Co-evolving Scenarios and Simulated Players to Locate Bugs that arise from the Interaction of Software Models of Video Games

Isis Roca[a,b,*], Óscar Pastor[b], Carlos Cetina[a], Lorena Arcega[a]

[a] *SVIT Research Group, Universidad San Jorge, Zaragoza, Spain*
[b] *PROS Research Centre, Universitat Politècnica de València, Valencia, Spain*

Abstract

Context: Game Software Engineering (GSE) is a field that focuses on developing and maintaining the software part of video games. A key component of video game development is the utilization of game engines, with many engines using software models to capture various aspects of the game.

Objective: A challenge that GSE faces is the localization of bugs, mainly when working with large and intricated software models. Additionally, the interaction between software models (i.e. bosses, enemies, or environmental elements) during gameplay is often a significant source of bugs. In response to this challenge, we propose a co-evolution approach for bug localization in the software models of video games, called CoEBA.

Method: The CoEBA approach leverages Search-Based Software Engineering (SBSE) techniques to locate bugs in software models while considering their interactions. We conducted an evaluation in which we applied our approach to a commercial video game, Kromaia. We compared our approach with a state-of-the-art baseline approach that relied on the bug localization approach used by Kromaia's developers and a random search used as a sanity check.

Results: Our co-evolution approach outperforms the baseline approach in precision, recall, and F-measure. In addition, to provide evidence of the significance of our results, we conducted a statistical analysis that shows significant differences in precision and recall values.

Conclusion: The proposed approach, CoEBA, which considers the interaction between software models, can identify and locate bugs that other bug localization approaches may have overlooked.

Keywords: Bug Localization, Model Interaction, Game Software Engineering, Search-Based Software Engineering, Model-Driven Engineering

1. Introduction

The video game industry has become a significant and lucrative software industry sector [1]. As video games continue to gain popularity, the development of these games has become increasingly complex, involving the creation of sophisticated software covering multiple aspects of the game, such as game mechanics, graphics, and physics. Game Software Engineering (GSE), therefore, has emerged as a field that focuses on the development and maintenance of software for video games [1].

In video game development, game engines serve as a basic resource that combines elements such as graphics and physics engines with a suite of accompanying tools. While there are popular game engines [2, 3], some studios opt for custom solutions tailored to their specific needs [4]. A recent literature review [5] identifies both widely used game engines such as Unity or Unreal and specific engines developed by the video game studios themselves. A key asset of game engines is software models. In the scope of this work, the concept of software model should not be confused with the concept model used in computer graphics and video games to refer to the visual representation of 3D shapes or geometry. An actual example of a model of the scenario of a final boss of Kromaia is presented in Figure 1. In addition, Kromaia developers can also use the C++ programming language to create content for the game. Game engines offer developers a choice between coding, often in languages such as C++, or using software models that are integrated into the engines (as illustrated in Figure 2 [6]). These models provide a higher level of abstraction, allowing developers to focus on designing the core content of the game. Typically, game engines incorporate their own modeling languages (for instance, Unreal employs Blueprint models), while recent trends in Model-Driven Game Development [5] highlight the adoption of other languages such as UML and Domain Specific Languages (DSLs).

This approach, similar to Model-Driven Engineering (MDE), has proven to be useful in simplifying video game development. However, applying MDE to video games requires specific bug localization approaches to ensure the quality of the

*Corresponding author.
Email addresses: iroca@usj.es (Isis Roca), opastor@pros.upv.es (Óscar Pastor), ccetina@usj.es (Carlos Cetina), larcega@usj.es (Lorena Arcega)

[1] T. B. R. Company, Gaming global market report 2023, https://www.thebusinessresearchcompany.com/report/gaming-global-market-report (2023).

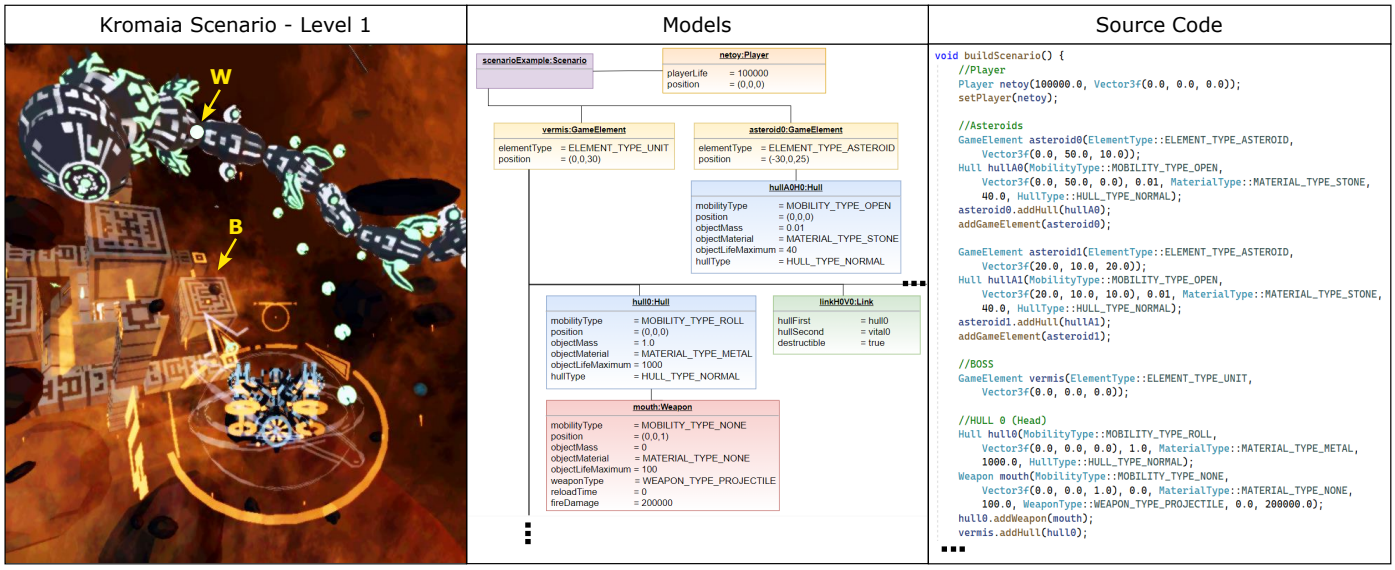

Figure 1: Example of a scenario developed using models and code.

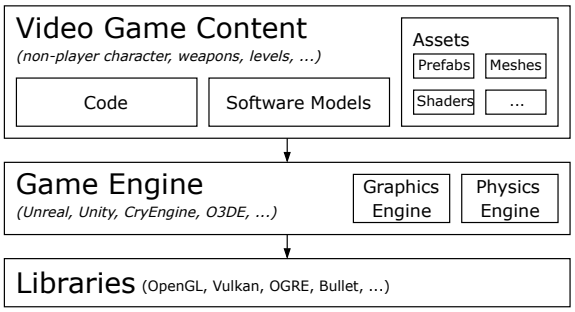

Figure 2: Overview of video game artifacts.

software. Currently, bug location approaches mainly focus on source code without considering other crucial elements such as software models. This is shown in a survey on bug location [7], which highlights that none of the existing approaches consider models as a potential source of bugs. This lack of appropriate approaches can have a negative impact on video game development. It often results in an increase in development time, which can lead to delays in delivery and, ultimately, the release of games with a high number of bugs. A good example of this is the blockbuster *Cyberpunk 2077* [2], which was released after nine years of development with numerous bugs. Such was the level of bugs that the game had to be pulled from store shelves, and even a year after its initial release, patches were still being released to fix them.

In this context, it is important to note that video games have different characteristics compared to classic software in both, developing and maintaining, due to the differences in the artifacts used (i.e., shaders, meshes, or prefabs) and the interactive nature of video games. Additionally, game developers face different challenges than non-game developers when locating bugs and reusing code [8].

In video game development, like the boss shown in Figure 1, content can be defined through code or models (see the central part of Figure 1 for the model and the right part for the code). Models offer higher-level concepts and abstraction from implementation details, providing a smaller bug localization search space. However, despite this advantage, existing literature on bug localization in models is scarce, posing a challenge. As models are key artifacts in video game development, bug localizations should be performed on models.

