimum, maximum, average amount of time necessary to complete a level	[8]
Playtesting	Playstyle coverage	Number of different playstyles (or ”personas”) applied by automated agents over the total number of playstyles available	[15] [14] [32]
Fun factor	Fun factor value	Measurement of the player’s enjoyment of the tested levels or scenes (in terms of arousal, stress, boredom, etc)	[33] [34]

3.2.4 Performance and Reliability

Software testing techniques for performance and reliability are similar and often overlap; the difference lies in the metrics and aspects tested: testing the performance means testing the efficiency in terms of of framerate, resources utilization (RAM, GPU, CPU...) [25], load time, battery usage, so the responsiveness and fluidity under a particular workload [40]; testing for reliability considers the ability of a software application to function correctly and without failures over a certain period, and aims to identify issues that can potentially damage functionalities [45].

In the context of video games, these testing aspects are very important to ensure the game has the fluidity and stability necessary to enhance the user experience and the success of the game itself.

Some of the testing types include stress testing, which involves testing under intentionally overloaded conditions, such as pressing the same buttons multiple times, or simulating multiple users logged on server; or soak testing, which is a duration test, evaluating the efficiency of the game when is on for long periods of time.

It's also important to address the compatibility by testing that multiple components can perform while sharing the same hardware or software environment, or compatibility with gaming support tools (pads, mouses, keyboards), and the co-existence between programs and game.

Moreover, video games are software product designed to be continuously updated over time, so it's essential to assure maintainability by testing how efficiently the game can be modified, and to perform regression testing, considering bugs related to following updates of the software.

Finally, to assure reliability is important to provide a good management of fault tolerance and recovery, so test the handling of faults consistently with the game flow and the recovery from crash.

Table 3.5. Performance metrics

Sub-category	Metric	Definition	Refs
Performance	Memory usage	Usage of memory during the execution of the test cases	[26]
Performance	FPS value	Maximum, minimum or average FPS (frames per second) rate during the execution of the test cases	[20] [21] [22]
Performance	CPU usage	CPU usage, measured in time needed to render a frame or in power consumption per frame	[23] [24] [25]
Performance	GPU usage	GPU usage measured in time needed to render a frame or in power consumption per frame	[23] [24] [25]
Performance	Battery usage	Consumption of battery during the execution of the test cases	[25]

Table 3.6. Reliability metrics

Sub-category	Metric	Definition	Refs
Fault tolerance/recovery	Number of bugs	Number of bugs discovered during the execution of test cases. A bug is a minor functional, behavioural or graphical issue in the execution of the game, not resulting to an unexpected closure of the software.	[26] [27] [28] [30]
Fault tolerance/recovery	Number of crashes	Number of crashes triggered during the execution of the test cases. A crash is a critical issue in the execution of the game, resulting to an unexpected closure of the software.	[26]

Chapter 4

Iv4XR Framework and Metrics Implementation

4.1 Testing tools

The market and scientific literature offer a variety of frameworks and tools for automated video game testing, ranging from simple to complex, paid or free. Some of these tools are video games specific, while others are generally used for software application testing.

Some GUI testing techniques that are applicable to game testing involve image recognition, a machine learning technique that trains models to identify objects within an image; and/or so called capture/playback or capture/replay techniques, which consists of recording the system screen, then automatically replay the same sequence of commands to get to that screen and comparing the two results [46].
Also for performance and reliability testing non-specific tools for video games can be used, which evaluate efficiency, server and network stability, resource consumption, and response to faults.

Regarding the testing of multimedia aspects, a very useful open-source framework is Rivergame [11]. It is a platform-independent tool that allows to test a variety of aspects using artificial intelligence techniques: visual changes in the scene and the correctness of the UI; the quality of animations, through a set of fixed points for each object that act as a skeleton to calculate its trajectory and verify it matches the desired one; and finally, the sound, by

comparing spectrograms to test sound effects and background music, and using NLP (Natural Language Processing) algorithms for dialogues, which convert the characters' voices into text.
Several other tools propose techniques to test various aspects of gameplay and game environment, mainly using agents trained with reinforcement learning techniques to explore as much as possible in order to find bugs and anomalous behaviors [9] [10].

4.2 Iv4XR Framework

Iv4XR stands for "intelligent verification/validation for extended reality (XR) based systems" [35]. Extended reality refers to highly interactive systems such as augmented reality and virtual reality systems, which are employed in a variety of industries beyond entertainment and video games.

Iv4XR is an agent-based framework that offers a variety of autonomous agents to automate different types of testing in systems like computer games or computer simulators, and also support the integration of other tools. Its functioning is essentially based on an agent-objective solution, illustrated in figure 4.1.

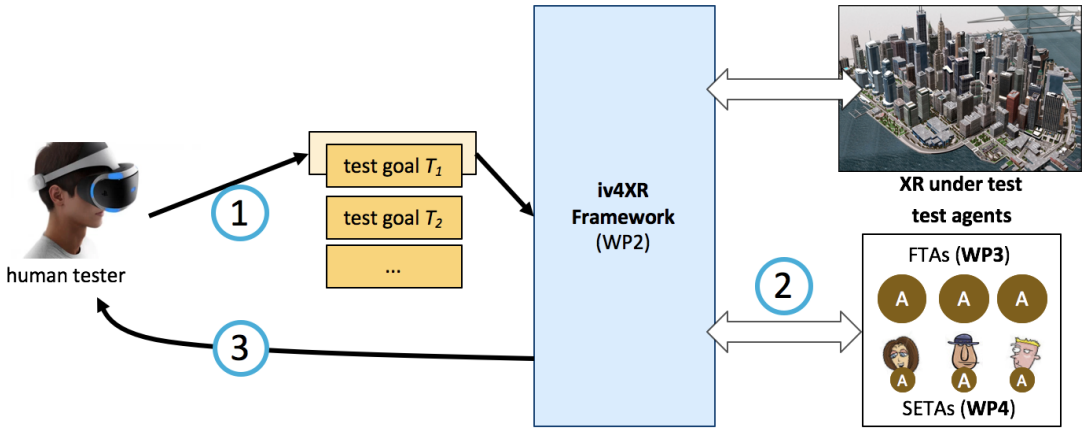

Figure 4.1. Iv4XR approach overview

The testing process begins with the tester/developer defining test goals. These goals are then fed into the Framework (step 1).

The Framework takes these goals and deploys intelligent agents (step 2). These agents, acting like simulated human testers, autonomously interact with the XR system under test (step 2). As they interact, they aim to achieve the test goals and verify the expected behaviors (predicates) defined within those goals.
Throughout the process, the Framework keeps the tester informed of the agents' findings (step 3). This information includes both real-time updates and a comprehensive final report, highlighting any violations of the expected behaviors. Test agents can be of different types depending on the type of testing that needs to be conducted.
In fact, in iv4XR there are three types of agents available: goal-solving test agents, exploratory functional test agents (FTAs) and socio-emotional test agents (SETAs).

Exploratory functional test agents (FTAs)

Exploratory functional test agents (FTAs) aren't based on predefined goals or tactics. Instead, they freely explore the System Under Test (SUT) without scripts or sequences. While exploring, FTAs analyze the system's behavior and identify potentially interesting actions the user might take.

FTAs leverage the iv4XR framework with the integration of TESTAR tool [53], an automatic scriptless system testing tool that works at the GUI level. TESTAR's operating is based on generating test sequences. It achieves this by connecting to the SUT in its initial state, then continuously selecting actions to transition the SUT to new states. Throughout this process, the TESTAR agent checks pre-defined conditions (oracles) to identify any system failures.

Socio-emotional test agents (SETAs)

Socio-emotional test agents (SETAs) are designed for testing different components of UX.
Due to the complex nature of UX, SETAs are designed for modularity, meaning they can be equipped with different modules to test different components of UX, according to what is most important to testers.
Each module empowers them to predict a specific UX component. This modular design also allows for easy expansion with new modules as research uncovers novel UX predictors.

One of the these modules for SETAs agents is an emotional prediction module leveraging the PAD model of emotion, which categorizes human emotions based on three key dimensions: Pleasure, Arousal, and Dominance. The prediction module is trained with machine learning algorithms on data from SUT in order to predict a user's emotional state based on their interactions within the XR environment [34].

A different way to predict emotions implemented for SETAs is a model based approach that uses the OCC (Ortony, Clore, Collins) theory of emotions to formalize relevant emotions [33].

Another module for SETAs agents implements Persona Agents, agents that behave like a specific player or subset of players, addressing the problem of players that can have pretty different playstyles. This is done by grouping user behavior togetherin clusters based on how similar it is, and then adjust the agents to act like the typical user from each group [32].

These modules actually allow for the testing of at least two metrics present in Table 3.4, namely fun factor value and playstyle coverage. However, other modules are currently under development that will allow for the testing of metrics related to difficulty estimation and dynamic plots coverage[31].

4.2.1 Goal-solving test agents

The primary type of agent employed in iv4XR and used in this work is the goal-solving test agent. Iv4XR approach for these agents draws inspiration from the concept of BDI (Belief-Desire-Intent) agents [47]: agents are given a simple goal or a set of goals ("goal structure") and operate through a series of update cycles.

During each cycle, the agent senses its environment, evaluates its current state, and makes decisions aimed at achieving its designated goals. The agent also evaluates whether the current goal has been achieved, should be discarded, or if another goal should be pursued.

In the case of an Iv4XR agent, "belief" concept refers to all the observations the agent has collected up to the current point: the most recent observation

is considered actual, while older ones may no longer be aligned with the current state of the SUT.

Iv4XR agents can make decisions based on this belief: for instance, if in its belief there is a certain object, the agent may assume it still exists in the actual game world and may decide to go to its location.

The update cycle functioning of these agents ensures high reactivity, and makes it especially well-suited for controlling a game.

To facilitate test automation, iv4XR introduces the concept of tactics for simpler tasks, which allow the developer to specify in which way the agents should try to solve their goals. Iv4xr's internal libraries handle the execution of tactics and communication between agents, making agent programming simpler and more abstract for the developer.

For more complex testing scenarios, multiple goals can be structured into hierarchical arrangements ("goal structures").

