Bug Localization in Game Software Engineering: Evolving Simulations to Locate Bugs in Software Models of Video Games

Rodrigo Casamayor, Lorena Arcega, Francisca Pérez, Carlos Cetina
rcasamayor@acm.org,{larcega,mfperez,ccetina}@usj.es
Universidad San Jorge. Escuela de Arquitectura y Tecnología
Zaragoza, Spain

ABSTRACT

Video games have characteristics that differentiate their development and maintenance from classic software development and maintenance. These differences have led to the coining of the term Game Software Engineering to name the emerging subfield that intersects Software Engineering and video games. One of these differences is that video game developers perceive more difficulties than other non-game developers when it comes to locating bugs. Our work proposes a novel way to locate bugs in video games by means of evolving simulations. As the baseline, we have chosen BLiMEA, which targets classic software engineering and uses bug reports and the defect localization principle to locate bugs. We also include Random Search as a sanity check in the evaluation. We evaluate the approaches in a commercial video game (Kromaia). The results for F-measure range from 46.80%. to 70.28% for five types of bugs. Our approach improved the results of the baseline by 20.29% in F-measure. To the best of our knowledge, this is the first approach that is designed specifically for bug localization in video games. A focus group with professional video game developers has confirmed the acceptance of our approach. Our approach opens a new research direction for bug localization for both game software engineering and possibly classic software engineering.

CCS CONCEPTS

• **Software and its engineering → Search-based software engineering**; **Software maintenance tools**.

KEYWORDS

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

ACM Reference Format:
Rodrigo Casamayor, Lorena Arcega, Francisca Pérez, Carlos Cetina. 2022. Bug Localization in Game Software Engineering: Evolving Simulations to Locate Bugs in Software Models of Video Games. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22), October 23–28, 2022, Montreal, QC, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3550355.3552440

MODELS '22, October 23–28, 2022, Montreal, QC, Canada

ACM ISBN 978-1-4503-9466-6/22/10…$15.00
https://doi.org/10.1145/3550355.3552440

1 INTRODUCTION

Today, video game development is one of the fastest growing industries in the world. Such is the relevance and depth that video games have in our society that, if we put it in terms of developer population, the video game industry is responsible for 8.8M active developers as of 2019 [29]. 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. This might suggest that the number of video game developers will continue to grow in the future as the Metaverse is developed.

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 (e.g., shaders, meshes, or prefabs). In addition, game developers perceive more difficulties than other non-game developers when locating bugs as well as reusing code [24].

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 [30] and Unreal Engine [18], but it is also possible for a studio to make its own specific engine (e.g., CryEngine [13]).

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

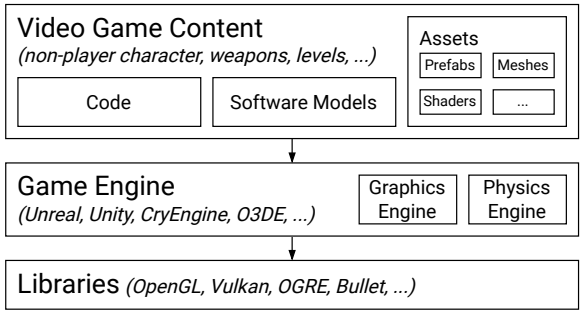

Figure 1: Overview of video game artifacts.

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 Figure 1). Unity and Unreal propose their own modeling language, and a recent survey in Model-Driven Game Development [36] reveals that UML and Domain Specific Language (DSL) models are also adopted by development teams.

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 [24]. Nevertheless, a fault localization survey [34] 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 [25].

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* [17]. 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.

There are significant differences between Classic Software Engineering and Game Software Engineering [1, 25]. Our work argues that the differences can become opportunities for tackling the challenge of bug localization in video games. Specifically, we propose leveraging game simulations to locate bugs in the software models of video games. In video games, it is common to include non-player characters (NPCs). They accompany the player in the adventure, are the enemies to beat, or simply populate the world recreated in the video game. These NPCs have pre-programmed behaviors and can be used to launch gameplay simulations. For example, in a first-person shooter game (such as the popular *Doom* video game), NPCs roam the levels looking for weapons and power-ups to confront other NPCs or the player himself.