Furthermore, due to the interactive nature of video games, the interaction between software models in the gameplay can be a significant source of bugs and cannot be ignored. For instance, the interaction between the player, the game environment, and the game mechanics can result in unpredictable behavior and bugs. The boss that appears in Figure 1 is a powerful enemy that appears at the end of the level. The bosses have weak points that must be attacked to destroy them. However, a boss can be invincible because a weak point is overlapped. This bug occurs when solid objects overlap each other when they are not supposed to. Letter W in Figure 1 indicates a weak point. When the boss hits some blocks (letter B in Figure 1) that are part of the scenario, this type of bug occurs. Someone may argue that this bug may occur due to the emergent behavior of the game. However, other bugs can occur even in deterministic games. For example, a boss can have wrong behavior because the movement is blocked. This bug occurs when the parts of the boss are incorrectly positioned. Incorrect positioning blocks the movement of other parts of the model; for example, the position of one part invades the space of the other. If they invade each other the physics have unpredictable behavior. This bug occurs due to an error in the boss's programming (in code or models). In games that do not provide linear experiences, the player has many possibilities to perform different behaviors. This leads to the concept of emergent gameplay. It describes situations where a game's mechanics and systems interact in ways that

[2] How buggy is Cyberpunk 2077, really? - `https://www.pcgamer.com/how-buggy-is-cyberpunk-2077-really/`

were not intentionally designed by the developers but rather created through the choices and actions of players. In addition, games can incorporate random elements. Such elements can include factors like randomized enemy behavior, variable item drops, or procedural level generation. Therefore, developers have the ability to create diverse and dynamic experiences for players, making each playthrough unique.

Given the complexity of video games thorough game testing becomes imperative. Game testing involves systematically evaluating the game's mechanics, interactions, and random elements to identify bugs, balance issues, and unintended emergent behaviors. In addition, games can also have many scenarios, which produce an excessively high amount of different combinations that are unaffordable to be playtested by a human [9].

In this work, we present an approach, called CoEBA, for bug localization in game software engineering that considers the interaction between different software models. Our approach applies Search-Based Software Engineering (SBSE) [10] techniques to locate bugs by exploring the search space evolving software models. Our CoEBA approach is designed to co-evolve simulated players and scenarios to identify potential bugs in software models while considering their interactions. Simulated players represent player behaviors during the gameplay, and scenarios represent the environment around the player including context, enemies, and bosses. In these scenarios is where model interactions occur. In order to validate the effectiveness of our approach, we compared it with two other approaches, a baseline [6] and a random search using the Kromaia case study [11]. Kromaia is a commercial video game that involves flying and shooting with a spaceship in 3D space and has been released on multiple platforms and translated into different languages.

To evaluate the CoEBA approach, we have formulated three research questions:

- RQ_1: What is the performance in terms of solution quality of the CoEBA, the baseline, and the random search approaches?
- RQ_2: Is there any improvement in performance taking into account model interaction?
- RQ_3: Are the differences in performance results obtained by the CoEBA, the baseline, and the random search approaches significant?

In our evaluation, we use widely accepted metrics such as precision, recall, and F-measure to evaluate the performance of our approach, the baseline approach, and a random search approach. Our results show that CoEBA outperformed the baseline and the random search approaches regarding recall, precision, and F-measure. The precision, recall, and F-measure values achieved by CoEBA are 45.76%, 84.83%, and 53.87%, respectively. We performed a statistical analysis to provide quantitative evidence of the impact of the results. This analysis shows that the impact on performance is significant.

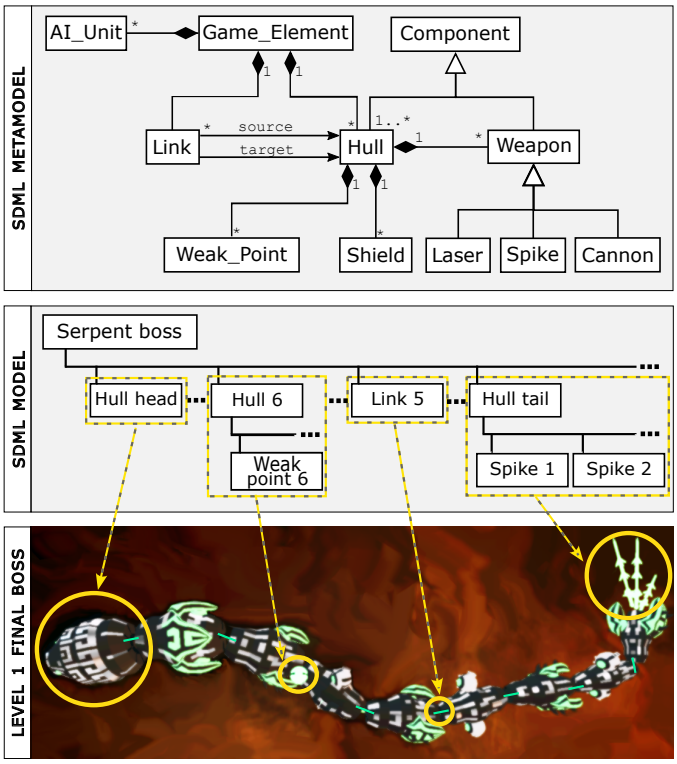

Figure 3: The Shooter Definition Modeling Language (SDML).

From the evaluations performed, we can say that the interaction between models is relevant to locating bugs. This is because it defines the behavior of the video game as a whole and provides insights to facilitate bug location improving the performance of the approach.

The structure of this paper is as follows. Section 2 provides an overview of the case study (Kromaia) and the motivation. Section 3 describes our proposed approach, CoEBA. In Section 4, we present the evaluation methodology of our approach with Kromaia. In section 5, we present the results. In sections 6 and 7, we discuss the results and analyse the threats to validity respectively. In Section 8, we present a review of the related work in this area. Finally, in Section 9, our conclusions are summarized.

2. Background and motivation

This section presents the case study we employed to evaluate this work. The case study involves the video game Kromaia[3]. Kromaia is a commercial video game released on PC and PlayStation 4. It is set in a three-dimensional space. The game is a blend of space exploration, action, and shooting genres. Each level requires the player to pilot their spaceship from a starting point to a target destination while exploring floating structures, avoiding asteroids, and collecting items along the way. The player must also defeat basic enemies attacking their

[3]To learn more about Kromaia, you can watch the official trailer released by Playstation: https://youtu.be/EhsejJBp8Go

spaceship by firing projectiles. Once the player reaches the destination, the final boss for that level appears, and the player must defeat the boss to progress to the next level.

The developers of Kromaia use a Domain Specific Language (DSL) called Shooter Definition Modeling Language (SDML) to describe the various characteristics and behaviors of the game's entities, including bosses, environmental elements, weapons, defenses, and movement behaviors. The developers of Kromaia implemented this DSL to define shooter-style video games. However, its applicability extends beyond this genre to other video games with similar themes. This modeling language has a fundamental role in the definition of various attributes in relation to game entities. It provides a comprehensive description of the anatomical structure of the characters, covering details such as the constituent parts, their physical characteristics, and how they are interconnected. In addition, the DSL delves into the configuration of vulnerable sections, weaponry, and protective measures within the character's structure or body. It also regulates the various movement behaviors associated with the character as a whole or its individual components. SDML follows the principles of Model-Driven Engineering (MDE), allowing developers to specify the rules of engagement between entities and the game environment, such as boundaries, obstacles, and environmental hazards. The upper part of Figure 3 presents an excerpt version of the SDML metamodel that provides a comprehensive understanding of the game's elements in Kromaia. The complete metamodel consists of over 20 concepts, 20 relationships, and more than 60 properties. The SDML models are interpreted at runtime to generate the corresponding entities in the video game. This means that the software models are created using SDML and translated into C++ objects at runtime by an interpreter used by the game engine. Kromaia was developed with a specific video game engine created by the company [11].

Kromaia's specific engine uses concepts similar to those of widespread engines such as Unity or Unreal. In fact, Kromaia's SDML models can be loaded into Unity's WebGL (`https://svit.usj.es/models22/bl-in-mgse`). This engine acts as a framework in the context of the video game architecture and allows developers to create content in two different ways:

- Programming, using the framework's Application Programming Interface (API).
- Software Models, which are created using SDML and translated to its programming equivalent at run-time by an interpreter that is used by the engine.

Therefore, developers should track only what they develop for each part of the game content. In the specific case of Kromaia, the developers started to create content with code, but to be more productive they switched to SDML models to create the content of the game (e.g., levels, NPCs, items, and weapons). More can be learned about the SDML model of Figure 3 in the following video: `https://youtu.be/Vp3Zt4qXkoY`

SDML allows the development of any game element such as bosses, enemies, or environmental elements. An actual example of a final boss of Kromaia is presented in the bottom part of Figure 3. The Serpent is the final boss that the player must defeat to complete level 1. The central part of the figure shows the model of the boss using the SDML. This boss is composed of one main hull (the head of the serpent) followed by eight hulls. The tail (last hull) has two shields and three spike weapons. As in this example, all bosses, enemies, and elements in Kromaia are implemented using SDML.

