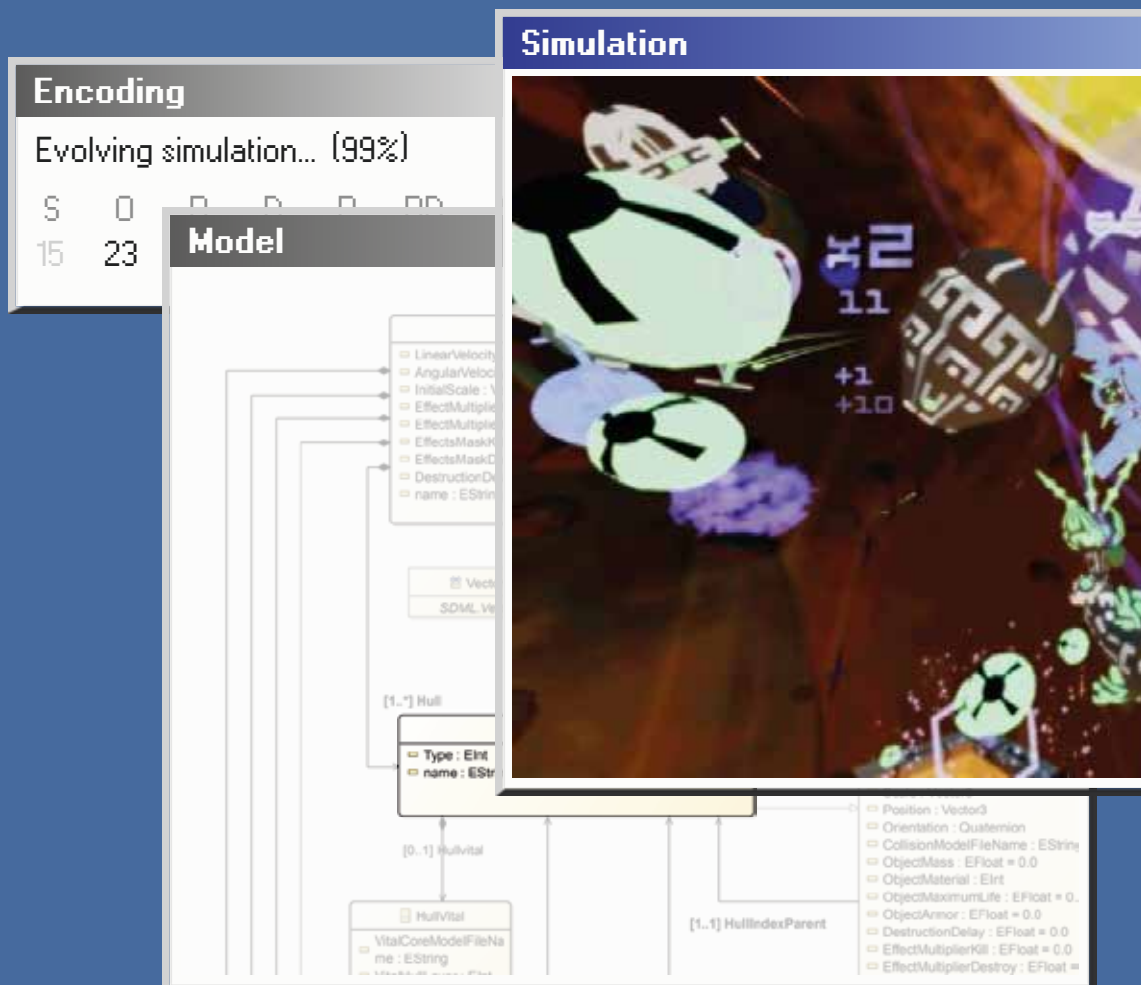


Towards Bug Localization in Models in Game Software Engineering

Rodrigo Casamayor Moragriega



Thesis Supervisors



Dr. Óscar Pastor López
Dra. María Francisca Pérez Pérez



UNIVERSITAT
POLITÀCNICA
DE VALÈNCIA



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Towards Bug Localization in Models in Game Software Engineering

October 2025

Author: Rodrigo Casamayor Moragriega

Thesis Supervisors: Dra. María Francisca Pérez Pérez
Dr. Óscar Pastor López

Dedicatoria

A Juan Carlos y a mis padres,
por su apoyo constante e incondicional a lo largo de este proceso de
crecimiento personal.

Agradecimientos

Quiero agradecer de manera expresa a mi directora, Paqui, y a Carlos, su acompañamiento, criterio y rigor a lo largo de este trabajo. Su confianza desde las primeras etapas y su orientación experta han sido determinantes para que esta tesis llegara a buen término.

Deseo agradecer también a Marcos su apoyo constante y la aportación de una mirada crítica y complementaria.

Extiendo también mi agradecimiento al Grupo de Investigación SVIT de la Universidad San Jorge, un colectivo excepcional que representa con coherencia lo que significa trabajar en equipo. Su generosidad intelectual y humana ha sido una parte esencial de este recorrido.

Abstract

Video games have unique characteristics that differentiate their development and maintenance from classic software engineering (CSE), which led to the emerging subfield of Game Software Engineering (GSE). One of these differences is that video game developers perceive more difficulties than other non-game developers when it comes to locating bugs. The aim of this thesis is to address bug localization in software models in GSE by leveraging the data (i.e., traces) generated from simulations of the behavior of non-player characters (NPCs). NPCs (e.g., bots in First-Person Shooters) are not controlled by the player and are key components of video games. Hence, exploiting the data from simulations can be instrumental in locating bugs.

The approaches that are proposed in this work explore: 1) how the data from simulations can be used to automatically locate potential buggy elements in software models, which refer to high-level representations that describe the structure and behavior of game content (e.g., NPCs) and are interpreted at run-time; 2) how the human effort for assessing candidate solutions influences the location of bugs; 3) whether component-specific operations can boost bug localization. The approaches proposed are evaluated at the scale of industrial settings with a commercial video game (Play Station 4 and Steam) and 29 professional video game developers. The results show that the approaches improve the quality of the solu-

tion of the model fragment that may contain the bug compared to the baselines, and the manual effort is reduced.

This thesis presents a compendium of three research papers published in academic journals and conference. The contributions demonstrate that these novel Search-Based Game Software Engineering approaches significantly improve the quality of the solution of located model elements. This is essential to the maintenance of video games for reducing the amount of tedious manual work and minimizing the number of bugs that go unnoticed.

Resumen

Los videojuegos presentan características únicas que diferencian su desarrollo y mantenimiento del de la ingeniería de software clásica (CSE), lo que ha dado lugar al surgimiento del subcampo de la Ingeniería de Software para Videojuegos (GSE). Una de estas diferencias es que los desarrolladores de videojuegos perciben más dificultades que otros desarrolladores no especializados en juegos a la hora de localizar errores. El objetivo de esta tesis es abordar la localización de errores en modelos de software dentro de la GSE aprovechando los datos (es decir, trazas) generados a partir de simulaciones del comportamiento de los personajes no jugadores (NPCs). Los NPCs (por ejemplo, bots en juegos de disparos en primera persona) no son controlados por el jugador y son componentes clave de los videojuegos. Por tanto, explotar los datos procedentes de simulaciones puede ser fundamental para localizar errores.

Las aproximaciones propuestas en este trabajo exploran: 1) cómo los datos de las simulaciones pueden utilizarse para localizar automáticamente posibles elementos defectuosos en modelos de software, que se refieren a representaciones de alto nivel que describen la estructura y el comportamiento del contenido del juego (por ejemplo, los NPCs) y que se interpretan en tiempo de ejecución; 2) cómo el esfuerzo humano dedicado a evaluar soluciones candidatas influye en la localización de errores; y 3) si las operaciones específicas de los componentes pueden mejorar

la localización de errores. Las aproximaciones propuestas se evalúan en un contexto industrial con un videojuego comercial (PlayStation 4 y Steam) y 29 desarrolladores profesionales de videojuegos. Los resultados muestran que las aproximaciones mejoran la calidad de la solución del fragmento del modelo que puede contener el error en comparación con los baselines, y que el esfuerzo manual se reduce.

Esta tesis presenta un compendio de tres artículos de investigación publicados en revistas y congresos académicos. Las contribuciones demuestran que estas novedosas aproximaciones basadas en búsqueda dentro de la Ingeniería de Software para Videojuegos mejoran significativamente la calidad de las soluciones al localizar los elementos del modelo defectuosos. Esto resulta esencial para el mantenimiento de los videojuegos, ya que reduce la cantidad de trabajo manual tedioso y minimiza el número de errores que pasan desapercibidos.

Resum

Els videojocs presenten característiques úniques que diferencien el seu desenvolupament i manteniment del de la enginyeria de programari clàssica (CSE), fet que ha donat lloc a l'aparició del subcamp de l'Enginyeria de Programari per a Videojocs (GSE). Una d'aquestes diferències és que els desenvolupadors de videojocs perceben més dificultats que altres desenvolupadors no especialitzats en jocs a l'hora de localitzar errors. L'objectiu d'aquesta tesi és abordar la localització d'errors en models de programari dins de la GSE aprofitant les dades (és a dir, traces) generades a partir de simulacions del comportament dels personatges no jugadors (NPCs). Els NPCs (per exemple, bots en jocs de trets en primera persona) no són controlats pel jugador i són components clau dels videojocs. Per tant, explotar les dades provinents de simulacions pot ser fonamental per a localitzar errors.

Les aproximacions proposades en aquest treball exploren: 1) com les dades de les simulacions poden utilitzar-se per a localitzar automàticament possibles elements defectuosos en models de programari, que fan referència a representacions d'alt nivell que descriuen l'estructura i el comportament del contingut del joc (per exemple, els NPCs) i que s'interpreten en temps d'execució; 2) com l'esforç humà dedicat a avaluar solucions candidates influïx en la localització d'errors; i 3) si les operacions específiques dels components poden millorar la localització d'errors.

Les aproximacions proposades s'avaluaren en un context industrial amb un videojoc comercial (PlayStation 4 i Steam) i 29 desenvolupadors professionals de videojocs. Els resultats mostren que les aproximacions milloren la qualitat de la solució del fragment del model que pot contindre l'error en comparació amb els baselines, i que l'esforç manual es redueix.

Aquesta tesi presenta un compendi de tres articles d'investigació publicats en revistes i congressos acadèmics. Les contribucions demostren que estes noves aproximacions basades en cerca dins de l'Enginyeria de Programari per a Videojocs milloren significativament la qualitat de les solucions en localitzar els elements defectuosos del model. Això és essencial per al manteniment dels videojocs, ja que redueix la quantitat de treball manual feixuc i minimitza el nombre d'errors que passen desapercebuts.

Contents

Dedicatoria	iii
Agradecimientos	v
Abstract - Resumen - Resum	vii
Contents	xiii
I Introduction	1
II Compendium of Articles	21
1 Bug localization in game software engineering: evolving simulations to locate bugs in software models of video games	23

2 Studying the Influence and Distribution of the Human Effort in a Hybrid Fitness Function for Search-Based Model-Driven Engineering	59
3 Boosting bug localization in software models of video games with simulations and component-specific genetic operations	107
III Discussion	173
IV Conclusions	185

Part I

Introduction

Introduction

This part introduces the context and objectives of this thesis. In addition, it includes a development overview of the research process, and describes the methodology applied. The last sections of the chapter present the structure of this work and the scientific articles that are included in the document.

Context

This thesis addresses the software aspect of video game development through search-based techniques. Accordingly, the key background areas in Software Engineering and Search-Based Software Engineering are first introduced, followed by an overview of video game development and the emerging field of Game Software Engineering. This contextual foundation frames the specific focus of this work on automating bug localization in game software models.