A very simple example of a tactic is shown in Figure 4.2.

```
var tactic1 = ANYof(
  action().do1(B → B.env().moveUp())    .on(g1),
  action().do1(B → B.env().moveDown()) .on(g2),
  action().do1(B → B.env().moveLeft())   .on(g3),
  action().do1(B → B.env().moveRight())  .on(g4))
```

Figure 4.2. Simple tactic to move the agent

This is a tactic to move the agent by choosing the direction based on guards (g1, g2..), that must be set to true to enable that direction. The combinator ANYof simply chooses randomly one of the directions enabled.

More structured tactics can be built by combining simpler ones: an example is reported in Figure 4.3.

```
Tactic navigateToTactic(id) {
  return
  FIRSTof(survivalTactic(),
          travelTo(id) .lift(),
          explore()    .lift(),
          Abort)                ; }
```

Figure 4.3. More complex tactic

Here, the combinator FIRSTof will execute the first enabled tactic. This means that, for example, if the game object with said id is not anymore present in the agent's belief, the travelTo tactic will not be enabled; or if the agent's health state goes too low, the survivalTactic will be enabled and executed; and so on.

4.2.2 Architecture and Environment interface

The overall iv4XR architecture is summarized in Figure 4.4.

The core library is called aplib, which stands for Agent Programming Library, and it contains: the implementation of tactics, goals, and the algorithms used to solve them; tools for collecting traces and data during test execution; and the library that handles the representation and management of the game environment. Aplib then interfaces with the different types of agents through specific goal libraries and with the SUT through a game environment interface.

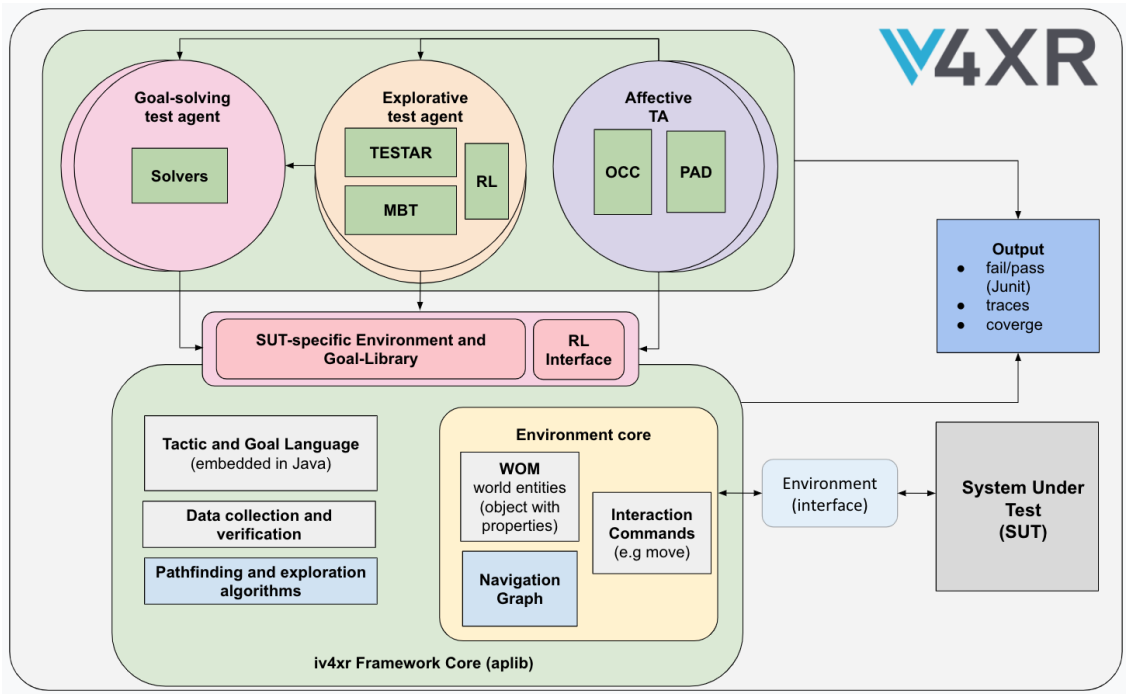

Figure 4.4. Iv4XR architecture

As shown in Figure 4.4, Iv4XR cannot be used without the implementation of an interface to connect the internal libraries and representations with the specific features of the SUT. This is due to the fact that video games are very diverse in the representation of their states, making it impossibile to find a standardized way to interface with them.

This interface, called Environment, allows test agents to be connected to the SUT and hence perform their test-goals; and it should be implemented by the developers at least to provide a method to observe the current state of the game and other methods to handle the atomic actions that the agent can perform within the game, for example to interact with object and characters, or moving to a certain point.

The level of complexity that the interface can reach is at the developers' discretion, which can decide how many actions to implement and how much of the game world can be observed.

In Figure 4.5 a more detailed description of what is needed to be implemented for the environment interface is shown.

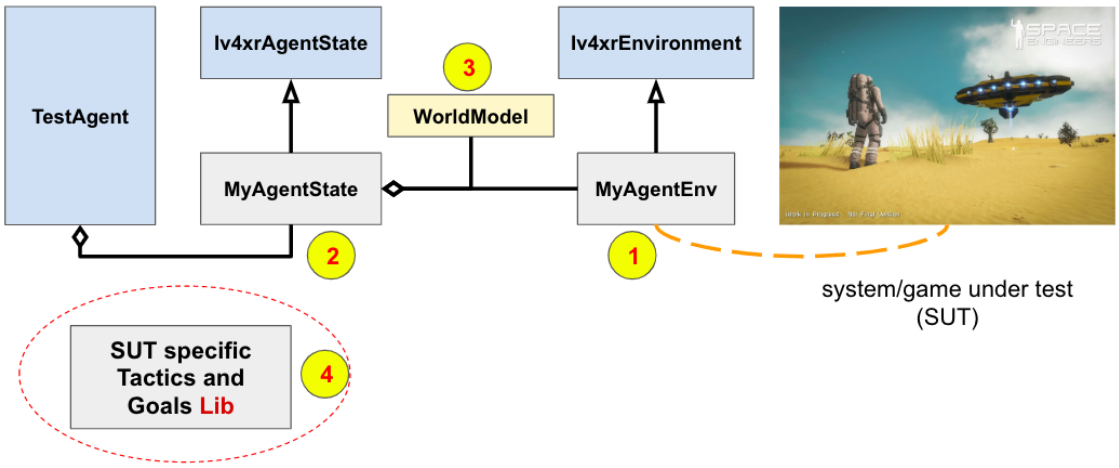

Figure 4.5. Iv4XR architecture related to environment implementation

Several key components are needed to get everything working:

1. implementation of Iv4xrEnvironment class: this class acts as a bridge between the agent and the system under test (SUT). For example, allows to send commands to the SUT and retrieve information about its current state.

2. implementation of Iv4xrAgentState: this component holds all the information the agent knows about the SUT. It includes a general representation of the game state called WorldModel.

3. implementation of WorldModel and WorldEntity: these are generic building blocks used to represent the game state (WorldModel) and individual objects within the SUT (WorldEntity).

4. building a translator: this custom-built component takes actual objects and the game state from the SUT and translates them into the generic WorldEntity and WorldModel formats. It essentially turns the game's unique language into something the agent can understand.

5. customized and game specific tactics and goals library, supported by tools like pathfinding and exploration algorithms provided by iv4XR.

A more accurate description of WorldModel and WorldEntity is reported in figure 4.6.

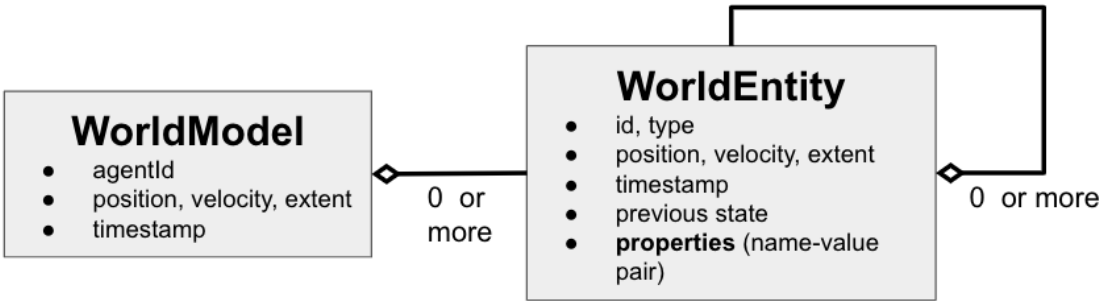

Figure 4.6. WorldModel and WorldEntities structure

A WorldModel, or WOM, provide a snapshot of a specific agent's perspective. It includes the agent's ID and location, along with details about the surrounding objects (WorldEntities). These WorldEntities, each representing a game object, hold information like their ID, location, and custom properties. Both WOMs and WorldEntities are time-stamped, ensuring up-to-date information about the agent and its environment.

Instead, in figure 4.7 is reported the general structure of Iv4xrAgentState, which represent the state of the agent, hence his belief. An instance of Iv4xrAgentState holds a WOM, which is the last game state observed,a Navigatable which represent a navigation graph, and an instance of Iv4xrEnvironment, essential to give the agent control over the game. Agent states also have an updateState() method, which will be periodically called to refresh the state informations.

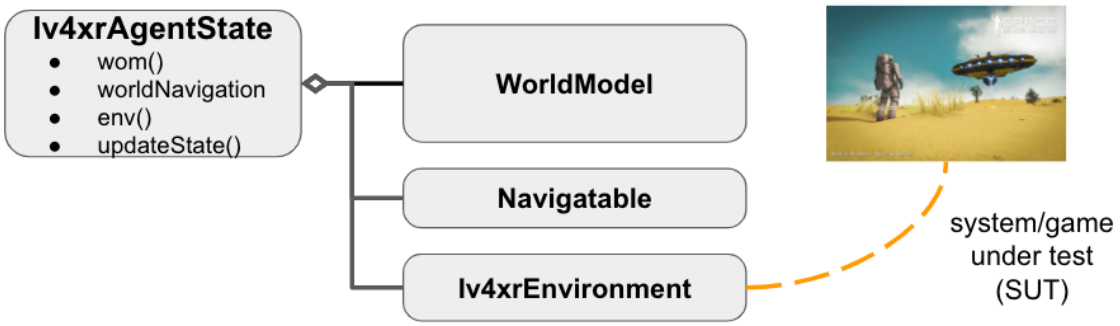

Figure 4.7. Agent state structure

4.3 Implementation of game metrics in iv4XR

The second objective of this thesis work is to implement a subset of the coverage metrics previously defined in the taxonomy into the iv4XR framework. The goal is to verify the effectiveness of the metrics formulated before in a process of game testing, and to investigate the feasibility of making the coverage model applicable to the testing of any video game.