In order to locate bugs related to the SDML models that define the behavior of the game's bosses, a previous work employs an approach that involves running game simulations [6]. These simulations are designed to simulate a one-on-one battle between a boss and a simulated player that represents the actions and decisions of a human player. The simulated player is powered by an algorithm that the Kromaia team developed. The developers used the algorithm during the game's development to evaluate the content they made. The algorithm focuses on simulating the player, so it is independent of changes in the other elements of the game (attacking weak points and avoiding taking damage or colliding with the scenario). However, the algorithm is game-dependent (space shooter), so it cannot be used as it is in other games. However, it would be possible to find similar algorithms in other games, e.g., those controlling opposing cars in a racing game, enemy generals commanding troops in an RTS game, or bots in an FPS game. The algorithm can be parametrized to define the fighting strategy. The simulated player parameters were provided by the developers based on the analysis of battles between human players and bosses. It considers aspects such as the weaponry used by each of them and the differences between the two entities in terms of agility, speed, endurance, or size. For example, the parameters can define the time that the player spends in each element, how the level is traversed, or the player's reaction when the player is hit. During the simulated battle, both the boss and the simulated player are programmed to strive for victory and avoid tie games at all costs, ensuring that there is a clear winner.

In addition to the bosses and the simulated player, environmental elements in Kromaia play a crucial role in shaping the gameplay experience. These elements, such as walls, spikes, and towers, significantly impact the actions and strategies of both the boss and the player during a duel. The placement and layout of these elements can create opportunities for the boss and the player to gain advantages or disadvantages and influence their movement patterns and attack styles. For example, walls can serve as a strategic cover for the boss, allowing them to evade incoming attacks or gain an advantageous position to launch a counter-attack. Similarly, the player may use walls to dodge or avoid the boss's attacks or launch surprise attacks behind the cover. Towers that shoot projectiles can provide an additional challenge for both the boss and the player, as they must be avoided or destroyed to gain an advantage.

Moreover, the presence of spikes or other environmental elements can further enrich the gameplay experience, as the player needs to be aware of their location and avoid taking damage from them. The placement of these environmental elements can force the player to alter their attack strategies or movement patterns to avoid taking damage, adding a layer of complexity to the game. Thus, to achieve a realistic simulation between the

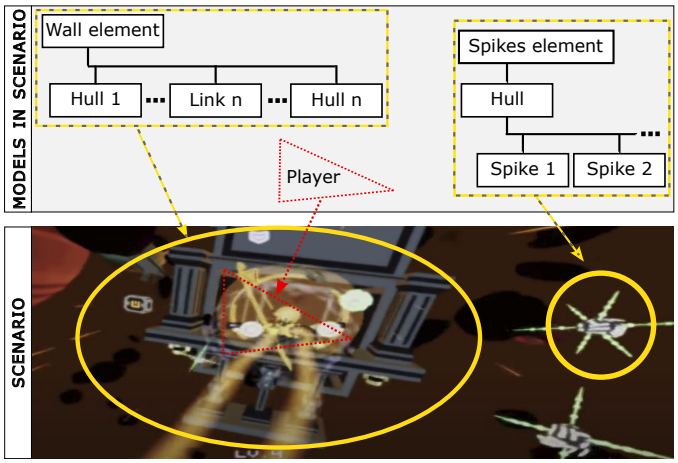

Figure 4: Example of model interaction in the scenario.

boss and the simulated player, it is essential to consider the impact of these environmental elements in conjunction. We refer as a scenario to the interaction of all these models (bosses and elements of the environment) that conform to the same metamodel. Figure 4 shows an example of a real scenario with walls and spikes and the player in the middle.

The figure depicted in Figure 5 presents two different simulations that emulate a player's behavior during a battle with a boss in the game. These simulations aim to predict how the boss and simulated player will behave under different conditions and to locate bugs in the software models of the game. The triangle symbol in the figure represents the simulated player, while the circles and lines that connect them correspond to game elements, such as the boss, towers, spikes, or walls. The dashed lines indicate the path followed by the simulated player or the projectiles, and the crosses represent the attacks performed by the simulated player, the boss, and the environmental elements.

The upper simulation presented in Figure 5 shows a duel between the simulated player and a boss without considering the environmental elements (a previous work [6]). The bottom example shows a simulation between the simulated player, a boss, and (taking into account) the environmental elements (the one used in this work). That is, this simulation considers several software models of the video game (i.e., boss, towers, spikes, or walls) and the interactions that these models have throughout the simulation. The $M1$ model represented in this figure is the level 1 final boss shown in Figure 3, the rest of the models ($M3$, $M4$, $M6$, $M7$, $M8$, $M12$, $M13$, $M21$, $M33$, and $M36$) represented show environmental elements that are present in the scenario and that have been defined using SDML (Fig 3).

The interaction among the software models of the environmental elements, the boss, and the simulated player can also generate unwanted bugs that may impact the game's overall quality. For instance, if the boss's software model is not appropriately designed to handle the presence of walls or towers, this could cause unintended behaviors that could lead to glitches and an unpleasant gaming experience for the player. Similarly, if the player's behavior is not adapted to the presence of environmental elements, such as walls or spikes, this could result

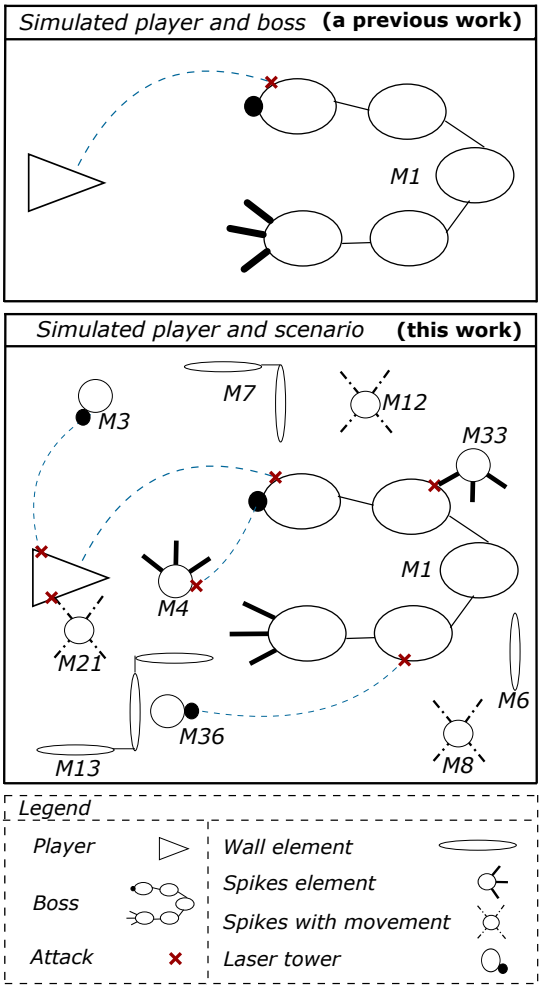

Figure 5: Example of two different simulations: simulated player and boss (one software model), simulated player, boss, and environmental elements (several software models).

in unintended collisions or ineffective attacks. To minimize the risk of unwanted bugs, it is crucial to incorporate the impact of environmental elements into the simulations and thoroughly test the gameplay. This can help to ensure that the interactions between the boss, the simulated player, and the scenario are smooth and error-free, leading to a more enjoyable and immersive gaming experience.

3. Co-Evolving scenarios and simulated players to locate bugs in the interaction of software models of video games

The gameplay of a video game can involve numerous combinations of scenarios (game elements in the scene) and player strategies, so the number of different model interactions that can be triggered is very high. Since the search space is too large, it is difficult to perform playtesting manually, exploring the space of possibilities exhaustively [9]. To tackle this challenge, we leverage SBSE through a co-evolutionary algorithm to explore the vast search space produced by combinations of scenarios (game elements in the scene) and simulated players (player behaviors in the battle).

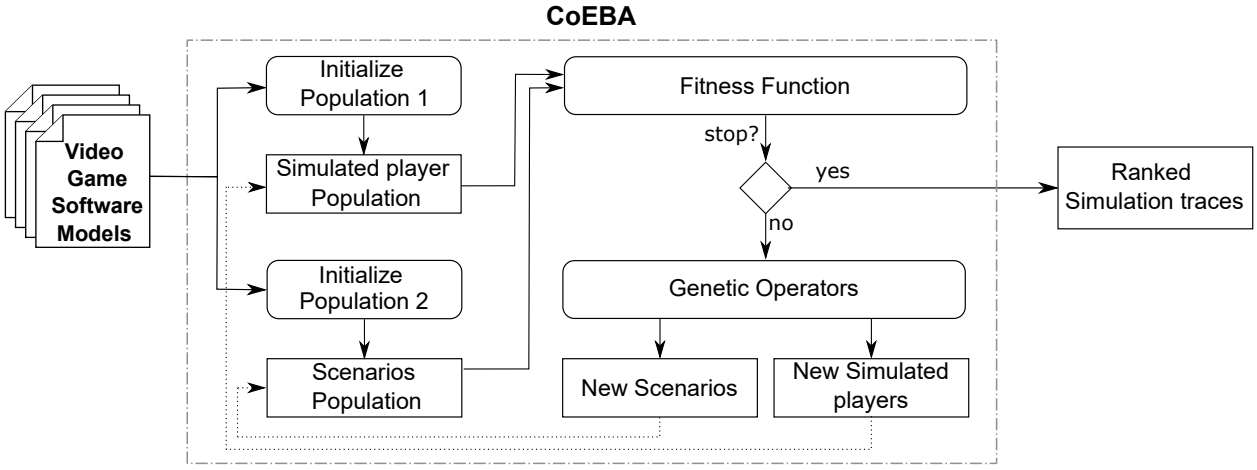

Figure 6: CoEBA: Co-Evolutionary Bug Localization approach for GSE.