Software Engineering

Software Engineering (SE) is the discipline concerned with the systematic and efficient development and maintenance of software systems. It encompasses a broad set of practices and methodologies to ensure software is delivered with quality, on time, and within budget. In practice, software engineering activities span the entire life cycle from requirements and design to coding, testing, and maintenance. Notably, maintenance (including bug fixing and evolution) can consume a majority of a software system's life cycle effort [7]. Ensuring software quality is therefore paramount: defects that remain in a system can lead to failures, increased costs, and user dissatisfaction. One critical maintenance challenge is bug localization, the process of identifying the source of a known software failure. Locating the exact model element or code section that causes a bug is often labor-intensive and requires significant developer expertise. Improving and automating bug localization techniques is highly desirable to reduce maintenance effort and improve software reliability.

Search-based Software Engineering

Search-Based Software Engineering (SBSE) is an approach to automate software engineering tasks by formulating them as optimization problems that can be solved via metaheuristic search algorithms. Instead of relying solely on human intuition or exhaustive methods, SBSE applies techniques such as genetic algorithms, simulated annealing, and other evolutionary or heuristic searches to explore large solution spaces for op-

timal or near-optimal solutions. The basic idea is to define a search space of possible solutions to a software engineering problem and a fitness function (or objective function) that quantitatively measures solution quality. The search algorithm then iteratively generates and evaluates candidate solutions, “evolving” them toward better fitness using genetic operations crossover and mutation. SBSE has been successfully applied across many areas of SE, from requirements prioritization and project scheduling to test case generation and software design optimization. Particularly relevant to this thesis, researchers used SBSE for debugging and maintenance tasks such as automated fault localization and program repair [8] (for example, genetic programming has been employed to evolve program fixes for bugs in code). These successes indicate that complex software problems which are hard to solve manually (or by exact methods) can often be tackled effectively with guided search. SBSE provides a promising paradigm to address the bug localization problem by automatically searching for evidence of defects, especially in domains where traditional debugging is difficult or costly.

Video Game Development

Video game development is a multidisciplinary endeavor combining software engineering with creative arts, design, and interactive storytelling. Modern video games are advanced software products but also complex works of creativity and art. This blend of technology and art introduces unique development challenges and constraints that distinguish game development from traditional software projects. The video game industry’s scale underscores its importance [5]: global video game revenues have surpassed those of the film and music industries combined in recent years, making it the world’s largest entertainment industry. Developing a game involves not only writing efficient and robust code (for game engines, graphics, physics, AI, etc.), but also ensuring an engaging player experience, fun gameplay, and artistic cohesion. These demands create significant differences between game development and classic software development. For instance, game projects often have highly iterative and fast-paced development cycles to tune gameplay, requiring constant communication among diverse teams and flexible design adaptation. Games

also tend to have a shorter product life cycle and intense time-to-market pressure, meaning development practices must accommodate frequent updates (e.g. patches, expansions) under tight deadlines. Moreover, ensuring a game is fun and bug-free is more subjective and complex than typical software quality assurance, as issues in a game can include subtle problems in gameplay logic, performance hiccups, or graphical glitches that affect user experience. All these factors make testing and debugging in games particularly challenging [6]. Indeed, game developers often report greater difficulty in diagnosing and locating bugs compared to non-game software projects. High-profile incidents in the game industry have shown that quality issues (e.g., severe bugs in a major release) can lead to public backlash and financial loss, underscoring the need for effective debugging practices. These industry realities motivate research into specialized engineering methods that cater to the distinctive needs of game development in bug localization.

Game Software Engineering

The distinctive characteristics of video game development have led to the emergence of Game Software Engineering (GSE) as a dedicated sub-field at the intersection of software engineering and game development. GSE focuses on adapting and extending software engineering principles, processes, and tools to better suit the game domain, acknowledging that games often demand different approaches than classical business or scientific software. Recent studies [9] indicate that GSE has matured into an independent research domain with its own community and body of knowledge. In fact, a systematic literature review [4] consolidating work on game development processes concludes that GSE has accumulated sufficient volume and unique practices to be considered a branch of SE in its own right. The differences between traditional SE and GSE outnumber the similarities, prompting researchers to develop new approaches tailored to game-specific challenges. Key areas of GSE research include game architecture and design, content generation, testing and QA for games, and maintenance of live game software. Importantly, maintenance and evolution tasks in games (such as bug fixing, patching, and content updates) are critical for player retention and long-term success.

For example, keeping a game updated with fixes and new content is essential to maintain an active player base. However, the GSE community has identified a gap between academic research and industry practice [4], particularly a need for more industrial case studies and practical tools. This work contributes to this agenda by addressing an important maintenance problem (bug localization) in an industrial game context. In GSE, bug localization refers to identifying defective elements in game software artifacts (such as model definitions or scripts) that are causing observed faults in the game. Given the noted difficulties game developers face in locating bugs, automating this process could significantly aid game maintenance.

Software models are fundamental in game development as they specify key game elements such as non-player characters (NPCs), environments, and gameplay mechanics. NPC behaviors can be simulated within the game engine, producing execution traces that capture their interactions and decisions. These simulations provide valuable data that can be used to locate bugs in the software models. By leveraging Search-Based Game Software Engineering (SBGSE), this thesis uses the data derived from NPC-driven simulations to guide the search for bugs in software models, with the aim of reducing manual effort and improving the solution quality of the located model elements.

The next sections outline the specific objectives of this thesis and the research methodology taken to achieve them.

Objectives and Development

Goals Pursued

The overall goal of this thesis is to improve bug localization in the software models of video games by exploring the use of Search-Based Software Engineering techniques through leveraging the data (i.e., traces) generated from simulations of the behavior of non-player characters (NPCs).

SBSE may help to explore the large number of possible simulations. Moreover, leveraging game-specific features (such as simulations and do-

main knowledge) to achieve better bug localization performance (in terms of quality of the solution) compared to conventional approaches. The proposed approaches are evaluated against baselines to determine whether the observed differences are statistically significant.

Specifically, this thesis aims to answer the following research questions:

RQ₁: *How can video game simulations be used in search-based techniques to locate bugs in software models, and is there any difference in performance in terms of solution quality compared to traditional bug localization approaches?*

RQ₂: *What is the influence of the human effort for assessing candidate solutions in search-based bug localization?*

RQ₃: *To what extent can component-specific genetic operations improve the effectiveness of search-based bug localization in video games?*

Industrial Case Study Involvement

To answer the research questions, the research associated to the articles included in this thesis produced a set of approaches: EMoSim (Evolutionary algorithm for bug localization in software Models leveraging game Simulations) with classic genetic operations, [2], SimuHaFF (Simulation and Human as the Fitness Function) [3], and EMoSim with component-specific genetic operations [1]. These approaches were evaluated in the context of an industrial case study: Kromaia, a commercial video game released for PC (Steam) and PlayStation 4 platforms in digital and physical formats. Kromaia serves as a real-world setting and complex testbed for this research. Notably, apart from external libraries, Kromaia consists of over 260,000 lines of private C++ source code comprising a custom game engine and game-specific code. Its software architecture is a combination of inheritance and composition, comparable to that of other professional game engines like Unity [10]. Moreover, the developers of Kromaia created a Domain-Specific Modeling Language for game content (levels, enemy behaviors, etc.), with a built-in model interpreter to load these models into the game at runtime. This means that much of the game’s content is defined as external models, which the engine in-

terprets into in-game objects. Consequently, this requires a specific GSE approach for bug localization in software models.

Research Development Progress

The research presented in this thesis started by investigating how domain-specific information from Game Software Engineering, particularly the use of NPC-based simulations, could be leveraged to locate bugs in video games. From this initial step, the work progressed iteratively, with each stage building upon the findings of the previous one.

The overall research context, together with its alignment with the research questions (RQ1–RQ3), is summarized in Figure 1. The top part of the figure shows the starting point of this work at the intersection of SBSE and Game Software Engineering, while the central part represents the research questions, and the lower part highlights the contributions that answer each research question. Next, these contributions are introduced, each corresponding to one of the included publications and collectively addressing the research objectives.

- **EMoSim:** This first approach, called EMoSim (Evolutionary algorithm for bug localization in software Models leveraging game Simulations), was published in the International Conference on Model Driven Engineering Languages and Systems (MODELS 2022) [2] and presented at MODELS 2022.

To the best of our knowledge, this is the first approach specifically designed for bug localization in video games. EMoSim evolves gameplay simulations to automatically locate bugs in video game models.

The key idea is to treat a video game’s boss simulations (i.e., encounters with bosses controlled by non-player characters) as the search space for an evolutionary algorithm. A simulated player is used to execute gameplay sessions against these bosses, producing traces that capture their behaviors and interactions.

The most common bug types were first identified together with professional developers (e.g., unreachable bosses, unbalanced abilities,

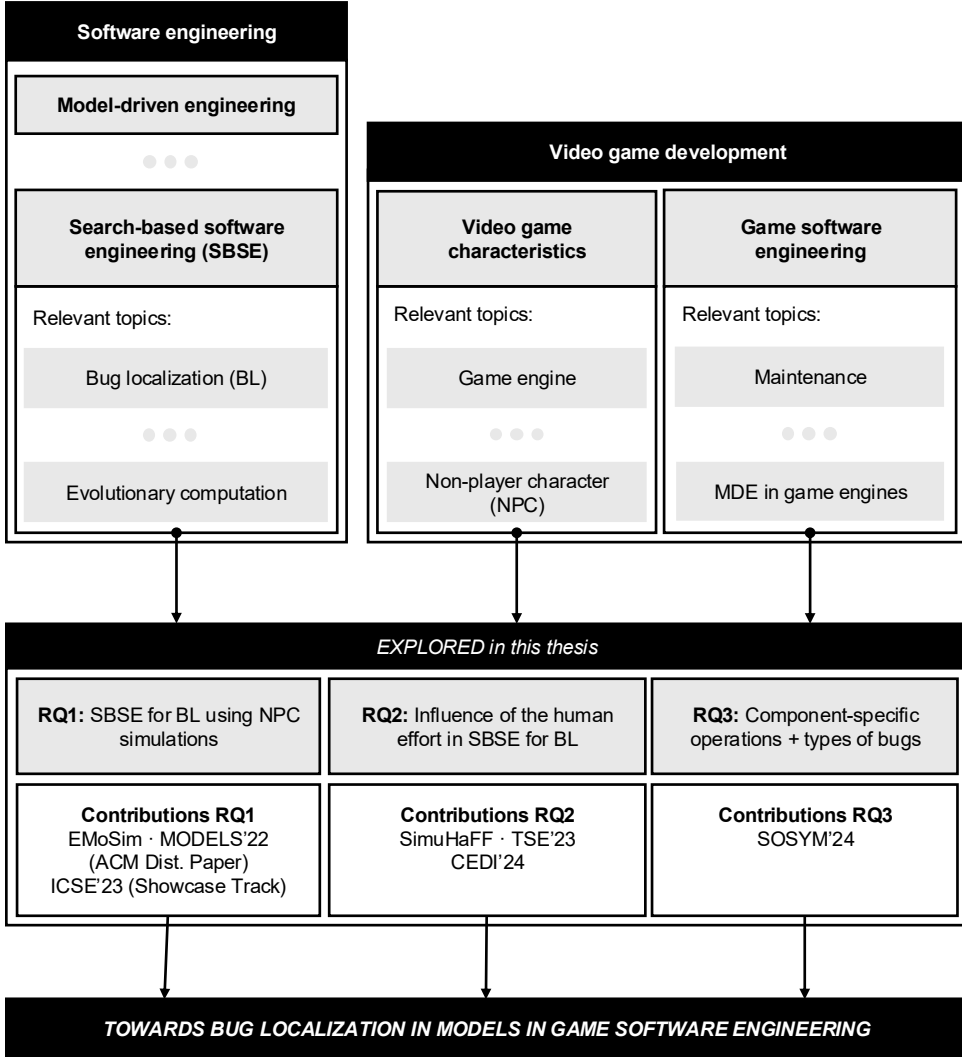


Figure 1: Thesis context and research overview.

progression inconsistencies). EMoSim then defines a fitness function able to detect when a simulation exposes one of these bugs.

By iteratively mutating and recombining gameplay scenarios, the algorithm evolves towards scenarios that reliably reproduce a bug, thereby highlighting the buggy model elements.