In this paper, we propose an evolutionary algorithm for bug localization in software models that leverages model simulations. Our approach, called EMoSim, evolves simulations that produce traces that are relevant for locating bugs. To put our approach in perspective, we have compared it with a baseline, BLiMEA [2, 3]. BLiMEA uses bug reports and the defect localization principle[21] (as many bug localization approaches do [34]), and it is specifically designed to locate bugs in software models. We also included Random Search as a sanity check in the evaluation. We have done one evaluation using the Kromaia case study. Kromaia is a video game about flying and shooting with a spaceship in a three-dimensional space [1]. It was released on PC, PlayStation, and translated to eight different languages. The metrics applied are widely accepted by the software engineering community in the domain of evolutionary algorithms, such as precision, recall, and F-measure.

The results show that EMoSiM outperforms the baseline and the random search approaches. The results for F-measure range from 46.80%. to 70.28% for five types of bugs. Our approach improved the results of the baseline by 20.29% in F-measure. The statistical analysis performed provides quantitative evidence of the impact of the approach and indicates that this impact is significant. Furthermore, a focus group has confirmed the acceptance of our approach.

To the best of our knowledge, this is the first approach that specifically deals with bug localization in video games. 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 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, similar agents 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 bugs. In Section 3, we describe our approach, EMoSim. In Section 4, we evaluate our approach in Kromaia and discuss the results. In Section 5, we examine the related work of the area. Finally, we present our conclusions in Section 6.

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. Each of the levels involves a player's spaceship flying from a starting point to a target destination reaching the goal before being destroyed. The level involves exploring floating structures, avoiding asteroids, and finding items along the route, while protected by basic enemies that try to damage the player's spaceship by firing projectiles. 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.

The bosses are specified with the Shooter Definition Model Language (SDML). SDML is a DSL model for the video game domain. This DSL follows the main ideas of MDE using models for Software Engineering. The models are created using SDML and interpreted at runtime. Specifically, SDML defines 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. Examples of the models, 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.

By leveraging game simulations, we want to locate the bugs that are related to the models that specify the bosses using SDML. The simulations used in this work simulate a duel between a boss and

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

a human player. The simulated player is an algorithm that is able to act like a human player. It was created by the developers of 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.

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 have unpredictable behavior.

It is important to clarify that bosses can be built either using SDML software models or directly with C++. 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 [15]. 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 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.

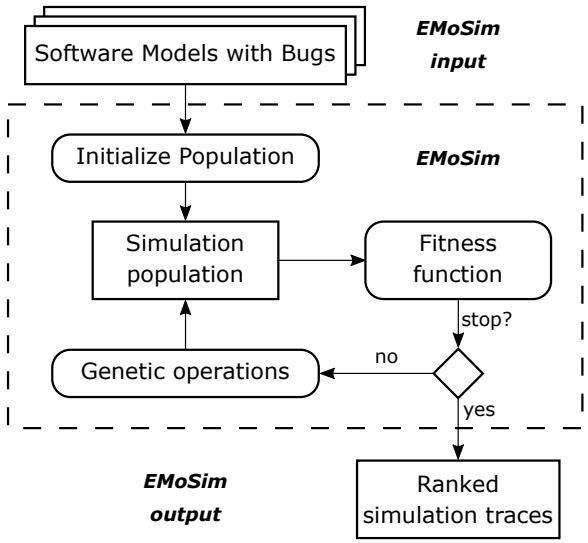

Figure 2: EMoSim approach.

3.1 Overview of the Approach

We call our approach EMoSim, Evolutionary algorithm for bug localization in software Models leveraging the game Simulations. The general structure of the approach is introduced in Figure 2. Our EMoSim approach takes as input a set of software models in which we want to locate the bug. The goal of EMoSim is to obtain a ranked list of simulation traces that are ordered by their relevance in locating the bug.

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. The evolutionary algorithm and its adaptation to the bug localization problem in software models of video games are described in the following sections.

3.2 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 [9]. 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 (crossover 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.

3.2.1 ***Individual representation.*** To represent a candidate solution (individual), 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 simulation. 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.