4.3.1 LabRecruits

To perform the testing, the targeted game is LabRecruits [61], iv4XR's designated testing game.
LabRecruits is a 3D computer game developed in Unity intended for AI testing. After launching, it's possible to load a level, which can be defined by testers through an ASCII-based CSV file.

A level is essentially a maze, which can be multi-level, and can contain decorative objects, such as plants, chairs, lights, tables, and a set of interactable objects, such as buttons and flags. Doors act as blocking-access gates, and buttons act as switches, controlling the opening and closing of the doors. Buttons and door can be in a many-to-many kind of relationship, to make the game resolution more complicated.
Flags can be of two different types: healing flags, that restore a part of health points when toggled, and finish flag, that mark the level as completed when toggled.

In a level there may be two types of hazards: an environmental one, in the form of a fire, which can be walked through but it will decrease player's health score; and enemies in the form of infected humans, that will chase the player if he gets too close to them and will damage his health if they manage to touch him.
Enemies cannot be fought, but after touching the player, they will freeze for a brief period, providing an opportunity for the player to escape.
There can also be present one or more NPC characters, that cannot be controlled by the player.

In figure 4.8 is reported an example of a LabRecruits level, with buttons, doors and fire hazards.

Figure 4.8. Screenshot of a LabRecruits level

Player start with an health score of 100 points that will decrease if he comes into contact with fire hazards or enemies and will increase if he touches a healing flags.
Some actions can be rewarded and increase the statistic named "score": coming into contact with a button for the first time and toggle it, touching heal flags of finish flags.
The goal is not the same for each level, and defining it is not mandatory, for example a level can be created with the intention to explore it, thus not containing a finish flag.

4.3.2 Coverage testing workflow

For the testing process, the implementation focus is on the gameplay category, particularly on coverage metrics for interactive objects (buttons, flags, doors) and entities with which the player interacts (fires, enemies, NPCs). The testing process has been performed using iv4XR's goal-solving autonomous agents, and test are implemented as JUnit test classes.

To be able to write tests, two classes are essential: class LabRecruitsTestServer, in order to establish a connection between the framework and the game, and class LabRecruitsEnvironment, which provides basic methods to interact with the instance of LabRecruits game currently connected.

To launch the communication channel, an instance of LabRecruitsTestServer is created and method waitForGameToLoad() is invoked. This will launch an instance of Lab Recruits and create a dedicated server. Class diagram of LabRecruitsTestServer is reported in figure 4.9.

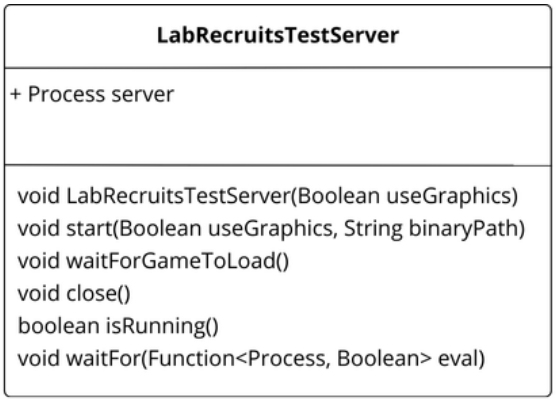

Figure 4.9. Class diagram of LabRecruitsTestServer

Once this server is running, a LabRecruitsEnvironment object can be created using an instance of LabRecruitsConfig, which represent the configuration of the selected level, and will automatically connect to the server, giving control over the launched game instance.

Next essential step is create a test agent with class LabRecruitsTestAgent, which is a subclass of iv4XR TestAgent. The state structure is provided by class BeliefState, which act as an internal model for each test agent, representing its understanding of the game world, and considers the in-game entities the agent encounters and the accessible areas it can navigate. Class diagrams are provided in figure 4.10.

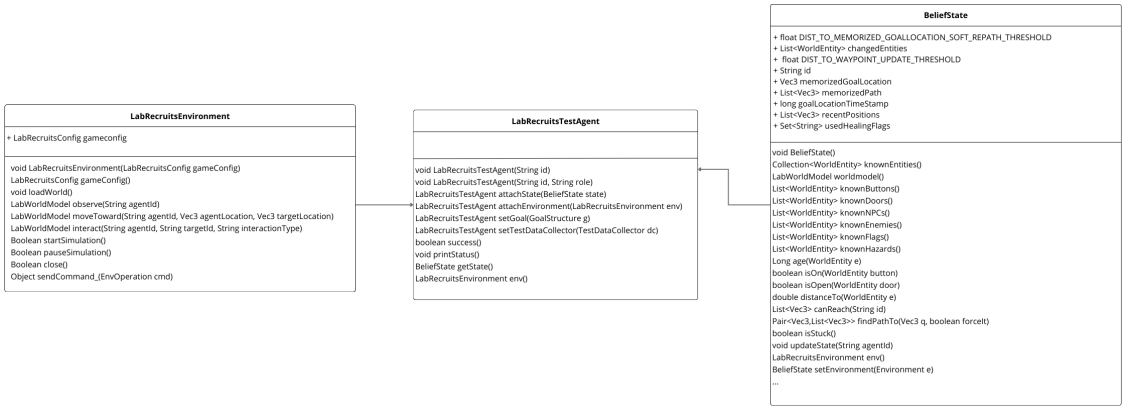

Figure 4.10. Class diagrams of LabRecruitsTestAgent, LabRecruitsEnvironment, BeliefState

Test execution

The basic workflow of executed tests, handled in class CoverageTest, is the following:
First, server is started, and instances of LabRecruitsEnvironment and LabRecruitsTestAgent are created. The selected level is load using method loadWorld().

A log structure, namely levelInfo, and containing relevant informations on the selected level's composition is created via class LRFloorMap, which essentially is written to handle the logical extracted map of a level, and kept for the final comparison.
LRFloorMap divide a level configuration parsed by a CSV file into logical cells (tiles) which can be of different types considering the game entities, identify the type and position of entities and the relationship buttons-door.
Class diagram of LRFloorMap is reported in figure 4.11.

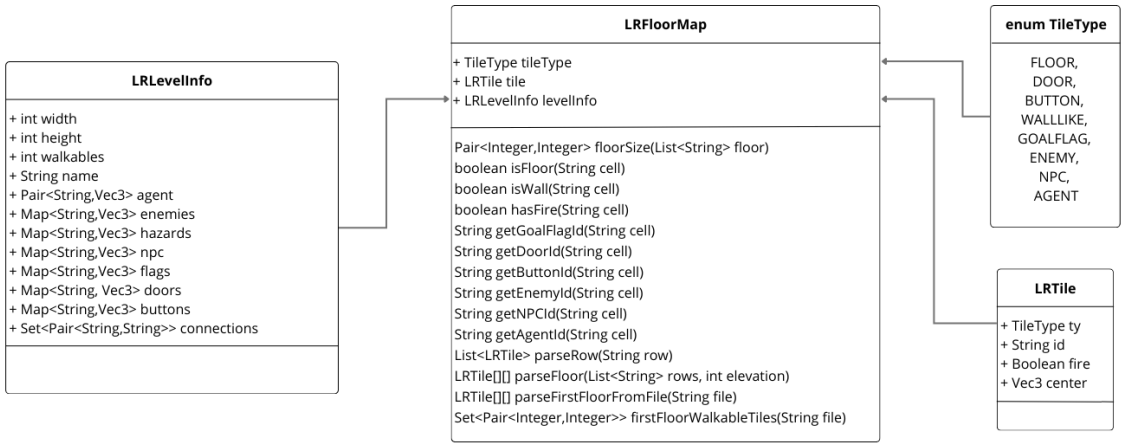

Figure 4.11. Class diagrams of LRFloorMap

A state of type BeliefState is created and attached to the agent using attachState method. A testing task of type GoalStructure is defined and attached to the agent using the method setGoal, along with the environment, using method attachEnvironment, and eventual data collectors, using method setTestDataCollector; all these method are from class LabRecruitsTestAgent.

Then, the simulation is started using the related method startSimulation() of LabRecruitsEnvironment, and until the testing goal is completed or failed, the state of the agent keeps updating through the method updateState() of class BeliefState.

Once the agent has completed its task, a comparison is made between the entities present in the log and those contained in the agent's state (i.e., those with which it interacted in the game), and the corresponding coverage metrics are calculated in form of percentages, using the method computeCoverage of class CoverageTest.

Finally, the simulation and the server connection are closed.
The final test results are graphically shown using showTestResults method and include percentages of total interactable items coverage, and buttons, doors, flags, NPCs, enemies and hazards coverage.

The testing task is designed to encourage interaction with as many buttons

and doors as possible, leading to significant exploration of the game world.
If the agent doesn't see anything, it explores for a short period using method explore, which essentially makes the agent to wander for a period of time specified by the budget variable.
The strategy is to interact with doors in order of distance, provided by method sortDoorsByDistance, from the agent: when the agent sees a door, a dedicated algorithm is invoked to determine which button will open that door based on the current state.
If the button exists and is reachable, it is pressed and the door is opened, and so on to the next door.
Also, if the agent sees a flag along the way, it will interact with it.

In figure 4.12 the overall diagram of test execution class is reported.