In the evaluation with the commercial video game Kromaia, EMoSim was compared against BLiMEA (a classic SE baseline relying on bug reports and defect localization) and Random Search. EMoSim achieved F-measure improvements up to 20.29% over the baseline across five bug types.

A focus group with professional developers confirmed the acceptance of the approach, showing that simulation-guided search effectively addresses the challenge that game developers find bugs hard to localize.

This work was recognized with the ACM SIGSOFT Distinguished Paper Award at MODELS 2022, where it was ranked as the second best paper, and was later invited to be presented in the Showcase Track at ICSE 2023, held in Melbourne, Australia, from May 17 to May 19, 2023.

- **Empowering Humans for Assessing Candidate Solutions—SimuHaFF:** This second work investigates whether including humans in the evaluation of candidate solutions could further improve bug localization. This work was published in the IEEE Transactions on Software Engineering (TSE) journal [3], presented at CEDI and also discussed with a focus group of professional developers.

SimuHaFF introduces the first hybrid fitness function that combines human evaluations with simulated gameplay traces. In this setup, simulations handle the large-scale exploration, while human experts contribute judgments on selected candidate solutions to avoid fatigue.

The motivation, as highlighted in the TSE paper, is that certain anomalies are better judged by a human (e.g., whether a detected imbalance is truly a bug or an intended feature).

To avoid human fatigue, the algorithm does not request feedback continuously but instead at specific moments. Several strategies were studied to distribute the human effort:

- Human evaluation at the beginning of the search (to guide exploration early).
- Human evaluation at the end of the search (to filter high-potential candidates).
- Interleaving human input at intervals.

The results showed that deferring human effort to the later stages produced the best trade-off: developers only reviewed a reduced set of promising candidates, but their insights significantly improved the precision of bug localization.

Hybridizing the fitness function outperformed the best baseline by 33.46% in F-measure. Furthermore, the study confirmed that human-in-the-loop SBSE is practical in GSE: it reduces tedious manual work, minimizes unnoticed bugs, and leverages expert insight without requiring continuous involvement.

- **Extending Bug Localization with Component-Specific Genetic Operations and New Bug Types:** In the third work, the research focused on improving the evolutionary algorithm itself with component-specific genetic operators. This work was published in the *Software and Systems Modeling (SoSyM)* journal [1].

Specifically, this extension of EMoSim addressed more types of bugs and introduced component-specific genetic operations that are tailored to the structure of video game models, rather than using the classic genetic operations (single-point crossover plus random mutation).

Video game models include heterogeneous components such as NPCs, bosses, world elements, and behaviors. Instead of recombining whole scenarios randomly, component-specific crossover and mutation act selectively on these elements (e.g., exchanging only certain bosses or behaviors between simulations).

This approach ensures that generated scenarios remain meaningful and aligned with the model structure. The algorithm leverages NPC traces to automatically highlight potential buggy model elements.

The evaluation on Kromaia compared component-specific operators against generic evolutionary search, BLiMEA, and Random Search. The new operators improved F-measure for all bug categories: by 7.93% over generic evolutionary operators, by 27.17% over BLiMEA, and by 46.34% over Random Search.

These results confirmed that evolving simulations with component-specific operators significantly boosts bug localization performance (in terms of solution quality) and extends the coverage to additional bug categories.

This contribution demonstrates that integrating game-model-specific knowledge into genetic operations is a promising direction for SBSE in GSE.

Through the above progression, the research trajectory moved from establishing a SBSE approach for game bug localization (EMoSim), to enriching it with human feedback (SimuHaFF), and finally to refining it with component-specific genetic operations. As a result of this work, a suite of evaluated approaches has been developed that collectively advance the state of the art towards bug localization in models in game software engineering.

Research Methodology

The research in this thesis was conducted using a Design Science methodology, following the framework proposed by Wieringa for Information Systems and Software Engineering [11]. The motivation for adopting design science is to create and evaluate artifacts (in this case, algorithms and approaches) that address identified problems in a real-world problem context.

In design science terms, each developed approach can be viewed as a treatment introduced into the problem context of game software engi-

neering and studied through an empirical cycle consisting of the following stages (illustrated in Figure 2): (A) Problem Investigation: examining prior work and industry needs to define research questions (for example, investigating how SBSE had been or could be applied to game development problems); (B) Treatment Design: devising the search-based approaches and planning experiments, including baseline selection, to answer those questions; (C) Treatment Validation: analyzing potential threats and ensuring the design is sound (for instance, using accepted evaluation metrics and an industrial case to increase external validity); (D) Treatment Implementation: implementing the algorithms and setting up the experiments so the approach can interact with the problem context (the game and its data); and (E) Implementation Evaluation: executing the experiments, gathering results, and interpreting them with respect to the research questions.

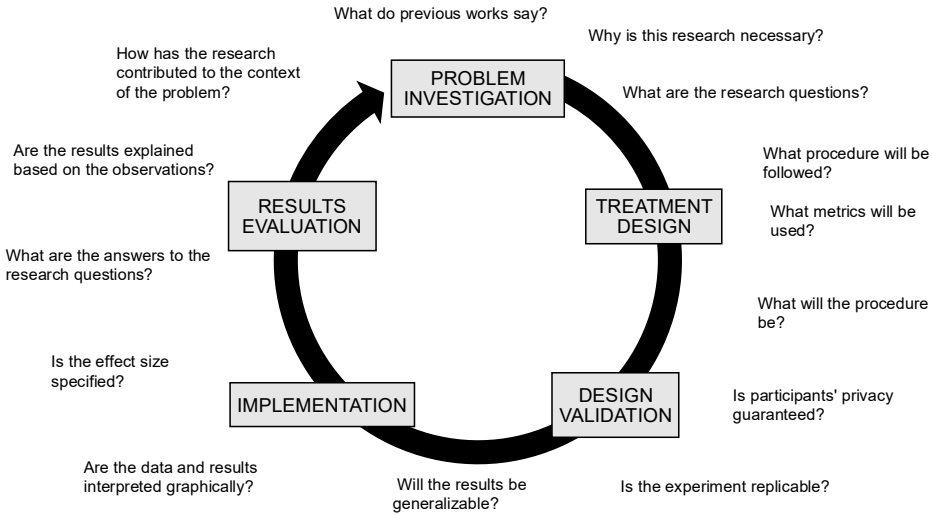


Figure 2: Empirical cycle used in the Design Science methodology.

This cyclical process involves assessing, after each evaluation, whether the treatment sufficiently solved the problem or if new questions emerged, which then informs the next cycle of research. Indeed, in this work, the outcome of the first approach raised new questions (e.g., the role of human insight, the potential of specialized operators) that led to the

second and third approaches, iterating through the design science cycle multiple times.

Structure of the Thesis

This work is structured as a compendium of publications, in accordance with the regulations of Universitat Politècnica de València for PhD theses by publication. The document is organized into four main parts as follows:

- **I. Introduction:** The first part (the current chapter) establishes the context of the research, outlines the objectives and methodology, and provides an overview of the work conducted (including summaries of the publications in the compendium).
- **II. Compendium of Articles:** The second part consists of the three published research articles that form the core of this thesis. Each article is presented as a separate chapter, maintaining its original content and bibliography, but formatted to fit the thesis layout. Together, these articles detail the problem addressed, the proposed solutions (EMoSim, SimuHaFF, and EMoSim with component-specific genetic operators), and their evaluation results.
- **III. Discussion:** The third part discusses the integrated results and implications of the research. It revisits the research questions in light of the findings from all studies, examines the contributions across different scopes (academic and industrial), and outlines potential directions for future work building upon this thesis.
- **IV. Conclusions:** The final part of the thesis highlights the conclusions drawn from the research. It reflects on the extent to which the thesis objectives were achieved, the impact of the results on the field of Game Software Engineering, and final remarks on the significance of the work.

Compendium Description

The following peer-reviewed articles were produced and published in different conferences and journals during the course of this PhD research and are included in Part II of this thesis:

1. **Bug Localization in Game Software Engineering: Evolving Simulations to Locate Bugs in Software Models of Video Games** (ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems) (Casamayor, Arcega, Pérez, Cetina, 2022) (**MODELS'22**) [2].

This conference paper presents the EMoSim approach and its evaluation in the Kromaia case study, establishing a novel simulation-driven approach for model bug localization.

2. **Studying the Influence and Distribution of the Human Effort in a Hybrid Fitness Function for Search-Based Model-Driven Engineering** (IEEE Transactions on Software Engineering) (Casamayor, Cetina, Pastor, Pérez, 2023) (**TSE'23**) [3].

This journal article introduces the SimuHaFF approach and studies the influence and distribution of the human effort for providing evaluations of candidate solutions and integrate them in a hybrid fitness function. The study shows that allocating algorithmic effort first and human input afterwards improves bug localization in video game software models.

3. **Boosting Bug Localization in Software Models of Video Games With Simulations and Component-specific Genetic Operations** (Software and Systems Modeling) (Casamayor, Arcega, Pérez, Cetina, 2024) (**SOSYM'24**) [1].

This journal article describes the enhanced EMoSim approach with component-specific genetic operations. It shows that component-specific genetic operations improve bug localization performance in terms of solution quality (higher F-measure than EMoSim, BLiMEA, and Random Search) and increase coverage by 50% more bug categories. This work highlights the benefit of integrating model-specific knowledge into search-based techniques for game software engineering.

As previously mentioned, each of these of these articles is incorporated as a chapter in the thesis, and they collectively represent the primary contributions of this research. The results and discussions in Part III draw directly on the findings reported in these works.

Additionally, the following oral and poster communications were presented at national and international conferences during the development of this PhD research:

- Talk at the 3rd International Summer School on Search- and Machine Learning-based Software Engineering (SMILESENG), held in Córdoba, Spain, from June 22 to June 24, 2022.
- MODELS'22 was presented in the Technical Track of the 25th International Conference on Model Driven Engineering Languages and Systems (MODELS 2022), held in Montréal, Canada, from October 26 to October 28, 2022.
- MODELS'22 was presented by invitation in the Showcase Track at the 45th International Conference on Software Engineering (ICSE 2023), held in Melbourne, Australia, from May 17 to May 19, 2023.
- MODELS'22 was exhibited by invitation as a poster at the 45th International Conference on Software Engineering (ICSE 2023), held in Melbourne, Australia, from May 17 to May 19, 2023.
- MODELS'22 was presented at the XXVII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2023), held in Ciudad Real, Spain, from September 12 to September 14, 2023.
- TSE'23 was presented at the III Congreso Español de Videojuegos (CEV 2024), within the VII Congreso Español de Informática (CEDI 2024), held in A Coruña, Spain, from June 17 to June 19, 2024.

In 2022, during the ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS 2022), MODELS'22 received the ***Distinguished Paper award*** from ACM SIGSOFT and was the second best paper presented at the conference.

Research Support

The research works included as a compendium in this thesis were partially funded as follows:

- [2] was partially supported by the Ministry of Science and Innovation under the project *Iniciando Líneas de Producto Software Mediante Búsquedas Interactivas Dirigidas por Modelos* (PID2021-128695OB-I00).
- [3] and [1] 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). Furthermore, [1] was partially supported by the Spanish Ministry of Science and Innovation under the Excellence Network AI4Software (Red2022-134647-T).

Bibliography

- [1] Rodrigo Casamayor et al. “Boosting bug localization in software models of video games with simulations and component-specific genetic operations”. In: *Software and Systems Modeling* (2025). ISSN: 1619-1374. DOI: 10.1007/s10270-024-01253-2 (cit. on pp. 8, 12, 16, 18).
- [2] Rodrigo Casamayor et al. “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*. Ed. by Eugene Syriani et al. ACM, 2022, pp. 356–366. DOI: 10.1145/3550355.3552440 (cit. on pp. 8, 9, 16, 18).
- [3] Rodrigo Casamayor et al. “Studying the Influence and Distribution of the Human Effort in a Hybrid Fitness Function for Search-Based Model-Driven Engineering”. In: *IEEE Trans. Software Eng.* 49.12 (2023), pp. 5189–5202. DOI: 10.1109/TSE.2023.3329730 (cit. on pp. 8, 11, 16, 18).