In co-evolutionary algorithms, instead of evolving a population of individuals representing the solution, they co-evolve subpopulations representing specific parts of the solution. There are two types of co-evolution: cooperation and competition [12, 13]. We have selected a competitive co-evolutionary algorithm for its ability to solve problems involving interactions between different entities. Such problems arise in situations where entities require adaptation and evolution in response to their opponents' strategies, as in games in which competitive co-evolutionary algorithms excel [14, 15]. Taking into account the problem we want to solve, we follow a competitive approach taking into account the simulated players and the environmental elements where the fitness value of an individual from one subpopulation depends on the fitness value of other individuals from the other subpopulation.

In this section, we first present an overview of our approach and then provide the details of the approach and our adaptation of the co-evolutionary algorithm to deal with the specific challenges of software models of video games.

3.1. Overview of our approach

Figure 6 presents our Co-Evolutionary Bug Localization approach for GSE called CoEBA. The left part shows the input for the approach: the set of software models that describe the video game's content. The center shows the main steps. The 'Initialize Population' step calculates the two initial populations. The 'Fitness' step assigns values that assess how good each individual of each population is. Finally, the 'Genetic Operations' step produces new generations of both populations. The approach outputs a ranked list of simulation traces that can trigger the bug.

3.2. CoEBA populations

CoEBA works with two different populations. The first population is composed of simulated players, which represent player behaviors during battles. The second population is composed of game scenarios, encompassing diverse combinations of game elements within a given scene. To represent each individual of each population, we use a vector representation.

For the first population, each vector's dimension represents a simulated player parameter. The simulated player emulates the behavior of a player in a determined scenario when the battle with the boss occurs. For example, the parameters can define how many steps the simulated player takes in each hull of the boss or each environmental element, the order in which the hulls and the environmental elements are visited following different patterns, the behavior of the player when they are attacked, or the direction used to visit the hulls of the boss or the environmental elements. Thus, an individual is defined as a set of parameters applied to a simulated player. The size of the individual corresponds to the number of parameters (dimensions) in the vector. The developers provided the simulated player parameters based on the analysis of real games played. Figure 7 upper part shows four different simulated players. The SP1 example shows a simulated player that attacks the first hull and then moves away. The SP2 example shows a simulated player that attacks the first hull, then skips one, and attacks the following hulls to the end of the boss.

For the second population, each vector's dimension represents a software model that can appear (or not) in the scenario (see the central part of Figure 7). That is, each vector represents the different types of environmental elements (software models) that can come onto the scene. Each vector in the population has a dimension corresponding to a specific scenario and contains information about the software models present in the scenario. The number of different types of environmental elements available in the game determines the number of dimensions in the vector. These environmental elements have been identified and defined by the Kromaia developers, and we have used their specifications to create our vector representation for the second population.

The central part of Figure 7 presents four different scenarios. Each example presents different types of software models that appear as environmental elements in the scenario. To represent the S1 example, the individual would have the dimension corresponding to $M3$ set to 1, while the dimension corresponding to $M2$ would be set to 0. In contrast, the S2 example would have the dimension corresponding to $M2$ set to 1, while the dimen-

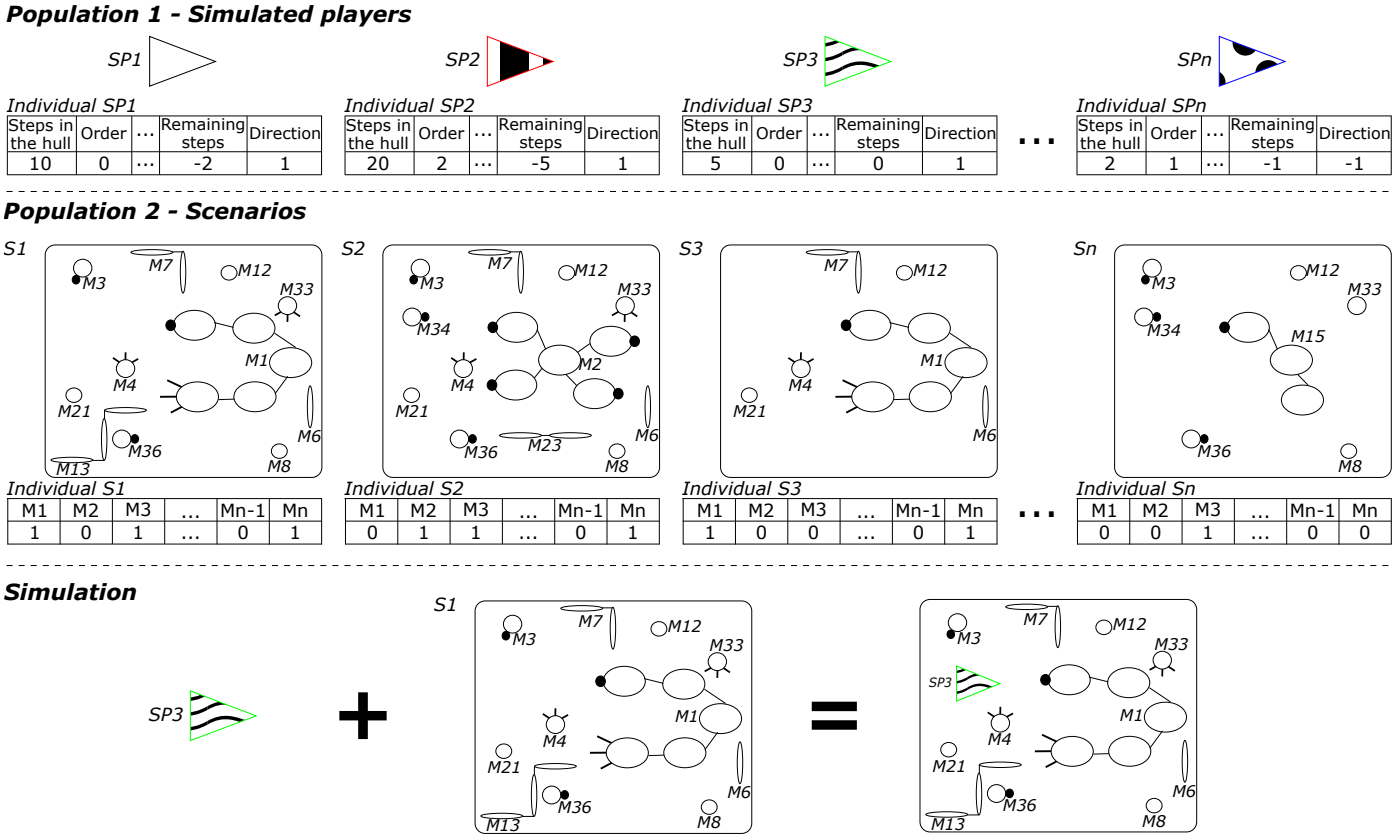

Figure 7: Examples of individuals and their representation from both populations simulated players and scenarios, and an example of a simulation (simulated player + scenario)

sion corresponding to $M13$ would be set to 0. The number and the type of these environmental elements can play an essential role while the combination of a simulated player and a scenario (that is, a simulation, see bottom part of Figure 7) is running. Hence, the interaction of the software models of the environmental elements, the player, and the boss when all of them are put together in the scenario can trigger unwanted bugs.

3.3. CoEBA fitness

The fitness function used by our approach assesses simulated players and scenarios based on their ability to detect game bugs. It is based on two functions for each population: 1) evaluating their alignment with how the game developers intended for the game to behave; 2) maximizing the coverage of the less suitable simulated players or scenarios in the generation.

In both populations, the first function involves a duel between the game's player and boss. The goal is to evaluate how closely the game's behavior aligns with the developers' intent. This function calculates its fitness value based on the percentage of player victories and the percentage of player health remaining after winning a simulated battle. The calculations for victory and health are performed in the same manner as described in previous studies [11, 6]. In a previous study by Blasco et al. [11] (content generation), simulations closest to the developers' intentions were rewarded. However, the opposite approach is taken for bug localization [6]. Hence, in this work, simulated players that deviate the most from what developers intended are rewarded and ranked first in the function.

