Boosting Bug Localization in Software Models of Video Games with Simulations and Component-Specific Genetic Operations

Rodrigo Casamayor[1,2*], Lorena Arcega[1], Francisca Pérez[2] and Carlos Cetina[2]

[1]SVIT Research Group, Universidad San Jorge, Zaragoza, Spain.
[2]PROS Research Center, VRAIN, Universitat Politècnica de València, Valencia, Spain.

*Corresponding author(s). E-mail(s): rcasamayor@usj.es;
Contributing authors: larcega@usj.es; franciscaperez@upv.es; cetina@upv.es;

Abstract

The development and maintenance of video games present unique challenges that differentiate them from Classic Software Engineering (CSE) such as the increased difficulty in locating bugs within video games. This distinction has given rise to Game Software Engineering (GSE), a subfield that intersects software engineering and video games. Our work proposes a novel way for bug localization in video games by evolving simulations via an evolutionary algorithm, which helps to explore the large number of possible simulations. Simulations generate data (i.e., traces) from the behavior of non-player characters (NPCs). NPCs are not controlled by the player and are key components of video games. We hypothesize that such traces can be instrumental in locating bugs. Our approach automatically locates potential buggy model elements from traces. Furthermore, we propose a novel way of applying genetic operations to evolve simulations by selectively combining their components, rather than combining all components as a whole. We evaluate our approach in the commercial video game Kromaia, and the results indicate that evolving simulations using our novel component-specific genetic operations boosts bug localization. Specifically, our approach improved the F-measure for all bug categories over randomly combining all components, the baseline (which focuses on CSE and utilizes bug reports), and Random Search by 7.93%, 27.17%, and 46.34%, respectively. This work opens a new research direction for further exploration in bug localization within GSE and potentially in CSE as well. Moreover, it encourages other researchers to explore alternative genetic operations rather than selecting them by default.

Keywords: Bug Localization, Video Games, Search-Based Software Engineering, Model-Driven Engineering

1 Introduction

Today, video game development is one of the fastest growing industries in the world and in terms of developer population, this industry is responsible for 8.8M active developers as of 2019 [1]. According to the same report, the total number of active software developers is 18.9M, so almost one out of every two developers is involved in the games sector. Furthermore, video game development is instrumental in achieving the vision of the Metaverse [2]. This might suggest that the number of video game developers will continue to grow in the future as the Metaverse is developed [3].

Video games present characteristics that differentiate their development and maintenance from

the development and maintenance of classic software; for example, how developers contribute to video games vs. non-games by working on different kinds of artifacts such as shaders (which determine the appearance of objects in a 3D environment), meshes (a collection of vertices, edges, and faces that define the shape and structure of a 3D object), or prefabs (pre-designed and reusable objects that encapsulate a combination of meshes, materials, scripts, and other components that define a particular game object). In addition, game developers perceive more difficulties than other non-game developers when locating bugs as well as reusing code [4].

Nowadays, most video games are developed by means of so-called *game engines*. A *game engine* refers to a development environment that integrates a graphics engine and a physics engine as well as a set of tools that wraps around them in order to accelerate development. The most popular ones are Unity [5] and Unreal Engine [6], but it is also possible for a studio to make its own specific engine (e.g., CryEngine [7]).

A key artifact of *game engines* are software models. Developers can create video game content directly using code (e.g., C++) or the software models of the engines. On the one hand, the code allows developers to have more control over the content. On the other hand, software models are much less bound to the underlying implementation and technology and raise the abstraction level using terms that are much closer to the problem domain. This means that developers are liberated from a significant part of the implementation details of physics and graphics and can focus on the content of the game itself (see Fig. 1). Unity and Unreal propose their own modeling language. This includes a combination of scripting languages as well as component/object-based structures that facilitate the creation and manipulation of game elements. In Unity, developers often use C# scripts to define the behavior of game objects, while in Unreal Engine, a combination of C++ code and Blueprints (a visual scripting system) is commonly employed. A recent survey in Model-Driven Game Development [8] reveals that UML and Domain Specific Language (DSL) models are also adopted by development teams.

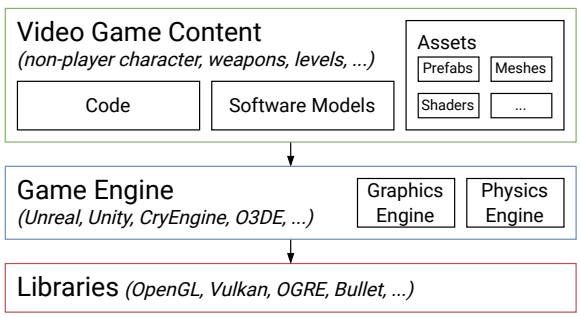


Fig. 1 Overview of video game artifacts.

Current approaches for locating bugs in software focus on source code, while the approaches needed to locate bugs in video games should consider other artifacts such as software models [4]. Nevertheless, a fault localization survey [9] reveals that none of the bug localization approaches consider models as the source of the bugs, which poses a considerable problem for developers since much of the video game content remains unexplored. Actually, Politowski et al. argue that the way in which developers deal with bugs must inevitably be different in video games than in traditional software since the artifacts used are also different [10].

The lack of specific bug localization approaches leads to a longer development time, which sometimes causes delays in the deadlines and postponement of the launch date. This results in the video game being released with an excessive number of bugs, such as the case of the blockbuster *Cyberpunk 2077* [11]. After nine years of development, *Cyberpunk 2077* was released with so many bugs that it was withdrawn from the stores, and, a year after its release, patches are still being released to fix its bugs. These bugs included NPCs floating in mid-air, characters glitching through solid objects, and quest malfunctions preventing mission progression [12].

There are significant differences between Classic Software Engineering and Game Software Engineering [10, 13]. Our work suggests these differences can be advantageous for bug localization in video games. Unlike traditional software, video games commonly include non-player characters (NPCs) that enhance gameplay by interacting with the player and the game environment. NPCs, such as enemies, allies, and bystanders, are programmed with specific behaviors and roles that are not controlled by the player. These NPCs can

be used to run gameplay simulations, which generate traces of their behavior. The traces can serve to detect outliers, which can likely indicate the location of a bug. Hence, we propose leveraging game simulations to locate bugs in the software models of video games.

In our recent work [14], we proposed an approach, called EMoSim, that automatically locates a set of potential buggy model elements from a simulation, which is automatically generated using an evolutionary algorithm. In each iteration of the algorithm, new simulations are automatically created (i.e., evolved) by means of genetic operations, which combine all of the components of two simulations that are taken as parents. As a result, the evolved simulations produce traces that are relevant for locating five different types of bugs (e.g., Weak Point is Hidden, which is a type of software model bug).

EMoSim was compared with a baseline, BLiMEA [15, 16], which is specifically designed to locate bugs in software models and uses bug reports and the defect localization principle [17] (as many bug localization approaches do [9]). The "defect localization principle" states that the most recent modifications to a project are the most relevant for certain Information Retrieval purposes [17–19]. In the context of bug localization, the primary source or root cause of a bug or defect in a software system typically resides within the most recently modified file. This serves as a guiding heuristic for identifying and prioritizing areas for investigation or debugging within the software development process. Also, Random Search was used as a sanity check in the evaluation with the Kromaia case study. Kromaia is a video game about a spaceship flying and shooting in a three-dimensional space [1]. It was released on PC, PlayStation, and translated to eight different languages. Within the Kromaia context, an NPC can be a simple enemy, the final boss of a level, or a totem (level element with a pre-programmed movement), among others. The results showed that EMoSiM significantly outperformed the baseline and Random Search, and a focus group confirmed its acceptance.

[1]See the official Playstation trailer to learn more about Kromaia: https://youtu.be/EhsejJBp8Go

In this paper, we extend our previous work, and the main contributions can be summarized as follows:

1. We classify the five types of bugs that were located in the previous work in four categories that were identified in a recent taxonomy of bugs in video games [20]. Thus, we report the results using the third party classification of the taxonomy in order to facilitate future comparisons of our results with other works that use the same classification.
2. We extend the bug types to cover 50% more categories of the existing taxonomy. Our previous work covered four categories of the taxonomy (30 bugs), whereas this work covers six categories (54 bugs).
3. We propose novel component-specific genetic operations that evolve simulations considering the different components of a simulation (the simulated player, the NPC and the itinerary) instead of evolving simulations using the classic genetic operations as in our previous work, which randomly combine all components of a simulation as a whole.
4. We implement our component-specific genetic operations, and we compare the results of the categories of bugs with the classic genetic operations, the baseline, and Random Search.
5. We present a more thorough and updated related work section that is focused on the topics covered in this paper: bug localization in games, bug localization in models, and bug localization in games that use models.

To report the performance in terms of solution quality, three metrics (recall, precision, and F-measure) are included since they are widely accepted by the software engineering community in the domain of evolutionary algorithms [14]. In addition, we perform a statistical analysis (following the guidelines by Arcuri and Briand [21]) in order to provide quantitative evidence of the impact of the results and to show whether this impact is significant.

The results show that EMoSiM with the component-specific genetic operations significantly outperforms EMoSiM with the classic genetic operations, the baseline, and Random Search in F-measure for all categories of bugs. The biggest improvements in recall and precision of EMoSiM with the component-specific genetic

operations are obtained when the recall is compared with EMoSiM with the classic genetic operations (improvement of 28.34%), and when precision is compared with Random Search (improvement of 35.59%). In F-measure, EMoSiM with the component-specific genetic operations improves the results of EMoSiM with the classic genetic operations by 7.93%, the baseline by 27.17%, and Random Search by 46.34%.

After analyzing the results, we claim the following:

- Our work suggests that the current ideas of leveraging bug reports and the defect localization principle may not be enough to locate bugs in video games.
- Our results show that the component-specific genetic operations improve the results of EMoSiM. This can motivate other researchers to consider other genetic operations instead of selecting the classic genetic operations by force of habit or default.
- Our results show that our idea of evolving simulations is a promising, novel way to locate bugs in video games.
- The discussion of our results helps advance the understanding of bugs in video games.
- Our approach benefits from the experience of NPCs in video games. However, agents similar to NPCs (e.g., simulation agents that represent humans for testing UI) can also be developed for apps to evaluate the potential benefits of evolving simulations to locate bugs in classic software engineering.

The remainder of the paper is structured as follows. In Section 2, we present the case study (Kromaia) and the types of bugs. In Section 3, we carry out the classification of the types of bugs in the categories of the taxonomy. In Section 4, we describe our approach, EMoSim. In Section 5, we evaluate our approach in Kromaia, and we present the results, which are discussed in Section 6. In Section 7, we review the threats to validity. In Section 8, we examine the related work on the topics covered in this paper. Finally, we conclude the paper in Section 9.

2 Background

The case study that we use to evaluate the work presented here is performed using the bosses of the video game Kromaia. The game in Kromaia takes place in a three-dimensional space. Fig. 2 shows the content for a gameplay of Kromaia. Each of the levels involves a player's spaceship (Fig. 2.H) flying from a starting point to a target destination in order to reach the goal before being destroyed. The level involves exploring floating structures, avoiding asteroids, and finding items along the route. Meanwhile, the boss (see the screenshot of Fig. 2.A), an enemy character that the player must defeat at the end of each level, is being protected by basic entities attempting to damage the player's spaceship with fired projectiles (Fig. 2.G). These entities, including enemy bosses, derive their structure from rigid bodies or solid objects known as hulls (Fig. 2.B). If the player manages to reach the destination, the final boss corresponding to that level appears and must be defeated in order to complete the level.

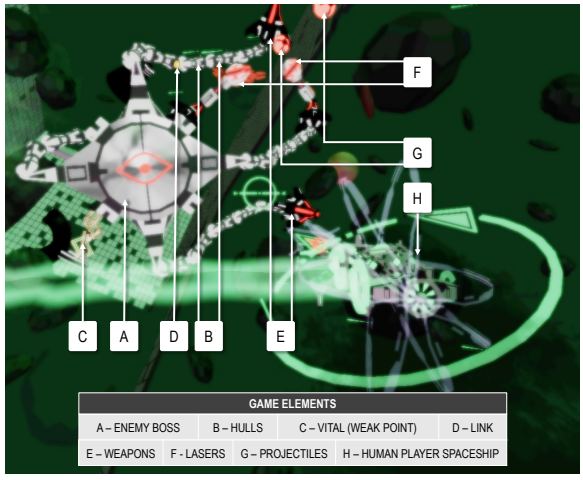


Fig. 2 Screenshot showing the game content in Kromaia.

The bosses are specified with the Shooter Definition Model Language (SDML). SDML is a custom DSL model for the video game domain of Kromaia. This DSL follows the main ideas of MDE using models for Software Engineering. The models are created using SDML and 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.

Specifically, SDML defines structural and behavioral aspects that are included in video game

entities: the anatomical structure (including which parts are used in it, their physical properties, and how they are connected to each other); the amount and distribution of vulnerable parts, weapons, and defenses in the structure/body of the character; and the movement behaviours associated to the whole body or its parts. This modeling language has concepts such as hulls, links, weak points, weapons, and AI components. At this point, it is important to highlight that this notion of *model* or *software model* should not be confused with *mesh* or *3D model*, which are terms used in video games and computer graphics for describing the visual representation of 3D shapes. Examples of the models of Kromaia, the metamodel, and an online visualizer to show the models as they would be seen in the Kromaia video game can be found at the following URL: https://svit.usj.es/models22/bl-in-mgse.

Fig. 3 shows a graphical representation of the SDML metamodel of a Kromaia boss. The SDML metamodel contains: more than 20 concepts, over 20 relationships, and more than 60 properties. A typical boss model contains around 1300 elements.

2.1 Video Game Simulations

Non-player characters (NPCs) are created through the development process of the video game. We propose to harness NPCs to run simulations that help to locate bugs. Our simulations reuse NPCs and the video game itself. Our simulations are not ad hoc developments.

Specifically, by leveraging game simulations, we want to locate the bugs that are related to the models that specify the bosses using SDML. Simulations use non-player characters (NPCs) to extract information of the game during runtime with the aim of uncovering bugs that arise from dynamic interactions during gameplay. This differs from classic test runs that check the well-formedness of the game models at design time with the aim of identifying bugs related to overall structure and initial configurations of the NPCs.