Figure 3 shows two examples of individuals. Each simulation emulates the behaviour of a player when the battle with the boss occurs. For example, the parameters can define how many steps the simulated player takes in each hull of the boss, 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.

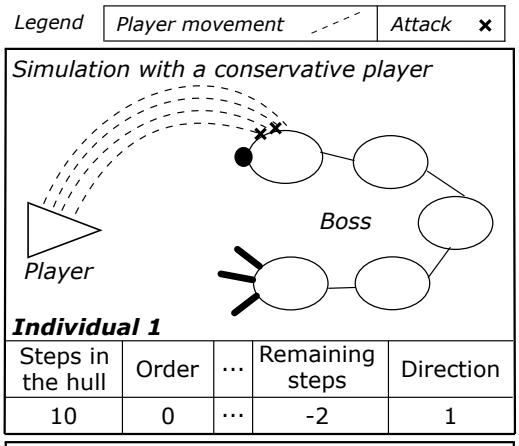

Figure 3: Representation of a simulation as an individual.

Each example in Figure 3 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.

3.2.2 ***Fitness function.*** Once a solution is created, it should be assessed using a fitness function quantifying the quality of the proposed simulation. The input of this step is a set of simulations; the output is the set of simulations, where each simulation 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 [8]. This fitness rewards simulations that have behaved as the developers intended for their game. In [8], 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. 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 [8]:

- $Victory$ 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. Finally, our algorithm ranks the simulations in ascending order with lower fitness values first. A lower fitness value means that the simulation has had a result that the developers would not approve of.

3.2.3 ***Selection.*** To select individuals, 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 solutions while still giving a chance to every solution.

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.

3.2.4 ***Genetic operators.*** To better explore the search space, the crossover and mutation operators are presented below:

- Crossover: We use a single, random, cut-point crossover. It starts by selecting and splitting two parent solutions 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 solutions 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.
  Each solution has the same length, which is the number of parameters for the simulation. When applying the crossover operator, the new solutions have the same length.
- 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.

As a result, new simulations are created. In other words, the new simulations represent other possible solutions 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. To do so, 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 EVALUATION

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

There are two aspects that we evaluate regarding the use of the simulations in our approach and the different bugs involved in the process. In order to address the evaluation of these aspects, we formulated the following three research questions:

- RQ_1: What is the performance in terms of solution quality of the EMoSim, the baseline, and the random search approaches?
- RQ_2: Is there any difference in performance among the different types of bugs?
- RQ_3: Are the performance results obtained by the EMoSim, the baseline, and the random search approaches significant?

Answering RQ_1 allows us to compare the performance results (in terms of recall, precision, and F-measure) of our approach and the baseline approach. 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 type of bug influences the results. Answering RQ_3 allow 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).

4.1 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, we extracted the oracle from the documentation of the video game. The oracle is the ground truth and is used to compare the results provided by the EMoSim, the baseline (BLiMEA [2]), and the random search (RS) approaches.

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 [34] 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, a total of 30 bugs were randomly selected from the entire documentation, i.e., six bugs for each type of the five types of bug. 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

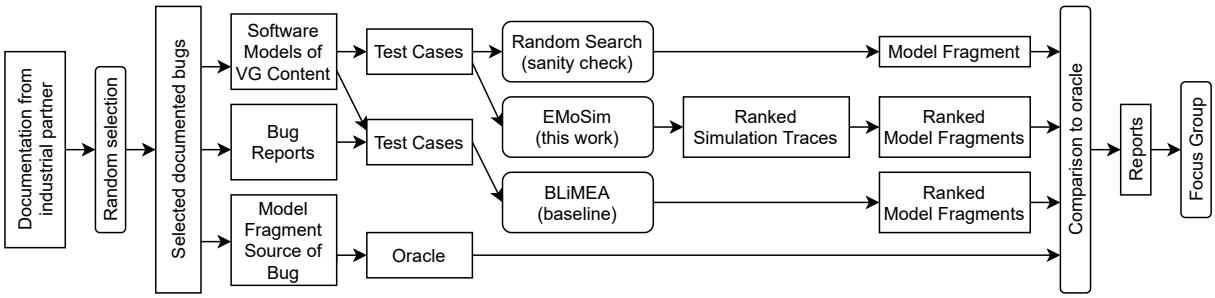

Figure 4: Evaluation process.

that bug was manifested for EMoSim and RS. All of them were obtained from the documentation.