- [4] Jorge Chueca et al. “The consolidation of game software engineering: A systematic literature review of software engineering for industry-scale computer games”. In: *Information and Software Technology* 165 (2024), p. 107330 (cit. on pp. 6, 7).
- [5] *E3 2021: Video games are big business, topping film and music combined*. <https://eu.usatoday.com/videos/tech/2021/06/10/e-3-2021-video-games-big-business-topping-film-and-music-combined/7637695002/>. [Online; accessed 21-November-2021]. 2021 (cit. on p. 5).
- [6] Raihana Ferdous et al. “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*. Ed. by Una-May O’Reilly and Xavier Devroey. Vol. 12914. Lecture Notes in Computer Science. Springer, 2021, pp. 56–71. DOI: 10.1007/978-3-030-88106-1_5 (cit. on p. 6).
- [7] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. “Search-based software engineering: Trends, techniques and applications”. In: *ACM Comput. Surv.* 45.1 (2012), 11:1–11:61. DOI: 10.1145/2379776.2379787 (cit. on p. 4).
- [8] Isis Roca et al. “Co-evolving scenarios and simulated players to locate bugs that arise from the interaction of software models of video games”. In: *Inf. Softw. Technol.* 169 (2024), p. 107412. DOI: 10.1016/J.INFSOF.2024.107412 (cit. on p. 5).
- [9] SlashData. *Global developer population report 2019*. <https://sdata.me/GlobalDevPop19>. [Online; accessed 21-November-2021]. 2019 (cit. on p. 6).
- [10] Unity Technologies. *Unity*. <https://unity.com>. [Online; accessed 21-November-2021]. 2005 (cit. on p. 8).
- [11] Roel J. Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Springer, 2014. ISBN: 978-3-662-43838-1. DOI: 10.1007/978-3-662-43839-8 (cit. on p. 13).

Part II

Compendium of Articles

Chapter 1

Bug localization in game software engineering: evolving simulations to locate bugs in software models of video games

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.

1.1 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].

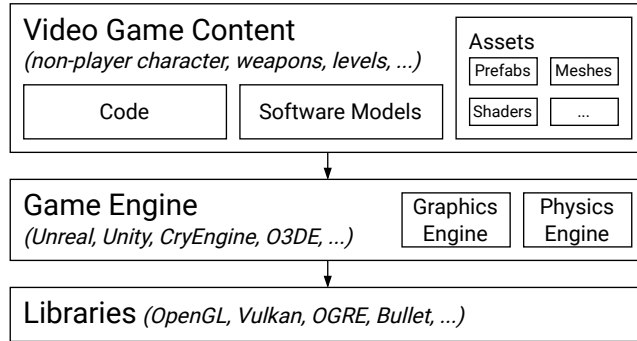


Figure 1.1: Overview of video game artifacts.

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 [32] and Unreal Engine [15], 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 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 3.1). 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 localiza-

tion 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[20] (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 space¹. 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:

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

- 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 1.2, we present the case study (Kromaia) and the bugs. In Section 1.3, we describe our approach, EMoSim. In Section 1.4, we evaluate our approach in Kromaia and discuss the results. In Section 1.5, we examine the related work of the area. Finally, we present our conclusions in Section 1.6.

1.2 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 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 inacces-

sible 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 [14]. 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.

1.3 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.

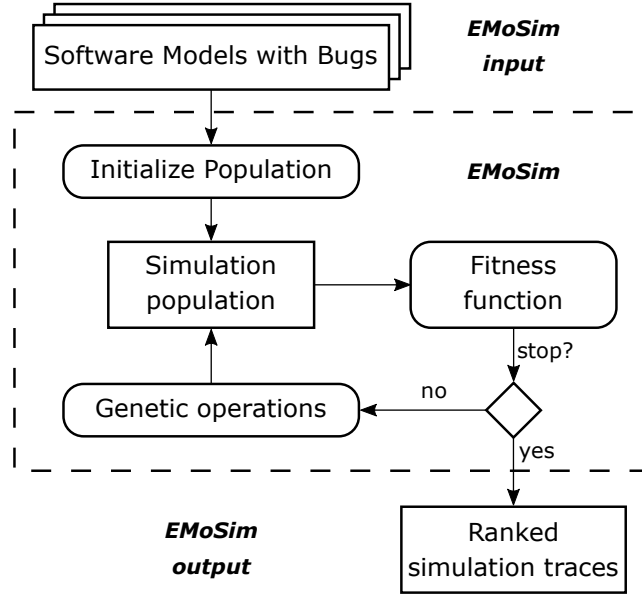


Figure 1.2: EMoSim approach.

1.3.1 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 1.2. 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.

1.3.2 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 (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.

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.

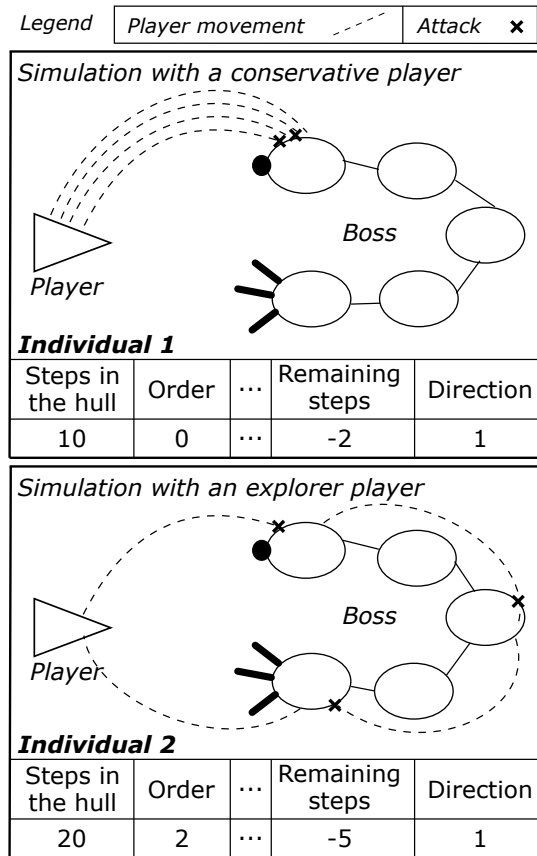


Figure 1.3: Representation of a simulation as an individual.

Figure 1.3 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.

Each example in Figure 1.3 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.

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):

$$victory = 1 - \frac{|V_O - V_P|}{V_O} \quad (1.1)$$

- *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):

$$health = 1 - \frac{\sum_{d=1}^{V_P} \frac{|H_O - H_P|}{H_O}}{V_P} \quad (1.2)$$

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.

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.

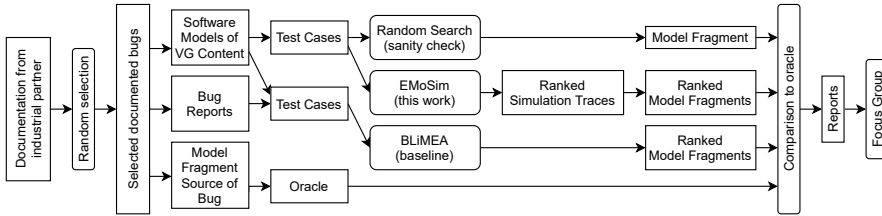


Figure 1.4: Evaluation process.

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.

1.4 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).

1.4.1 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 [3]), 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 that bug was manifested for EMoSim and RS. All of them were obtained from the documentation.

1.4.2 *Experimental setup*

Figure 1.4 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 [22]. 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 ($\frac{TP}{TP+FN}$) measures the number of elements of the model fragment from the oracle that are correctly retrieved by the proposed model fragment.

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

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

Table 1.1: Mean values and standard deviations for Recall, Precision, and F-measure for each type of bug.

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

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.

1.4.3 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 [2]. For EMoSim, we started from those reported in [8] (as we used the same simulation) and made sure they converged.

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>.

1.4.4 Results

In this section, we present the results obtained in EMoSim (this work), BLiMEA (baseline), and the RS (sanity check) approaches in Kromaia. Table 3.3 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 (RQ₁), 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 3.3 shows the mean values and standard deviations for recall, precision, and F-measure for each type of bug. For the second research question (RQ₂), 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.

1.4.5 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].

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 [18]. 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 3.4 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 1.2: 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	$1x10^{-9}$	$3x10^{-10}$	$3x10^{-10}$

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 $\hat{A}_{12}$ [33, 19]. $\hat{A}_{12}$ measures the probability that running one algorithm

Table 1.3: 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)

yields higher values than running another algorithm. If the two algorithms are equivalent, then $\hat{A}_{12}$ will be 0.5.

Table 3.5 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 [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 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.

1.4.6 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) [2]. 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 3.1). 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.

1.4.7 Threats to Validity

To acknowledge the threats to the validity of our work, we use the classification suggested by De Oliveira et al. [23].

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 meta-language 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 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.

1.5 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.

1.5.1 Bug localization in games

Ariyurek et al. [6] 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 [7], 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. [31] 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.

1.5.2 Bug localization in games that use models

Iftikhar et al. [21] 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.

1.5.3 Bug localization in models

Outside the game domain, several techniques are used for bug localization in models. Arcega et al. [3] 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. [30] 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. [28] 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 [27]. 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.

1.6 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.

Acknowledgements

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

Bibliography

- [1] Apostolos Ampatzoglou and Ioannis Stamelos. “Software engineering research for computer games: A systematic review”. In: *Information and Software Technology* 52.9 (2010), pp. 888–901. ISSN: 0950-5849. DOI: <https://doi.org/10.1016/j.infsof.2010.05.004> (cit. on p. 26).
- [2] Lorena Arcega et al. “An approach for bug localization in models using two levels: model and metamodel”. In: *Software and Systems Modeling* 18.6 (2019), pp. 3551–3576. DOI: 10.1007/s10270-019-00727-y (cit. on pp. 27, 41, 45).
- [3] Lorena Arcega et al. “Bug Localization in Model-Based Systems in the Wild”. In: *ACM Trans. Softw. Eng. Methodol.* 31.1 (2021), 10:1–10:32. ISSN: 1049-331X. DOI: 10.1145/3472616 (cit. on pp. 27, 38, 52).
- [4] Andrea Arcuri and Lionel Briand. “A Hitchhiker’s guide to statistical tests for assessing randomized algorithms in software engineering”. In: *Software Testing, Verification and Reliability* 24.3 (May 2014), pp. 219–250. ISSN: 0960-0833. DOI: 10.1002/stvr.1486 (cit. on pp. 41–43, 48).
- [5] Andrea Arcuri and Gordon Fraser. “Parameter tuning or default values? An empirical investigation in search-based software engineering”. In: *Empirical Software Engineering* 18.3 (2013), pp. 594–623. ISSN: 1573-7616. DOI: 10.1007/s10664-013-9249-9 (cit. on pp. 48, 49).
- [6] Sinan Ariyurek, Aysu Betin-Can, and Elif Surer. “Automated Video Game Testing Using Synthetic and Humanlike Agents”. In: *IEEE Transactions on Games* 13.1 (2021), pp. 50–67. DOI: 10.1109/TG.2019.2947597 (cit. on p. 50).
- [7] Sinan Ariyurek, Aysu Betin-Can, and Elif Sürer. “Enhancing the Monte Carlo Tree Search Algorithm for Video Game Testing”. In: *IEEE Conference on Games, CoG 2020, Osaka, Japan, August 24-27, 2020*. IEEE, 2020, pp. 25–32. DOI: 10.1109/CoG47356.2020.9231670 (cit. on p. 50).
- [8] Daniel Blasco et al. “An evolutionary approach for generating software models: The case of Kromaia in Game Software Engineering”. In: *J. Syst. Softw.*

- 171 (2021), p. 110804. DOI: 10.1016/j.jss.2020.110804 (cit. on pp. 34, 35, 41, 49).
- [9] Ilhem Boussaïd, Patrick Siarry, and Mohamed Ahmed-Nacer. “A survey on search-based model-driven engineering”. In: *Automated Software Engineering* 24.2 (2017), pp. 233–294. ISSN: 1573-7535. DOI: 10.1007/s10515-017-0215-4 (cit. on p. 32).
- [10] Loli Burgueño et al. “Static Fault Localization in Model Transformations”. In: *IEEE Transactions on Software Engineering* 41.5 (2015), pp. 490–506. ISSN: 0098-5589. DOI: 10.1109/TSE.2014.2375201 (cit. on p. 52).
- [11] Norman Cliff. *Ordinal methods for behavioral data analysis*. Lawrence Erlbaum Associates, Inc, 1996, p. 212. DOI: <https://doi.org/10.4324/9781315806730> (cit. on p. 44).
- [12] William Jay Conover. *Practical Nonparametric Statistics, 3rd Edition*. 3. ed. Wiley series in probability and statistics. USA: Wiley, 1999. VIII, 584. ISBN: 0471160687 (cit. on p. 42).
- [13] Crytek. *CryEngine*. <https://www.cryengine.com>. [Online; accessed 21-November-2021]. 2002 (cit. on p. 26).
- [14] África Domingo et al. “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*. Ed. by Schahram Dustdar et al. Vol. 12127. Lecture Notes in Computer Science. Springer, 2020, pp. 353–367 (cit. on p. 30).
- [15] Epic Games. *Unreal Engine*. <https://www.unrealengine.com>. [Online; accessed 21-November-2021]. 1998 (cit. on p. 26).
- [16] Raihana Ferdous et al. “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*. Ed. by Una-May O’Reilly and Xavier Devroey. Vol. 12914. Lecture Notes in Computer Science. Springer, 2021, pp. 56–71. DOI: 10.1007/978-3-030-88106-1_5 (cit. on p. 51).