Concretely, the simulations used in this work simulate a duel between a boss and a human player. The simulated player is an algorithm that acts as an agent and stand-in for the human player, which is capable of mimicking human behavior. It was created by the developers of

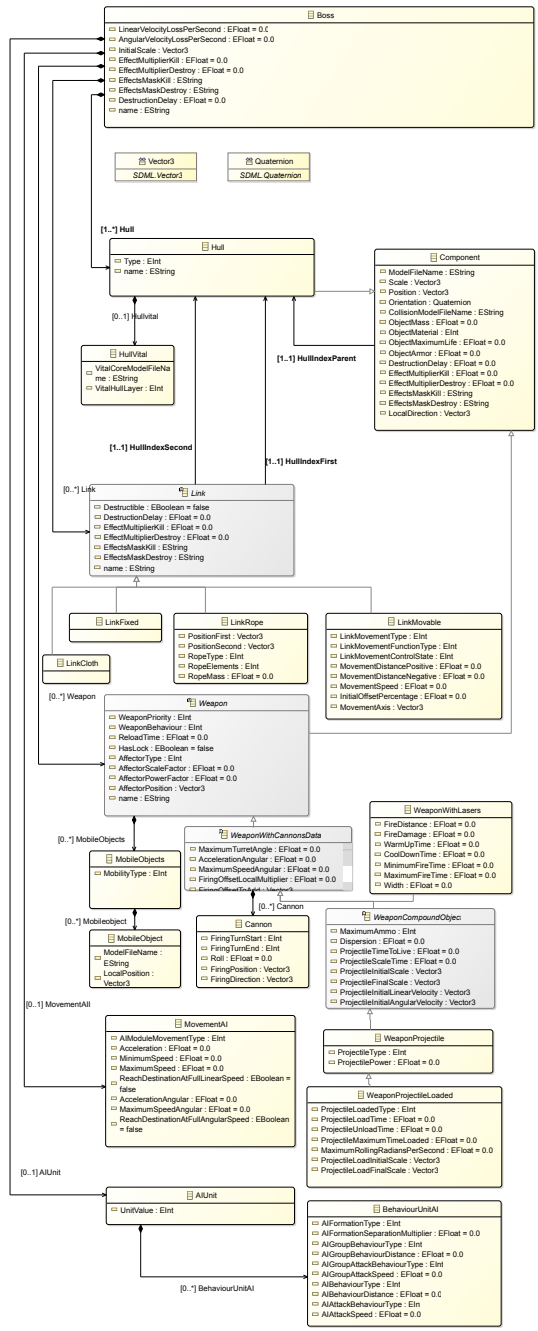


Fig. 3 SDML metamodel of a Kromaia boss.

the Kromaia video game. We used their algorithm for our approach. During the simulation, the simulated player faces the boss in order to destroy the weak points that are available at that moment, whereas the boss acts according to the anatomy, behaviour, and attack/defense balance that is included in its model, trying to defeat the

simulated player. In the simulation, both the boss and the simulated player try to win the match and do not avoid confrontation, try to prevent draw/tie games, and try to ensure that there is a winner. The algorithm can fight a boss by applying different strategies. Hence, the algorithm can be parametrized to define the fighting strategy. The simulation parameters were provided by the developers based on the analysis of battles between human players and bosses.

2.2 Bug Types

The developers that implemented the bosses provided us with the types of bugs that are the most common when creating the models. The most common bugs listed by the developers are the following:

- A boss is invincible because a Weak Point is Hidden (WPH). This bug occurs when a vital point in the boss is inaccessible or invisible. Vital points are vulnerable parts of the bosses. If they are inaccessible or invisible, the player cannot reach them; thus, the player will never be able to defeat the boss.
- A boss is invincible because a Weak Point is Overlapped (WPO). This bug occurs when solid objects overlap each other when they are not supposed to. This can trigger scenarios similar to those described in the previous bug.
- A boss has wrong behaviour because of Bad Link Indexes (BLI). This bug occurs when the links between the parts of a boss are incorrectly assigned. This causes the physics to become erratic; thus, the movement of the boss will not be as expected.
- A boss has wrong behaviour because a Hull is Not Linked (HNL). This bug occurs when the hulls are not attached to any other part of the boss. In this case, the hull works independently without taking into account the rest of the model.
- A boss has wrong behaviour because a Hull Movement is Blocked (HMB). This bug occurs when the hulls are incorrectly positioned. Incorrect positioning blocks the movement of other parts of the model; for example, the position of one hull invades the space of the other. If they invade each other the physics has unpredictable behavior.

It is important to clarify that bosses can be built either using SDML software models or directly with C++. The developers of Kromaia started to create content with code, but they switched to SDML models to create the content of the game (e.g., levels, NPCs, items, and weapons). The intuition of the developers is that they make fewer mistakes and are more efficient working with the models than with the code, and an experiment confirmed this [22]. However, even though the models abstract implementation details in contrast to the code, these can be sources of bugs such as those indicated above.

In the interviews that we conducted with the developers that led to identifying these types of bugs, the developers acknowledged that these types of bugs are not limited to bosses. These types of bugs have occurred in other enemies and in other games that they developed in the past. Therefore, in the evaluation, we consider the types of bugs presented above, and our evolutionary approach leverages the simulations to locate the most relevant model elements for the bugs.

3 Classifying Bug Types in Taxonomy Categories

Truelove et al. [20] propose a taxonomy that is based on the root causes of the bugs and their effects on video games. This taxonomy consists of 20 categories, each shedding light on specific aspects of bug manifestations. For instance, the “Action” category in their taxonomy encompasses bugs that are related to the errors in the ability/inability to perform actions. We use this taxonomy to classify the five types of bugs presented in Section 2. Thus, our work is aligned with the state-of-the-art, and we report the results using a third party classification that facilitates future comparisons of our results with other works that use the same classification.

To do the bug classification and relate it to the bugs from [20], we involved multiple people by employing a methodology similar to those used in systematic literature reviews (SLRs) that involve inter-rater reliability (IRR) [23]. IRR ensures that classifications are consistent and reliable by having multiple raters who independently classify the data and then, comparing their classifications. We did the following steps to map the classification:

(1) one author initially classified the bugs identified in our work; (2) the other authors reviewed and validated this classification; and (3) to further ensure accuracy, two professional video game developers independently reviewed the classification. This step served as an additional layer of validation to verify our classification.

In our classification alignment with [20], we want to emphasize that the presented mappings between our bug types and the categories from [20] are not intended as rigid one-to-one correspondences. Instead, our approach involves a nuanced interpretation based on thematic relevance and shared conceptual ground. We systematically analyzed the intrinsic characteristics of each bug type, aiming to capture the broader implications within the context of both classifications. The resulting mappings are not a mechanical exercise but rather a methodical process, reflecting the thoughtful consideration given to the relationships between our bug types and the categories outlined in [20]. We aligned each bug type with the category in [20] by considering the primary impact on gameplay and player experience. For example, the "Weak Point is Hidden" (WPH) bug occurs when a boss's vital point is inaccessible or invisible, preventing the player from performing necessary actions. We classify this under the "Action" category because it affects the player's ability to interact with the game effectively. Similarly, the "Clipping" (CLP) bug is characterized by objects intersecting incorrectly, breaking immersion. This bug fits within the "Collision of Objects" category as it directly involves incorrect handling of object collisions.

Our classification (*type of bug → category of bug in the taxonomy*) is as described below.

WPH→Action. The "Weak Point is Hidden" (WPH) bug occurs when a boss in a game is invincible because a vital point is inaccessible or invisible to the player. We classify this bug in the category of "Action" because it refers to errors in the ability or inability to perform actions in the game. Examples of this type of bug include button mapping bugs, incorrect feedback from the game, and inconsistent movement or aiming mechanics. While the WPH and Action bugs may seem distinct, they can often be related. Specifically, the WPH bug can contribute to the Action bug by preventing the player from being able to perform certain actions that would be necessary to defeat the boss. For example, if the WPH bug makes a vital point of the boss inaccessible (see Fig. 4.A) the player may not be able to use certain weapons or attacks that would be effective against that point. As a result, the player may experience frustration and confusion, which can contribute to the Action bug.

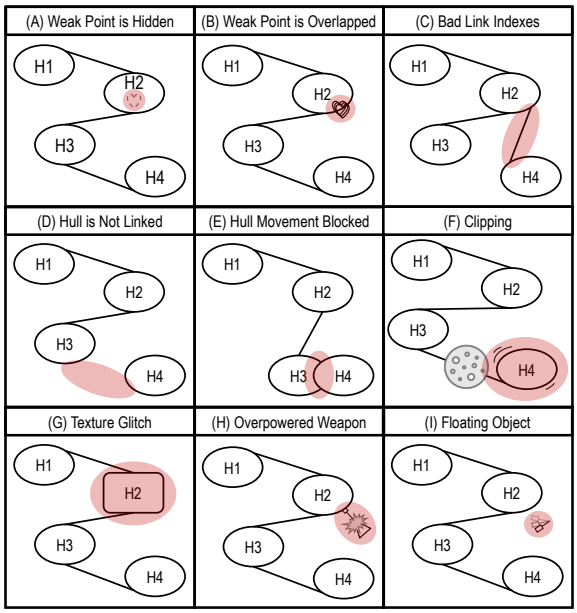


Fig. 4 Graphical representation of the bug types.

WPO→Action. The "Weak Point is Overlapped" (WPO) bug occurs when a boss is invincible because a weak point is overlapped by another object when it should not be. Similar to the WPH bug, the WPO bug can contribute to the category of "Action" by preventing the player from being able to perform certain actions that would be necessary to defeat the boss. For example, if a solid object overlaps with a weapon or attack that the player is attempting to use against a weak point (as Fig. 4.B depicts), the attack may not register or may cause unexpected results, making it difficult or impossible for the player to progress in the game. As a result, the player may become frustrated and disengaged from the game, leading to the "Action" bug.

BLI→Artificial Intelligence, Game Graphics. The "Bad Link Indexes" (BLI) bug occurs when the links between the parts of a boss are incorrectly assigned as depicted in Fig. 4.C, causing the physics of the video game to become erratic and the movement of the boss to behave unexpectedly. The BLI bug can manifest in a

variety of ways, but it often causes bosses to move in ways that are unintended or unpredictable. This can impact player experience, as it can make it difficult for players to anticipate or react to the behavior of bosses in the video game. Hence, BLI is related to the categories "Artificial Intelligence" (AI) and "Game Graphics". When the BLI bug affects a boss's movement in unexpected ways, it can also cause the boss's AI to behave unpredictably, leading to an "AI"-related bug. For example, if the boss moves in a way that is not expected, it may trigger an AI response that is inappropriate for the situation. This can cause the boss to behave in ways that are not intended by the game designers, leading to a negative user experience. In addition, the BLI bug can also contribute to "Game Graphics" bugs since the erratic movement of the boss can cause visual artifacts or other anomalies in the game world. For example, if the boss moves in a way that is not expected, it may clip through other objects in the game world or cause other visual glitches. This can negatively impact the visual experience of the video game.

HNL→Game Graphics, Position of Object. The "Hull is Not Linked" (HNL) bug occurs when the hulls of a boss are not attached to any other part of the model. In this case, the hull works independently without taking into account the rest of the boss. This can lead to unexpected behavior and movement that can negatively impact player experience. This bug can manifest in a variety of ways such as bosses moving in unexpected ways or parts of the boss appearing detached from the rest of the model. The HNL bug contributes to the categories "Game Graphics" and the "Position of Object". When the HNL bug affects the movement and behavior of bosses, it can also cause visual anomalies in the game world, leading to "Game Graphics" bugs. For example, Fig. 4.D shows a hull of a boss that appears to be floating independently of the rest of the model, breaking the immersion and negatively impacting the visual experience of the video game. In addition to its impact on "Game Graphics", the HNL bug can also affect the "Position of Object" in the game world. If the hull of a boss is not properly linked to the rest of the model, it may appear to be in the incorrect position or orientation within the game space over time.

HMB→Artificial Intelligence, Position of Object. The "Hull Movement is Blocked" (HMB) bug occurs when the hulls of a boss are incorrectly positioned, and their movement is blocked by other parts of the model. This can lead to unexpected behavior, such as bosses moving in unexpected ways or not responding to player actions. This type of bug can also have ripple effects on other aspects of the video game, such as the game's "Artificial Intelligence" (AI) and the "Position of Object" within the game world. This bug can cause the boss to behave in unexpected ways, leading to an "AI" bug where an NPC or AI does not behave in the intended manner. For example, if the movement of the boss is blocked by incorrectly positioned hulls (Fig. 4.E), it may not respond to player actions as expected, leading to player frustration and a negative experience overall. In addition to its impact on AI, the HMB bug can also affect the "Position of Object" in the game world. When the movement of the boss is blocked, it can result in objects not being in the correct position or orientation within the game space over time.

In our previous work [14], we identified the five types of bugs that were presented above. In this work, we classified those five types of bugs in four categories of the taxonomy. In addition, we added four types of bugs that also affect the behavior of a boss and are common when creating the model. Thus, we cover 50% more categories of the existing taxonomy. In the following paragraphs, we describe these new bug types and their classification in the taxonomy.

CLP→Collision of Objects. Bug "Clipping" (CLP) occurs when two objects intersect with each other and the physics engine does not handle it correctly. For example, Fig. 4.F depicts a boss's link clipped through an asteroid, looking visually jarring and breaking the immersion. CLP is a specific example of the category "Collision of Objects". In games, collision detection and response are critical to gameplay and immersion. Objects colliding with each other should behave in a realistic and expected manner. Therefore, it is important to thoroughly test and debug collision detection and response to ensure that objects behave correctly when they collide or interact with each other.

TXG→Game Graphics. The "Texture Glitch" (TXG) bug occurs when the game's textures are not rendered correctly, leading to visual artifacts or distortion. For example, if a boss's hull texture is stretched and distorted (as Fig. 4.G represents), it can look visually unappealing and break the immersion. It can happen for a wide range of reasons, including incorrect texture mapping, memory errors, or bugs in the *game engine*. When a boss has a wrong appearance due to a TXG, it can make the game feel visually unpolished and unappealing. It can also make it difficult for the player to distinguish important game elements, making it harder to play the game. Therefore, it is essential to ensure that the game's graphics are rendered correctly to provide an immersive and enjoyable experience for the player.