4.2 Experimental setup

Figure 4 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 baseline, BLiMEA, produces a ranking of model fragments that are the most relevant to the bug. EMoSim produces a ranking of traces. The trace contains all of the model elements that the 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 [23]. 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, and
- 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. However, in order to evaluate the performance 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.

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.

4.3 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 of the bugs as suggested in [4]. For BLiMEA, we used the same parameters as reported in [3]. For EMoSim, we started from those reported in [8] (as we used the same simulation) and made sure they converged.

Table 1: Mean values and standard deviations for Recall, Precision, and F-measure for each type of bug.

	EMoSiM			Baseline			RS		
	Recall ± (σ)	Precision ± (σ)	F-measure ± (σ)	Recall ± (σ)	Precision ± (σ)	F-measure ± (σ)	Recall ± (σ)	Precision ± (σ)	F-measure ± (σ)
BLI	63.61 ± 19.70	38.12 ± 7.23	46.80 ± 7.83	31.94 ± 9.45	27.55 ± 8.32	28.55 ± 5.13	87.5 ± 9.99	3.48 ± 0.31	6.70 ± 0.59
HMB	77.78 ± 20.18	32.87 ± 3.58	45.76 ± 6.46	46.26 ± 23.53	19.77 ± 6.59	26.53 ± 8.20	95 ± 4.59	1.88 ± 0.17	3.69 ± 0.33
HNL	75 ± 14.41	35.23 ± 5.06	47.66 ± 6.54	38.34 ± 22.97	14.40 ± 3.81	19.86 ± 4.45	89.44 ± 7.12	1.88 ± 0.21	3.67 ± 0.40
WPH	61.67 ± 24.65	29.93 ± 3.15	39.08 ± 9.77	42.16 ± 24.86	18.95 ± 6.56	25.60 ± 11.68	82.77 ± 10.84	1.65 ± 0.19	3.23 ± 0.37
WPO	60.93 ± 5.26	83.30 ± 3.66	70.28 ± 3.99	40.94 ± 5.26	55.82 ± 3.80	47.17 ± 4.57	84.13 ± 17.36	8.25 ± 1.02	15.01 ± 1.96
ALL	68.03 ± 18.56	42.53 ± 19.65	49.22 ± 12.29	39.89 ± 18.64	26.32 ± 15.48	28.93 ± 11.32	87.90 ± 10.68	3.26 ± 2.45	6.16 ± 4.38

For purposes of replicability, the implementation source code and the data (software models and oracles) are publicly available, including the CSV files used as input in the statistical analysis at the following URL: http://www.gamesoftwareengineering.com/models22/bl-in-mgse.

4.4 Results

In this section, we present the results obtained in EMoSim (this work), BLiMEA (baseline), and the RS (sanity check) approaches in Kromaia. Table 1 shows the mean values and standard deviations for recall, precision, and F-measure for each approach. EMoSim and BLiMEA obtained better results than the RS. For the first research question (RQ_1), the EMoSim approach obtained the best results in precision and F-measure, providing an average value of 42.53 in precision and 49.22 in F-measure.

Table 1 shows the mean values and standard deviations for recall, precision, and F-measure for each type of bug. For the second research question (RQ_2), the WPO bug type obtained the best results in precision and F-measure, providing an average value of 83.30 in precision and 70.28 in F-measure. Overall, in terms of precision, and F-measure, EMoSim outperformed the other two approaches. RS yielded better recall values since it includes all of the model elements (relevant and not relevant) and does not discriminate (as the precision value shows). That is why when comparing for F-measure (which harmonizes recall and precision), RS comes out worse.

4.5 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 [4].

4.5.1 ***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 shows that it is more powerful than the others when working with real data [19]. In addition, according to Conover [12], 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 $7.901x10^{-13}$, $5.761x10^{-13}$, and $1.42x10^{-14}$ for recall, precision, and F-measure, respectively. 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 2 shows the $p-Values$ of Holm's post hoc analysis for the case study and the performance indicators for the algorithms. All the $p-Values$ obtained are smaller than their corresponding significance threshold value (0.05), indicating that the differences in performance among the three algorithms are significant.