For the second fitness function, we select the top individuals from each population, which are the worst simulated players and scenarios. For assessing each simulated player, our approach considers the number of scenarios that did not achieve expected victory and health values in the duel. The more scenarios that generate unwanted behavior with the simulated player, the higher the fitness value. In order to normalize the numerical value, we establish this as a percentage of the undesired scenarios against the total number present in the population. Similarly, our approach considers the number of simulated players that did not achieve expected victory and health values in the duel to assess each scenario. The more simulated players that generate unwanted behavior with the scenario, the higher the fitness value. In order to normalize the numerical value, we establish this as a percentage of the undesired simulated players against the total number present in the population. Thus, resulting values can vary from 0 to 1.

The fitness value of a simulated player is the average value between the first and the second fitness values described above. Subsequently, our algorithm orders simulated players and scenarios in an ascending manner with those having lower fitness values ranked first. A lower fitness value means that the simulated player or the scenario produces undesirable outcomes, which do not align with the intended objectives of the develop-

ers.

3.4. CoEBA genetic operators

To generate subsequent populations, genetic operators are applied, which involve selecting the individuals that will serve as parents for the new individuals. Following the selection operator, other genetic operators, such as crossover and mutation, are applied to manipulate the individuals. These operators help in generating new sets of individuals from the existing ones. The selection process ensures that the fittest individuals are more likely to be selected as parents, thus increasing the chances of producing offspring with better fitness scores. Furthermore, the use of genetic operators like crossover and mutation helps to introduce diversity into the population, preventing premature convergence and promoting exploration of the search space.

Selection. In order to select individuals from both subpopulations, our approach considers their fitness values. Our approach employs the roulette wheel selection mechanism. This selection approach helps to prevent premature convergence [16], a phenomenon that can reduce the algorithm's performance. In essence, the roulette wheel selection mechanism assigns a probability of selection to each individual in the population that is directly proportional to their fitness score. By using this technique, the algorithm guarantees that solutions with greater fitness have a higher chance of being selected, emulating the concept of "survival of the fittest". This aids in the convergence towards optimal or nearly optimal solutions. Meanwhile, the roulette wheel selection mechanism maintains diversity by providing less fit solutions with a possibility of being chosen. This diversity is crucial as it prevents the algorithm from becoming trapped in local optima and encourages the exploration of the solution space, which is especially critical for complex optimization problems. The selected individuals will be the ones that generate the next individuals.

Crossover. In the CoEBA approach, the crossover operator plays a significant role in generating new and diverse individuals. This operator involves the combination of genetic material from two parent solutions to create offspring solutions with new characteristics. CoEBA uses a single, random, cut-point crossover, where two-parent solutions are split at a random cut-point to generate two sub-vectors. The offspring solutions are then produced by combining the first part of one parent with the second part of the other parent for the first offspring and combining the first part of the second parent with the second part of the first parent for the second offspring. It should be noted that the length of the solutions remains constant during the crossover operation. Thus, the new solutions obtained through the crossover operation are the same length as the parent solutions. This method helps to promote diversity and can lead to the discovery of new individuals that may not have been present in the previous population.

Mutation. In our approach, the mutation operator plays a crucial role in maintaining diversity in the population and exploring new regions of the search space. The mutation operator randomly changes one or more parameters in a solution, introducing novel genetic material that can lead to improved solutions. For the simulated player population, the mutation operator starts by selecting certain positions within the vector representation of the individual. These selected dimensions are then replaced by a different value of the corresponding parameter. However, the values for mutation are not randomly generated numbers. Instead, they are obtained from a catalog of values compiled by developers based on their experience. Similarly, the mutation operator for the scenario population also starts by selecting certain positions within the vector representation of the individual. These selected dimensions are then replaced by a different value of the corresponding environmental element. By utilizing a pre-collected catalog of values, the mutation operator can efficiently and effectively explore the search space of both populations. The use of a pre-collected catalog of values also ensures that the mutations are relevant and appropriate for the game's mechanics, increasing the likelihood of finding effective and realistic solutions.

CoEBA generates a collection of simulation traces ranked according to their expected probability of triggering the intended bug. Overall, the aim of CoEBA is to highlight potential sources of bugs in the interaction between the software models of the video game.

4. Evaluation methodology

This section presents the methodology used in the evaluation of the approach, including the preparation of the evaluation, the experimental process, the measures used to evaluate the algorithm performance, and the explanation of the statistical analysis performed.

In order to evaluate the CoEBA approach, we have formulated the following research questions:

- RQ_1: What is the performance in terms of solution quality of the CoEBA, the baseline, and the random search approaches?
- RQ_2: Is there any improvement in performance taking into account model interaction?
- RQ_3: Are the differences in performance results obtained by the CoEBA, the baseline, and the random search approaches significant?

Answering the first research question (RQ_1) enables us to evaluate and contrast the performance outcomes of our proposed approach with the baseline approach, using metrics such as recall, precision, and F-measure. Additionally, we compare our approach and a random search (RS) as a sanity check. The second research question (RQ_2) seeks to determine whether model interactions affect the results. Finally, answering the third research question (RQ_3) allows us to conduct a thorough comparison between the approaches, provide formal and quantitative evidence (statistical significance) that the differences in the results are not due to chance, and demonstrate that these differences are significant in practice (i.e., effect size).

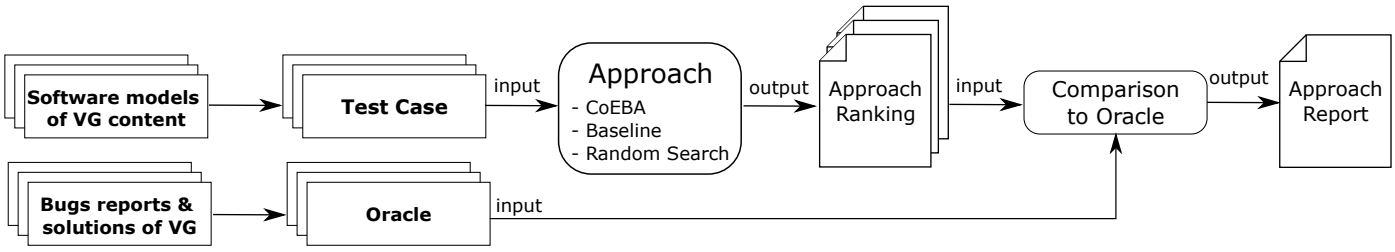

Figure 8: Overview of the evaluation process.

4.1. Evaluation preparation

The Kromaia developers provide the software models, the bug reports, the bug solutions, and the video game documentation. We employ all this information to build the oracle used in this evaluation. The oracle is the ground truth and is used to compare the results provided by the CoEBA approach, the baseline, and a random search.

The baseline is a previously developed approach for bug localization in GSE [6]. It uses an evolutionary algorithm for generating simulations that produce relevant traces for locating bugs. We use a standard random search (RS) as a sanity check. If RS outperforms an intelligent search method, we can conclude that there is no need to use a metaheuristic search. The algorithm starts with a random population of simulated players and a random population of scenarios. The simulated players are assessed using the CoEBA fitness function to obtain the best simulated player. Then, new random populations are generated and assessed. The search moves to a new simulated player if the fitness value is better than the current best simulated player. The loop is repeated until the stop condition is met.

To prepare the oracle, a total of 30 bugs were selected from the entire documentation. These 30 bugs cover the 6 different level bosses and have been provided by the Kromaia developers. These bugs are the most common when developers create the models.

We created the test case needed as input by the approaches for each bug. Each test case included the set of product models where the bug was manifested.

4.2. Experimental process

Figure 8 shows an overview of the process followed to evaluate our approach. The inputs to this process are used to run the different approaches, generating a set of results for each bug.

However, since these approaches perform genetic operations, chance could affect the results. Therefore, each approach is executed 30 times for each bug in order to minimize the effect of chance, as recommended in [17]. We employed the same parameters for the baseline approach as reported in [6]. For CoEBA, we began with the parameters reported in [11], since we utilized the same simulation algorithm and verified that they converged.

Once the approaches are executed, the next step is to compare them with each other and with a RS approach using statistical methods. To do this, the best solutions for each bug are selected from the ranking generated by each approach [18]. Then, these solutions are compared to the oracle (the actual solution) to generate the final report.

The approaches generate a list of traces. The trace contains all the model elements used by the interpreter at runtime during the simulation. All model elements that appear in the trace form the most relevant model fragment for the bug according to the trace. We can then compare the model fragments with an oracle to check accuracy.

Overall, the process involves running different approaches multiple times for each bug, selecting the best solutions, and comparing them with the actual solution to generate a report. This process is designed to minimize the effect of chance and provide a statistically rigorous evaluation of the different approaches.