OPW→Interaction Between Object Properties. The "Overpowered Weapon" (OPW) bug occurs when a weapon or ability has unintended effects on the game world or other objects. Fig. 4.H shows an example of a weapon that deals too much damage or has too wide of an area of effect. OPW is a specific example of the category known as "Interaction Between Object Properties". This category refers to when the properties of two or more objects do not behave properly when interacting with each other. In the case of OPW, the interaction between the weapon's properties and the game world is not balanced properly. This can cause unintended effects such as bosses becoming too weak, enemies being killed too easily, or puzzles being solved too quickly. It is important to carefully balance the properties of objects in the game to ensure a fun and challenging gameplay experience.

FLO→Position of Object. The "Floating Object" (FLO) bug occurs when an object is not properly anchored to the whole boss's body or other objects in the game world, leading to it floating or moving unnaturally. For example, Fig. 4.I shows a weapon in the game world that is floating in mid-air. It can be frustrating for players to see objects floating around unnaturally or not anchored correctly, and it can break the immersion of the game. This bug can also affect the gameplay mechanics since objects that are not properly anchored may not behave as intended, leading to unintended consequences. Therefore, this bug is classified in the category "Position of Object", which refers to any bugs related to an object that is not in the correct position or orientation within the game space over time. This category encompasses a wide range of bugs, from objects that are floating in mid-air to objects that are stuck in walls or floors, and it is a crucial aspect of ensuring a smooth and enjoyable gameplay experience for the player.

Table 1 summarizes the classification of the nine types of bugs in the six categories of bugs of the taxonomy. The columns of the table show the following: the categories of the taxonomy (Column 1); the descriptions of the categories (Column 2); and the types of bugs included in each category (Column 3). The table does not include all of the categories detailed in the taxonomy [20] since not all of the bugs are related to software models (e.g., audio and camera). This classification was verified by a professional video game developer. Afterwards, another professional video game developer reviewed and confirmed the classification. We use this classification of six categories of bugs to report the results in the evaluation. Nevertheless, we acknowledge the fact of using a different classification as a threat to validity.

4 Evolving Simulations to Locate Bugs in Software Models of Video Games

This section describes how our evolutionary algorithm tackles the challenge of bug localization in video games. We first present an overview of our approach and subsequently provide the details of the approach and our adaptation of the evolutionary algorithm to work with game simulations.

4.1 Overview of the Approach

We call our approach "Evolutionary algorithm for bug localization in software Models leveraging the game Simulations" (EMoSim). The general structure of the approach is introduced in Fig. 5. Our EMoSim approach takes as input a set of software models in which we want to locate a bug. The goal of EMoSim is to obtain a ranked list of simulation traces that are ordered by their relevance in locating the bug.

To do this, the approach initializes a population of individuals. Each individual refers to the overall configuration (set of parameters) applied to a duel between the boss and the simulated

Table 1 Classification of types of bugs in the taxonomy of categories of bugs.

Category of the taxonomy	Description	Bug type(s)
Action	Errors in the ability/inability to perform actions.	Weak Point is Hidden (WPH), Weak Point is Overlapped (WPO)
Artificial Intelligence	An NPC or AI does not behave in the intended manner.	Bad Link Indexes (BLI), Hull Movement is Blocked (HMB)
Collision of Objects	Objects do not behave properly when they collide or contact each other.	Clipping (CLP)
Game Graphics	A certain visual aspect of the game world is being rendered incorrectly.	Bad Link Indexes (BLI), Hull is Not Linked (HNL), Texture Glitch (TXG)
Interaction Between Object Properties	Two or more object properties do not behave properly when interacting with each other.	Overpowered Weapon (OPW)
Position of Object	An object is not in the correct position or orientation within the game space over time.	Hull is Not Linked (HNL), Hull Movement is Blocked (HMB), Floating Object (FLO)

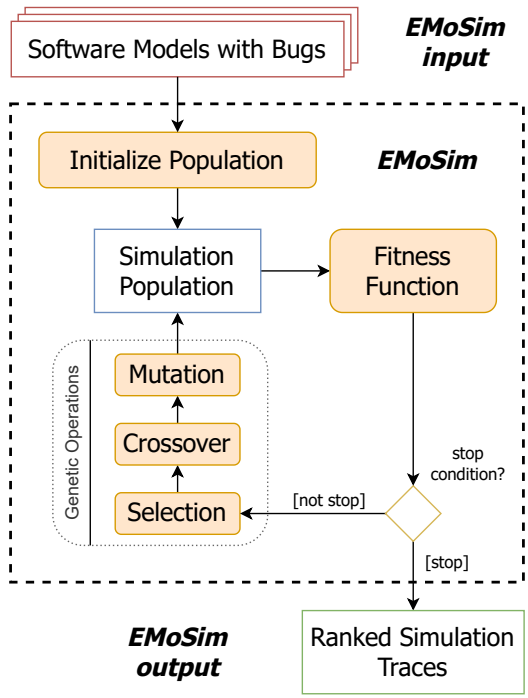


Fig. 5 EMoSim approach.

player. For example, a parameter of the individual is how many attacks the simulated player will do to the first hull of the boss. Thus, we use the terms individual and simulation interchangeably. The search space for our approach is determined by the number of possible simulations. To explore the search space, EMoSim uses an evolutionary algorithm that enables the exploration of a large number of possible simulations. For each simulation, our approach collects data (specific values) from the execution of the duel. For example, the number of steps (i.e., time units) that a simulated player took to attack each hull of the boss. This data (i.e., trace) is used to assess each individual of the population with the fitness function, which is calculated with data that the developers consider relevant, such as the percentage of simulated player victories.

If the stop condition of the evolutionary algorithm is not met, new individuals (i.e., simulations) are generated through the genetic operations in order to create conditions in the duel between the boss and the simulated player where bugs are more likely to appear. For example, the conditions of a simulation with high values of parameters, such as the probability of the boss being hit and the remaining steps to be taken by the player when hit, after running the duel, may result in an outlier value of 300 steps in a hull in the trace, which may indicate a bug. Hence, the search of simulations that is performed by the evolutionary algorithm helps to identify conditions for bug localization, addressing the dynamic nature of video games. If the stop condition of the evolutionary algorithm is met, the simulation traces are ranked in ascending order with the lowest fitness values first. The resulting relevant model fragment for the bug being located is extracted from a simulation trace.

Next, we describe the evolutionary algorithm and its adaptation to the bug localization problem in software models of video games.

4.2 Adapting the EMoSim Approach

Evolutionary algorithms are inspired by Darwin's evolutionary theory, where a population of individuals is modified through crossover and mutation operators [24]. Hence, to develop an evolutionary algorithm, the following elements must be defined:

- Representation of the individuals.

- Evaluation of the individuals using a fitness function for each objective to determine a quantitative measure of their ability to solve the problem under consideration.
- Selection of the individuals to transmit from one generation to another.
- Creation of new individuals using genetic operators (cross-over and mutation) to explore the search space.

The following subsections describe the design of these elements of our evolutionary algorithm for bug localization in software models of video games.

4.2.1 Individual Representation

To represent an individual (i.e., simulation), we use a vector representation. Each vector's dimension represents a parameter of the simulation. Thus, an individual is defined as a set of parameters applied to a duel between the boss and the simulated player. The size of the individual corresponds to the number of parameters (dimensions) in the vector. The simulation parameters were provided by the developers based on the analysis of battles between real players and bosses. The current number of simulation parameters is 12 (as Table 2 shows): three parameters are about the simulated player (upper part of the table), five about the boss (middle part of the table), and four more about the itinerary performed by the simulated player or the boss during the simulation (bottom part of the table). The data types are Int and Float scalars. New individuals are generated by the evolutionary algorithm.

Fig. 6 shows two examples of individuals. The graphical representation of each individual emulates the behavior of a player when the duel with the boss occurs by taking into account the parameters. For example, the parameters that are included in each individual (i.e., simulation) can define how many steps the simulated player takes in each hull of the boss, the order in which the hulls are visited following different patterns (one by one, visit one skip one, visit one skip three...), if the player requires all of the remaining steps in the hull when he/she is attacked by it, or the direction used to visit the hulls of the boss. Steps are a measurement unit of the time spent by the simulated player on a certain element of the model.

Each example in Fig. 6 corresponds to different parameters applied to a simulation. In both cases, the triangle corresponds to the simulated player, the circles and lines that connect them correspond to the boss, the dashed and dotted lines correspond to the path that follows the simulated player in his/her strategy, and the crosses correspond to the attacks that the simulated player performs to the hulls. The upper example shows the simulation of a conservative player in which the player attacks the first hull and then moves away. The lower example shows the simulation of an explorer player in which the player attacks the first hull then skips one and attacks the following hulls to the end of the boss.

4.2.2 Fitness Function

Once an individual is created, its parameters are used to execute the duel between the boss and the simulated player. This simulation is performed by a custom stand-alone simulator, implemented in Kotlin, which interprets XML models derived from the Kromaia game. The simulator uses the same algorithm than the engine of the game itself, and the advantage of using the simulator over the engine as is in the game is that unnecessary parts

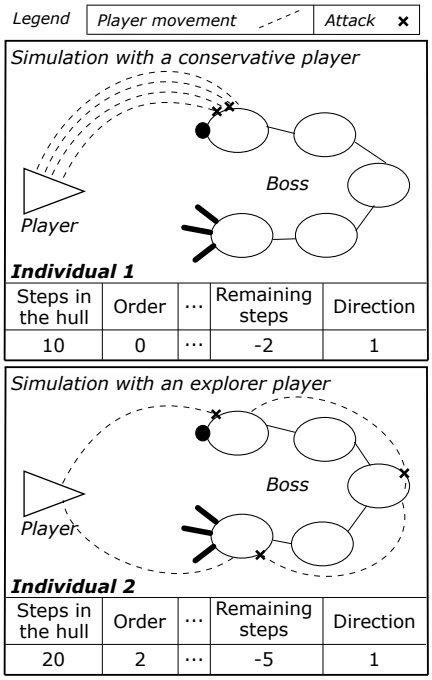


Fig. 6 Representation of a simulation as an individual.

Table 2 Simulation parameters provided by the developers.

Name	Description	Data type	Default value	Value range
playerInitialShields	Shields that the simulated player has at the start.	Int	5	[0, ∞)
hitProbabilityPlayer	Probability of the boss being hit by the simulated player.	Float	0.25	[0, 1]
recoveryStepDistance	Steps that the simulated player requires to regain control after being attacked.	Int	10	[0, ∞)
bossInitialShields	Shields that the boss has at the start.	Int	Dynamic[a]	[0, ∞)
hitProbabilityWeaponMe	Probability of the simulated player being hit by a melee weapon.	Float	0.014	[0, 1]
hitProbabilityWeaponBu	Probability of the simulated player being hit by a bullet weapon.	Float	0.014	[0, 1]
hitProbabilityWeaponHo	Probability of the simulated player being hit by a homing weapon.	Float	0.014	[0, 1]
hitProbabilityWeaponLa	Probability of the simulated player being hit by a laser weapon.	Float	0.014	[0, 1]
hullVisitSteps	Steps that the simulated player stops on each hull.	Int	20	[1, ∞)
hullOrderVisit	Order in which the simulated player visits the hulls, i.e., one by one, two by two, ...	Int	1	[1, ∞)
hullStepCount	Steps for the simulated player to flee or dodge after being hit.	Int	-1	(-∞, -1]
hullRouteDirection	Direction of the route followed by the simulated player: 1 forward and -1 backward.	Int	1	1, -1

[a]Based on the value specified in the model.

like graphics or sound are not run. Thus, the simulation reproduces the behavior and mechanics of the Kromaia game to ensure fidelity, but without running the actual game engine. The information collected from the duel serves to assess the quality of each individual (i.e., simulation) using a fitness function. Hence, the input of this step is a set of individuals; the output is the set of individuals, where each individual has been assigned a fitness value regarding its relevance for the bug.

We use a fitness function that is similar to the fitness function presented in [25]. This fitness rewards simulations that have behaved as the developers intended for their game. In [25], the goal of the work was to generate game content (bosses); therefore, it made sense to reward those bosses who behaved as the developers desired. In this work, the goal is different. Since we are looking for bugs, we use the same fitness function except that we rank the simulation traces in reverse order. This means that we rank as first the simulations that are farthest from what the developers expected. The idea is that if they have strayed from what the developers expected, they might be relevant when locating a bug.

For each simulation, our approach collects information about the battle and key events in order to calculate the fitness value and to obtain the trace. The information retrieved from the simulation is the data that the developers regard as relevant, using their domain knowledge. Hence, our approach takes into account the percentage of simulated player victories (*victory*) and the percentage of simulated player health left once the player wins a duel (*health*).

The calculation of *victory* and *health* is performed in the same way as in [25]:

- $Victory$ is the difference between the number of simulated player victories (V_P) and the optimal number of victories (V_O, 33%, according to the developers of Kromaia and their criteria):

$$victory = 1 - \frac{|V_O - V_P|}{V_O} \tag{1}$$

- $Health$, which refers to completed duels that end in simulated player victories, is the average difference between the player's health percentage once the duel is over (H_P) and the optimal health level that the player should have at that point (H_O, 20%, according to the developers of Kromaia):

$$health = 1 - \frac{\sum_{d=1}^{V_P} \frac{|H_O - H_P|}{H_O}}{V_P} \tag{2}$$

To normalize the values of both *victory* and *health*, our approach limits the range of values that each calculation can take between 0 and 1. Values less than 0 will get a value equal to 0 (which means 0%). Likewise, values greater than 1 will get a value equal to 1 (which means 100%).

The fitness value of a simulation is the average value between the *victory* and the *health* values described above.

4.2.3 Obtaining the Ranked Simulation Traces and Their Resulting Model Fragments

When the stop condition is met, our algorithm ranks the simulations (i.e., individuals) in ascending order with the lowest fitness values first. A lower fitness value means that the simulation has had a result that the developers would not approve of.

For each individual, apart from calculating the fitness value after executing the simulation, the trace is also obtained with the aim of extracting the relevant model fragment where the bug may be located. The trace contains the specific values that are gathered after the duel between the boss and the simulated player, such as how many steps (i.e., time units) the simulated player takes to attack each hull of the boss. The upper part of Fig. 7 depicts an example of a model of a Kromaia boss and a trace of an individual.