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

	Recall	Precision	F-measure
EMoSim vs Baseline	$4.9x10^{-6}$	0.00056	$1.6x10^{-6}$
EMoSim vs RS	$6x10^{-7}$	$3x10^{-10}$	$3x10^{-10}$
Baseline vs RS	$1x10^{-9}$	$3x10^{-10}$	$3x10^{-10}$

4.5.2 ***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 [4]. 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 $\hat{A}_{12}$ [20, 33]. $\hat{A}_{12}$ 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 3 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 [11] values for recall, precision, and F-measure. From the results, we can determine how much the quality of the solution is influenced using our approach (EMoSim) compared to the baseline (BLiMEA) and Random Search. The magnitude of improvement using EMoSim instead of the baseline can be interpreted as being large according to the magnitude scales [26] of the Cliff Delta values. According to the $\hat{A}_{12}$ value, EMoSim

Table 3: Effect size measures for comparing each pair of algorithms in Kromaia.

	$\hat{A}_{12}$		
	Recall	Precision	F-measure
EMoSim vs Baseline	0.8668252	0.8180737	0.8834721
EMoSim vs RS	0.1872771	1	1
Baseline vs RS	0.02378121	1	0.9881094
	Cliff's Delta		
	Recall	Precision	F-measure
EMoSim vs Baseline	0.7336504 (large)	0.6361474 (large)	0.7669441 (large)
EMoSim vs RS	-0.6254459 (large)	1 (large)	1 (large)
Baseline vs RS	-0.9524376 (large)	1 (large)	0.9762188 (large)

obtains better results in F-measure than the baseline in 88.35% of the runs. Hence, EMoSim has an actual impact on performance. EMoSim obtains better results in precision and F-measure than Random Search in 100% of the runs.

4.6 Discussion

Despite the rise of video games and the arrival of the metaverse (where some video games can be seen as an embryo of the metaverse), video game research has not received much attention yet. Within the software engineering research community, the efforts of Pascarella et al. [24] are relevant in understanding the differences between Classic Software Engineering (CSE) and Game Software Engineering (GSE). Similarly, Politowski et al. [25] dedicated efforts to understand the testing differences between CSE and GSE. Our work also advances the understanding of the differences between CSE and GSE in relation to Bug Localization (BL).

For decades, in CSE, the scientific community has exploited the textual descriptions of bug reports to locate bugs [34]. More recently, the defect principle has also proven useful for locating bugs [37]. However, GSE can be a completely different animal when it comes to BL as we argue below.

The baseline of our evaluation was successful in the context of CSE, for example, by locating bugs in the firmware that controls induction hobs from the brands of the BSH group (Bosch, Siemens, Gaggenau, Neff, Balay, among others) [3]. This success was obtained by exploiting the information from the bug reports and the defect principle as other approaches in the literature [34]. However, in the context of this GSE work, the baseline results obtained lower values in the main measures (precision, recall, and F-measure). After analyzing the results, both the bug report and the defect principle fail to successfully guide the localization of the bug.

Many would argue that the main requirement of video games is to be fun. This requirement may have an impact on the bugs of video games. By checking the bug reports or the conditions for the defect principle, it does not seem like either of these are connected to fun. However, simulations may be better suited to providing an idea of fun or lack of fun. In EMoSim, the simulations are ordered by how far they are from what the developers consider to be the ideal experience (and in their opinion the most fun). Our intuition is that thinking about the lack of fun heuristics can help design better guides to locate bugs in video games. All in all, video game developers seem to be more concerned about bugs that imperceptibly frustrate the player than a null pointer exception.