- [17] PC Gamer. *How buggy is Cyberpunk 2077, really?* <https://www.pcgamer.com/how-buggy-is-cyberpunk-2077-really/>. [Online; accessed 21-November-2021]. 2020 (cit. on p. 26).
- [18] Salvador García et al. “Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: Experimental analysis of power”. In: *Information Sciences* 180.10 (May 2010). Special Issue on Intelligent Distributed Information Systems, pp. 2044–2064. ISSN: 0020-0255. DOI: <http://dx.doi.org/10.1016/j.ins.2009.12.010> (cit. on p. 42).
- [19] Robert J. Grissom and John J. Kim. *Effect sizes for research: A broad practical approach*. New Jersey, United States: Lawrence Erlbaum Associates Publishers, 2005 (cit. on p. 43).
- [20] Ahmed E. Hassan and Richard C. Holt. “The Top Ten List: Dynamic Fault Prediction”. In: *Proceedings of the 21st IEEE International Conference on Software Maintenance*. ICSM ’05. USA: IEEE Computer Society, 2005, 263–272. ISBN: 0769523684. DOI: 10.1109/ICSM.2005.91 (cit. on p. 27).
- [21] Sidra Iftikhar et al. “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*. Ed. by Timothy Lethbridge, Jordi Cabot, and Alexander Egyed. IEEE Computer Society, 2015, pp. 426–435. DOI: 10.1109/MODELS.2015.7338274 (cit. on p. 51).
- [22] Hisao Ishibuchi, Yusuke 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. DOI: 10.1109/CEC.2006.1688438 (cit. on p. 39).
- [23] Márcio de Oliveira Barros and Arilo Claudio Dias Neto. *Threats to Validity in Search-based Software Engineering Empirical Studies*. Tech. rep. 0006/2011. Jan. 2011 (cit. on p. 48).
- [24] Luca Pascarella et al. “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*. Ed. by Andy Zaidman, Yasutaka Kamei, and Emily Hill. ACM, 2018, pp. 392–402. DOI: 10.1145/3196398.3196418 (cit. on pp. 25, 26, 45).
- [25] Cristiano Politowski, Fábio Petrillo, and Yann-Gaël 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. DOI: 10.1109/AST52587.2021.00018 (cit. on pp. 26, 45).
- [26] Jeanine Romano et al. “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 (cit. on p. 44).
- [27] Jesús Sánchez Cuadrado, Esther Guerra, and Juan Lara. “Quick Fixing ATL Transformations with Speculative Analysis”. In: *Softw. Syst. Model.* 17.3 (July 2018), 779–813. ISSN: 1619-1366. DOI: 10.1007/s10270-016-0541-1 (cit. on p. 52).
- [28] Jesús Sánchez Cuadrado, Esther Guerra, and Juan Lara. “Static Analysis of Model Transformations”. In: *IEEE Transactions on Software Engineering* 43.9 (2017), pp. 868–897. DOI: 10.1109/TSE.2016.2635137 (cit. on p. 52).
- [29] SlashData. *Global developer population report 2019*. <https://sdata.me/GlobalDevPop19>. [Online; accessed 21-November-2021]. 2019 (cit. on p. 25).
- [30] Javier Troya et al. “Spectrum-Based Fault Localization in Model Transformations”. In: *ACM Trans. Softw. Eng. Methodol.* 27.3 (Sept. 2018). ISSN: 1049-331X. DOI: 10.1145/3241744 (cit. on p. 52).
- [31] Rosalia Tufano et al. “Using Reinforcement Learning for Load Testing of Video Games”. In: *Proceedings of the 44th International Conference on Software Engineering (ICSE 2022), Pittsburgh, USA*. 2022 (cit. on p. 50).
- [32] Unity Technologies. *Unity*. <https://unity.com>. [Online; accessed 21-November-2021]. 2005 (cit. on p. 26).

- [33] András Vargha and Harold D. Delaney. “A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong”. In: *Journal of Educational and Behavioral Statistics* 25.2 (2000), pp. 101–132. DOI: 10.3102/10769986025002101. eprint: <http://jeb.sagepub.com/content/25/2/101.full.pdf+html> (cit. on p. 43).
- [34] W. Eric Wong et al. “A Survey on Software Fault Localization”. In: *IEEE Trans. Software Eng.* 42.8 (Aug. 2016), pp. 707–740. ISSN: 0098-5589. DOI: 10.1109/TSE.2016.2521368 (cit. on pp. 26, 27, 38, 45).
- [35] Yan Zheng et al. “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. DOI: 10.1109/ASE.2019.00077 (cit. on p. 50).
- [36] Meng Zhu and Alf Inge Wang. “Model-driven Game Development: A Literature Review”. In: *ACM Comput. Surv.* 52.6 (2020), 123:1–123:32. DOI: 10.1145/3365000 (cit. on p. 26).
- [37] Thomas Zimmermann et al. “Mining Version Histories to Guide Software Changes”. In: *Proceedings of the 26th International Conference on Software Engineering*. ICSE ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 563–572. ISBN: 0-7695-2163-0 (cit. on p. 45).

Chapter 2

Studying the Influence and Distribution of the Human Effort in a Hybrid Fitness Function for Search-Based Model-Driven Engineering

Search-Based Software Engineering (SBSE) offers solutions that efficiently explore large complex problem spaces. To obtain more favorable solutions, human participation in the search process is needed. However, humans cannot handle the same number of solutions as an algorithm. We propose the first hybrid fitness function that combines human effort with human simulations. Human effort refers to human participation for providing evaluations of candidate solutions during the search process, whereas human simulations refer to recreations of a scenario in a specific situation for automatically obtaining the evaluation of candidate solutions. We also propose three variants for the hybrid fitness function that vary in the distribution of human effort in order to study whether the variants influence the performance in terms of solution quality. Specifically, we leverage our hybrid fitness function to locate bugs in software models for the video games of game software engineering. Video games are a fertile domain for these hybrid functions because simulated players are naturally developed as part of the video games (e.g., bots in First-Person Shooters). Our evaluation is at the scale of industrial settings with a commercial video game (Play Station 4 and Steam) and 29 professional video game developers. Hybridizing the fitness function outperforms the results of the best baseline by 33.46% in F-measure. A focus group confirms the acceptance of the hybrid fitness function. Hybridizing the fitness function significantly improves the bug localization process by reducing the amount of tedious manual work and by minimizing the number of bugs that go unnoticed. Furthermore, the variant that obtains the best results is a counter-intuitive result that was under the radar of the interactive SBSE community. These results can help not only video game developers to locate bugs, but they can also inspire SBSE researchers to bring hybrid fitness functions to other software engineering tasks.

2.1 Introduction

Search-Based Software Engineering (SBSE) addresses problems throughout the Software Engineering lifecycle (from requirements to maintenance) using search-based algorithms. Increasingly, the community has been paying more attention to SBSE [35] since it offers automated or semi-automated solutions that efficiently explore large complex problem spaces. For example, SBSE can be used to locate model fragments from an input query in a family of software products that has been developed and maintained by a company over years [30]. Only three key ingredients are needed to apply SBSE: 1) a representation (encoding) of the problem (e.g., a bit string); 2) the definition of the set of operations (e.g., mutation); and 3) the definition of the fitness function (e.g., similarity to the input query). Then, candidate solutions (which are encoded following the representation chosen) are evolved (by applying the operations) and are assessed (by the fitness function) in an iterative process until a stop condition is met (e.g., a time slot).

Although SBSE reformulates software engineering problems as search problems, some contexts require the human's subjective evaluation in the search process in order to obtain the most favorable solutions that alleviate some limitations of SBSE techniques (e.g., vocabulary mismatch and tacit knowledge). This refers to interactive SBSE (iSBSE) [57]. For example, Wang et al. [71] suggest that there are complex approaches that are partially successful on multiple fault problems that need more human intervention. Marculescu et al. [48] involve a human in the fitness function when the optimization goal depends on "human preference, intuition, emotion, and psychological aspects". Feldt [26] emphasizes the importance of collaboration between a human and a system in identifying and comprehending a bug's root cause. Most previous iSBSE works [33, 4, 51, 6, 41, 50, 49, 13, 1, 39, 54, 58, 44, 25, 40, 31] interleave the human's subjective evaluation with the evaluation that is performed by the algorithm until the stop condition is met. Although other works use the human's subjective evaluation differently (before [72, 47, 38, 64] or after [46, 70, 24, 12, 29, 34, 17, 43, 55, 18, 75, 2, 3] the algorithm), none of these iSBSE works compare whether a different distribution of the human evaluations improves the quality of the solutions.

For the first time, our previous work [54] completely replaced the fitness function with a human for Search-Based Model-Driven Engineering (SB-MDE), specifically, for locating features in software models. Software models are a more favorable context for humans than code because of their higher abstraction. Although involving a human as fitness function (HaFF) improves the results, human participation must be limited in order to prevent fatigue [66]. All in all, humans are not immune to fatigue, so logically they cannot handle and process the same number of evaluation requests as an algorithm. According to Takagi's [67] recommendations for avoiding human fatigue, both the size of the population to be evaluated and the number of iterations should be 10 or 20.

For more demanding problems (e.g., large-scale software systems) where human fatigue may appear due to the high number of human evaluations that are needed to locate a solution, it is necessary to combine the effort of an algorithm that automatically obtains the best solution possible with the human. Hybridizing the fitness function by exploiting both the algorithm and the human as fitness function in SBMDE can enrich the solution quality. Hence, in this work, we propose a hybrid fitness function that is the first to combine human effort with human simulations. On the one hand, human effort refers to human participation for providing manual evaluations of candidate solutions during the search process. On the other hand, human simulations refer to recreations of a scenario in a specific situation for automatically obtaining the evaluation of candidate solutions. Building a simulated human may be a bigger challenge than the task at hand. However, in the domain of video games that Game Software Engineering (GSE) tackles [5], there are human simulations that have been built as part of the development of the video game. Some examples of human simulations in video games are: the rival drivers in a racing game, the bots in a First-Person Shooter (FPS), or the generals of the enemy troops in a Real-Time Strategy (RTS).