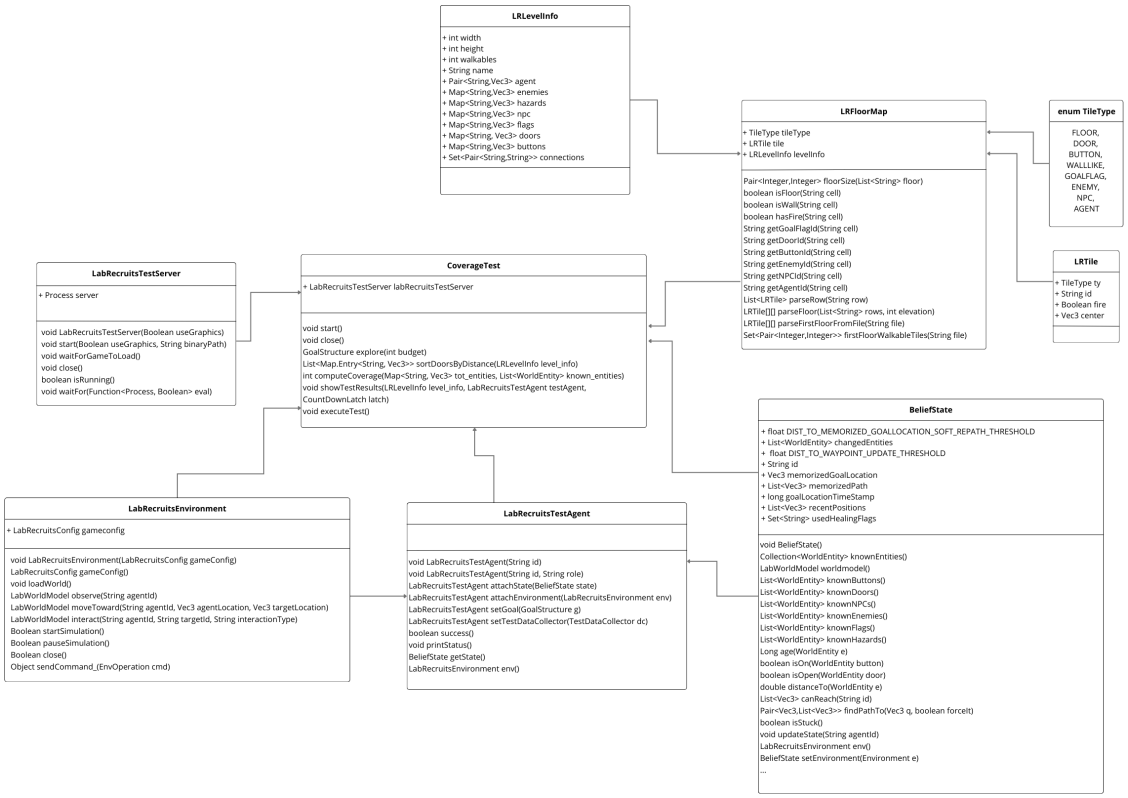

Figure 4.12. Class diagrams of CoverageTest

Chapter 5

Coverage test session results

The coverage metrics for objects and interactable entities coverage were evaluated in two different subcases: considering the newly described testing methodology applied to different levels of LabRecruits, and considering demo tests provided by the developers instead.

Each test has been executed two times in order to verify if the results are consistent or can change between different runs.

5.1 Session of custom-written tests

The testing methodology applied in this phase is the one described previously, where the testing goal assigned to the agent is formulated to be as general as possible (not specific to a particular level) and to make it open as many doors as possible based on its belief.

Five different levels were considered, on which the test was applied, and the coverage metrics for interactable entities were evaluated.

Level 1

Level considered is called buttons_doors_1. This is a relatively small level, containing 3 doors, 4 buttons, and a single fire hazard. The test results are shown in Figure 5.1.

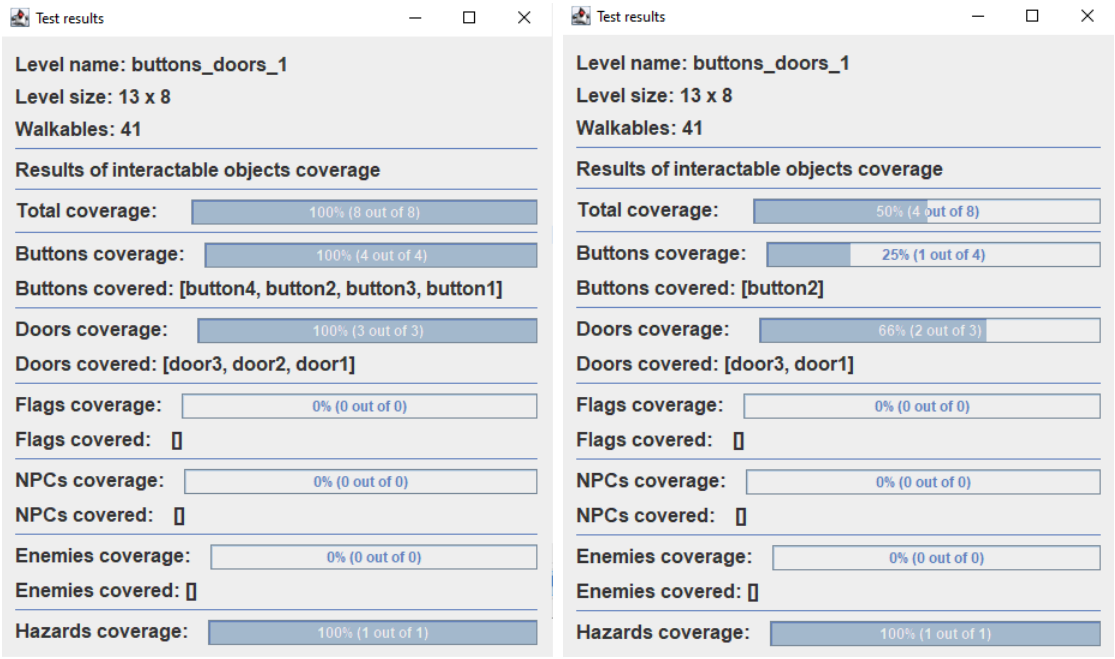

Figure 5.1. Test results on level 1

As can be seen from the results, during the first run the test was indeed able to achieve 100% coverage metrics, in the sense that the agent came into contact with all the entities present in the level.

During the second run, on the other hand, there was no total coverage: this was caused by the fact that depending on how the agent decides to advance, sometimes it fails to "see" the button hidden by the wall as shown in the figure 5.2, therefore it does not interact with it and cannot open all the doors.

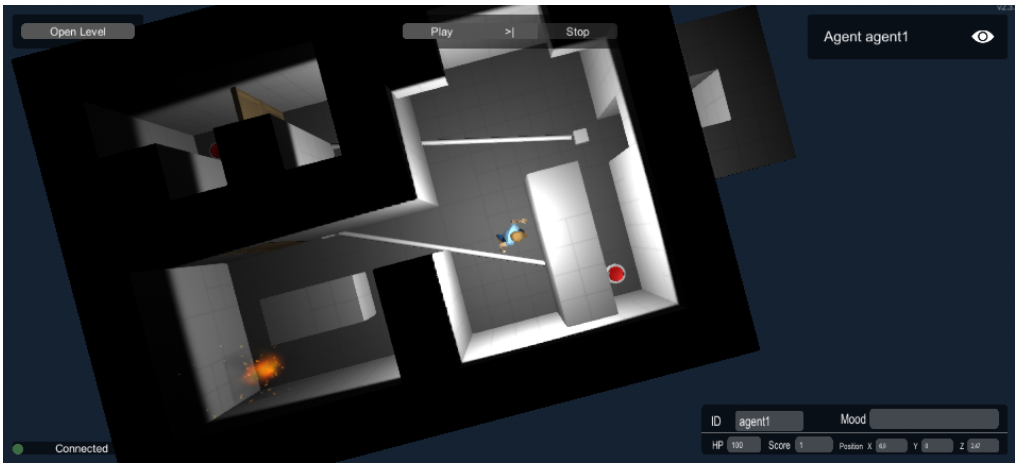

Figure 5.2. Test level 1 screenshot

Level 2

Level considered is named HZRDDirect, and it's a medium size level containing 3 doors, 3 buttons, and plenty of fire hazards, as shown in figure 5.4. The results of the two runs for this level were the same, and are shown in figure 5.3.

In this level the formulated test has reached 100% of coverage of interactable entities without any problem, while, as will be reported in the section dedicated to demo tests, in that case total coverage will not be achieved.

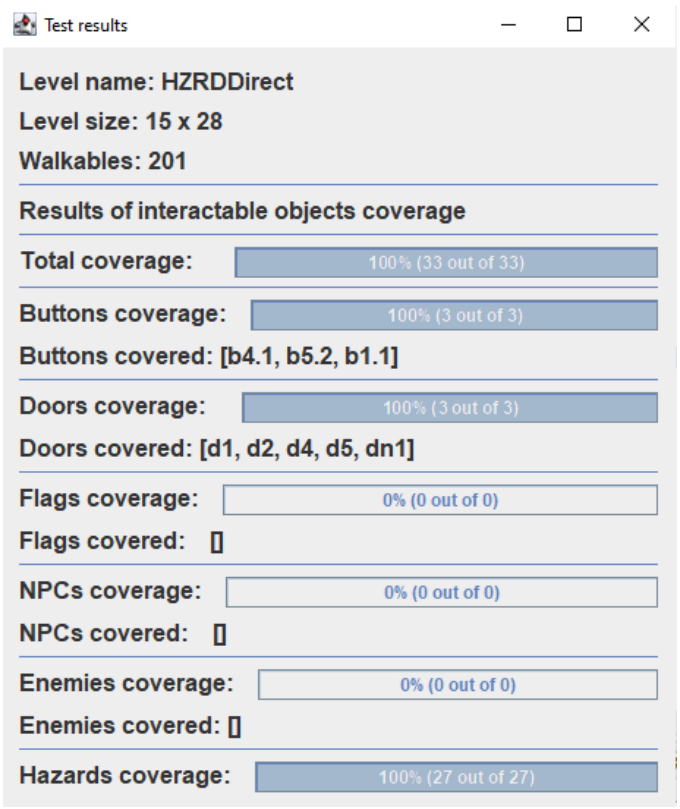

Figure 5.3. Test results on level 2

Figure 5.4. Test level 2 screenshot

Level 3

The level considered here is called fbk_mediumlevel. It's a level slightly bigger than previous one, and contains 5 buttons, 3 doors and 1 flag. It is structured to have long empty corridors between doors.
Results of two runs are reported in figure 5.5.