As occurs in CSE, GSE developers accelerate the development through the use of frameworks or libraries (see Figure 1). This can potentially be a source of bugs. For example, the developers are very careful not to completely overlap a weak point with hulls (hulls can only be destroyed after the weak point is destroyed). This can cause a blocking bug by making the enemy indestructible and the player unable to progress through the game. The developers have to prevent this situation by taking into account the behavior of the boss and the interaction of the boss with other objects. Our approach revealed blocking bugs where the boss was indestructible because one weak point overlapped another weak point. The bugs that our approach locates are not due to errors in the well-formedness of the models, but rather are produced by errors that appear due to the interaction of different elements at runtime. The approach based on evolving simulations has allowed developers to detect this previously unknown source of bugs because it is at runtime when the interactions trigger the bugs. Nevertheless, more research is needed in the bugs caused by the well-formedness of the models.

The results also show that the different types of bugs (WPH, WPO, BLI, HNL, and HMB) obtain different values of recall, precision, and F-measure. This reveals that, in order to put the performance of different approaches into perspective, future research on BL in video games must report the types of bugs that have been used in their evaluation. It is not enough to talk about bugs in video games in general.

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 parameters of model elements.

One may think that in order to use EMoSim it is necessary to develop the simulations that guide the search from scratch. However, in the case of video games, Non-Playable Characters (NPCs) are created as part of the usual development. These NPCs act as enemies, cooperate with the player, or simply wander the video game worlds for a more realistic feel. These NPCs are the raw material for the simulations, which significantly reduces the effort of applying EMoSim. A simulated player can interact with the NPCs (as is the case in this work), and it is even possible for several NPCs to interact with each other (e.g., several bots in a deathmatch map of a first-person shooter game). Our approach is a twist on the usual use of NPCs in video games, using them to locate bugs.

As part of the development (e.g., an app), in the case of CSE, there are no agents that are equivalent to the NPCs of video games. However, future research should explore the results that the approach of this work in BL for CSE would obtain if these agents were built. A simulation-based approach like EMoSim can potentially help to find bugs related to user experience in CSE.

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?, (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.

4.7 Threats to Validity

To acknowledge the threats to the validity of our work, we use the classification suggested by De Oliveira et al. [14].

1) Conclusion Validity threats. We considered random variation by executing the approaches 30 times for each of the bugs as suggested in [4]. We used measurements that are widely accepted in the software engineering research community (recall, precision, and F-measure) [5] 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 [5]. We addressed the lack of a meaningful comparison baseline by comparing the results obtained from 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 [5], 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 [8]. 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 documentation of the game. We randomly selected six bugs for each type of the five types of bug from the entire documentation 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 [5].

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 to any software model that conforms to MOF (the OMG metalanguage for defining modeling languages), and the text elements that are associated to the models are extracted automatically using the reflective methods provided by the Eclipse Modeling Framework. In addition, our approach requires the three main ingredients of SBSE approaches: encoding, operators, and fitness function. The operators are the widespread crossover and mutation operators. 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, FPS games, or RTS games. For those cases were 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.

5 RELATED WORK

This section presents the related works. It is divided into three subsections taking into account the topics covered in this paper: bug localization in games, bug localization in games that use models, and bug localization in models.

5.1 Bug localization in games

Ariyurek et al. [7] 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 in terms of the errors found. In their following work [6], they extended the MCTS agent with several modifications (Transpositions, Knowledge-Based Evaluations, Tree Reuse, MixMax, Boltzmann Rollout, Single Player MCTS, and Computational Budget) for game testing purposes. Their results showed that MCTS modifications improve the bug-finding performance of the agents.

Tufano et al. [32] 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.

Zheng et al. [35] 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.

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.

5.2 Bug localization in games that use models

Iftikhar et al. [22] 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. [16] 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.

5.3 Bug localization in models

Outside the game domain, several techniques are used for bug localization in models. Arcega et al. [2] 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. [31] 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. [27] 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 [28]. 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. [10] 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.

The above approaches that take 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.

6 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 shows that bug reports and the defect localization principle are not enough to locate bugs in video games. We propose a novel route to locate bugs in video games by means of evolving video game simulations that produce traces that are relevant to locating bugs. To locate bugs, we leverage non-player characters, which is one of the key features that are inherent to the video game domain.

Our proposal of evolving simulations offers significantly better results. A focus group has confirmed the acceptance of our proposal. Our discussion of the results helps to advance the understanding of bugs in video games. Our work opens a novel research direction for bug localization in video games that could also potentially be used in Classic Software Engineering.