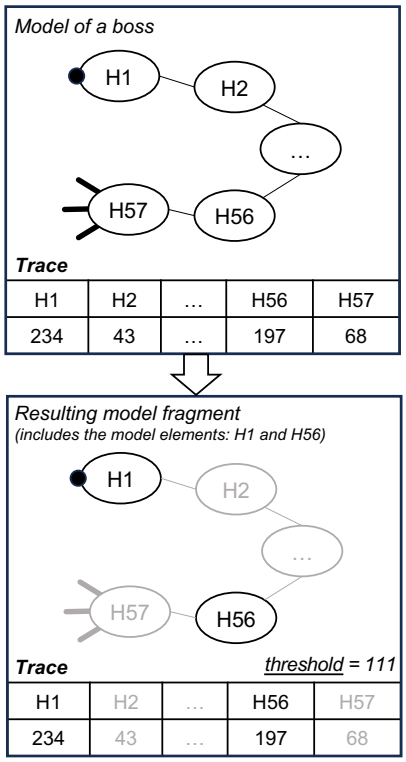


Fig. 7 Example of a simulation trace and the model fragment mapping.

In order to extract the relevant model fragment from a simulation trace, the approach does the following:

1. **Grouping and Counting by Element**: The approach groups elements within the simulation trace and counts their occurrences, resulting in a map that tallies how many times each distinct element appears in the trace.
2. **Filtering for Relevant Elements**: This grouped count is then filtered based on a threshold. Elements with occurrences above this threshold are retained.

The threshold is an outlier that is calculated at run-time according to the central tendency of the trace values. The higher frequency of an element in the trace (exceeding the threshold) likely indicates its relevance for locating the bug. The intuition is that if the simulated player took an extraordinary amount of steps attacking one hull element, that hull element may be hidden or invincible. Therefore, the elements surviving the filter—those appearing more frequently than the threshold—can be considered to be the relevant model elements.

The lower part of Fig. 7 shows the model fragment that results after selecting those model elements that are above the threshold in the trace. In the depicted example, the trace of the selected individual indicates Hull 1: 234 steps, Hull 2: 43 steps, ..., Hull 56: 197 steps, Hull 57: 68 steps, and our approach sets 111 as the threshold. Hence, Hull 1 and Hull 56 are the model elements that conform the resulting model fragment, which can be considered relevant for locating the bug.

4.2.4 Selection

If the stop condition is not met, the evolutionary algorithm continues with the genetic operations (either classic and component-specific). To do this, it is necessary to select individuals from the population that are going to be evolved. Then, we use the wheel selection mechanism, i.e., the selection of an individual is directly proportional to its relative fitness in the population. This mechanism gives a higher probability of selection to the fittest individuals while still giving a chance to every individual.

In each iteration, the algorithm selects individuals from the population (P_n) for the next

generation of the population (P_{n+1}). The selected individuals will be the ones that generate the next individuals using genetic operations.

4.2.5 The Classic Genetic Operations

Most of the works use the classic genetic operations (single-point crossover plus random mutation) [26] to generate new individuals from the modification of the elements of the individual as a whole. These genetic operations are presented below:

- Crossover: We use a single, random, cut-point crossover. It starts by selecting and splitting two parent individuals at random. When two parent individuals are selected, a random cut point is determined to split them into two sub-vectors. Then, the crossover creates two child individuals by putting the first part of the first parent with the second part of the second parent for the first child and putting the first part of the second parent with the second part of the first parent for the second child.
  Fig. 8-a) depicts a representation of the crossover applied to the Parent A and Parent B simulations and the resulting offspring simulations. Each individual (S_1 and S_2 in the figure) has the same length, which is the number of parameters for the simulation.
- Mutation: This operator consists of randomly changing one or more parameters in the simulations. Given an individual, the mutation operator first randomly selects some positions in the vector representation of the individual. Then, the selected dimensions are replaced by another value of the parameter. These values are not randomly generated numbers; they are selected from a catalogue of values that the developers have collected from battles between real players and bosses. Fig. 8-b) depicts a representation of the mutation applied to the individuals that were obtained from the crossover (S_1 and S_2) and the resulting new simulations (Child A and Child B). Each replaced value in the individual is highlighted in solid gray.

As a result, new simulations are created in order to test a large number of possible game scenarios by changing the simulation parameters, which represent different elements of the boss and player profiles. In other words, the new simulations represent other possible individuals that might be relevant for locating the bug. Overall, the aim of the approach is to find the most relevant simulation to locate the target bug. The most relevant simulation seeks to represent conditions in the duel between the boss and the simulated player where bugs are more likely to appear. For example, the conditions of a simulation with high values of parameters, such as the probability of the boss being hit and the remaining steps to be taken by the simulated player when hit, after running the duel, may result in a trace with an outlier value of 300 steps in a hull, which may indicate a bug.

To find the most relevant simulation, the algorithm of EMoSim performs a search that is guided by a fitness function. This search is done among the different simulations (previously obtained by applying the mutation and crossover operations) that could be relevant to locate the bug.

4.2.6 The Component-Specific Genetic Operations

Most previous works choose the classic genetic operations [26] by default, and they do not consider that there are other operations that could be potentially better for the quality of the solutions. In our previous work [14], we also chose the classic genetic operations by default without considering whether doing changes in the genetic operations could affect the quality of the solutions. Hence, in this paper, we propose component-specific operations that perform the genetic operations considering three different components of a simulation:

- The *simulated player* represents the entity (i.e., the player's spaceship) whose behavior is being modeled within the simulation. This component comprises three parameters: the number of shields that the player has at the start, the probability of the boss being hit by the player, and the distance in recovery steps after an attack.
- The *boss* represents the entity (i.e., the final boss of a level) within the simulation that is not controlled by the player. This component comprises five parameters such as the probability of the player being hit by a melee weapon, by a bullet weapon, or by a laser weapon. We never mutate the model that defines the boss

but only the parameters of the simulation (i.e., the parameters of the duel between the boss and the simulated player).
- The *itinerary* refers to the path or sequence of actions taken by the simulated player or NPCs within the simulation. This component comprises four parameters: the steps that the player uses to attack each hull, the order in which the player visits the hulls (i.e., one by one, two by two, ...), the remaining steps to be taken by the player when hit, and the direction of the route followed by the player: 1 forward and -1 backward.

The lower part of Fig. 8 depicts an example of the component-specific operations by taking into account the three different components of the parent simulations. To start with, each parent simulation is split into three subsets (see Fig. 8-c), which correspond to the three different components of the simulation (simulated player, boss, and itinerary). The single-point crossover operation is then applied at the component level, creating six offspring simulations that inherit genetic information from both parents (see Fig. 8-d). The main advantage of this single-point crossover at the component level is that the offspring simulations combine information from parents that only contain one component instead of containing all components. Combining all components may produce offsprings that lack meaningful coherence and, consequently, worsen the results. Then, the resulting six offspring simulations of the components are randomly mutated to change one or more parameters (see Fig. 8-e). These parameters are selected from a catalogue of values that the developers have collected from battles between real players and bosses.

As a result, the resulting six offspring simulations are merged into two resulting children (see Child C and Child D in Fig. 8). Each child simulation inherits genetic information from all three components, with a more targeted and fine-tuned genetic representation. Thus, this new component-specific genetic operations produce simulations that better capture the complexities of video game scenarios.

5 Evaluation

This section presents an evaluation of the approach: the oracle preparation, the experimental setup, the results obtained, and the statistical analysis performed.

We evaluate the use of the simulations in our approach and the different categories of bugs involved in the process. In order to address the evaluation, we formulated the following four research questions:

- RQ_1: What is the performance in terms of solution quality of EMoSim with the component-specific genetic operations, EMoSim with the classic genetic operations, the baseline, and Random Search?
- RQ_2: Is there any difference in performance between the classic genetic operations and the component-specific genetic operations?
- RQ_3: Is there any difference in performance among the different categories of bugs in video games?
- RQ_4: Are the performance results obtained by EMoSim with the component-specific genetic operations, EMoSim with the classic genetic operations, the baseline, and Random Search significant?

Answering RQ_1 allows us to compare the performance results (in terms of recall, precision, and F-measure) of the two variants of our approach (with the classic genetic operations and the component-specific genetic operations), and the baseline. In addition, we compare our approach with a Random Search (RS) sanity check. If RS outperforms an intelligent search method, we can conclude that there is no need to use metaheuristic search. Answering RQ_2 with the same metrics allows us to know if the use of component-specific genetic operations instead of the classic genetic operations influences the results. Answering RQ_3 with the same metrics allows us to know if the bug category influences the results. Answering RQ_4 allows us to properly compare the approaches, to provide formal and quantitative evidence (statistical significance) that the approaches do in fact have an impact on the comparison metrics (i.e., that the differences in the results were not obtained by mere chance), and to show that those differences are significant in practice (effect size).

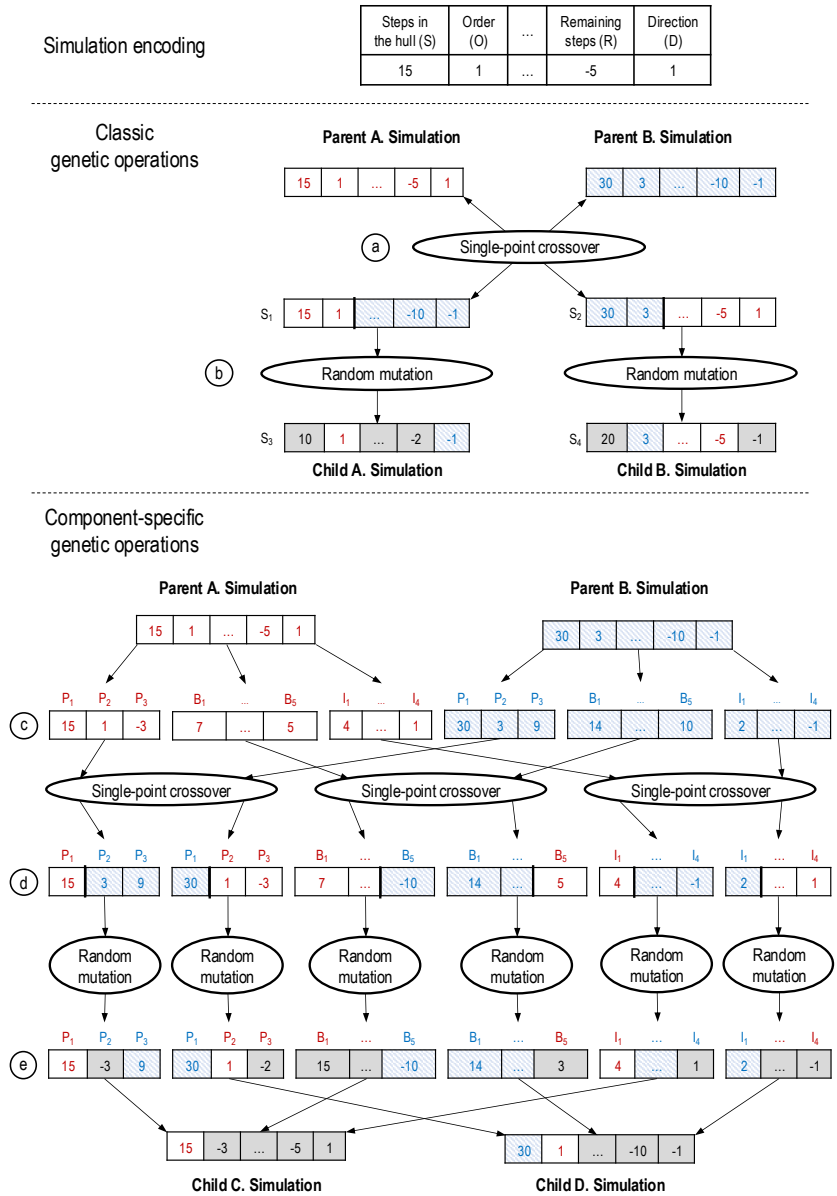


Fig. 8 The component-specific genetic operations using simulation encoding.

5.1 Oracle Preparation

To evaluate the approach, we applied it to the Kromaia video game, which is a commercial video game released on PC and PlayStation 4. For the case study, the data repository provided by our industrial partner includes the models with bugs and the bug reports. Our industrial partner also provides the model fragments that are source of the bugs (oracles) as part of the data repository. As the left side of Fig. 9 shows, a random selection of 54 bugs is made from the provided data repository. From the selection of 54 bugs, their corresponding data and documentation is extracted such as the oracle. The oracle is the ground truth and is used to compare the results provided by EMoSim in its two variants (using component-specific or classic genetic operations), the baseline (BLiMEA [16]), and RS.

The BLiMEA approach uses an evolutionary algorithm that iterates through the models of a system and assesses model fragments as possible sources of bugs. Although this approach is not specific for video games, the approach can be useful for locating bugs in systems that use models. A recent survey [9] on bug localization techniques did not identify any other approach that considers models as the source of the bugs or a specific approach for bug localization in video games.

BLiMEA uses a multi-objective evolutionary algorithm with two fitness functions: Information Retrieval (IR) and modification timespan. This approach receives a bug description and a set of product models as input. The output is a set where each model fragment has been assigned two fitness values: the similarity to the bug description and the timespan to the most recent model-fragment modifications. Since BLiMEA needs more information as input than EMoSiM, we augmented the test cases with the information that BLiMEA needs.

To prepare the oracle, we extend the 30 bugs from our previous work [14] to a total of 54 bugs that were randomly selected from the entire data repository. Thus, the six categories of bugs are supported and four new types of bugs (CLP, TXG, OPW, and FLO) are supported with six bugs each. These bugs contained natural language bug descriptions and the approved model fragments that contained the target bugs. Each model had more than 1000 model elements. For each of the bugs, we created two different test cases that were needed as input by the approaches. One of them included the set of product models where that bug was manifested and a bug description for BLiMEA. The second one included the set of product models where that bug was manifested for EMoSim and RS. All of them were obtained from the data repository.