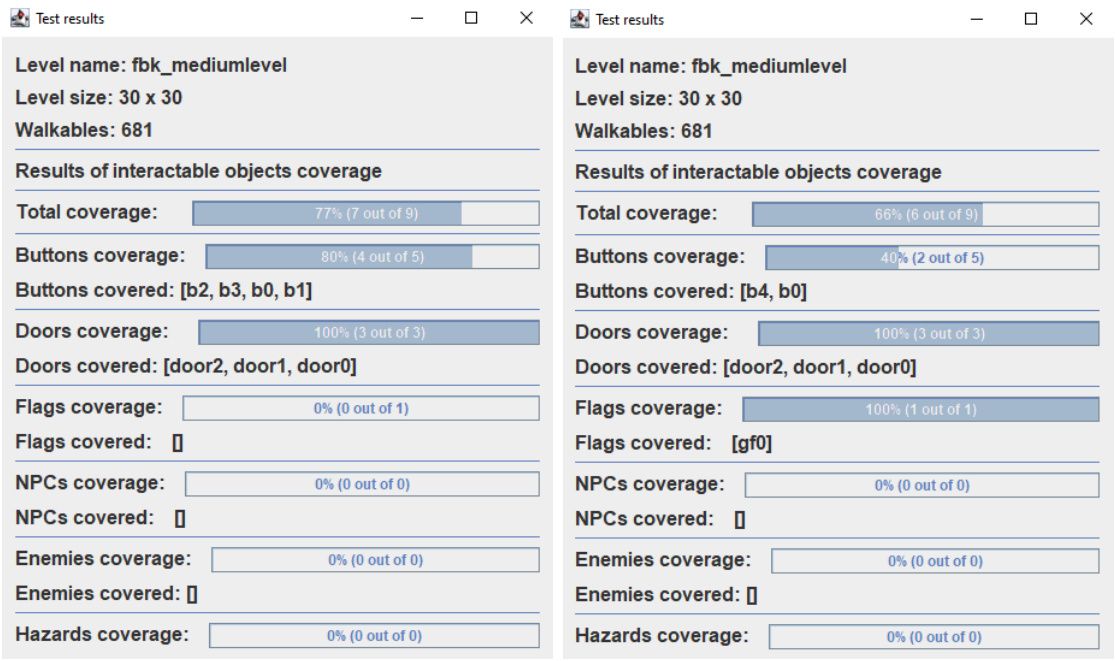

Figure 5.5. Test results on level 3

As can be seen from the images, the test had two different final outcomes but in both cases failed to achieve total coverage of the interactable entities. The first run achieved to cover almost all buttons (4 out of 5), but didn't cover the flag; while the second run covered less buttons (2 out of 5) but was successful to cover the flag.
This is due to the fact that there are many corners and angles in this level, and the agent often has difficulties to overcome them and gets stuck, unable to continue (despite the internal implementation of an unstuck technique), an example of this in figure 5.6.

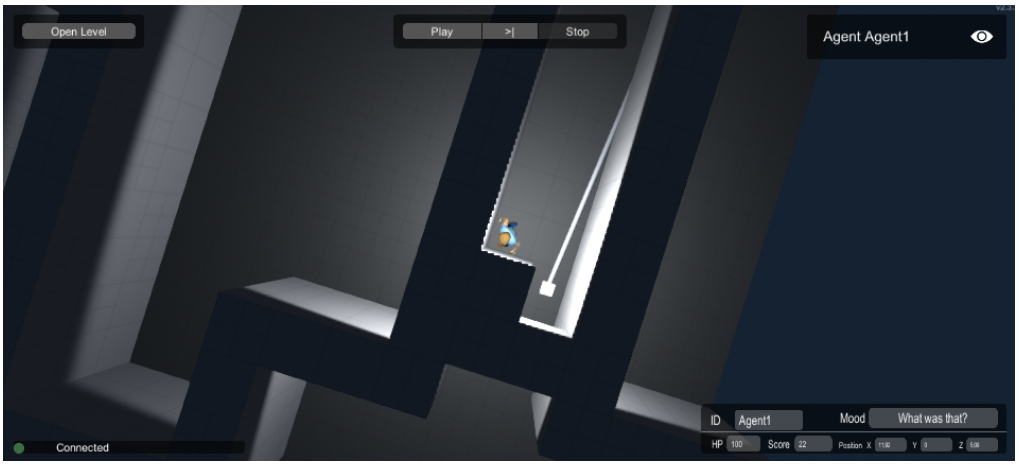

Figure 5.6. Test level 3 screenshot

Level 4

The level considered is named R8_fire3, and it's a big level that contains 11 buttons, 11 doors, 2 flags, 5 enemies in the form of infected humans (represented graphically as red characters) and 53 fire hazards.
Test results of the two runs on this level are reported in figure 5.7.

In this case as well, the test yielded two different outcomes: in the first run, the coverage was lower, at 37%, covering fewer doors, buttons, and fire hazards.
In the second run, there was a higher total coverage of 59%, covering more doors, buttons, and fire hazards.
In both cases, only one of the two flags was covered, and 2 out of 5 enemies were encountered.
This was due to the fact that at a specific point in the level, shown in figure 5.8, the agent gets stuck between a wall and one of the enemies that attacks it, unable to free itself and eventually being killed by the enemy.

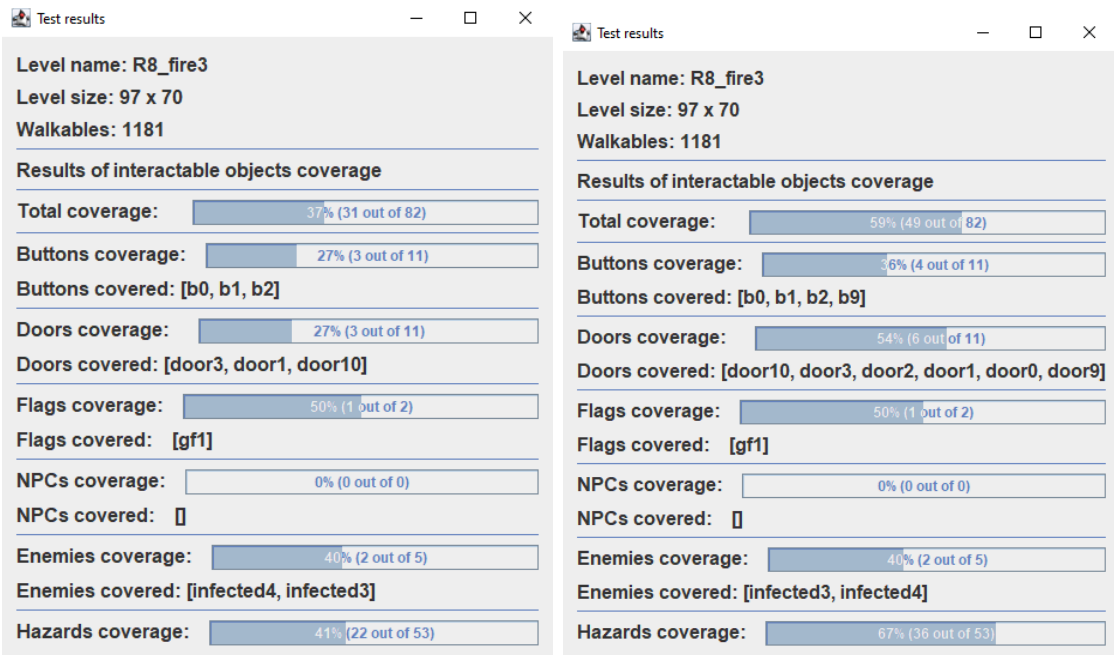

Figure 5.7. Test results on level 4

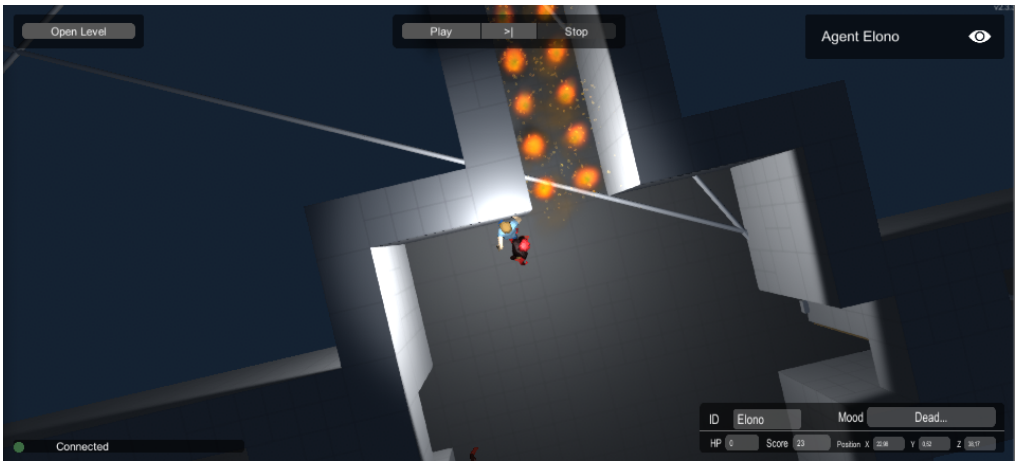

Figure 5.8. Test level 4 screenshot

Level 5

The level tested here is called lab1. This is a medium-sized level but full of walls, decorative objects, and interactable entities, without many empty or open spaces and with intricate paths, as can be seen in figures 5.11 and 5.12. In particular, it contains 20 buttons 13 doors, 1 flag and 15 fire hazards.
Test results are reported in figures 5.9 and 5.10.

The test results of the two runs on this level differ very little, considering that the first run covers only one button and one fire hazard more than the second.
However, neither run achieves total coverage, reaching a maximum of 32% in the first run.
This is probably due to the way the level is structured, which is not particularly compatible with the applied testing methodology: the fact that there are many doors and many buttons confuses the agent, which could for example notice a door that can be opened by more than one button, but only have knowledge of one of these buttons which may not be reachable from its current position (e.g., because behind a wall).

On levels like this that present multiple and intricate button-door relationships and complex opening patterns, the testing methodology used does not perform very well, due to the potential to encounter obstacles that hinder the ability of the agent to reach all interactive elements.