4.3. Algorithm performance

A confusion matrix is a table that is commonly used to evaluate the performance of a classification model (our approach under evaluation) on a set of test data (the solution) where the truth values are known (derived from the oracle). In our case, each solution is a trace that consists of a subset of model elements that are part of a model (where the bug is present). Since the classification granularity is at the level of model elements, the presence or absence of each element is considered a classification. Thus, our confusion matrices distinguish between two values: TRUE (presence) and FALSE (absence). The confusion matrix then arranges the comparison results into four categories:

- True positive (TP): an element present in the predicted solutions that is also present in the actual solution.
- True Negative (TN): an element not present in the predicted solution that is not present in the actual solution.
- False Positive (FP): an element present in the predicted solution that is not present in the actual solution.
- False Negative (FN): an element not present in the predicted solution that is present in the actual solution.

The confusion matrix holds the results of the comparison between the predicted and actual results; it is just a specific table layout to help the visualization of the performance of a classifier. However, to evaluate the performance of the approach, it is necessary to derive some measurements from the values of the confusion matrix. In this case, the three measurements are precision, recall, and F-measure.

- Precision measures the number of elements from the prediction (the result of the approach) that are correct according to the ground truth (the oracle).

$$Precision = \frac{TP}{TP + FP} \quad (1)$$

- Recall measures the number of elements of the ground truth (the oracle) that are correctly retrieved by the prediction (the result of the approach).

$$Recall = \frac{TP}{TP + FN} \quad (2)$$

- F-measure combines both recall and precision as the harmonic mean of precision and recall.

$$F - measure = 2 \times \frac{Precision \times Recall}{Precision + Recall} \quad (3)$$

Precision and recall values can range between 0% to 100%. A value of 100% in precision and 100% in recall implies that both the predicted solution and the solution from the oracle are the same.

4.4. Implementation details

We conducted some parameter optimization to determine the optimal values for our algorithm's parameters. Nevertheless, this paper is not meant to optimize the algorithm's performance on a specific problem, but instead to compare the quality of different solutions provided by the algorithms. Hence, most of the default values suffice to evaluate the performance of search-based techniques within the testing context [19, 20]. We apply the crossover operation with a probability of 0.9 and the mutation operation with a probability that depends on the number of hulls in the boss during simulated player evolution or on the number of environmental elements during scenario evolution: 1/(Number). The population size is set to 100.

In addition, there are two fundamental performance metrics for evolutionary algorithms: one measuring solution quality and the other measuring algorithm speed. This study is focused on solution quality, which is measured through precision, recall, and F-measure (Section 4.3). Therefore, a budget has been assigned for each execution of the approach to ensure the quality of the solution. The focus of this paper is the solution quality and not the performance of each algorithm in terms of time. After conducting preliminary tests, we have determined that the time of convergence, or the point at which the search reaches its highest point and ceases to improve, occurs after around 40 seconds of execution. To ensure convergence, we set a wall clock time of 60 seconds for each run.

The replication package including models, bugs, oracles, and source code, is available at https://doi.org/10.5281/zenodo.10231570.

4.5. Statistical analysis

To compare our approaches accurately, we analyzed all the data derived from the empirical analysis using statistical methods in accordance with the guidelines provided in [17]. Our statistical analysis aims to: 1) provide formal and quantitative evidence (statistical significance) that our approaches do, in fact, have an impact on the comparison metrics (i.e., that the variations in the results were not achieved by mere chance); and 2) show that those differences are significant in practice (effect size).

4.5.1. Statistical significance

Hypotheses are used in statistical analysis to make inferences and draw conclusions about a population based on sample data. In the context of comparing algorithms, hypothesis testing helps determine whether there is sufficient evidence to support the idea that there is a difference between the algorithms being compared.

The null hypothesis, denoted as H_0, represents the default assumption that there is no significant difference between the algorithms. It assumes that any observed differences in the data are due to random variation or chance. In other words, H_0 states that there is no effect or relationship between the algorithms being compared.

The alternative hypothesis, denoted as H_1, is the opposite of the null hypothesis. It suggests that there is a significant difference between at least one pair of algorithms being compared. H_1 states that the observed differences in the data are not merely due to chance but are instead a result of a true difference or effect.

When conducting hypothesis testing, the goal is to gather enough evidence from the data to either reject the null hypothesis in favor of the alternative hypothesis or fail to reject the null hypothesis. This is done by calculating a test statistic and comparing it to a critical value or using a $p - value$ to determine the statistical significance of the results. If the evidence is strong enough to reject the null hypothesis, it provides support for the alternative hypothesis and indicates that there is a significant difference between the algorithms.

The $p - values$ obtained from the test is used to make this determination, and a $p - values$ under 0.05 is considered statistically significant [17]. Non-parametric techniques like the Quade test are suitable for analyzing data that does not follow a normal distribution, and it is more powerful than other tests for real data [21] with a low number of algorithms (no more than four or five algorithms) [22].

However, the Quade test cannot determine which algorithm performs the best. To perform this comparison, we need to conduct a post hoc analysis that compares the performance of each algorithm against all other alternatives through pair-wise comparisons.

4.5.2. Effect size

When comparing different algorithms, more is needed to determine solely whether there are statistically significant differences between them. It is equally important to assess the practical significance or magnitude of any improvements observed.

This can be achieved through the use of effect size measures. One commonly used is the *Cohen′s d* value [23]. The *Cohen′s d* value ranges from 0 to infinity, with higher values indicating a more substantial effect size. A value of 0.8 and above represents a high interaction effect, while a value between 0.5 and 0.8 signifies a medium interaction effect, and a value between 0.2 and 0.5 represents a small interaction effect. Values ranging from 0 to 0.2 indicate almost no interaction effect.

5. Results

This section presents the results obtained by the three approaches: 1) our approach, CoEBA; 2) the baseline approach; and 3) the RS approach; in evaluating bugs in Kromaia. The results are presented in Table 1, which shows the mean values and standard deviations for each measure obtained by each approach. The measures used to compare them are recall, precision, and F-measure.

Table 1: Mean values and standard deviation of the Precision, Recall, and F-measure values obtained by each approach.

	Precision ± (σ)	Recall ± (σ)	F-measure ± (σ)
CoEBA	45.76 ± 28.70	84.83 ± 33.92	53.87 ± 24.71
Baseline	36.34 ± 28.47	68.95 ± 42.87	44.46 ± 29.03
Random Search	02.14 ± 10.06	08.07 ± 25.34	02.64 ± 09.64

The CoEBA and baseline approaches obtained better results than the RS approaches. Furthermore, the CoEBA approach outperformed the baseline approach and obtained the best results in all three measures (precision, recall, and F-measure) for the first research question (RQ_1). Specifically, the CoEBA approach achieved an average value of 45.76% in precision, 84.83% in recall, and 53.87% in F-measure.

The second research question (RQ_2) asks whether the interaction between models contributes to the improvement in bug location. The results show that CoEBA achieved an improvement of 9.42% in precision, 15.88% in recall, and 9.41% in F-measure compared to the baseline approach, which did not take into account the interaction between models. Therefore, answering to RQ_2, the results indicate that take into account the interaction of software models pays off for bug localization in GSE.

5.1. Stadistical analysis

The $p - value$ obtained in the Quade test is $< 2.2x10^{-16}$ for recall, precision, and F-measure (results are significantly different).

In the post hoc analysis, we perform statistical tests to determine if there are significant differences in the results between specific pairs of algorithms. Table 2 presents the results of Holm's post hoc analysis, which includes the $p - value$ and performance indicators for the algorithms. Most $p - values$ are below the significance threshold of 0.05, indicating significant differences in performance between the algorithms. We can see that precision and recall values are significant.

Table 2: Holm's post hoc $p - values$ for each pair of algorithms.

	Precision	Recall	F-measure
CoEBA vs Baseline	0.0088	$< 2.2x10^{-16}$	0.57
CoEBA vs RS	$< 2.2x10^{-16}$	$< 2.2x10^{-16}$	$< 2.2x10^{-16}$
Baseline vs RS	$< 2.2x10^{-16}$	$< 2.2x10^{-16}$	$< 2.2x10^{-16}$

Table 3: Cohen's d values of the Precision, Recall, and F-measure values obtained by each approach.

	Precision	Recall	F-measure
CoEBA vs Baseline	0.329345	0.410637	0.349083
CoEBA vs RS	2.028632	2.563807	2.731206
Baseline vs RS	1.602291	1.728865	1.933129

Furthermore, Table 3 presents the *Cohen′s d* values for recall, precision, and F-measure obtained by each approach. Regarding the *Cohen′s d* values for the CoEBA and the baseline approaches, the results show a small effect size in precision but highlight a more significant effect on recall.

Table 4: Classification of the precision and recall results. Obtained from [24].