5.2 Experimental Setup

Fig. 9 shows an overview of the process that was followed in the evaluation. The left part of the figure shows the inputs of the evaluation process, which are the models with bugs and bug reports from the industrial partner. The input (data repository from the industrial partner) is marked with a red border line. The output (ranked model fragments) of our approach, RS, and the baseline is marked with a blue border, and the intermediate one (ranked simulation traces) is marked with a green border. The various approaches are filled with different pattern backgrounds: EMoSim (component-specific) branch, which is the novelty of this paper, uses the evolutionary algorithm (as described in Section 4) with the component-specific genetic operations (as described in Section 4.2.6). It has the orange "cross-hatch" background and a thicker border line. EMoSim (classic), which is our previous work, also uses the evolutionary algorithm, but the genetic operations are the classic ones (as described in Section 4.2.5). It has the purple "hatch" background. Random Search (sanity check) has the "solid" background; and BLiMEA (baseline) has the "zigzag line" background. The box that represents the reports that contain information about the performance results (in terms of solution quality) is highlighted with a dashed line.

The baseline, BLiMEA, produces a ranking of model fragments that are the most relevant to the bug. Random Search is used as a sanity check to determine if our approach performs better than mere chance. To produce a model fragment, RS starts with the random selection of an individual (i.e., the set of parameters applied to a simulation) from the population. Then, new individuals are generated by taking into account all components of the selected individual as a whole using the genetic operations. These two steps are repeated until the stop condition is met. Once the stop condition is met, an individual is selected randomly and the simulation (that is described by the parameters of the selected individual) is executed to obtain its trace. Afterwards, RS sets a random threshold value to select those model elements whose value in the trace is above the threshold. Random thresholds can help in exploring different regions of the search space without bias. Thus, the search remains stochastic and simple, and not overly focused on any particular region based on a predetermined threshold. For example, if the trace of the selected individual indicates Hull 1: 3 steps, Hull 2: 245 steps, ..., Hull 19: 87 steps and RS randomly sets 4 as threshold, Hull 2 and Hull 19 are the model elements that conform the model fragment. Both EMoSim (component-specific) and EMoSim (classic) produce a ranking of traces. The trace contains all of the model elements that the

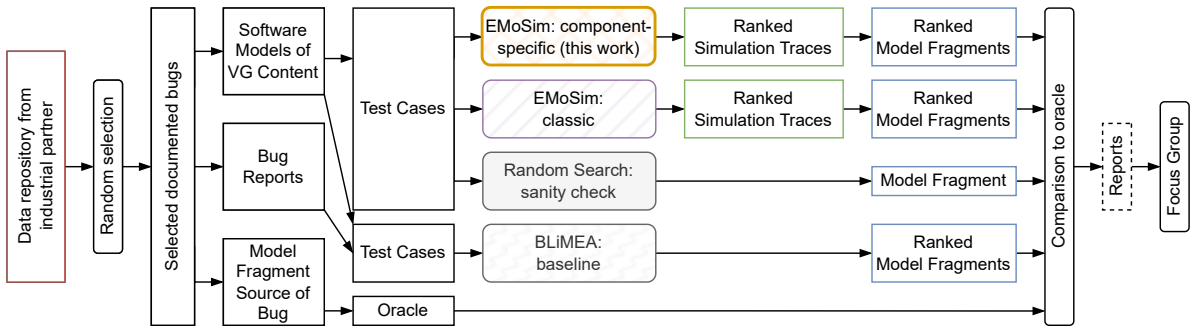


Fig. 9 Evaluation process.

interpreter has used at runtime during the simulation. All of the model elements that appear in the trace form the most relevant model fragment according to the trace for the bug. Then, we can compare the model fragments with an oracle in order to check accuracy.

After running the approaches, in order to compare them, we take the best solutions from each of the approaches for each of the bugs (the first solution in the ranking) as suggested in [27]. Then, we compare them to the actual solution (from the oracle) that contains the model fragment of the target bug in order to get a confusion matrix.

A confusion matrix is a table that allows the visualization of the performance of a classification algorithm. In our case, each solution is a model fragment that is composed of a subset of the model elements that are present in the model (where the bug is being located). Since the granularity will be at the level of model elements, the presence or absence of each model element will be considered as a classification. Therefore, our confusion matrices will distinguish between two values (TRUE/presence and FALSE/absence). The confusion matrix arranges the results of the comparison into four categories:

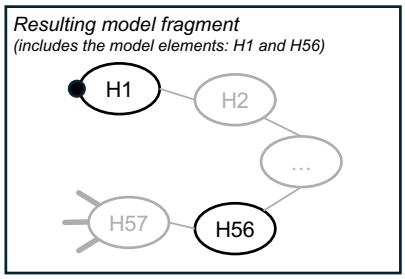
- True positive (TP): a model element present in the predicted model fragment that is also present in the model fragment from the oracle.
- True Negative (TN): a model element not present in the predicted model fragment that is not present in the model fragment from the oracle.
- False Positive (FP): an element present in the predicted model fragment that is not present in the model fragment from the oracle.
- False Negative (FN): an element not present in the predicted model fragment that is present in the model fragment from the oracle.

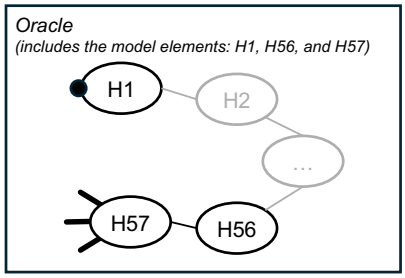
The confusion matrix holds the results of the comparison between the predicted model fragments and the model fragments from the oracle. The result of the sum of all of the categories (TP+TN+FP+FN) is the number of model elements (n) of the model that contains the predicted model fragment. Fig. 10 shows an example of the comparison made between the resulting model fragment (the upper part of the figure) and the oracle (the middle part of the figure) to obtain the confusion matrix (the lower part of the figure). In the figure, the model element H1 is in both the resulting model fragment and the oracle, so it is considered a True Positive (TP), whereas the model element H57 is in the oracle but not in the resulting model fragment, so it is considered a False Negative (FN). However, in order to evaluate the performance in terms of solution quality of the approach, it is necessary to extract some measurements from the confusion matrix. Specifically, we create a report that includes three performance measurements (recall, precision, and F-measure) for each of the test cases for the approaches.

Recall $\left(\frac{TP}{TP+FN}\right)$ measures the number of elements of the model fragment from the oracle that are correctly retrieved by the proposed model fragment.

Precision $\left(\frac{TP}{TP+FP}\right)$ measures the number of elements from the proposed model fragment that are correct according to the ground truth (the oracle).

F-measure $\left(2 * \frac{Precision*Recall}{Precision+Recall}\right)$ corresponds to the harmonic mean of precision and recall.





Comparison and confusion matrix

N=57		Predicted	
		FALSE	TRUE
Actual	FALSE	TN = 54	FP = 0
	TRUE	FN = 1	TP = 2

Fig. 10 Example of the calculation of the confusion matrix.

Recall values can range between 0 (i.e., no single model element from the model fragment from the oracle is present in any of the model fragments of the predicted solution) and 1 (i.e., all of the model elements from the oracle are present in the predicted solution).

Precision values can range between 0 (i.e., no single model element from the model fragment predicted is present in the model fragment from the oracle) and 1 (i.e., all of the model elements from the predicted solution are present in the model fragment from the oracle). A value of 1 in precision and 1 in recall implies that both the predicted model fragment and the model fragment from the oracle are the same.

5.3 Implementation Details

Each time that we run an approach, we obtain a set of results for a bug. As the approaches perform genetic operations, chance could affect the results. In order to minimize the effect of chance, we execute each of the approaches 30 times for each bug as suggested in [21]. For BLiMEA, we used the same parameters as reported in [15]. For EMoSim, we started from those reported in [25] (as we used the same simulation) and made sure they converged.

To determine the stop condition, we ran some prior tests to determine the convergence time. According to the tests, the time needed to converge was below 8 seconds for locating each bug. Therefore, we established the stop condition at 10 seconds (adding a margin to ensure convergence), ensuring that the approaches run long enough to obtain the best solutions.

For purposes of replicability, the implementation source code and the data (software models and oracles) are publicly available, including the confusion matrices obtained as result, and the CSV files (in terms of recall, precision, and F-measure) used as input in the statistical analysis at the following URL: https://bitbucket.org/svitusj/bl-in-mgse. The specific code that is executed to obtain the traces is in the file "Source Code/EMoSim/approach/simulation/EAlgorithm.kt" (lines 31-107).

5.4 Results

In this section, we present the results obtained in the two variants of EMoSim (with the component-specific genetic operations and the classic genetic operations), BLiMEA (baseline), and the RS (sanity check) approaches in Kromaia. Table 3 shows the mean values and standard deviations for recall, precision, and F-measure for each approach in the six categories of bugs that were presented in Section 3. Fig. 11 shows a bar chart that depicts the resulting F-measure values for each approach for the six categories of bugs. Both variants of EMoSim and BLiMEA obtained better results than the RS.

Table 3 Mean values and standard deviations for Recall, Precision, and F-measure for each category of bugs.

	EMoSim (component-specific genetic operations)						
	Action	Artificial Intelligence	Collision of Objects	Game Graphics	Interaction Between Object Properties	Position of Object	Overall
Recall ± (σ)	**93.15 ± 16.78**	**96.48 ± 8.62**	**98.33 ± 4.08**	**97.64 ± 5.78**	**100**	**97.78 ± 5.44**	**97.55 ± 8.04**
Precision ± (σ)	59.23 ± 12.99	**40.96 ± 9.88**	36.37 ± 3.65	**36.92 ± 2.59**	**36.58 ± 6.67**	**34.94 ± 3.88**	**40.82 ± 16.43**
F-measure ± (σ)	**70.92 ± 12.8**	**57.03 ± 9.73**	**52.94 ± 3.32**	**53.52 ± 3.16**	**53.29 ± 6.67**	**51.38 ± 4.29**	**55.73 ± 12.59**
	EMoSim (classic genetic operations)						
	Action	Artificial Intelligence	Collision of Objects	Game Graphics	Interaction Between Object Properties	Position of Object	Overall
Recall ± (σ)	59.17 ± 23.82	68.34 ± 20.06	74.34 ± 12.28	65.83 ± 20.63	80 ± 23.29	72.59 ± 18.29	69.21 ± 22.99
Precision ± (σ)	**64.03 ± 6.98**	36.33 ± 4.53	**37.53 ± 6.13**	35.95 ± 2.51	35.59 ± 9.87	34.14 ± 1.57	39.5 ± 15.66
F-measure ± (σ)	59.88 ± 17.09	46.97 ± 7.96	49.37 ± 5.6	45.86 ± 6.81	48.66 ± 12.47	45.96 ± 4.69	47.8 ± 13.09
	BLiMEA (baseline)						
	Action	Artificial Intelligence	Collision of Objects	Game Graphics	Interaction Between Object Properties	Position of Object	Overall
Recall ± (σ)	46.11 ± 16.26	35.47 ± 9.31	35.14 ± 8.21	34.35 ± 9.11	36.95 ± 10.3	38.87 ± 10.38	41.15 ± 19.98
Precision ± (σ)	41.08 ± 7.28	23.52 ± 6.75	22.05 ± 7.63	20.79 ± 3.08	22.58 ± 6.6	18.3 ± 3.47	25.64 ± 15.62
F-measure ± (σ)	41.43 ± 3.87	27.81 ± 6.22	27.78 ± 6.15	25.56 ± 3.85	27.56 ± 6.62	24.62 ± 4.68	28.56 ± 11.62
	Random Search (sanity check)						
	Action	Artificial Intelligence	Collision of Objects	Game Graphics	Interaction Between Object Properties	Position of Object	Overall
Recall ± (σ)	56.67 ± 18.23	70.93 ± 16.53	72.09 ± 13.72	70.42 ± 16.11	81.67 ± 17.48	68.7 ± 10.17	68.79 ± 18.24
Precision ± (σ)	7.13 ± 3.95	4.84 ± 2.3	9.77 ± 4.99	4.25 ± 1.88	3.78 ± 1.48	3.16 ± 1.33	5.23 ± 4.19
F-measure ± (σ)	12.57 ± 6.73	9.03 ± 4.16	16.84 ± 8.05	8 ± 3.45	7.2 ± 2.75	6.03 ± 2.47	9.39 ± 6.78

RQ_1 answer. EMoSim with the component-specific genetic operations obtained the best performance results, providing an average value of 97.55 in recall, 40.82 in precision, and 55.73 in the overall F-measure. EMoSim with the classic genetic operations obtained 47.8 in the overall F-measure, which also outperformed the baseline and RS. Although the overall value of recall of RS outperformed the baseline and was slightly lower than EMoSim with the classic genetic operations, the overall value of precision indicates that many elements of the resulting model fragments were not present in the model fragments from the oracle. For this reason, RS obtained the worst F-measure.

RQ_2 answer. There are differences in performance between the classic genetic operations and the component-specific genetic operations. As Table 3 shows, the biggest improvement is when the overall recall is compared (by 28.34%). In the overall F-measure, EMoSiM with the component-specific genetic operations improved the results of EMoSiM with the classic genetic operations by 7.93%. Also, EMoSiM with the component-specific genetic operations improved the results of the baseline by 27.17% and RS by 46.34% in the overall F-measure. The best values for each category are highlighted in bold in the table.

5.5 Statistical Analysis

To properly compare the approaches, all of the data resulting from the empirical analysis was analyzed using statistical methods following the guidelines in [21].