In this work, we study the influence of hybridizing the fitness function in terms of solution quality in the context of Bug Localization (BL) in software models for GSE. Moreover, we propose three variants of the hybrid fitness function in order to study the influence in the quality of the solution of different ways of distributing the human effort during

BL: Variant 1 interleaves the assessment between the human and the algorithm; Variant 2 relies on the human assessment first, and then the assessment of the algorithm; and Variant 3 relies on the assessment the algorithm first, and then the human assessment.

In the evaluation, 29 professional video game developers were involved, acting as the human component of the fitness function with the objective of locating 29 bugs in Kromaia, which is a commercial video game about flying and shooting with a spaceship in a three-dimensional space¹. Kromaia was released on PC, PlayStation and translated to eight different languages.

To assess the performance of our hybrid fitness function in terms of solution quality, we apply metrics (recall, precision, and F-measure) that have been widely accepted by the software engineering community in the domain of evolutionary algorithms [9]. In order to put the performance of the variants into perspective, we set two baselines that include: only using the human as fitness function as proposed in [54], and only using an algorithmic fitness function as proposed in [19]. In order to compare the results of the variants with the baselines, we perform a statistical analysis (following the guidelines by Arcuri and Briand [8]) in order to provide quantitative evidence of the impact of the results and to show that this impact is significant.

The results show that all of the variants of our hybrid fitness function significantly outperformed the baselines in F-measure. Specifically, Variant 3 (the algorithm assessments first and the human afterwards) was the one that yielded the best results, improving the results of the best baseline (only using the human as fitness function) by 40.3% in recall, 26.46% in precision, and 33.46% in F-measure. All of the comparisons show significant differences except when the recall of Variant 1 (interleaving the assessment between the human and the algorithm) and Variant 2 (the human assessment first and the algorithm afterwards) is compared to the baseline that only uses the algorithm as fitness function. The significant improvement in precision and F-measure in all of the comparisons comes from the influence of the human on the unattended algorithm. According

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

to the magnitude scales of the Cliff Delta values [59], the magnitude of improvement using Variant 3 instead of the baselines can be considered large.

To the best of our knowledge, this is the first SBMDE work that empowers humans with a hybrid fitness function that combines human assessment with simulations during BL in GSE and which obtains more favorable solutions than the baselines at an industrial scale. Specifically, we claim that:

- Our hybrid fitness function significantly improves the results of BL compared to the baselines. Furthermore, we created a focus group that confirms the acceptance of using a hybrid function in the context of BL for GSE. Our results can also motivate other researchers to use a hybrid fitness function to benefit other software maintenance activities.
- The variant of the hybrid fitness function that obtains the best results distributes all of the effort of the algorithm at the beginning and the human afterwards (Variant 3). This result is counter-intuitive from an initial survey that we made. The group of video game developers responded that Variant 1 would obtain the best results (interleaving simulations automatically produced by the evolutionary algorithm with human evaluations).
- Previous works that have involved the human in the fitness function (e.g., the human refines the algorithm's solution [40]) interleave the human participation and the algorithm execution. These works do not consider other approaches to distribute human effort. Our work reveals that interleaving is not the approach that achieves the best results. We acknowledge that our results are obtained in a different context of simulation-human-based hybrid fitness function, but our results might motivate other researchers to reconsider the decision of using interleaving as the default approach.
- The current shortage of bug-localization approaches in video games results in longer development times, delayed deadlines, and postponed release dates. Hybridizing the fitness function significantly

improves the BL process by reducing the amount of tedious manual work and minimizing the number of bugs that go unnoticed.

The remainder of this paper is organized as follows. Section 2.2 introduces bug localization in the GSE domain. Section 2.3 reviews the related work. Section 2.4 describes the variants that we propose for hybridizing and distributing human effort as fitness function. Section 2.5 presents the evaluation. The results are reported in Section 2.6 and discussed in Section 2.7. Section 2.8 describes the threats to validity. Finally, Section 2.9 concludes the paper.

2.2 Background

This section introduces how SBMDE is used to locate bugs in the software models of a commercial video game by leveraging game simulations. In this context, we also motivate the need for an approach that involves a human in the evaluation of candidate solutions. This section also introduces iSBSE.

2.2.1 *SBMDE for locating bugs in a commercial video game*

Our previous work [19] uses SBMDE to find the model fragments that are the source of bugs in a commercial video game (Kromaia). The content of Kromaia is specified using the Shooter Definition Model Language (SDML), which is a Domain Specific Language (DSL) model for the video game domain [14]. This DSL adheres to the fundamental principles of MDE using models for Software Engineering. The models are built with SDML and interpreted at runtime.

SDML defines aspects that are included in video game entities such as bosses, which must be defeated in order to complete a level. The definition of a boss includes: the anatomical structure (including which parts are used in it, their physical characteristics, and how they are connected to one another); the quantity and distribution of vulnerable parts, weapons, and defenses in the structure/body of the character; and the movement behaviors associated to the whole body or its parts. This

modeling language includes concepts such as hulls, links, weak points, weapons, and AI components.

Fig. 2.1 shows a simplified example of the graphical representation of a boss and a player. For further information, the following URL contains examples of the models, the metamodel, and an online visualizer that displays the models as they would appear in the Kromaia video game: <https://svit.usj.es/tse23/bl-in-mgse>.

To locate the bugs in Kromaia, our previous work [19] uses game simulations. The simulations mimic a duel between a boss and an algorithm that can behave like a human player (i.e., the simulated player). During a simulation, the boss acts in accordance with the anatomy, behavior, and attack/defense balance that are included in its model, trying to defeat the simulated player, whereas the simulated player confronts the boss in order to destroy the available weak points. Both the boss and the simulated player try to win the fight, avoid draw/tie games, and make sure there is a winner in the simulation.

Fig. 2.1 shows two examples of simulations. Each simulation emulates the behavior of a player when the battle with the boss occurs. The algorithm can fight a boss by applying different strategies. The fighting strategy can therefore be defined by parametrizing the algorithm. Thus, different player profiles can be formed. 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 (forward or backward).

Each example in Fig. 2.1 corresponds to different fighting strategies applied to a simulation. In both cases, the triangle corresponds to the simulated player, the set of connected circles and lines correspond to the boss, the dashed and dotted lines correspond to the path that follows the simulated player in his/her strategy to defeat the boss, and the crosses correspond to the attacks that the simulated player performs to the hulls. The upper example of Fig. 2.1 shows the simulation of a conservative player in which the player attacks each hull one at a time. The

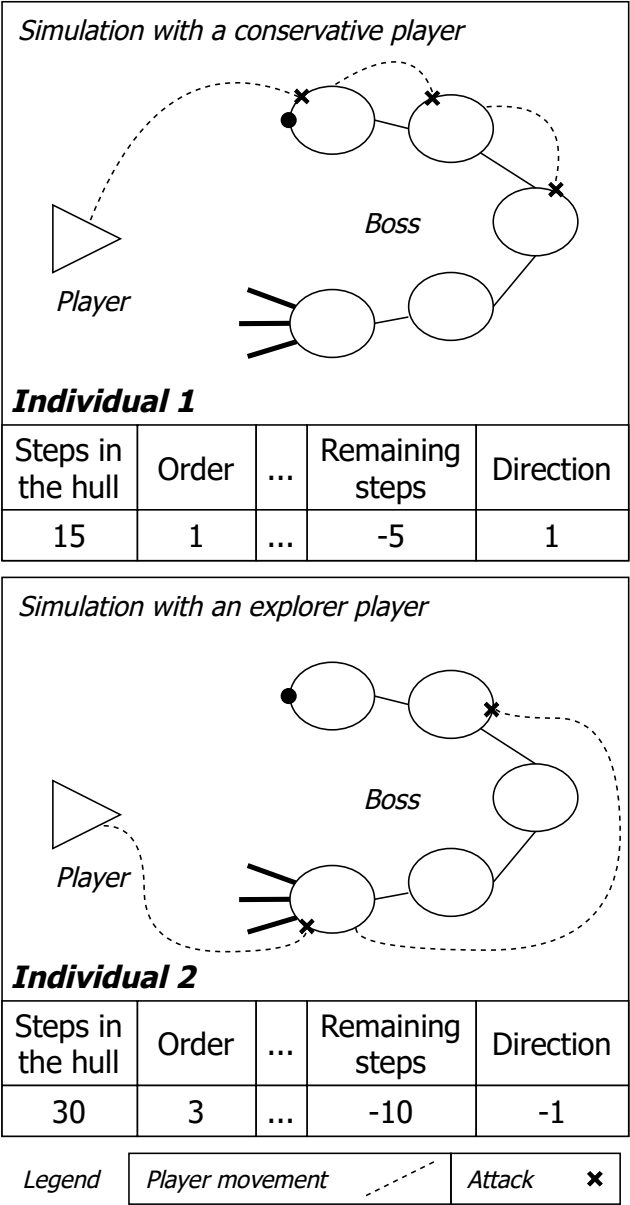


Figure 2.1: Examples of simulations between a player and boss.

lower example of the figure shows the simulation of an explorer player in which the player attacks the last hull of the boss, skips two hulls, and then attacks the following hulls up to the head of the boss.

Fig. 2.1 shows a set of parameters applied to a simulation for each example. Each set of parameters defines a candidate solution (individual). Each individual is encoded as a vector representation whose size 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. Furthermore, a scenario is defined as the union between the simulation parameters (i.e., duel settings), the agent parameters (i.e., the parameters that define the behavior of a player in the simulation), and the software model (e.g., the model of the boss) to be analyzed for potential bug localization. The fitness function of our previous SBMDE work [19] rewards simulations that are the farthest from what is expected (i.e., the further from what is expected, the more relevant when it comes to locating a bug).

To obtain more favorable solutions, the human should be involved in the search process as previous works have detected [54, 71, 48, 26]. However, human participation must be limited to prevent fatigue from demanding problems where the humans cannot handle the same number of candidate solutions as an algorithm [67, 54]. For these demanding problems, we propose to study the influence and distribution of human effort in a hybrid fitness function during the location of model fragments that are sources of bugs.

2.2.2 *Interactive SBSE*

Interactive SBSE (iSBSE) has been formalized as an emergent subarea within SBSE that promotes active human effort by providing intermediate results for inspection by the humans. Human feedback is later integrated into either the problem formulation or the search process so that the algorithm progressively adapts the search to the human's preferences [67, 58]. A recent survey on iSBSE [57] acknowledges that any attempt to involve the human in the search process with the aim of adapting the results to the human's preferences can be viewed as iSBSE.

The use of interactivity during the search process has shown more favorable solutions than fully automated solutions since some limitations of SBSE techniques (e.g., vocabulary mismatch and tacit knowledge) are alleviated [57, 71, 48, 26, 54].

Nevertheless, iSBSE also poses new challenges like dealing with the inherent human fatigue. Table 2.1 shows the strategies for addressing human fatigue of the 33 iSBSE works, which are also compared in the next section (related work). Most of the iSBSE works (81.82%) do not report strategies for addressing human fatigue. Specifically, 57.58% of the iSBSE works do not explicitly mention fatigue (“-” in Column 2 of the table), whereas 24.24% of the works explicitly mention that they do not limit the time or iterations for human evaluations (“No limit” in Column 2 of the table). The rest of the iSBSE works (18.18%) limit the human participation with a set of interaction-related parameters to adapt the human participation to their needs (“Parameter Limit: *parameter*” in Column 2 of the table). These parameters limit the number of interactions, evaluations, or the time that the human can spend evaluating. The previous parameters limit the human participation to avoid fatigue.

As in the previous works, we also limit the number of human evaluations and iterations in this work to address human fatigue since humans cannot evaluate the same solutions as an algorithm. To set this limit, we follow Takagi’s [67] recommendations, which are that both the size of the population to be evaluated and the number of iterations should be 10 or 20. Therefore, we address fatigue as in previous works (limiting the human participation), but, in this work, we focus on studying the influence of different ways of distributing the human effort during BL on the quality of the solution.