Measure	Acceptable	Good	Excellent
Precision	20% – 29%	30% – 49%	> 50%
Recall	60% – 69%	70% – 79%	> 80%

Although the effect size according to *Cohen′sd* is small, we must take into account Hayes et al. classification [24]. Their work classifies performance for precision and recall as *Acceptable*, *Good*, or *Excellent*, as shown in Table 4. We started from a baseline approach with *Good* results in precision and through CoEBA approach we also obtained *Good* results in precision, but bordering on *Excellent*. On the other hand, the baseline approach obtained *Acceptable* results in recall while CoEBA approach achieved *Excellent* results.

The statistical analysis performed allows us to answer the research question (RQ_3) that asks if the performance results obtained by the CoEBA, the baseline, and the random search approaches are significant. Although the differences in F-measure values are not significant when we compare the CoEBA and the baseline approaches, the differences in precision and recall values are significant.

6. Discussion

The video game industry has seen a significant increase in popularity, but video game research has not received enough attention yet. Recent research by Politowski et al. [9] has highlighted the differences in testing between classical software engineering (CSE) and game software engineering (GSE) and the need for further work in automating playtesting. They conclude that video game testing is still a long way to go. Our work contributes to this effort by addressing the crucial role of software

model interaction in video game testing. The behavior of video games as a whole also includes the interaction between video game software models, and it is essential to explore this area to improve game testing. By investigating the co-evolution of simulated players and scenarios in bug location, our work helps to take another step towards improving video game testing and ultimately enhancing the quality of video games.

The evaluation presented in this work provides a detailed analysis of the values for three performance metrics obtained by applying our approach to locate the bugs of a commercial video game. The study conducted by Hayes et al. in [24] classified the results obtained into three categories: acceptable, good, and excellent. According to the Hayes et al. criteria, the CoEBA approach obtains good results for precision and excellent results for recall. It is worth mentioning that the precision and recall values obtained by CoEBA are significantly higher than the values obtained by the baseline. These suggest that CoEBA is a promising approach for bug localization in video game software models. It is important to note that this study represents only a first step in exploring the potential of CoEBA. Future research could investigate its application to more complex video game scenarios that involve additional tasks, such as finding objects or completing missions.

The recent work of Politowski et al. [9] includes a survey for video game developers. The main concerns highlighted in the paper relate to the high cost of building testing tools and the need for general testing tools that can work with a wide range of game types and game engines. Our approach does not share the concerns mentioned earlier. Our approach does not require developing simulation algorithms from scratch. Non-Playable Characters (NPCs) are commonly developed in video games as part of the regular development process. These NPCs serve different purposes, such as being enemies, cooperating with players, or inhabiting the virtual world to increase realism. NPCs can be utilized as the raw material for simulated players and scenarios, reducing the effort required to apply CoEBA. In addition, our approach locates bugs that arise from the interaction of software models and, hence, can be used in games that use software models. However, many game engines use models to capture different aspects of the game.

Additionally, some respondents expressed concerns about the potential for AI to replace game testers. When game developers perform playtesting, they have to test the complete video game to check several things, such as if it is fun or has problems like bugs or glitches. The solutions proposed by our approach are ordered by how far they are from what the developers consider to be the ideal experience (and, in their opinion, the most fun). However, our approach is intended to locate bugs rather than determine if the game is fun. Our approach can be an ally so that game testers can focus on tasks that are difficult for a machine to perform. This alliance could save time in video game developments avoiding delays in the deadlines and postponement of the launch date.

Finally, the approach we have developed considers the environmental elements present in the scenario and uses this information to apply a probability of damage dependent on the type of element. However, this is just the beginning of the possibilities that this approach presents. For example, future work involves exploring the interactions between more complex environmental elements such as mini-bosses. Additionally, this work serves as a starting point for other types of simulations where the objective is not only to defeat the boss but also to complete missions or find specific objects within the game environment. The potential for further exploration and expansion of this approach is vast, and it presents exciting opportunities for future research in the field of video game development and bug localization.

7. Threats to validity

In the following section, we will discuss potential threats to the validity of our work. To identify these threats, we will refer to the guidelines provided by De Oliveira et al. [25].

Conclusion validity threats.

- *Not accounting for random variation*: We addressed this threat by performing 30 independent runs for each algorithm on each bug.
- *Not using formal hypothesis and statistical tests*: We followed established guidelines [19] to employ standard statistical analysis and avoid this threat. To address the lack of good descriptive analysis, we utilized a variety of metrics, including precision, recall, and F-measure, to analyze the confusion matrix obtained from the experiments.

Internal validity threats.

- *Poor parameter settings threat*: We used standard values for the algorithms that have been tested in similar algorithms for bug location in video games [11, 6].
- *Lack of real problem instances*: For the evaluation, we utilized a commercial video game and obtained the necessary problem artifacts directly from the game developers and documentation. We selected the same bugs as the baseline algorithm to ensure a fair comparison.

Construct validity threats.

- *Lack of assessing the validity of cost measures*: To ensure the validity of our approach, we conducted a rigorous comparison with both the baseline and the sanity check. Additionally, our evaluation employed established metrics, such as precision, recall, and F-measure, which have been widely used in previous software engineering studies [19].
- *Lack of clarity on data collection*: To mitigate this potential threat, we utilized the SDML models of the contenders to conduct the simulation and relied on two primary indicators, the victory percentage, and health level, which were obtained from the developers and utilized to evaluate the approaches.

External validity threats.

- *Lack of a clear object selection strategy*: We tested the effectiveness of our approach in a real-world context by evaluating it on a commercial video game. To ensure the relevance of our results, we collected instances directly from the game's development team and documentation.

8. Related Work

This section summarizes the research efforts from two areas that are related to this work: bug localization in MDE and bug localization in video games using simulations.

8.1. Bug localization in MDE

Several techniques are employed for bug localization in models beyond the video game domain. In software development, models can play various roles and provide domain information useful in bug localization. Hence, bugs in models cannot be ignored in MDE and our work focuses on this area. In [26], the authors use models at runtime to locate bugs resulting from dynamic reconfigurations. They employ search-based software engineering to generate a list of reconfiguration sequences that could potentially lead to the model at runtime containing the bug. In [27], their approach considers the domain information embedded in models and metamodels, which is not present in the source code. Additionally, Arcega et al. [28] evaluate the application of existing model-based bug localization approaches to mitigate the effects of starting localization in the wrong place. They also consider software engineers' ability to refine the results at different stages. By combining their approaches with manual refinement, they obtain the best results. Finally, Khorram et al. [29] propose a generic framework for coverage computation and fault localization of domain-specific models. They consider a test suite for an executable model and analyze the model's execution traces to extract its covered elements which compose the coverage matrix for the test suite. The applied fault localization techniques are capable of identifying the defects injected in the models based on coverage measurements.

Despite the significance of the aforementioned works in bug localization within MDE, none of them are tailored toward video game development. These approaches were not created considering the specific characteristics of video games, nor have they ever undergone evaluations in the context of video games. Moreover, as far as we know, no research has been conducted on the impact of model interactions as a source of bugs in video game development.

8.2. Bug localization in video games using simulations

Testing video games is commonly referred to as playtesting since it requires testers to play the game to analyze it. However, relying solely on human playtesters has limitations, and there is a need for new approaches to automate game testing. Research efforts that introduce alternative methods, such as simulations, are becoming more prevalent to address these challenges [9].

Several recent research works have employed agents to automate video game testing. For instance, Albaghajati and Ahmed [30] proposed a co-evolutionary genetic algorithm-based approach that uses agents to test and verify video games. The approach involves two agents, one of which generates faulty game states while the other uses colored Petri nets to analyze the game states and detect errors. Ariyurek et al. [31, 32] explored the impact of different Monte Carlo Tree Search (MCTS) modifications on the ability of agents to find bugs in games while operating within specific computational budgets. They introduced various changes to the MCTS algorithm and demonstrated that such modifications could enhance the agents' capability to find bugs. Furthermore, they also evaluated the human-like qualities of the agents, taking into account the computational resources available. Pfau et al. [33] employ agents to play adventure games as fast as possible (similar to a speedrun). Their approach can be utilized to execute various Visionaire Engine-based games and for assessing daily builds, hardware compatibility, performance tests, and quality assurance playthroughs in a generic manner. Shirzadehhajimahmood et al. [34] introduced an agent-based testing framework with a search algorithm and on-the-fly model construction. Their approach enables the agent to navigate through the 3D game world and overcome obstacles while solving the testing tasks. They also employ an Extended Finite State Machine (EFSM) model to capture the general properties of the game, which assists in the search process.

The approaches mentioned above also employ player simulation to test games and locate bugs, but they do not consider software models as a potential source of bugs. Since models are more and more used in video game development [5], ignoring them as a possible cause of bugs is not advisable.

Casamayor et al. [6] developed an evolutionary algorithm that uses model simulations to locate bugs in software models. They evolve game simulations that generate traces to identify the location of bugs. However, our approach differs from theirs as our work considers that the interactions of software models that occur during simulations can cause bugs to appear.