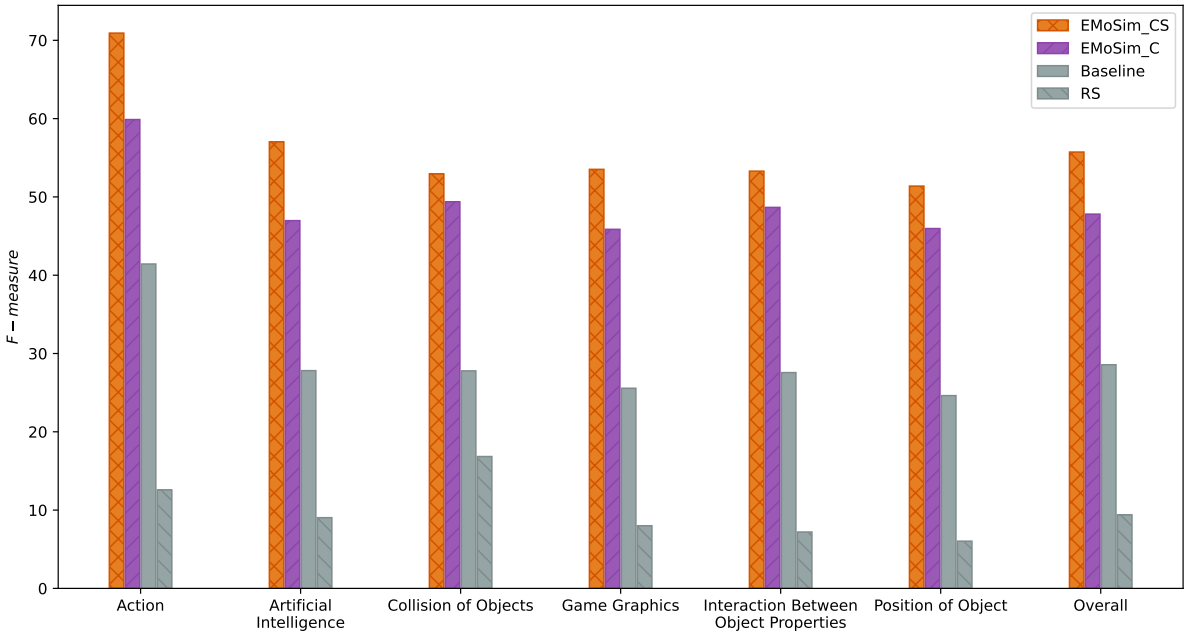


Fig. 11 Bar chart that represents the F-measure values for each category of bugs.

5.5.1 Statistical Significance

The test that we must follow depends on the properties of the data. Since our data does not follow a normal distribution in general, our analysis requires the use of non-parametric techniques. There are several tests for analyzing this kind of data; however, the Quade test has shown that it is more powerful than the others when working with real data [28]. In addition, according to Conover [29], the Quade test has shown better results than the others when the number of algorithms is low (no more than four or five algorithms).

The $p - Values$ obtained in the test are $< 2.2x10^{-16}$ for recall, precision, and F-measure. Since the $p-Values$ are smaller than 0.05, we can state that there are differences among the algorithms for the performance indicators of recall, precision, and F-measure.

However, with the Quade test, we cannot know which of the algorithms gives the best performance. In this case, the performance of each algorithm should be individually compared against all of the other alternatives. In order to do this, we perform an additional post hoc analysis. This kind of analysis performs a pair-wise comparison among the results of each algorithm, determining whether statistically significant differences exist among the results of a specific pair of algorithms.

Table 4 shows the $p - Values$ of Holm's post hoc analysis for the case study and the performance indicators for the algorithms: EMoSim with the component-specific (EMoSim_CS) or the classic (EMoSim_C) genetic operations, BLiMEA (Baseline), and RS. All of the $p-Values$ obtained are smaller than their corresponding significance threshold value (0.05), indicating that the differences in performance are significant (highlighted in bold in the table), except when comparing precision in EMoSim_CS vs EMoSim_C and recall in EMoSim_C vs RS.

Table 4 Holm's post hoc $p - Values$ for each pair of algorithms in Kromaia.

	Recall	Precision	F-measure
EMoSim_CS vs EMoSim_C	$\mathbf{< 2x10^{-16}}$	0.83	$\mathbf{2.1x10^{-8}}$
EMoSim_CS vs Baseline	$\mathbf{< 2x10^{-16}}$	$\mathbf{3x10^{-5}}$	$\mathbf{< 2x10^{-16}}$
EMoSim_CS vs RS	$\mathbf{< 2x10^{-16}}$	$\mathbf{< 2x10^{-16}}$	$\mathbf{< 2x10^{-16}}$
EMoSim_C vs Baseline	$\mathbf{9.7x10^{-8}}$	$\mathbf{2.7x10^{-5}}$	$\mathbf{2.8x10^{-12}}$
EMoSim_C vs RS	0.89	$\mathbf{< 2x10^{-16}}$	$\mathbf{< 2x10^{-16}}$
Baseline vs RS	$\mathbf{2x10^{-9}}$	$\mathbf{2.6x10^{-16}}$	$\mathbf{3.2x10^{-14}}$

RQ$_3$ answer. There are differences in performance among the different categories of bugs in video games. The Action category (which includes the WPH and WPO types of bugs) obtained the best results in EMoSiM (using either the component-specific or the classic genetic operations) for precision and F-measure (obtaining 70.92 for component-specific and 59.88 for classic in F-measure). The lowest F-measure value was obtained in the Position of Object category (which includes the HNL, HMB and FLO types of bugs) in EMoSiM with the component-specific genetic operations (51.38) and in the Game Graphics category (which includes the BLI, HNL and TXG types of bugs) in EMoSiM with the classic genetic operations (45.86).

5.5.2 Effect Size

When comparing algorithms with a large enough number of runs, statistically significant differences can be obtained even if they are so small as to be of no practical value [21]. Thus, it is important to assess if an algorithm is statistically better than another and to assess the magnitude of the improvement. Effect size measures are needed to analyze this.

For a non-parametric effect size measure, we use Vargha and Delaney's $\hat{A}_{12}$ [30, 31]. $\hat{A}_{12}$ measures the probability that running one algorithm yields higher values than running another algorithm. If the two algorithms are equivalent, then $\hat{A}_{12}$ will be 0.5.

Table 5 shows the values of the effect size statistics between pair-wise comparisons of algorithms in Kromaia. Specifically, the upper part of the table shows the $\hat{A}_{12}$ values, whereas the lower part of the table shows Cliff's Delta [32] values for recall, precision, and F-measure.

RQ$_4$ answer. From the results, we can determine that the performance results obtained by EMoSim with the component-specific genetic operations, EMoSim with the classic genetic operations, the baseline, and RS are significant in F-measure. The magnitude of improvement using EMoSim instead of the baseline and RS can be interpreted as being large based on the magnitude scales [33] of the Cliff Delta values. Hence, EMoSim has an actual impact on performance. The highest differences between EMoSim and the baseline and RS are obtained when the component-specific operations are used, obtaining better results in F-measure for 97% and 100% of the runs, respectively.

6 Discussion

The results show that the component-specific genetic operations improve on the results of the classic genetic operations. This reveals that it is worth exploring other genetic operations that could obtain better results instead of selecting the classic genetic operations by default as we did in our previous work [14] (randomly combining all components of an individual as a whole) since it is the most popular choice in the Search-Based Model-Driven Engineering community [26].

The results also show that the six different categories of bugs (e.g., Action, Artificial Intelligence and Collision of Objects) obtain different values of recall, precision, and F-measure using either the EMoSim variant with the component-specific genetic operations or the classic genetic operations. This reveals that, in order to put the performance of different approaches into perspective, future research on BL in video games must report the categories of bugs of the taxonomy that have been used in their evaluation. It is not enough to talk about bugs in video games in general, since the nature of bugs can be influenced by various game genres, development tools, or modeling languages used in different games. While our study focuses on a broad categorization of bugs, the nuances and specificity related to each video game or the type of game development environment used could certainly influence bug patterns. These distinctions may warrant further exploration to understand the intricacies of bug occurrences in specific game contexts.

Table 5 Effect size measures for comparing each pair of algorithms in Kromaia.

	$\hat{A}_{12}$		
	Recall	Precision	F-measure
EMoSim_CS vs EMoSim_C	0.9378782	0.488252	0.6863653
EMoSim_CS vs Baseline	0.9911	0.8348167	0.9700961
EMoSim_CS vs RS	0.9446422	1	1
EMoSim_C vs Baseline	0.8252047	0.8131007	0.8636525
EMoSim_C vs RS	0.5229619	1	0.99288
Baseline vs RS	0.1573514	0.9676041	0.9302243
	Cliff's Delta		
	Recall	Precision	F-measure
EMoSim_CS vs EMoSim_C	0.8757565 (large)	-0.02349591 (negligible)	0.3727305 (medium)
EMoSim_CS vs Baseline	0.9822001 (large)	0.6696333 (large)	0.9401922 (large)
EMoSim_CS vs RS	0.8892844 (large)	1 (large)	1 (large)
EMoSim_C vs Baseline	0.6504094 (large)	0.6262015 (large)	0.7273051 (large)
EMoSim_C vs RS	0.04592382 (negligible)	1 (large)	0.9857601 (large)
Baseline vs RS	-0.6852973 (large)	0.9352083 (large)	0.8604486 (large)

The reason why EMoSim did not achieve better results (closer to 100% precision and recall) is because sometimes the bugs are related to parameters of model elements. In this work, the granularity of the simulation traces is at the model element level (not the parameter level). To improve the results, future research should extend the approach to work with the granularity that is at the level of the parameters of model elements. Moreover, future research should explore whether parameters of model elements should be taken into account to assess individuals by the fitness function, or by a multi-objective approach.

Despite the rise of video games and the arrival of the metaverse, video game research remains underexplored. Notable contributions to understanding the differences between Classic Software Engineering (CSE) and Game Software Engineering (GSE) include the work of Pascarella et al. [4] and Politowski et al. [10]. Our study builds on this by examining Bug Localization (BL) within GSE.

In CSE, bug localization traditionally relies on textual descriptions of bug reports and the defect localization principle [9, 18]. However, these methods fall short in the context of GSE. The baseline of our evaluation, which was successful in CSE for locating firmware bugs in BSH group products [15] (Bosch, Siemens, Gaggenau, Neff, Balay, among others) showed lower precision, recall, and F-measure in GSE. This discrepancy arises because bug reports and defect localization principles do not effectively guide bug localization in video games.

A key requirement for video games is to be fun, a factor often absent in traditional bug reports. Simulations can better reflect the fun factor by measuring how far they deviate from the ideal experience envisioned by developers. Our intuition is that incorporating fun-related heuristics can improve bug localization in video games, focusing on issues that frustrate players rather than technical errors like null pointer exceptions.

As occurs in CSE, GSE developers accelerate the development through the use of frameworks or libraries (see Fig. 1). This can potentially be a source of bugs. For example, overlapping weak points in a boss's hull can make an enemy indestructible (hulls can only be destroyed after the weak point is destroyed), blocking game progress. Unlike baseline approaches, our method detects such runtime interaction bugs through evolving simulations, providing insights missed by traditional methods. Nevertheless, more research on the bugs caused by the well-formedness of the models is needed.

EMoSim leverages Non-Playable Characters (NPCs) already present in game development, reducing the effort needed for simulations. These NPCs, designed for various in-game roles, can be used to simulate player interactions and identify bugs. While CSE lacks equivalent agents to NPCs, future research could explore building similar agents for simulation-based bug localization.

Furthermore, we ran a focus group to acquire feedback from four software engineers of the industrial partner. One of them has been developing video games for 15 years, two have developed video games for six years, and the last one only has two years of experience developing video games. They all participated in the development of Kromaia, either from its inception (the most experienced developer) or creating new content for the game (the other three developers). Specifically, the focus group was composed of the following open questions: (1) What do you think of the results of the approaches?; (2) How do you feel about locating bugs in video games using simulation traces?; and (3) How do you imagine the use of EMoSim in video games of other genres and in more complex video games?

The engineers stated that the results of EMoSim were far superior to the results of the baseline. In their opinion, the idea of the baseline to favor results related to the latest modifications does not help find errors. In the event of a bug, the developers recognized that the first thing that they intuitively do is to check the latest modifications; however, in their experience, these are not usually the source of bugs since many of the bugs go unnoticed until the game is completed and played from start to finish.

On the other hand, the developers found the information of the traces to be very relevant in locating bugs. All four agreed that this meant moving from a bug localization based on gut-feeling to a bug localization based on evidence. In their opinion, the information of the traces is underutilized and traces contain latent information about the frustrating or difficult moments for the player.

The engineers also mentioned that EMoSim can be used for other video game genres and more complex problems (such as locating bugs in whole levels). To do this, they informally revisited some game genres (e.g., first person shooters, fighting games, or strategy games) to conclude that they provide the basic ingredients for applying EMoSim. They imagined how they are going to use the non-playable characters (known as NPCs in the video game domain) of those games as part of the simulation for the fitness function of EMoSim. The less experienced developer also stressed the importance of the presentation of the information of traces. In his opinion, it would help to convert the traces into heat maps on the software models where the model elements have different colors based on the number of occurrences in the trace.

Finally, we revisited the developers of the industrial partner 15 months later. The developers stated that the ideas of addressing simulations by the different components (simulated player, NPC and itinerary), which are titled as component-specific genetic operations in this work, are taken for granted nowadays. However, the developers acknowledged that these ideas were new to them until we presented this work. One of the strong points of this work is that it does not need additional developments. This means that the elements of the simulations are built naturally as part of the development. Nowadays, the developers recognize that they naturally separate the components of the simulations like our component-specific genetic operations do when they develop new video games. Hence, the strength of this work of minimizing overhead is maintained with regard to the natural development of video games.

7 Threats to Validity

To acknowledge the threats to the validity of our work, we use the classification suggested by De Oliveira et al. [34].

1) Conclusion Validity Threats. We considered random variation by executing the approaches 30 times for each bug as suggested in [21]. We used measurements that are widely accepted in the software engineering research community (recall, precision, and F-measure) [35] to analyze the obtained confusion matrix, and we showed the average of the results. We also used a statistical test (Quade test) and effect size measurements ($\hat{A}_{12}$ and Cliff's Delta) following accepted guidelines [35]. We addressed the lack of a meaningful comparison baseline by comparing the results obtained in the two variants of our EMoSim approach with a baseline and a sanity check.

2) Internal Validity Threats. We used values from the literature for the approaches to address the poor parameter settings. As suggested by Arcuri and Fraser [35], default values are good enough to measure the performance. We also used two main indicators (victory and health) to calculate the fitness of a simulation as performed in [25].

To address the lack of real problem instances, the evaluation of our work was performed using a commercial video game, and the problem artifacts were directly obtained from the developers and the data repository of the game, such as the catalogue of values that is used in the genetic operations and the software models. We randomly selected six bugs for each type of the nine types of bug from the entire data repository because the number of bugs in each category is similar. However, further research should be done in this direction.