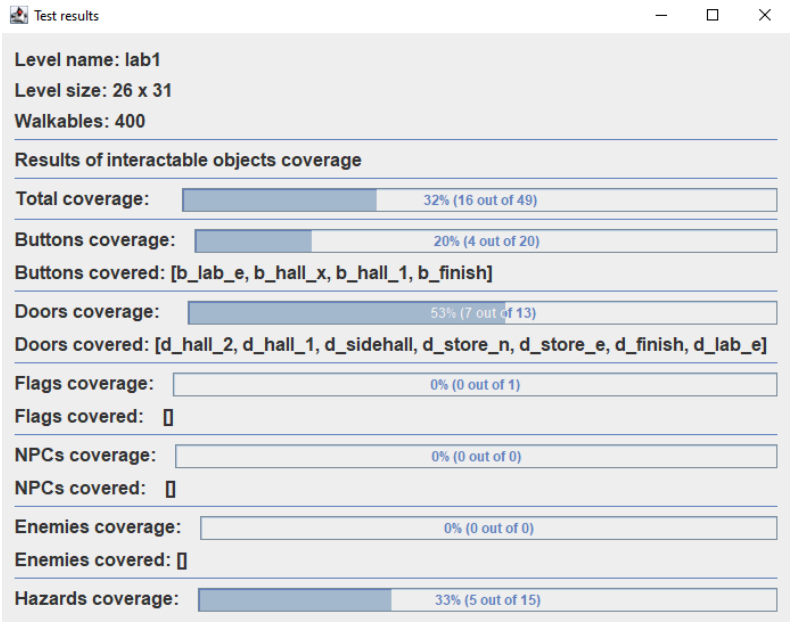

Figure 5.9. Test results on level 5 - first run

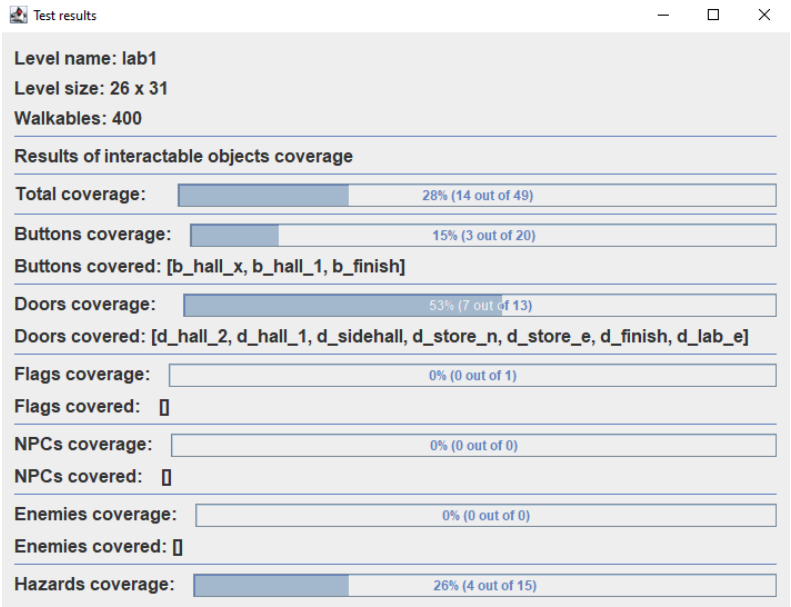

Figure 5.10. Test results on level 5 - second run

Figure 5.11. Test level 5 screenshot 1

Figure 5.12. Test level 5 screenshot 2

5.2 Session of provided demo-tests

In this section, the evaluation of the coverage metrics for interactable entities was performed on 4 demo tests provided by LabRecruits' developers.

These tests are specifically designed for each level and involve formulating the testing goal as a sequence of specific actions (such as to interact with an entity with given id, or to move forward to an indicated positon) to be executed sequentially to test a particular path or objective.

Demo test 1

First demo test considered is called RoomReachability test, and is formulated on level buttons_doors_1 considered also in 5.1.

The primary objective of this test is to confirm that the player can access the eastern closet. To achieve this, the player must be able to open the door guarding the closet, which likely requires navigating a sequence of switches and unlocking additional doors beforehand.
This is done by formulating the testing goal with sequential instructions to interact with the right button in order to open the door on the right path.

The result of coverage metrics evaluation on this test are reported in figure 1, and as can be seen achieved a total coverage.

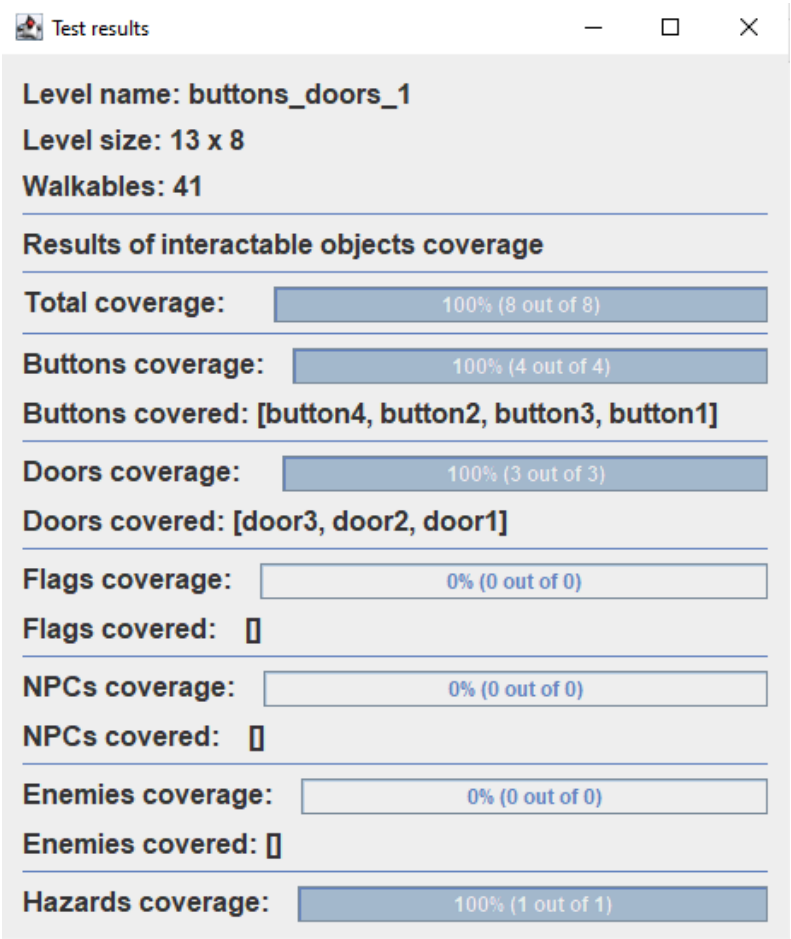

Figure 5.13. Test results on demo test 1

Demo test 2

The demo test considered is called FireHazardLevel_1 test, and is written on HZRDDirect level considered in 5.1, which contains plenty of fire hazards.

This level features a room designated as the target area, marked by a fire extinguisher.
The objective of the test is to ensure the agent can reach this target room while maintaining a sufficient health level (defined as retaining at least half of its initial health).
To achieve this, the agent follows a set of predetermined waypoints designed to minimize risk.
However, it must still navigate independently between these waypoints.
Coverage results on this test are reported in figure 5.14.

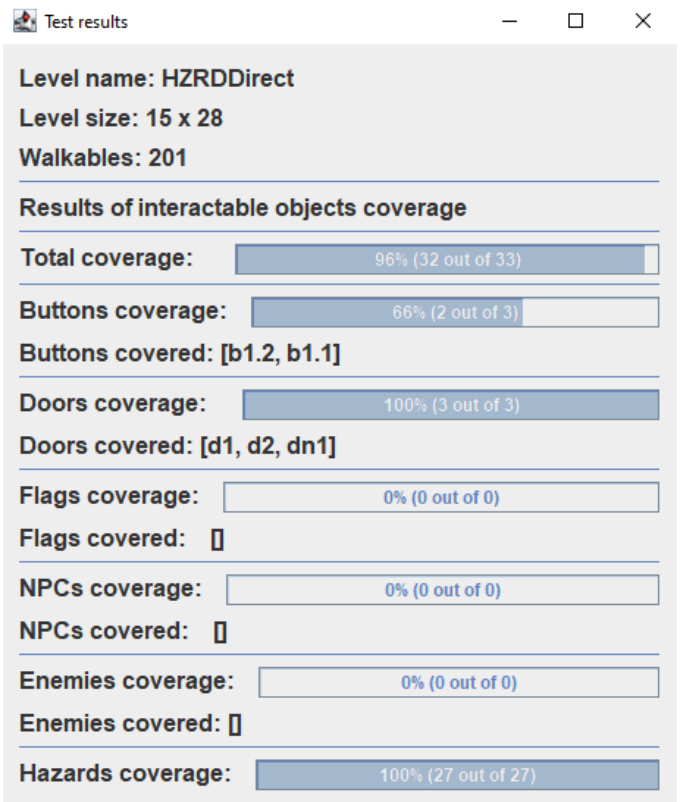

Figure 5.14. Test results on demo test 2

As shown, it was achieved a total coverage of 96%, covering all the interactable entities except for one button.

Comparing this result with the 100% coverage reached during the execution of the custom made test reported in 5.3, it can be observed that the more general testing methodology likely leads the agent to explore the level more thoroughly compared to a test generated based on a specific path that does not cover all entities.

Demo test 3

Demo test 3 is formulated on R8_fire3 also considered in 5.1, which had fire hazards and enemies.

This test scenario simulates a specific sequence of actions to try to reach the flag labelled as Finish point.
The objective is to verify that the agent can successfully reach the Finish point while maintaining its health within a specific range (between 20 and 50 health points) after passing a specified door. Additionally, the agent must have accumulated at least 34 points at this stage.
Upon reaching the Finish point, the agent's health should be fully restored to 100, and its total points should be at least 524.
This approach requires defining the specific actions needed to open each door, that involves specifying interactions with certain buttons as prerequisites for unlocking specific doors.

Coverage test results are reported in figure 5.15.