ACKNOWLEDGMENTS

This work has been partially supported by the Ministry of Science and Innovation under the project Iniciando Lineas de Producto Software Mediante Busquedas Interactivas Dirigidas por Modelos (PID2021-128695OB-I00).

REFERENCES

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

[2] Lorena Arcega, Jaime Font Arcega, Øystein Haugen, and Carlos Cetina. 2021. Bug Localization in Model-Based Systems in the Wild. *ACM Trans. Softw. Eng. Methodol.* 31, 1, Article 10 (oct 2021), 32 pages. https://doi.org/10.1145/3472616

[3] Lorena Arcega, Jaime Font, Øystein Haugen, and Carlos Cetina. 2019. An approach for bug localization in models using two levels: model and metamodel. *Software and Systems Modeling* 18, 6 (2019), 3551–3576. https://doi.org/10.1007/s10270-019-00727-y

[4] Andrea Arcuri and Lionel Briand. 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219 – 250. https://doi.org/10.1002/stvr.1486

[5] Andrea Arcuri and Gordon Fraser. 2013. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering* 18 (2013), 594–623.

[6] Sinan Ariyurek, Aysu Betin-Can, and Elif Sürer. 2020. 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, 25–32. https://doi.org/10.1109/CoG47356.2020.9231670

[7] Sinan Ariyurek, Aysu Betin-Can, and Elif Surer. 2021. Automated Video Game Testing Using Synthetic and Humanlike Agents. *IEEE Transactions on Games* 13, 1 (2021), 50–67. https://doi.org/10.1109/TG.2019.2947597

[8] Daniel Blasco, Jaime Font, Mar Zamorano, and Carlos Cetina. 2021. An evolutionary approach for generating software models: The case of Kromaia in Game Software Engineering. *J. Syst. Softw.* 171 (2021), 110804. https://doi.org/10.1016/j.jss.2020.110804

[9] Ilhem Boussaïd, Patrick Siarry, and Mohamed Ahmed-Nacer. 2017. A survey on search-based model-driven engineering. *Automated Software Engineering* 24, 2 (01 Jun 2017), 233–294. https://doi.org/10.1007/s10515-017-0215-4

[10] Loli Burgueño, Javier Troya, Manuel Wimmer, and Antonio Vallecillo. 2015. Static Fault Localization in Model Transformations. *IEEE Transactions on Software Engineering* 41, 5 (May 2015), 490–506. https://doi.org/10.1109/TSE.2014.2375201

[11] Norman Cliff. 1996. *Ordinal methods for behavioral data analysis.* Lawrence Erlbaum Associates, Inc. 212 pages. https://doi.org/10.4324/9781315806730

[12] William Jay Conover. 1999. *Practical Nonparametric Statistics, 3rd Edition.* Wiley, USA.

[13] Crytek. 2002. CRYENGINE | The complete solution for next generation game development by Crytek. https://www.cryengine.com. [Online; accessed 21-November-2021].

[14] Márcio de Oliveira Barros and Arilo Claudio Dias Neto. 2011. *Threats to Validity in Search-based Software Engineering Empirical Studies.* Technical Report 0006/2011.

[15] África Domingo, Jorge Echeverría, Oscar Pastor, and Carlos Cetina. 2020. 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 (Lecture Notes in Computer Science, Vol. 12127)*, Schahram Dustdar, Eric Yu, Camille Salinesi, Dominique Rieu, and Vik Pant (Eds.). Springer, 353–367.

[16] Raihana Ferdous, Fitsum Meshesha Kifetew, Davide Prandi, I. S. W. B. Prasetya, Samira Shirzadehhajimahmood, and Angelo Susi. 2021. 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 (Lecture Notes in Computer Science, Vol. 12914)*, Una-May O'Reilly and Xavier Devroey (Eds.). Springer, 56–71. https://doi.org/10.1007/978-3-030-88106-1_5

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

[18] Epic Games. 1998. Unreal Engine: The most powerful real-time 3D creation tool. https://www.unrealengine.com. [Online; accessed 21-November-2021].