9. Conclusion

Bug localization in video games that use Model-Driven Engineering (MDE) requires specialized approaches due to the unique characteristics of the game software. The interaction between software models during gameplay can be a significant source of bugs, particularly in games that provide non-linear experiences with various possibilities for player behavior and scenarios.

To address this issue, this work aimed to evaluate how the interaction between software models affects bug localization in video games. The paper compared two approaches: CoEBA, which co-evolves simulated players (player behavior) and scenarios (environmental elements), and the baseline approach which does evolve simulations but does not take into account model interaction. The evaluation was performed on Kromaia case studies.

The results of this paper showed that CoEBA provided significantly better bug localization results than the baseline approach. The results of this paper show that model interaction

through co-evolving simulated players and scenarios could be a promising direction for future research in the field of bug location for video games.

Acknowledgements

This work was supported in part by the Spanish Ministry of Science and Innovation under the Excellence Network AI4Software (Red2022-134647-T), in part by the Ministry of Economy and Competitiveness (MINECO) through the Spanish National R+D+i Plan and ERDF funds under the Project VARIATIVA under Grant PID2021- 28695OBI00, and in part by the Gobierno de Aragón (Spain) (Research Group S05_20D).

References

[1] A. Ampatzoglou, I. Stamelos, Software engineering research for computer games: A systematic review, Information and Software Technology 52 (9) (2010) 888–901. doi:https://doi.org/10.1016/j.infsof.2010.05.004.

[2] U. Technologies, Unity real-time development platform — 3d, 2d vr & ar engine, `https://unity.com`, [Online; accessed 21-November-2021] (2005).

[3] E. Games, Unreal engine: The most powerful real-time 3d creation tool, `https://www.unrealengine.com`, [Online; accessed 21-November-2021] (1998).

[4] Crytek, Cryengine — the complete solution for next generation game development by crytek, `https://www.cryengine.com`, [Online; accessed 21-November-2021] (2002).

[5] M. Zhu, A. I. Wang, Model-driven game development: A literature review, ACM Comput. Surv. 52 (6) (nov 2019). doi:10.1145/3365000.

[6] R. Casamayor, L. Arcega, F. Pérez, C. Cetina, Bug localization in game software engineering: Evolving simulations to locate bugs in software models of video games, in: Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems, MODELS '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 356–366. doi:10.1145/3550355.3552440.

[7] W. E. Wong, R. Gao, Y. Li, R. Abreu, F. Wotawa, A survey on software fault localization, IEEE Trans. Software Eng. 42 (8) (2016) 707–740. doi:10.1109/TSE.2016.2521368.

[8] L. Pascarella, F. Palomba, M. Di Penta, A. Bacchelli, How is video game development different from software development in open source, 2018. doi:10.1145/3196398.3196423.

[9] C. Politowski, Y.-G. Guéhéneuc, F. Petrillo, Towards automated video game testing: Still a long way to go, in: Proceedings of the 6th International ICSE Workshop on Games and Software Engineering: Engineering Fun, Inspiration, and Motivation, GAS '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 37–43. doi:10.1145/3524494.3527627.

[10] M. Harman, B. F. Jones, Search-based software engineering, Information and software Technology 43 (14) (2001) 833–839.

[11] D. Blasco, J. Font, M. Zamorano, C. Cetina, An evolutionary approach for generating software models: The case of kromaia in game software engineering, J. Syst. Softw. 171 (2021) 110804. doi:10.1016/j.jss.2020.110804.

[12] L. Miguel Antonio, C. A. Coello Coello, Coevolutionary multiobjective evolutionary algorithms: Survey of the state-of-the-art, IEEE Transactions on Evolutionary Computation 22 (6) (2018) 851–865. doi:10.1109/TEVC.2017.2767023.

[13] M. Boussaa, W. Kessentini, M. Kessentini, S. Bechikh, S. Ben Chikha, Competitive coevolutionary code-smells detection, in: G. Ruhe, Y. Zhang (Eds.), Search Based Software Engineering, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 50–65.

[14] A. B. Cardona, J. Togelius, M. J. Nelson, Competitive coevolution in ms. pac-man, in: 2013 IEEE Congress on Evolutionary Computation, 2013, pp. 1403–1410. doi:10.1109/CEC.2013.6557728.

[15] E. Z. Elfeky, S. Elsayed, L. Marsh, D. Essam, M. Cochrane, B. Sims, R. Sarker, A systematic review of coevolution in real-time strategy games, IEEE Access 9 (2021) 136647–136665. doi:10.1109/ACCESS.2021.3115768.

[16] M. Affenzeller, S. M. Winkler, S. Wagner, A. Beham, Genetic Algorithms and Genetic Programming - Modern Concepts and Practical Applications, CRC Press, United Kingdom, 2009.

[17] A. Arcuri, L. Briand, A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering, Software Testing, Verification and Reliability 24 (3) (2014) 219–250. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1486, doi:https://doi.org/10.1002/stvr.1486.

[18] H. Ishibuchi, Y. Nojima, Tsutomu Doi, Comparison between single-objective and multi-objective genetic algorithms: Performance comparison and performance measures, in: 2006 IEEE International Conference on Evolutionary Computation, 2006, pp. 1143–1150. doi:10.1109/CEC.2006.1688438.

[19] A. Arcuri, G. Fraser, Parameter tuning or default values? an empirical investigation in search-based software engineering, Empirical Software Engineering 18 (2013) 594–623.

[20] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, A. De Lucia, Parameterizing and assembling ir-based solutions for se tasks using genetic algorithms, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Vol. 1, 2016, pp. 314–325. doi:10.1109/SANER.2016.97.

[21] S. García, A. Fernández, J. Luengo, F. Herrera, Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: Experimental analysis of power, Information Sciences 180 (10) (2010) 2044 – 2064, special Issue on Intelligent Distributed Information Systems. doi:http://dx.doi.org/10.1016/j.ins.2009.12.010.

[22] W. J. Conover, Practical Nonparametric Statistics, 3rd Edition, Wiley, USA, 1999.

[23] J. Cohen, Statistical power analysis, Current directions in psychological science 1 (3) (1992) 98–101.

[24] J. H. Hayes, A. Dekhtyar, S. K. Sundaram, Advancing candidate link generation for requirements tracing: The study of methods 32 (1) (2006). doi:10.1109/TSE.2006.3.

[25] M. de Oliveira Barros, A. C. D. Neto, Threats to validity in search-based software engineering empirical studies, Tech. Rep. 0006/2011 (2011).

[26] L. Arcega, J. Font, C. Cetina, Evolutionary algorithm for bug localization in the reconfigurations of models at runtime, in: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 90–100. doi:10.1145/3239372.3239392.

[27] L. Arcega, J. Font, Ø. Haugen, C. Cetina, An approach for bug localization in models using two levels: model and metamodel, Software and Systems Modeling 18 (6) (2019) 3551–3576. doi:10.1007/s10270-019-00727-y.

[28] L. Arcega, J. F. Arcega, O. Haugen, C. Cetina, Bug localization in model-based systems in the wild, ACM Trans. Softw. Eng. Methodol. 31 (1) (oct 2021). doi:10.1145/3472616.

[29] F. Khorram, E. Bousse, A. Garmendia, J.-M. Mottu, G. Sunyé, M. Wimmer, From coverage computation to fault localization: A generic framework for domain-specific languages, in: Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2022, Association for Computing Machinery, New York, NY, USA, 2022, p. 235–248. doi:10.1145/3567512.3567532.

[30] A. Albaghajati, M. Ahmed, A co-evolutionary genetic algorithms approach to detect video game bugs, Journal of Systems and Software 188 (2022) 111261. doi:https://doi.org/10.1016/j.jss.2022.111261.

[31] S. Ariyurek, A. Betin-Can, E. Sürer, Enhancing the monte carlo tree search algorithm for video game testing, in: IEEE Conference on Games, CoG 2020, Osaka, Japan, August 24-27, 2020, IEEE, 2020, pp. 25–32. doi:10.1109/CoG47356.2020.9231670.

[32] S. Ariyurek, A. Betin-Can, E. Surer, Automated video game testing using synthetic and humanlike agents, IEEE Transactions on Games 13 (1) (2021) 50–67. doi:10.1109/TG.2019.2947597.

[33] J. Pfau, J. D. Smeddinck, R. Malaka, Automated game testing with icarus: Intelligent completion of adventure riddles via unsupervised solving, CHI

PLAY '17 Extended Abstracts, Association for Computing Machinery, New York, NY, USA, 2017. doi:10.1145/3130859.3131439.

[34] S. Shirzadehhajimahmood, I. S. W. B. Prasetya, F. Dignum, M. Dastani, An online agent-based search approach in automated computer game testing with model construction, in: Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation, A-TEST 2022, Association for Computing Machinery, New York, NY, USA, 2022, p. 45–52. doi:10.1145/3548659.3561309.