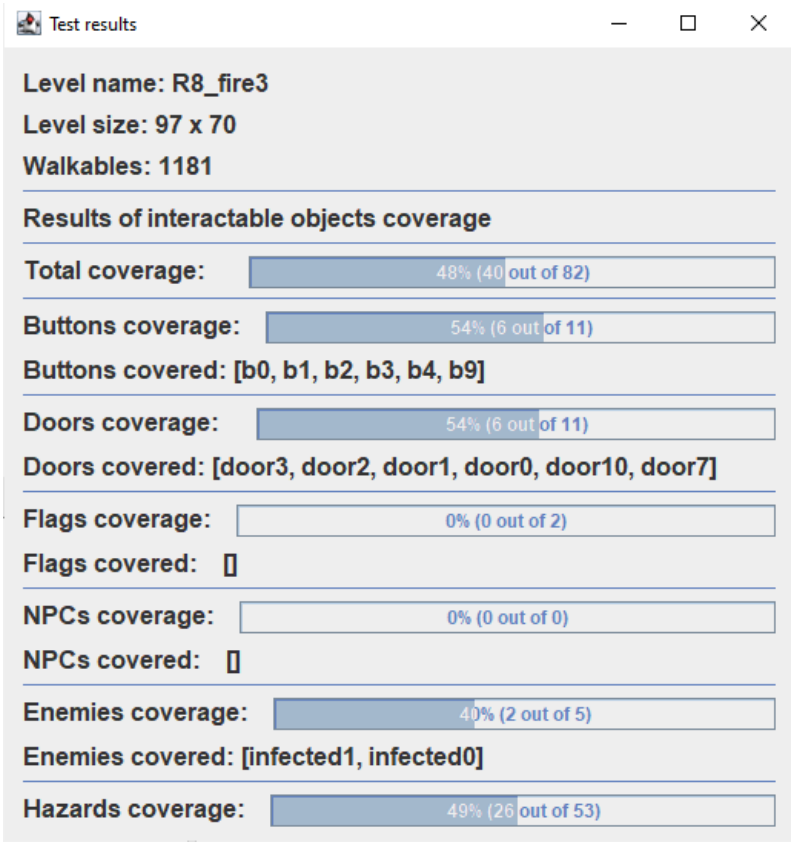

Figure 5.15. Test results on demo test 3

The agent failed to fullfill this testing goal as it died before the reaching of the finish flag, because of enemies and fire hazards.

Comparing this result with the ones obtained during the runs executed before in 5.1, it can be seen that the 48% of total coverage achieved here is mid way to the ones obtained during the previous run, which were 37% and 59% respectively.

However, the best result has been obtained before with 59% of total coverage, but it's worth mentioning that it also covered one flag that was not covered here instead.

Demo test 4

The last demo test in examination is called Lab1Test and is written for level called lab1, considered also in 5.1.

As for the precedent demo test, also in this case the objective is to verify that the level can be completed by reaching the finish flag placed in the room with a fire extinguisher.
Also for this testing goal the approach is to specify to the agent which doors to open in order to reach the final flag, through a sequence of sequential actions.

Results of the coverage on this test are reported in figure 5.16.

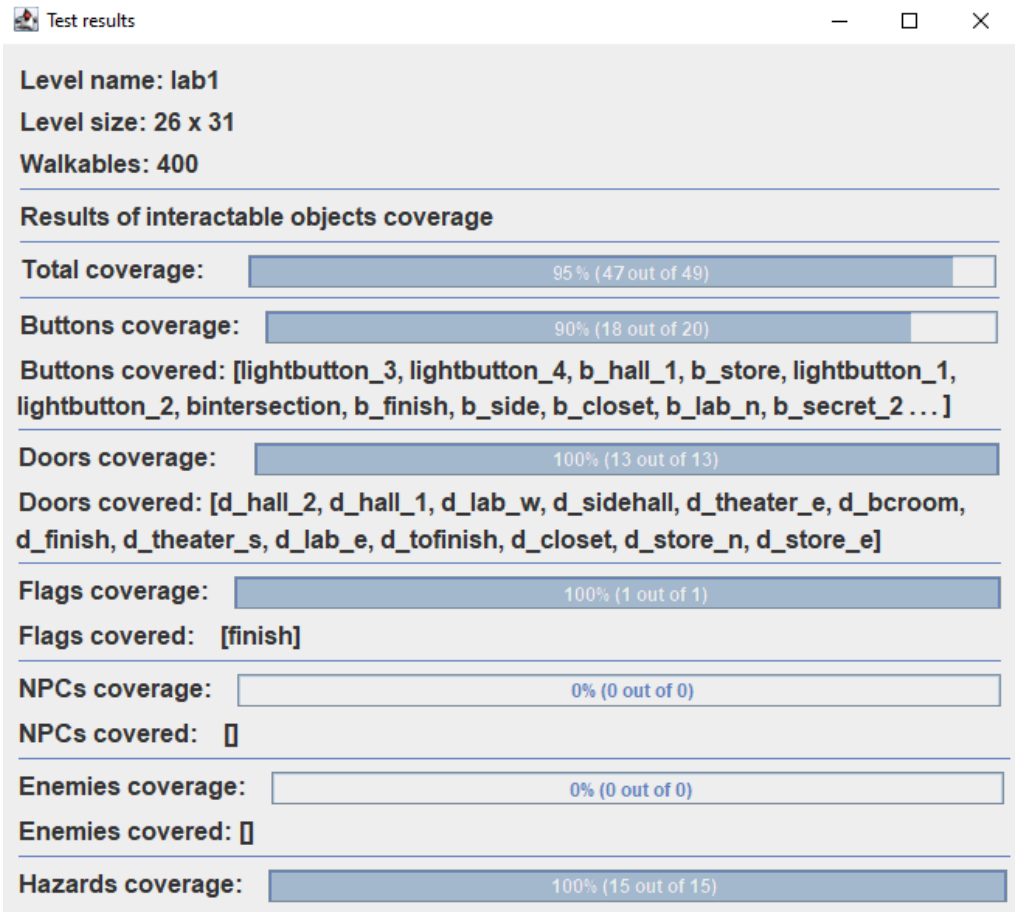

Figure 5.16. Test results on demo test 4

The one considered here is the level on which the methodology of testing

used in 5.1 performed worst, due to the complexity of door-buttons relationship.
The result of the demo test is 95% of total coverage, very good considering that it only missed to cover two buttons, but reached the finish flag, as shown in figure 5.17, rather than the coverage results of previous case in 5.9.

Figure 5.17. Demo test 4 screenshot

5.3 Final considerations

During the conduction of these two testing sessions, several observations have been made.

In general, crafting more generic and adaptable tests that can accommodate diverse level configurations yields superior coverage results.
This approach encourages the agent to interact with a wider range of entities within the game environment, leading to a more thorough exploration of the level's layout and mechanics.

However, this methodology proves less effective when dealing with highly dense levels that involve intricate door/button combinations and complex navigation paths.
In such scenarios, a more suitable approach involves constructing tests specifically tailored to the level, guiding the agent through the intended path in a sequential manner.

This targeted approach ensures that the agent explores the level in a more efficient and effective manner, allowing to specify to the agents key elements and interactions that you want to test.

Chapter 6

Conclusion and future work

The work of this thesis highlights the problem of the scarcity of literature related to video game testing, which focuses mostly on technical performance and bug finding, neglecting fundamental aspects such as multimedia components and user experience (UX), and concentrates mainly on system-level testing, neglecting unit and integration testing. Considering this lack, video game testing could greatly benefit from more elaborate and automated techniques and a more structured strategy.

Therefore, through a systematic literature review and the application of Straussian methodology of grounded theory, a taxonomy containing 26 specific coverage metrics for game testing has been created, divided into 6 macro-categories.
This taxonomy can be an important starting point for formulating a standardized video game testing methodology that provides comparable results, using a common strategy to evaluate testing done with different approaches, with consequent time savings and reduction of any inconsistencies.

An example of this can be found in the second part of this thesis, as the application of defined and standard coverage metrics for the LabRecruits game during the test sessions made it possible to evaluate the quality of different testing methods and compare their effectiveness.

In the future, it would certainly be possible to expand the taxonomy to make it as comprehensive as possible, covering every possible aspect of a

video game with specific metrics, and deepening research on the aspects that have been somewhat neglected so far.
It would also be possible to evaluate in the current academic or industrial landscape which testing tools or frameworks could potentially implement such metrics.
As an additional future step, it would be possible to conduct in-depth comparisons through empirical studies and thus assess various game testing methodologies and approaches using a set of chosen metrics from our established taxonomy, expanding the produced comparable results and allowing to discuss their quality.
Finally, a future work could be the developing of testing tools and framework that natively implement the defined taxonomy, in order to save time and make the application of coverage metrics more accessible.

However, considering the implementation of the metrics in iv4XR and the relative results obtained in chapter 4 and 5, it is clear that there are still significant limitations on the eventual possibility of making the taxonomy and its implementation a general standard applicable to every video game.

Firstly, each game has in-game entities, a game world, levels, and in general a structure, gameplay, and objects/characters defined in ways that can be very different from each other, making it necessary to adapt the metrics to the specifics, as in the case of iv4XR where interactable objects could be doors, buttons, flags.
This is due to the fact that video games are more valuable when they offer a unique experience, on the contrary, for other software applications it's preferable to have similar functioning across the systems, in order to facilitate usability for the end user and compatibility across platforms.
Therefore, the challenge would be to define general metrics that can satisfactorily cover every possible variation of in-game entities, characters, and every element of the game world.
One could think for example of creating subcategories for the type of video game and start from there to create coverage models valid within those subcategories.

As emerged from the systematic literature review, iv4XR is currently the most comprehensive video game testing framework in the academic landscape (which is why it was considered in this work), as it provides libraries to implement a wide range of agent-based testing typologies, and is under

continuous development as it is currently adding features to test also aspects of the user experience related to the agent's emotional state.

During the second phase of this work, the possibility of evaluating the implementation of coverage metrics in iv4XR on a game other than LabRecruits had been considered, in order to assert its effectiveness.

However, in order to test a video game using iv4XR, it is still necessary to write a dedicated interface (environment) that acts as a "translation" connection of the game elements to the framework.
This step is currently not bypassable, and therefore prevents the generalization of tests to different video games.

Looking ahead, potential directions to mitigate the generalization issue could include developing more and more standardized interfaces or adapters that bridge the gap between iv4XR and various game engines, enabling seamless integration and cross-game testing capabilities.

Bibliography

[1] Tom Wijman, *Newzoo's video games market size estimates and forecasts for 2022.* Newzoo.com, May 2023. https://newzoo.com/resources/blog/the-latest-games-market-size-estimates-and-forecasts.

[2] https://www.cyberpunk.net/ca/en/.

[3] Gabriel Ullmann, Cristiano Politowski, Yann-Gäel Guéhéneuc, Fabio Petrillo, *What Makes a Game High-rated? Towards Factors of Video Game Success.* IEEE/ACM 6th International Workshop on Games and Software Engineering (GAS), 2022.

[4] Cristiano Politowski, Yann-Gaël Guéhéneuc, Fabio Petrillo, *Towards Automated Video Game Testing: Still a LongWay to Go.* Proceedings of the 6th International ICSE Workshop on Games and Software Engineering, 2022.

[5] Vahid Garousi, Mika V. Mäntylä, *When and what to automate in software testing? A multi-vocal literature review.* Information and Software Technology, Volume 76, 2016, Pages 92-117.