2.3 Related work

Table 3.6 shows the related work, which is organized in two main parts in order to take into account the topics that are covered in the paper (BL and human effort in a hybrid fitness function). Also, the lower part of the table includes a row to compare the related work with our work. The columns of the table show: the related work (Column 1); if the work

Table 2.1: Strategies for addressing human fatigue in iSBSE related work

Strategy for addressing fatigue	
Ghannem et al. [33]	—
Amal et al. [4]	—
Lin et al. [46]	—
Yue et al. [72]	—
Van Rooijen and Hamann [70]	—
Lu et al. [47]	—
Debreceni et al. [24]	—
Batot and Sahraoui [12]	No limit
Fleck et al. [29]	—
Martínez et al. [51]	Parameter Limit: interactions
Gómez-Abajo et al. [34]	—
Calinescu et al. [17]	—
Araujo et al. [6]	No limit
Kessentini et al. [41]	No limit
Martínez et al. [50]	—
Marculescu et al. [49]	—
Kolchin [43]	—
Filho et al. [38]	Parameter Limit: evaluations
Bindewald et al. [13]	Parameter Limit: interactions
Procter et al. [55]	—
Le Calvar et al. [18]	—
Zubcoff et al. [75]	—
Alkhazi et al. [2]	—
Alkhazi et al. [3]	No limit
Silva et al. [64]	—
Alizadeh et al. [1]	No limit
Kessentini et al. [39]	No limit
Pérez et al. [54]	Parameter Limit: interactions and evaluations
Ramírez et al. [58]	Parameter Limit: time
Kuviatkovski et al. [44]	No limit
Delgado-Pérez et al. [25]	Parameter Limit: interactions
Kessentini et al. [40]	No limit
Freire et al. [31]	—

includes human interaction before, during, or after of the search process (Column 2); if the human participation is done during the search process

in the fitness function by assessing candidate solutions (Column 3); if the fitness function is a hybrid combining a simulation and the effort of a human (Column 4); whether or not the distribution of the human effort in the fitness function is studied (Column 5); and if the evaluation performed explicitly mentions that industry is involved (Column 6). In each cell of the table, we use either a check mark (to indicate that the work explicitly addresses what is mentioned in the column) or a cross mark otherwise.

The upper part of Table 3.6 includes 18 BL works which mainly cover: BL in games, BL in models, and BL in games that use models. Using the query presented in [19], we collected new papers up to January 2023 and identified those already covered to maintain consistency. For instance, Troya et al. [68] locate faulty rules in model transformations by applying Spectrum-Based Fault Localization (SBFL), where the human picks one assertion from the ranking to locate and fix the faulty rule that made the assertion fail. Arcega et al. [7] evaluate different model-based BL approaches in order to mitigate the effect of starting the localization in the wrong place. Software engineers are enabled to modify the solution obtained (e.g., by adding or removing model elements to a model fragment that the BL approach has obtained). Our previous work in BL in software models of video games, Casamayor et al. [19], uses an evolutionary algorithm where the candidate solutions are automatically assessed by a fitness function that collects information about the simulations. Only two BL works [68, 7] include human interaction, but neither of them consider the human in the assessment of candidate solutions in the fitness function, as Column 3 of Table 3.6 shows. Therefore, none of these BL works have a hybrid fitness function that combines a simulation and the effort of a human or the study of how human effort should be distributed to obtain the best quality solution possible.

The middle part of Table 3.6 shows the works of two recent surveys that include SBMDE works from 1998 to 2016 [15], and iSBSE works from 1999 to 2017 [57]. Using the queries presented in the surveys [15, 57], we updated both surveys from 2016 and 2017 until January 2023, respectively, and we identified the common works (see the 33 works in the middle part of the table). All of the works include human interaction, but

Table 2.2: Related work in bug localization and iSBSE

	Human interaction	In the fitness function			Industrial scale
		Human assessment	Hybrid: simulation and human	Studies human effort distribution	
Burgueño et al. [16]	✗	✗	✗	✗	✗
Ifikhar et al. [36]	✗	✗	✗	✗	✓
Sánchez-Cuadrado et al. [62]	✗	✗	✗	✗	✗
Sánchez-Cuadrado et al. [61]	✗	✗	✗	✗	✗
Troya et al. [68]	✓	✗	✗	✗	✗
Ariyurek et al. [10]	✗	✗	✗	✗	✗
Zheng et al. [74]	✗	✗	✗	✗	✓
Ariyurek et al. [11]	✗	✗	✗	✗	✗
Cheng et al. [20]	✗	✗	✗	✗	✗
Zhang et al. [73]	✗	✗	✗	✗	✗
Arcega et al. [7]	✓	✗	✗	✗	✓
Ferdous et al. [28]	✗	✗	✗	✗	✗
Quach et al. [56]	✗	✗	✗	✗	✗
Casamayor et al. [19]	✗	✗	✗	✗	✓
Ciborowska et al. [21]	✗	✗	✗	✗	✓
Khanfir [42]	✗	✗	✗	✗	✓
Liang et al. [45]	✗	✗	✗	✗	✓
Tufano et al. [69]	✗	✗	✗	✗	✓
Ghannem et al. [33]	✓	✓	✗	✗	✗
Amal et al. [4]	✓	✗	✗	✗	✗
Lin et al. [46]	✓	✗	✗	✗	✗
Yue et al. [72]	✓	✗	✗	✗	✗
Van Rooijen and Hamann [70]	✓	✗	✗	✗	✗
Lu et al. [47]	✓	✗	✗	✗	✗
Debreteni et al. [24]	✓	✗	✗	✗	✓
Batot and Sahraoui [12]	✓	✗	✗	✗	✗
Fleck et al. [29]	✓	✗	✗	✗	✗
Martínez et al. [51]	✓	✓	✗	✗	✗
Gómez-Abajo et al. [34]	✓	✗	✗	✗	✗
Calinescu et al. [17]	✓	✗	✗	✗	✗
Araujo et al. [6]	✓	✓	✗	✗	✓
Kessentini et al. [41]	✓	✓	✗	✗	✗
Martínez et al. [50]	✓	✓	✗	✗	✗
Marculescu et al. [49]	✓	✓	✗	✗	✓
Kolchin [43]	✓	✗	✗	✗	✗
Filho et al. [38]	✓	✗	✗	✗	✗
Bindewald et al. [13]	✓	✓	✗	✗	✗
Procter et al. [55]	✓	✗	✗	✗	✗
Le Calvar et al. [18]	✓	✗	✗	✗	✗
Zubcoff et al. [75]	✓	✗	✗	✗	✗
Alkhazi et al. [2]	✓	✗	✗	✗	✗
Alkhazi et al. [3]	✓	✗	✗	✗	✗
Silva et al. [64]	✓	✗	✗	✗	✗
Alizadeh et al. [1]	✓	✓	✗	✗	✓
Kessentini et al. [39]	✓	✓	✗	✗	✗
Pérez et al. [54]	✓	✓	✗	✗	✓
Ramírez et al. [58]	✓	✗	✗	✗	✗
Kuviatkovski et al. [44]	✓	✗	✗	✗	✗
Delgado-Pérez et al. [25]	✓	✗	✗	✗	✓
Kessentini et al. [40]	✓	✓	✗	✗	✗
Freire et al. [31]	✓	✓	✗	✗	✗
Our work	✓	✓	✓	✓	✓

only 36.36% of the works involve human assessment in the fitness function to evaluate candidate solutions. Of the works that include human assessment in the fitness function, only four of them involve industry in the evaluation. For example, Ghannem et al. [33] involve human assessment in the fitness function by combining structural similarity and designers' ratings to evaluate the model refactorings. Araujo et al. [6] present an architecture that combines interactive optimization and machine learning to address the next release problem in software engineering, incorporating human expertise and preferences for efficient requirement selection. Marculescu et al. [49] suggest optimizations for test cases using a dynamic fitness function that changes the weights of the objective values according to the assessment that is made by a human. The works by Kessentini et al. [39, 40] involve the human in the fitness function by interacting at the solution level by accepting/rejecting/modifying specific edit operations, and then computing the weighted probability of edit operations and target model elements.

Martinez et al. [51] completely replace the fitness function with a human, but they do not provide evidence supporting the idea that the human is beneficial. Their work makes a comparison between the human (combined with crossover and mutation operations) and Random Search. One of our previous works, Pérez et al. [54], completely replaces the fitness function with a human in a real-world industrial case study for Feature Location in Models (FLiM). Although the results show that the human as fitness function is beneficial, it is acknowledged that a hybrid approach could be explored to avoid human fatigue for more demanding problems. As Column 4 of Table 3.6 shows, none of the works explore a hybrid fitness function that combines simulations with humans, as this work does.

It is important to highlight that all of the works that combine human assessment with an algorithm in the fitness function [33, 6, 49, 13, 1, 39, 40, 31] do not address either BL or GSE, and they interleave the human effort with the assessment of the algorithm. For example, even though the works by Kessentini et al. [39, 40] study if the number of human interactions and time can be reduced by changing the algorithm, neither of them (see Column 5 of Table 3.6) study whether the distribution of the

human effort should be different in order to improve the quality of the solution (e.g., doing all human assessment first and then the algorithm assessment instead of interleaving the human-algorithm).

To address this lack, we propose three variants of the hybrid fitness function in order to study the influence in the quality of the solution of different ways of distributing the human effort during BL. The variants: 1) interleave the assessment of the candidate solutions between the human and the algorithm (distribution in the hybrid fitness function that is used by most of the related works); 2) rely on the assessment performed by the human, and then the assessment of the algorithm; or 3) the reverse (the assessment performed by the algorithm, and then the assessment of the human). Specifically, we study whether the selected variant influences the solution quality for SBMDE in the context of BL in GSE.

2.4 Hybridizing the fitness function

Fig. 3.5 shows an overview of our approach, SimuHaFF (Simulation and Human as the Fitness Function), for BL in GSE. Our approach consists of an evolutionary algorithm where the fitness function is hybrid. It is comprised of the assessment of candidate solutions by both the score automatically provided by an algorithm using simulations and the score provided by a human. The upper part of Fig. 3.5 depicts the set of software models, which the algorithm takes as input to locate the bug. Afterwards, the population of scenarios is initialized by a random selection from the input models. The goal is to obtain a ranked list of simulation traces that are ordered by their relevance in locating the bug.

The hybrid fitness function assesses each of the scenarios. The fitness provided by the algorithm runs the simulations as described in Section 2.2. In order to enable the human to assess candidate solutions, the approach displays the model fragments that are candidate solutions to be potential sources of bug. Then, the human provides the fitness score to each candidate solution, which is given on a scale of 1 to 7 with a maximum score of 7 (for the best candidates of having the bug) and a minimum of 1. The human evaluation is performed using a seven-point scale rather than a

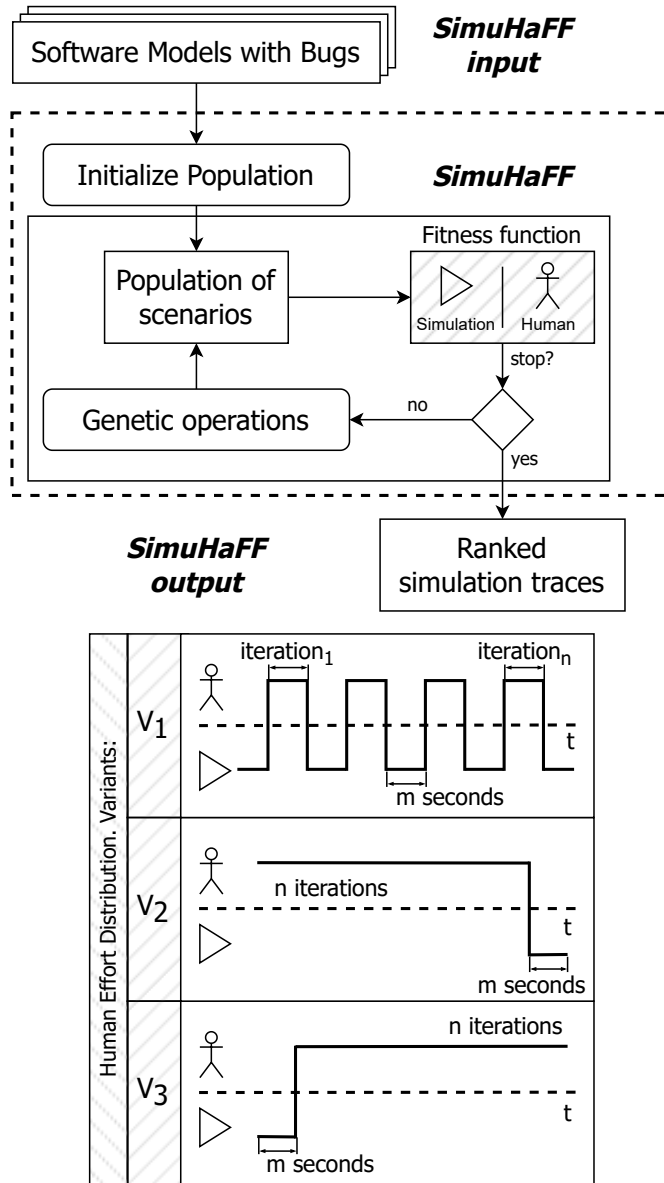


Figure 2.2: Overview and variants for hybridizing the fitness function.

broad rating to reduce the fatigue of the developers, as recommended in [67].