3) Construct Validity Threats. We addressed the threat of the lack of assessing the validity of cost measures by performing a fair comparison of our approach with the baseline and the sanity check. Moreover, our evaluation was performed using three measurements (recall, precision, and F-measure) that are widely used in the software engineering research community [35].

4) External Validity Threats. Our approach was evaluated in a commercial video game, whose instances were collected from real-world problems to mitigate the threat of the lack of a clear object selection strategy. To mitigate the generalization threat, our approach has been designed to be generic and applicable not only to the Kromaia video game but also for locating bugs in other different video games. Our approach can be applied if the software model conforms to MOF (the OMG metalanguage for defining modeling languages) and the simulated players are available. While the software model conforms to MOF, our implementation uses reflective methods provided by the Eclipse Modeling Framework. Thus, the text elements that are associated to the models are extracted automatically independently of the metamodel that is used. From the information that is extracted from the model, the population is initialized taking into account the encoding provided (e.g., how many parameters are in an individual). In addition to the encoding, our approach requires the other two main ingredients of SBSE approaches: operators and fitness function. The operators are the component-specific or the classic crossover and mutation operations. The encoding and the fitness function depend on the simulated player. We can apply our approach to other video games where simulated players are available. These simulated players are available in popular game genres such as car games (rival drivers), FPS games (bots), or RTS games (rival generals). Once the main ingredients of SBSE approaches are provided, the evolutionary algorithm (as used in our implementation) can be executed. Nevertheless, our results should be replicated with other video games and with other classifications of types of bugs before ensuring their external validity. For those cases where there is no simulated player, the developers should ponder the tradeoff of the cost of developing the simulated player and the benefits of locating bugs with our approach.

8 Related Work

Table 6 shows the related work, which takes into account the topics covered in this paper: bug localization in games, bug localization in models, and bug localization in games that use models. The upper part of Table 6 includes 24 bug localization works in chronological order, from 2015 until June 2024. Also, the lower part of the table includes two rows to compare our previous works [14, 59] as well as a row to compare this work with the other bug localization works. The columns of the table show: the related work (Column 1); the year of publication (Column 2); where it covers bug localization whether in games, in models, or in games using models (Columns 3-5); if doing bug localization in games, we classify which types of bugs are covered using the classification of the taxonomy of bugs in video games [20] (Column 6); if the fitness function is a simulation (Column 7); and if the evaluation performed explicitly mentions that industry is involved (Column 8). In each cell of the table, except columns 2 and 6, we use either a check mark (to indicate that the work explicitly addresses what is mentioned in the column) or a cross mark otherwise.

8.1 Bug Localization in Games

Some papers do bug localization in games as shown in Table 6. For instance, Ariyurek et al. [41] developed gameplay agents that are governed by reinforcement learning (RL), Monte Carlo Tree Search (MCTS), and inverse RL to mimic human behaviour in simple action-adventure games, with the goal of identifying bugs. In general, the authors found that the agents were capable of matching or even outperforming human testers

Table 6 Related work in bug localization.

	Year	Bug localization in			Categories of bugs found (in games)	Simulations as fitness function	Industrial scale
		Games	Models	Games that use models			
Burgueño et al. [36]	2015	✗	✓	✗	–	✗	✗
Iftikhar et al. [37]	2015	✗	✗	✓	Context State, Position of Object	✗	✓
Sánchez-Cuadrado et al. [38]	2017	✗	✓	✗	–	✗	✗
Sánchez-Cuadrado et al. [39]	2018	✗	✓	✗	–	✗	✗
Troya et al. [40]	2018	✗	✓	✗	–	✗	✗
Ariyurek et al. [41]	2019	✓	✗	✗	Action, Crash, Event Occurrence	✗	✗
Zheng et al. [42]	2019	✓	✗	✗	Action, Audio, Crash, Game Graphics, Value	✗	✓
Ahumada and Bergel [43]	2020	✓	✗	✗	Action, Triggered Event	✗	✗
Ariyurek et al. [44]	2020	✓	✗	✗	Action, Crash, Event Occurrence	✗	✗
Bergdahl et al. [45]	2020	✓	✗	✗	Action, Exploit	✗	✗
Cheng et al. [46]	2020	✗	✓	✗	–	✗	✗
Wu et al. [47]	2020	✓	✗	✗	Context State, Crash	✗	✓
Zhang et al. [48]	2020	✗	✓	✗	–	✗	✗
Ferdous et al. [49]	2021	✗	✗	✓	Action, Triggered Event	✗	✗
Quach et al. [50]	2021	✗	✓	✗	–	✗	✗
Arcega et al. [16]	2022	✗	✓	✗	–	✗	✓
Ciborowska et al. [51]	2022	✗	✓	✗	–	✗	✓
Khanfir [52]	2022	✗	✓	✗	–	✗	✓
Liang et al. [53]	2022	✗	✓	✗	–	✗	✓
Tufano et al. [54]	2022	✓	✗	✗	Implementation Response	✗	✗
Liu et al. [55]	2023	✗	✓	✗	–	✗	✓
Politowski et al. [56]	2023	✓	✗	✗	Action, Context State, Interaction Between Object Properties	✗	✗
Pérez et al. [57]	2023	✗	✓	✗	–	✗	✓
Roca et al. [58]	2024	✗	✗	✓	Action, Artificial Intelligence, Game Graphics, Position of Object	✓	✓
Our previous work [14]	2022	✗	✗	✓	Action, Artificial Intelligence, Game Graphics, Position of Object	✓	✓
Our previous work [59]	2023	✗	✗	✓	Action, Artificial Intelligence, Game Graphics, Position of Object	✓	✓
This work (extension)	2024	✗	✗	✓	Action, Artificial Intelligence, Game Graphics, Position of Object, Collision of Objects, Interaction Between Object Properties	✓	✓

in terms of the errors found. In their following work [44], for game testing purposes, they extended the MCTS agent with several modifications (Transpositions, Knowledge-Based Evaluations, Tree Reuse, MixMax, Boltzmann Rollout, Single Player MCTS, and Computational Budget). Their results showed that MCTS modifications improve the bug-finding performance of the agents.

Zheng et al. [42] rely on deep reinforcement learning (DRL) to make progress in automated video game testing. However, the challenge with the existing DRLs is that most of them focus on winning the game rather than game testing. Their work focuses on the balance between winning the game (i.e., advancing in the game) and exploring the game space (i.e., increasing the possibility of discovering bugs). They mainly leverage evolutionary algorithms and multi-objective optimization to explore the game space by optimizing the population iteratively, so more states of the game could be explored and tested, whereas DRL contributes to accomplishing the mission. They evaluate its effectiveness in two real-world commercial video games, having previously performed an empirical study to characterize game bugs by analyzing a total of 1,349 real bugs from four industrial games. They leverage the game itself as the simulation, whereas EMoSim leverages NPC-based game simulations, which isolate and scrutinize aspects of gameplay dynamics (such as the duel between the boss and the simulated player) for bug localization. The information obtained from the simulation is used by EMoSim as fitness function to locate the model fragments that are the source of bugs.

Tufano et al. [54] presented RELINE, which is an approach that uses RL to load test video games. RELINE can be instantiated on different games using different RL models and reward functions. Their proof-of-concept study performed on two subject systems shows the feasibility of their approach: Given a reward function that is able to reward the agent when artificial performance bugs are identified, the agent adapts its behavior continuing to play the game, whereas EMoSim looks for those bugs.

Politowski et al. [56] suggest an approach to automate game testing to balance video games

with autonomous agents by comparing the difficulty levels between game versions and issues with the game design, and the game demands for skill or luck.

The above approaches do not take into account software models as the source of the bugs. Models are used in many video game developments; hence, the models can be the source of the bugs.

8.2 Bug Localization in Models

Outside the game domain, several techniques are used for bug localization in models. For example, Arcega et al. [16] evaluate how to apply the existing model-based approaches in order to mitigate the effect of starting the localization in the wrong place. They also take into account that software engineers can refine the results at different stages. They compare different combinations of the application of bug localization approaches and human refinement. The combination of their approaches together with manual refinement obtains the best results.

Troya et al. [40] present an approach to apply Spectrum-Based Fault Localization (SBFL) for locating the faulty rules in model transformations. Their approach takes advantage of the information recovered after the model transformation is run. The inputs of their approach are a model transformation, a set of assertions, and a set of source models. Their approach finds the violated assertions and uses the information of the model transformation coverage to rank the transformation rules according to their suspiciousness of containing a bug.

Sánchez-Cuadrado et al. [38] combine static analysis and constraint solving to discover errors in ATL transformations. They developed a tool that uses static analysis to detect problems based on the textual information and generates witness models using OCL path conditions and constraint solving. In their subsequent works, they present an approach that proposes suitable quick fixes for ATL transformation errors [39]. Their approach performs speculative analysis to provide information on the impact of the application of each applicable quick fix and generates a dynamic quick fix rank. In addition, they constructed a static ranking empirically through the automated application of quick fixes on transformations.

Burgueño et al. [36] present a static approach to trace errors in model transformations, taking as input an ATL model transformation and a set of constraints that specify its expected behavior. Their approach automatically extracts the footprints of both artifacts and compares transformation rules and constraints one by one, obtaining the overlap of common footprints. The output is three matching tables that can be used by software engineers to trace the rules that might be the cause of broken constraints due to faulty behavior.

Pérez et al. [57] analyze how collaboration affects maintenance tasks such as Traceability Link Recovery (TLR), Bug Localization (BL), and Feature Location (FL) on software models in the context of a worldwide industrial supplier of railway solutions.

The above approaches and the approaches that are included in Table 6 that take bug localization in models into account are not specific to video game development. They were not designed with the peculiarities of video games in mind nor have they ever been evaluated in video games.

8.3 Bug Localization in Games That Use Models

Others do bug localization in games that use models as Table 6 shows. Iftikhar et al. [37] propose a software model-based methodology for automated video game testing. They use UML class diagrams and UML state machines for the modeling (domain and behavioral, respectively). Their approach automates test case generation, execution, and oracle generation. They conduct their evaluation using two platform games.

Ferdous et al. [49] present a search-based test generation approach applied to software models. They capture an abstraction of the desired game behavior in an extended finite state machine (EFSM) and derive abstract tests of the software model using search-based algorithms, which are then specified into action sequences that are executed in the game under test. They used a 3D game to evaluate the suitability of the approach and five search algorithms for test generation on three different software models of the game.

These approaches rely on UML or state machines, whereas our approach is not restricted to these models. As far as we could determine,

there are very few studies in game software engineering on the use of software models as the source of bugs or performing the testing taking into account software models as the main artifact.

Among the bug localization works that are included in Table 6, 29.63% do it in games, 48.15% do it in models, but only two other works prior to ours do it in games that use models. These works that do bug localization in games that use models (Iftikhar et al. [37] and Ferdous et al. [49]) locate only two categories of bugs, "Context State" and "Position of Object", and "Action" and "Triggered Event", respectively. Furthermore, only the former does so on an industrial scale. Our previous work in BL in games that use models [14] includes the novelty of using a fitness function that collects information about the simulations. It covers more categories of bugs than the previous works [37, 49] (four categories instead of two), and three of these categories (Artificial Intelligence, Game Graphics and Position of Object) have not been covered previously in bug localization works in games that use models. Roca et al. [58] use genetic operations that 1) combine all components of the simulation as a whole (unlike our approach); and 2) explore the interaction between different models, which is out of the scope of this work.

Our previous work [59] uses our EMoSim approach with the classic genetic operations as baseline [14] in order to study whether involving the human in a hybrid fitness function influences the quality of bugs being located without taking into account the bug types.

This work extends our previous work [14] with the classification of types of bugs in the existing taxonomy of bugs in video games to facilitate future comparisons of our results with other works, and it adds more bugs to cover 50% more categories of bugs of the taxonomy. These additional categories have not been covered previously in bug localization works in games that use models. Also, this work proposes novel component-specific genetic operations, and it compares the results that are obtained in the categories of bugs using the classic genetic operations, the baseline, and RS.

9 Conclusion

For years, bug reporting and the defect localization principle have proven to be useful for locating bugs in software. Bug localization in Game Software Engineering has received little attention despite the rise of video games and the problems that their developers have in locating bugs.

Our work presents a classification of bug types in the context of video games within the categories identified in a recent taxonomy. This facilitates future comparisons of our work with other works that use the same classification.

Also, our approach evolves simulations and uses novel component-specific genetic operations that produce relevant traces to locate bugs. The results show that our approach, using either the classic genetic operations or the novel component-specific operations, outperforms the traditional method of relying solely on bug reports and the defect localization principle to locate bugs in video games.

We propose a novel route to locate bugs in video games not only by evolving video game simulations that produce traces that are relevant to locating bugs, but also by exploring other genetic operations that do not randomly combine all components as a whole. To locate bugs, we leverage non-player characters, which is one of the key features that are inherent to the video game domain. Our approach involves the use of non-player characters to generate simulations that consider the different components of the game in order to evolve simulations rather than randomly combining them. The positive feedback received from a focus group indicates the acceptance of our proposal. Our discussion of the results contributes to advancing the understanding of bugs in video games.

Furthermore, our work suggests that evolving simulations can also be applied to Classic Software Engineering to locate bugs (e.g., simulation agents that represent humans for testing UI, or simulated transitions in state machines), which expands the scope of our approach beyond the video game domain. We hope that our research inspires more studies and collaborations to address the challenges of bug localization in video games that could also potentially be used in Classic Software Engineering. We also hope that our work motivates other researchers to consider alternative

genetic operations (instead of selecting the classic genetic operations by default) to potentially obtain better the results.

Finally, the study of the effect of our approach remains an open challenge for the field of Automated Program Repair (APR). APR aims to reduce the cost of fixing bugs by automatically producing patches. In APR, the automated localization of bugs is essential. The accuracy of the localization used by APR may have a significant effect on the success of APR. However, most APR tools require the location to be done on code statements instead of considering other software artifacts such as model elements. Moreover, not only can taking advantage of evolving simulations be relevant for locating bugs in software models, but it can also be relevant for fixing the located bugs automatically.

Acknowledgments. This work was supported 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 PID2021-128695OB-I00 and in part by the Gobierno de Aragón (Spain) (Research Group T61_23R). Moreover, this work was partially supported by the Spanish Ministry of Science and Innovation under the Excellence Network AI4Software (Red2022-134647-T).