[6] Vinicius H. S. Durelli, Rafael S. Durelli, Simone S. Borges, et al, *Machine Learning Applied to Software Testing: A Systematic Mapping Study.* IEEE Transactions on Reliability (Volume: 68, Issue: 3, 2019).

[7] Cristiano Politowski, Fabio Petrillo, Yann-Gäel Guéhéneuc, *A Survey of Video Game Testing.* IEEE/ACM International Conference on Automation of Software Test (AST), 2021.

[8] Samira shirzadehhajimahmood, I. S. W. B. Prasetya, Frank Dignum, Mehdi Dastani, and Gabriele Keller, *Using an Agent-Based Approach for Robust Automated Testing of Computer Games.* A-TEST 2021: Proceedings of the 12th International Workshop on Automating TEST Case Design, Selection, and Evaluation, 2021.

[9] Cong Lu, Raluca Georgescu, Johan Verwey, *Go-Explore Complex 3D Game Environments for Automated Reachability Testing.* IEEE Transactions on Games, 2022.

[10] Sinan Ariyurek, *Automated Video Game Testing Using Reinforcement Learning Agents*. Graduate school of informatics of the middle east technical university, 2022.

[11] Ciprian Paduraru, Miruna Paduraru, Alin Stefanescu, *RiverGame - a game testing tool using artificial intelligence*. IEEE Conference on Software Testing, Verification and Validation (ICST), 2022.

[12] Xudong Li, Dajun Zhou, Like Zhang, Yanqing Jing, *Human-like UI Automation through Automatic Exploration*. ISBDAI '20: Proceedings of the 2020 2nd International Conference on Big Data and Artificial Intelligence, 2020.

[13] Xiaoyin Wang, *VRTest: An Extensible Framework for Automatic Testing of Virtual Reality Scenes*. IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), 2022.

[14] Oleguer Canal Anton, *Automatic game-testing with personality: Multitask reinforcement learning for automatic game-testing*. KTH, School of Electrical Engineering and Computer Science (EECS), 2021.

[15] Samantha Stahlke, Atiya Nova, Pejman Mirza-Babaei, *Artificial Players in the Design Process: Developing an Automated Testing Tool for Game Level and World Design*. CHI PLAY '20: Proceedings of the Annual Symposium on Computer-Human Interaction in Play, 2020.

[16] Simon Liu, Li Chaoran, Li Yue et al, *Automatic generation of tower defense levels using PCG*. FDG '19: Proceedings of the 14th International Conference on the Foundations of Digital Games, 2019.

[17] Gustavo Andrade, Geber Ramalho, Hugo Santana, Vincent Corruble, *Automatic computer game balancing: a reinforcement learning approach*. AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems, 2005.

[18] Sven Charleer, Francisco Gutiérrez, Kathrin Gerling, Katrien Verbert, *Towards an Open Standard for Gameplay Metrics*. CHI PLAY '18 Extended Abstracts: Proceedings of the 2018 Annual Symposium on Computer-Human Interaction in Play Companion Extended Abstracts, 2018.

[19] Raphaël Marczak, Jasper van Vught, Gareth Schott, Lennart E. Nacke, *Feedback-based gameplay metrics: measuring player experience via automatic visual analysis*. IE '12: Proceedings of The 8th Australasian Conference on Interactive Entertainment: Playing the System, 2012.

[20] Shengmei Liu, Atsuo Kuwahara, James J Scovell, Mark Claypool, *The Effects of Frame Rate Variation on Game Player Quality of Experience*.

CHI '23: Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems, 2023.

[21] Mark Claypool, Kajal Claypool, *Perspectives, frame rates and resolutions: it's all in the game*. FDG '09: Proceedings of the 4th International Conference on Foundations of Digital Games, 2009.

[22] Saman Zadtootaghaj, Steven Schmidt, Sebastian Möller, *odeling Gaming QoE: Towards the Impact of Frame Rate and Bit Rate on Cloud Gaming*. IEEE Tenth International Conference on Quality of Multimedia Experience (QoMEX), 2018.

[23] Benedikt Dietich, Nadja Peters, Sangyoung Park, Samarjit, Chakraborty, *Estimating the Limits of CPU Power Management for Mobile Games*. IEEE 35th International Conference on Computer Design, 2017.

[24] Farouk Messaoudi, Gwendal Simon, Adlen Ksentini, *Dissecting games engines: The case of Unity3D*. IEEE International Workshop on Network and Systems Support for Games (NetGames), 2015.

[25] https://blog.gamebench.net/game-performance-metrics-that-matter. Gamebench, 2019.

[26] Johannes Pfau, Jan David Smeddinck, Rainer Malaka, *Automated Game Testing with ICARUS: Intelligent Completion of Adventure Riddles via Unsupervised Solving*. CHI PLAY '17 Extended Abstracts: Extended Abstracts Publication of the Annual Symposium on Computer-Human Interaction in Play, 2017.

[27] Rodrigo Casamayor, Lorena Arcega, Francisca Pérez, Carlos Cetina, *Bug Localization in Game Software Engineering: Evolving Simulations to Locate Bugs in Software Models of Video Games*. MODELS '22: Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems, 2022.

[28] Simon Varvaressos, Kim Lavoie, Sébastien Gaboury, Sylvain Hallé, *Automated Bug Finding in Video Games: A Case Study for Runtime Monitoring*. Computers in Entertainment Volume 15 Issue 1, 2017.

[29] https://www.browserstack.com/guide/code-coverage-vs-test-coverage.

[30] Geeta Rani, Upasana Pandey, Aniket Anil Wagde, Vijaypal Singh Dhaka, *A deep reinforcement learning technique for bug detection in video games*. International Journal of Information Technology, 2022.

[31] Carolina Veloso, Rui Prada, *Validating the plot of Interactive Narrative games*. 2021 IEEE Conference on Games (CoG), 2021.

[32] Fernandes Pedro M., Jonathan Jørgensen, Niels NTG Poldervaart,

Adapting Procedural Content Generation to Player Personas Through Evolution. IEEE Symposium Series on Computational Intelligence (SSCI), 2021.

[33] Ansari Saba Gholizadeh, et al., *An Appraisal Transition System for Event-Driven Emotions in Agent-Based Player Experience Testing*. International Workshop on Engineering Multi-Agent Systems, Springer, Cham, 2021.

[34] Fernandes Pedro M., Manuel Lopes, Rui Prada, *Agents for automated user experience testing*. IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2021.

[35] https://iv4xr-toolkit.eu/.

[36] *AI Agents in Software Testing*. https://testrigor.com/

[37] Muhammad Usmana, Ricardo Britto, Jürgen Börstler, Emilia Mendes, *Taxonomies in software engineering: A Systematic mapping study and a revised taxonomy development method*. Information and Software Technology, Volume 85, 2017.

[38] Riccardo Coppola, Emil Alégroth, *A taxonomy of metrics for GUI-based testing research: A systematic literature review*. Information and Software Technology, Volume 152, 2022.

[39] Functional testing, https://en.wikipedia.org/wiki/Functional_testing

[40] Johan Hoberg, *Game Testing: Exploring the Test Space*, https://www.gamedeveloper.com/programming/game-testing-exploring-the-test-space, 2014.

[41] *Testing and quality assurance tips for Unity projects*, https://unity.com/how-to/testing-and-quality-assurance-tips-unity-projects

[42] *UI Testing: A Detailed Guide*, https://www.browserstack.com/guide/ui-testing-guide

[43] Johan Hoberg, *Differences between Software Testing and Game Testing*, https://www.gamedeveloper.com/programming/differences-between-software-testing-and-game-testing, 2014.

[44] *What is UX testing with example*, https://www.browserstack.com/guide/what-is-ux-testing#:~:text=A%20user%20experience%20test%20is,application%20from%20the%20user's%20perspective.

[45] *Reliability Testing – Software Testing*, https://www.geeksforgeeks.org/software-testing-reliability-testing/

[46] Xiongfei Wu, Jiaming Ye, Ke Chen et al., *Widget Detection-based Testing*

for Industrial Mobile Games. IEEE/ACM 45th International Conference on Software Engineering, 2023.

[47] *Belief–desire–intention software model*, https://en.wikipedia.org/wiki/Belief%E2%80%93desire%E2%80%93intention_software_model

[48] Maurizio Leotta, Diego Clerissi, Filippo Ricca, Paolo Tonella, *Capture-replay vs. programmable web testing: An empirical assessment during test case evolution*. 20th Working Conference on Reverse Engineering (WCRE), 2013.

[49] Yoonsik Cheon, *Automated Random Testing to Detect Specification-Code Inconsistencies*. Departmental Technical Reports (CS), 2007.

[50] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks, *Evaluating Fuzz Testing*. Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security.

[51] Axel Bons, Beatriz Marín, Pekka Aho, Tanja E.J. Vos, *Scripted and scriptless GUI testing for web applications: An industrial case*. Information and Software Technology, Volume 158, June 2023.

[52] https://www.selenium.dev/

[53] https://testar.org/

[54] Henna-Riikka Ruonala, *Agile Game Development: A Systematic Literature Review*. M.S. thesis, University of Helsinki, Faculty of Science, Department of Computer Science, 2016.

[55] Emerson Murphy-Hill, Thomas Zimmermann and Nachiappan Nagappan, *Cowboys, Ankle Sprains, and Keepers of Quality: How Is Video Game Development Different from Software Development?*. ICSE 2014: Proceedings of the 36th International Conference on Software Engineering.

[56] Barbara A Kitchenham, *Systematic review in software engineering: where we are and where we should be going*. Proceedings of the 2nd international workshop on Evidential assessment of software technologies, 2012.

[57] Vahid Garousi, Michael Felderer, and Mika V Mäntylä, *The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature*. Proceedings of the 20th international conference on evaluation and assessment in software engineering, 2016.

[58] Vahid Garousi, Michael Felderer, Mika V. Mäntylä, *Guidelines for including grey literature and conducting multivocal literature reviews in software engineering*. Information and Software Technology, Volume 106, February 2019, Pages 101-121.

[59] Paul Ralph, *Toward methodological guidelines for process theories and taxonomies in software engineering*. IEEE Transactions on Software Engineering, 2018.

[60] Johanna C Van Niekerk, JD Roode,*Glaserian and Straussian grounded theory: similar or completely different?*. Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists.

[61] https://github.com/iv4xr-project/labrecruits.