[19] Salvador García, Alberto Fernández, Julián Luengo, and Francisco Herrera. 2010. Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: Experimental analysis of power. *Information Sciences* 180, 10 (2010), 2044 – 2064. https://doi.org/10.1016/j.ins.2009.12.010 Special Issue on Intelligent Distributed Information Systems.

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

[21] Ahmed E. Hassan and Richard C. Holt. 2005. The Top Ten List: Dynamic Fault Prediction. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM '05)*. IEEE Computer Society, USA, 263–272. https://doi.org/10.1109/ICSM.2005.91

[22] Sidra Iftikhar, Muhammad Zohaib Iqbal, Muhammad Uzair Khan, and Wardah Mahmood. 2015. 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*, Timothy Lethbridge, Jordi Cabot, and Alexander Egyed (Eds.). IEEE Computer Society, 426–435. https://doi.org/10.1109/MODELS.2015.7338274

[23] Hisao Ishibuchi, Yusuke Nojima, and Tsutomu Doi. 2006. Comparison between Single-Objective and Multi-Objective Genetic Algorithms: Performance Comparison and Performance Measures. In *2006 IEEE International Conference on Evolutionary Computation*. 1143–1150. https://doi.org/10.1109/CEC.2006.1688438

[24] Luca Pascarella, Fabio Palomba, Massimiliano Di Penta, and Alberto Bacchelli. 2018. 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*, Andy Zaidman, Yasutaka Kamei, and Emily Hill (Eds.). ACM, 392–402. https://doi.org/10.1145/3196398.3196418

[25] Cristiano Politowski, Fábio Petrillo, and Yann-Gaël Guéhéneuc. 2021. 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, 90–99. https://doi.org/10.1109/AST52587.2021.00018

[26] Jeanine Romano, Jeffrey D. Kromrey, Jesse Coraggio, and Jeff Skowronek. 2006. 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*. 1–3.

[27] Jesús Sánchez Cuadrado, Esther Guerra, and Juan Lara. 2017. Static Analysis of Model Transformations. *IEEE Transactions on Software Engineering* 43, 9 (2017), 868–897. https://doi.org/10.1109/TSE.2016.2635137

[28] Jesús Sánchez Cuadrado, Esther Guerra, and Juan Lara. 2018. Quick Fixing ATL Transformations with Speculative Analysis. *Softw. Syst. Model.* 17, 3 (July 2018), 779–813. https://doi.org/10.1007/s10270-016-0541-1

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

[30] Unity Technologies. 2005. Unity Real-Time Development Platform | 3D, 2D VR & AR Engine. https://unity.com. [Online; accessed 21-November-2021].

[31] Javier Troya, Sergio Segura, Jose Antonio Parejo, and Antonio Ruiz-Cortés. 2018. Spectrum-Based Fault Localization in Model Transformations. *ACM Trans. Softw. Eng. Methodol.* 27, 3, Article 13 (Sept. 2018), 50 pages. https://doi.org/10.1145/3241744

[32] Rosalia Tufano, Simone Scalabrino, Luca Pascarella, Emad Aghajani, Rocco Oliveto, and Gabriele Bavota. 2022. Using Reinforcement Learning for Load Testing of Video Games. In *Proceedings of the 44th International Conference on Software Engineering (ICSE 2022), Pittsburgh, USA*.

[33] András Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132. https://doi.org/10.3102/10769986025002101 arXiv:http://jeb.sagepub.com/content/25/2/101.full.pdf+html

[34] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Trans. Software Eng.* 42, 8 (2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368

[35] Yan Zheng, Changjie Fan, Xiaofei Xie, Ting Su, Lei Ma, Jianye Hao, Zhaopeng Meng, Yang Liu, Ruimin Shen, and Yingfeng Chen. 2019. 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, 772–784. https://doi.org/10.1109/ASE.2019.00077

[36] Meng Zhu and Alf Inge Wang. 2020. Model-driven Game Development: A Literature Review. *ACM Comput. Surv.* 52, 6 (2020), 123:1–123:32. https://doi.org/10.1145/3365000

[37] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. 2004. Mining Version Histories to Guide Software Changes. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. IEEE Computer Society, Washington, DC, USA, 563–572. http://dl.acm.org/citation.cfm?id=998675.999460