References

[1] SlashData, "Global developer population report 2019," https://sdata.me/GlobalDevPop19, 2019, [Online; accessed 21-November-2021].

[2] Y. K. Dwivedi, D. L. Hughes, A. M. Baabdullah, S. Ribeiro-Navarrete, M. Giannakis, M. M. Al-Debei, D. Dennehy, B. A. Metri, D. Buhalis, C. M. K. Cheung, K. Conboy, R. Doyle, R. Dubey, V. Dutot, R. Felix, D. P. Goyal, A. Gustafsson, C. Hinsch, I. Jebabli, M. Janssen, Y. Kim, J. Kim, S. Koos, D. Kreps, N. Kshetri, V. Kumar, K. Ooi, S. Papagiannidis, I. O. Pappas, A. Polyviou, S. Park, N. Pandey, M. M. Queiroz, R. Raman, P. A. Rauschnabel, A. Shirish, M. Sigala, K. Spanaki, G. W. Tan, M. K. Tiwari, G. Viglia, and S. F. Wamba, "Metaverse beyond the hype: Multidisciplinary perspectives on emerging challenges, opportunities, and agenda for research, practice and policy," *Int. J. Inf. Manag.*, vol. 66, p. 102542, 2022. [Online]. Available: https://doi.org/10.1016/j.ijinfomgt.2022.102542

[3] H. Wang, H. Ning, Y. Lin, W. Wang, S. Dhelim, F. Farha, J. Ding, and M. Daneshmand, "A survey on the metaverse: The state-of-the-art, technologies, applications, and challenges," *IEEE Internet Things J.*, vol. 10, no. 16, pp. 14 671–14 688, 2023. [Online]. Available: https://doi.org/10.1109/JIOT.2023.3278329

[4] L. Pascarella, F. Palomba, M. D. Penta, and A. Bacchelli, "How is video game development different from software development in open source?" in *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, A. Zaidman, Y. Kamei, and E. Hill, Eds. ACM, 2018, pp. 392–402.

[5] Unity Technologies, "Unity," https://unity.com, 2005, [Online; accessed 21-November-2021].

[6] Epic Games, "Unreal Engine," https://www.unrealengine.com, 1998, [Online; accessed 21-November-2021].

[7] Crytek, "CryEngine," https://www.cryengine.com, 2002, [Online; accessed 21-November-2021].

[8] M. Zhu and A. I. Wang, "Model-driven game development: A literature review," *ACM Comput. Surv.*, vol. 52, no. 6, pp. 123:1–123:32, 2020.

[9] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Trans. Software Eng.*, vol. 42, no. 8, pp. 707–740, 2016. [Online]. Available: https://doi.org/10.1109/TSE.2016.2521368

[10] C. Politowski, F. Petrillo, and Y. Guéhéneuc, "A survey of video game testing," in *2nd*

IEEE/ACM International Conference on Automation of Software Test, AST@ICSE 2021, Madrid, Spain, May 20-21, 2021. IEEE, 2021, pp. 90–99.

[11] P. Gamer, "How buggy is Cyberpunk 2077, really?" https://www.pcgamer.com/how-buggy-is-cyberpunk-2077-really/, 2020, [Online; accessed 21-November-2021].

[12] Steam Community, "Cyberpunk 2077 bug reports and discussions," 2020, accessed: 2024-06-20. [Online]. Available: https://steamcommunity.com/app/1091500/discussions/0/2996548763043050958/?l=spanish&tscn=1609341594

[13] A. Ampatzoglou and I. Stamelos, "Software engineering research for computer games: A systematic review," *Information and Software Technology*, vol. 52, no. 9, pp. 888–901, 2010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584910000820

[14] R. Casamayor, L. Arcega, F. Pérez, and 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 2022, Montreal, Quebec, Canada, October 23-28, 2022*, E. Syriani, H. A. Sahraoui, N. Bencomo, and M. Wimmer, Eds. ACM, 2022, pp. 356–366. [Online]. Available: https://doi.org/10.1145/3550355.3552440

[15] L. Arcega, J. Font, Ø. Haugen, and C. Cetina, "An approach for bug localization in models using two levels: model and metamodel," *Software and Systems Modeling*, vol. 18, no. 6, pp. 3551–3576, 2019.

[16] ——, "Bug localization in model-based systems in the wild," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 1, pp. 10:1–10:32, 2022. [Online]. Available: https://doi.org/10.1145/3472616

[17] A. E. Hassan and R. C. Holt, "The top ten list: Dynamic fault prediction," in *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ser. ICSM '05. USA: IEEE Computer Society, 2005, p. 263–272. [Online]. Available: https://doi.org/10.1109/ICSM.2005.91

[18] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 563–572. [Online]. Available: http://dl.acm.org/citation.cfm?id=998675.999460

[19] B. Sisman and A. C. Kak, "Incorporating Version Histories in Information Retrieval Based Bug Localization," in *9th IEEE Working Conference on Mining Software Repositories*, 2012.

[20] A. Truelove, E. S. de Almeida, and I. Ahmed, "We'll fix it in post: What do bug fixes in video game update notes tell us?" in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 736–747. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00073

[21] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1486

[22] Á. Domingo, J. Echeverría, O. Pastor, and C. Cetina, "Evaluating the benefits of model-driven development - empirical evaluation paper," in *Advanced Information Systems Engineering - 32nd International Conference, CAiSE 2020, Grenoble, France, June 8-12, 2020, Proceedings*, ser. Lecture Notes in Computer Science, S. Dustdar, E. Yu, C. Salinesi, D. Rieu, and V. Pant, Eds., vol. 12127. Springer, 2020, pp. 353–367.

[23] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, vol. 33, no. 1, 1977.

[24] I. Boussaïd, P. Siarry, and M. Ahmed-Nacer, "A survey on search-based model-driven engineering," *Automated Software Engineering*, vol. 24, no. 2, pp. 233–294, Jun 2017. [Online]. Available: https://doi.org/10.1007/s10515-017-0215-4

[25] D. Blasco, J. Font, M. Zamorano, and C. Cetina, "An evolutionary approach for generating software models: The case of Kromaia in Game Software Engineering," *J. Syst. Softw.*, vol. 171, p. 110804, 2021. [Online]. Available: https://doi.org/10.1016/j.jss.2020.110804

[26] F. Pérez, T. Ziadi, and C. Cetina, "Utilizing automatic query reformulations as genetic operations to improve feature location in software models," *IEEE Trans. Software Eng.*, vol. 48, no. 2, pp. 713–731, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2020.3000520

[27] H. Ishibuchi, Y. Nojima, and 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.

[28] S. García, A. Fernández, J. Luengo, and 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*, vol. 180, no. 10, pp. 2044–2064, 2010, special Issue on Intelligent Distributed Information Systems. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020025509005404

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

[30] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[31] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*. New Jersey, United States: Lawrence Erlbaum Associates Publishers, 2005.

[32] N. Cliff, *Ordinal methods for behavioral data analysis*. Lawrence Erlbaum Associates, Inc, 1996.

[33] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys?" in *annual meeting of the Florida Association of Institutional Research*, 2006, pp. 1–3.

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

[35] A. Arcuri and G. Fraser, "Parameter tuning or default values? an empirical investigation in search-based software engineering," *Empirical Software Engineering*, vol. 18, pp. 594–623, 2013.

[36] L. Burgueño, J. Troya, M. Wimmer, and A. Vallecillo, "Static fault localization in model transformations," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 490–506, May 2015.

[37] S. Iftikhar, M. Z. Iqbal, M. U. Khan, and W. Mahmood, "An automated model based testing approach for platform games," in *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, ON, Canada, September 30 - October 2, 2015*, T. Lethbridge, J. Cabot, and A. Egyed, Eds. IEEE Computer Society, 2015, pp. 426–435. [Online]. Available: https://doi.org/10.1109/MODELS.2015.7338274

[38] J. Sánchez Cuadrado, E. Guerra, and J. de Lara, "Static analysis of model transformations," *IEEE Transactions on Software Engineering*, vol. 43, no. 9, pp. 868–897, 2017.

[39] ——, "Quick fixing atl transformations with speculative analysis," *Softw. Syst. Model.*, vol. 17, no. 3, p. 779–813, Jul. 2018. [Online]. Available: https://doi.org/10.1007/s10270-016-0541-1

[40] J. Troya, S. Segura, J. A. Parejo, and A. Ruiz-Cortés, "Spectrum-based fault localization in model transformations," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 3, Sep. 2018. [Online]. Available: https://doi.org/10.1145/3241744

[41] S. Ariyurek, A. Betin-Can, and E. Surer, "Automated video game testing using synthetic and humanlike agents," *IEEE Transactions on Games*, vol. 13, no. 1, pp. 50–67, 2019.

[42] Y. Zheng, C. Fan, X. Xie, T. Su, L. Ma, J. Hao, Z. Meng, Y. Liu, R. Shen, and Y. Chen, "Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning," in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 772–784. [Online]. Available: https://doi.org/10.1109/ASE.2019.00077

[43] T. Ahumada and A. Bergel, "Reproducing bugs in video games using genetic algorithms," in *2020 IEEE Games, Multimedia, Animation and Multiple Realities Conference (GMAX)*, 2020, pp. 1–6.

[44] S. Ariyurek, A. Betin-Can, and 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. [Online]. Available: https://doi.org/10.1109/CoG47356.2020.9231670

[45] J. Bergdahl, C. Gordillo, K. Tollmar, and L. Gisslén, "Augmenting automated game testing with deep reinforcement learning," in *IEEE Conference on Games, CoG 2020, Osaka, Japan, August 24-27, 2020*. IEEE, 2020, pp. 600–603. [Online]. Available: https://doi.org/10.1109/CoG47356.2020.9231552

[46] X. Cheng, N. Liu, L. Guo, Z. Xu, and T. Zhang, "Blocking bug prediction based on xgboost with enhanced features," in *44th IEEE Annual Computers, Software, and Applications Conference, COMPSAC 2020, Madrid, Spain, July 13-17, 2020*. IEEE, 2020, pp. 902–911. [Online]. Available: https://doi.org/10.1109/COMPSAC48688.2020.0-152

[47] Y. Wu, Y. Chen, X. Xie, B. Yu, C. Fan, and L. Ma, "Regression testing of massively multiplayer online role-playing games," in *IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020*. IEEE, 2020, pp. 692–696. [Online]. Available: https://doi.org/10.1109/ICSME46990.2020.00074

[48] J. Zhang, R. Xie, W. Ye, Y. Zhang, and S. Zhang, "Exploiting code knowledge graph for bug localization via bi-directional attention," in *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*. ACM, 2020, pp. 219–229. [Online]. Available: https://doi.org/10.1145/3387904.3389281

[49] R. Ferdous, F. M. Kifetew, D. Prandi, I. S. W. B. Prasetya, S. Shirzadehhajimahmood, and A. Susi, "Search-based automated play testing of computer games: A model-based approach," in *Search-Based Software Engineering - 13th International Symposium, SSBSE 2021, Bari, Italy, October 11-12, 2021, Proceedings*, ser. Lecture Notes in Computer Science, U. O'Reilly and X. Devroey, Eds., vol. 12914. Springer, 2021, pp. 56–71. [Online]. Available: https://doi.org/10.1007/978-3-030-88106-1_5

[50] S. Quach, M. Lamothe, B. Adams, Y. Kamei, and W. Shang, "Evaluating the impact of falsely detected performance bug-inducing changes in JIT models," *Empir. Softw. Eng.*, vol. 26, no. 5, p. 97, 2021. [Online]. Available: https://doi.org/10.1007/s10664-021-10004-6

[51] A. Ciborowska and K. Damevski, "Fast changeset-based bug localization with

BERT," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 946–957. [Online]. Available: https://doi.org/10.1145/3510003.3510042

[52] A. Khanfir, "Effective and scalable fault injection using bug reports and generative language models," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, A. Roychoudhury, C. Cadar, and M. Kim, Eds. ACM, 2022, pp. 1790–1794. [Online]. Available: https://doi.org/10.1145/3540250.3558907

[53] H. Liang, D. Hang, and X. Li, "Modeling function-level interactions for file-level bug localization," *Empir. Softw. Eng.*, vol. 27, no. 7, p. 186, 2022. [Online]. Available: https://doi.org/10.1007/s10664-022-10237-z

[54] R. Tufano, S. Scalabrino, L. Pascarella, E. Aghajani, R. Oliveto, and G. Bavota, "Using reinforcement learning for load testing of video games," in *Proceedings of the 44th International Conference on Software Engineering (ICSE 2022), Pittsburgh, USA*, 2022.

[55] D. Liu, Y. Feng, Y. Yan, and B. Xu, "Towards understanding bugs in python interpreters," *Empir. Softw. Eng.*, vol. 28, no. 1, p. 19, 2023. [Online]. Available: https://doi.org/10.1007/s10664-022-10239-x

[56] C. Politowski, F. Petrillo, G. El-Boussaidi, G. C. Ullmann, and Y. Guéhéneuc, "Assessing video game balance using autonomous agents," in *7th IEEE/ACM International Workshop on Games and Software Engineering, GAS@ICSE 2023, Melbourne, Australia, May 15, 2023*. IEEE, 2023, pp. 25–32. [Online]. Available: https://doi.org/10.1109/GAS59301.2023.00011

[57] F. Pérez, R. Lapeña, A. C. Marcén, and C. Cetina, "How the quality of maintenance tasks is affected by criteria for selecting engineers for collaboration," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 3, pp. 63:1–63:22, 2023. [Online]. Available: https://doi.org/10.1145/3561384

[58] I. Roca, Ó. Pastor, C. Cetina, and L. Arcega, "Co-evolving scenarios and simulated players to locate bugs that arise from the interaction of software models of video games," *Inf. Softw. Technol.*, vol. 169, p. 107412, 2024. [Online]. Available: https://doi.org/10.1016/j.infsof.2024.107412

[59] R. Casamayor, C. Cetina, O. Pastor, and F. Pérez, "Studying the influence and distribution of the human effort in a hybrid fitness function for search-based model-driven engineering," *IEEE Trans. Software Eng.*, vol. 49, no. 12, pp. 5189–5202, 2023. [Online]. Available: https://doi.org/10.1109/TSE.2023.3329730