In order to study how the distribution of the human effort in a hybrid fitness function influences the results, we propose three variants that take into account Takagi's [67] recommendations for avoiding human fatigue. These recommendations are the reduction of both the size of the population to be evaluated and the number of iterations (to 10 or 20). The variants (V1-V3) that we propose are:

Variant 1 interleaves simulations that are performed by the algorithm with human evaluations. The simulations are executed by pre-setting a time m (in seconds) each time that the human provides an evaluation. In total, the human performs n evaluations. Fig. 3.5-V₁ depicts how the simulations are interleaved with human evaluations in this variant. The upper part above the dotted line represents the human evaluation, whereas the lower part shows the evaluation that is automatically produced by the evolutionary algorithm using simulations.

Variant 2 relies on n consecutive human evaluations and then launching the simulations of the algorithm for m seconds. Fig. 3.5-V₂ depicts the distribution of the human evaluations first and then the simulations of the algorithm.

Variant 3 reverses V2 by launching the simulations of the algorithm for m seconds and then the n consecutive human evaluations. Fig. 3.5-V₃ shows the distribution of this variant.

Once the candidate solutions are assessed by both the algorithm and the human, new scenarios are generated if the stop condition of the algorithm is not met. To do this, existing scenarios are selected using the wheel selection mechanism, where the selection of a scenario is inversely proportional to its relative fitness in the population. To modify the scenario, we use the widespread single-point crossover and random mutation (the most popular choice in SBMDE [53]).

Finally, when the stop condition of the algorithm is met, the output is a ranking of simulation traces, which are ordered by their relevance in locating the bug.

2.5 Evaluation

This section presents the evaluation of our work: the research questions that we aim to answer, the evaluation process (including recall, precision, and F-measure to measure the quality of the solutions; baselines; and statistical analysis), and the implementation details.

2.5.1 Research questions

We aim to answer the following research questions:

RQ₁: *What is the performance in terms of solution quality using the three variants for distributing the human effort in the hybrid fitness function and the baselines in BL?*

RQ₂: *Is the difference in the quality of the solution between the variants and the baselines significant?*

RQ₃: *How much is the quality of the solution influenced by using each variant compared to each baseline?*

2.5.2 Planning and execution

Fig. 2.3 shows an overview of the evaluation process to answer the research questions. The upper part of the figure shows the software models with bugs selected from the bug catalog provided by our industrial partner, which are the inputs for the test cases.

In the figure, the output of the test cases are two baselines and the three variants of our approach. The first baseline, HaFF, has the fitness function completely replaced by a human, and it outputs a sorted collection of model fragments that are considered to be the most relevant for the bug. The second baseline, EMoSim, has the unattended algorithm pro-

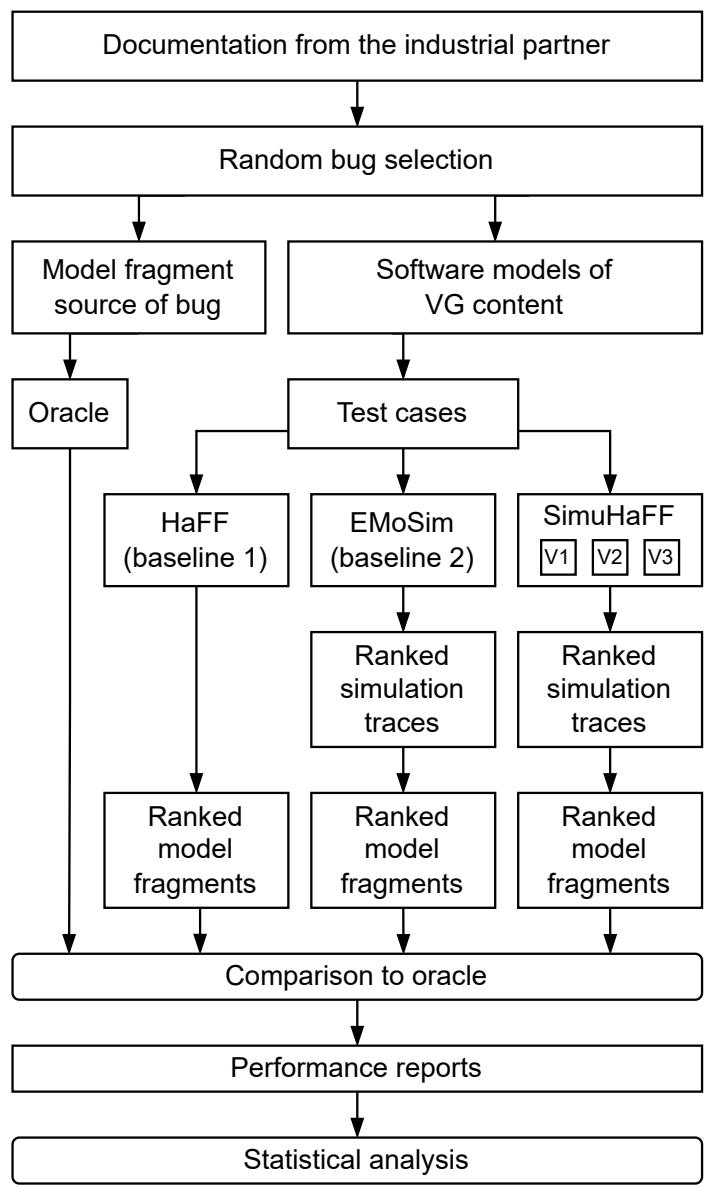


Figure 2.3: Overview of the evaluation process.

ducing simulation traces as the fitness function, but it outputs a ranking of simulation traces as described in Section 2.2. The trace contains all of the model elements that the interpreter has used at run-time 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. The three variants of our SimuHaFF approach work as described in Section 2.4 to form the model fragments that are considered to be the most relevant for the bug.

Answering RQ₁: After executing the two baselines and the three variants of our approach, we take the first solution in the ranking for each of the bugs as suggested in [37]. Afterwards, we compare the solution against the oracle (i.e., the ground truth) in order to get a confusion matrix. The confusion matrix is a table that provides detailed information about the performance of a classification algorithm. In our work, each candidate solution is a model fragment that is composed of a subset of model elements (where the bug is to be found). The presence or absence of model elements is considered as a classification since the granularity is at the model element level. Accordingly, our confusion matrix distinguishes between two specific values: TRUE (present) or FALSE (absent).

A confusion matrix groups the results of a comparison into four separate categories: True positive (TP), when an element that is present in the predicted model fragment is also present in the model fragment from the oracle; True Negative (TN), when an element that is not present in the predicted model fragment is not present in the model fragment from the oracle; False Positive (FP), when an element that is present in the predicted model fragment is not present in the model fragment from the oracle; and False Negative (FN), when an element that is not present in the predicted model fragment is present in the model fragment from the oracle.

Once the TP+TN+FP+FN are calculated, some measurements are extracted from the confusion matrix in order to assess the performance in terms of the solution quality of the approach. For each baseline and variant of our approach, we specifically generate a report with three performance metrics (recall, precision, and F-measure) that are widely accepted in the software engineering research community [60]: Recall ($\frac{TP}{TP+FN}$)

measures the number of elements of the model fragment from the oracle that are correctly retrieved by the proposed model fragment; Precision ($\frac{TP}{TP+FP}$) measures the number of elements from the proposed model fragment that are correct according to the ground truth (the oracle); and F-measure ($2 * \frac{Precision * Recall}{Precision + Recall}$) corresponds to the harmonic mean of precision and recall.

Recall and precision values can range between 0 and 1. A recall value of 0 means that no single model element from the model fragment from the oracle is present in any of the model fragments of the predicted solution), whereas a value of 1 means that all of the model elements from the oracle are present in the predicted solution. A precision value of 0 means that no single model element from the model fragment predicted is present in the model fragment from the oracle, whereas a value of 1 means that 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.

Answering RQ₂: To properly compare the approaches, all of the data resulting from the empirical analysis was analyzed using statistical methods following the guidelines in [8]. 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 [32]. The Quade test compares the results of multiple algorithms to determine whether there are significant differences among them. In addition, according to Conover [23], the Quade test has shown better results than the others when the number of algorithms is low (no more than four or five algorithms). 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 Holm's 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.

Answering RQ₃: 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 [8]. Thus, it is important to assess if an algorithm is statistically better than another and to assess the magnitude of the improvement. To assess how much the quality of the solution is influenced by using SimuHaFF compared to the baselines, the magnitude of the improvement should be assessed through *effect size* measures. For a non-parametric effect size measure, we used Cliff’s delta [22]. Cliff’s delta is an ordinal statistic that describes the frequency with which an observation from one group is higher than an observation from another group compared to the reverse situation. It can be interpreted as the degree to which two distributions overlap with values ranging from -1 to 1. For example, when comparing Base2_EMoSim and V3_SimuHaFF a value of 0 means there is no difference, a value of -1 means that all samples in Base2_EMoSim are lower than all samples in V3_SimuHaFF, and a value of 1 means the opposite (all samples in Base2_EMoSim are higher than all samples in V3_SimuHaFF). Moreover, threshold values were defined in [59] for the interpretation of Cliff’s delta effect size ($|d| < 0.147 \rightarrow$ "negligible"; $|d| < 0.33 \rightarrow$ "small"; $|d| < 0.474 \rightarrow$ "medium", $|d| \geq 0.474 \rightarrow$ "large"). We record a Cliff’s delta value for each pair-wise comparison in recall, precision, and F-measure.

2.5.3 Implementation details

In order to compare the baseline and variants of our approach fairly, we chose the parameters shown in Table 3.2 to calibrate the evolutionary algorithm and the fitness. As the table shows, the number of iterations the human performs is 10 and the number of candidate solutions (individuals) that the human evaluates per iteration is also 10. 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 with the automatic fitness run long enough to obtain the best solutions. In this way, the number of possible simulations is defined by how many scenarios can be simulated during that time. Even though the population size is at most 100 scenarios, we only present to the human the best 10 in all of the variants and in the baseline that includes the human in the fitness function (HaFF) in order to prevent human fatigue [54]. We assume that the best candidate solutions are those with the lowest fitness value since they are ordered by ascending fitness. For the rest of parameters in the table, we used those settings that are commonly used in previous works [54][19].

Table 2.3: Parameter settings

	Parameter description	Value
Fitness		
HaFF and SimuHaFF	Iterations	10
	Individuals for subjective evaluation	10
EMoSim and SimuHaFF	% _{playerWin} : Percentage goal of player's winnings	0.33
	% _{playerWinLife} : Percentage goal of life left when player wins	0.35
	Stop: Time (in seconds) from convergence	10
Evolutionary algorithm		
HaFF	Size: Population size	10
EMoSim and SimuHaFF	Size: Population size	100
	PhitByWeapon: Weapon hit probability	0.014
	PhitByPlayer: Player hit probability	0.25
Crossover and mutation	μ : Number of parents	2
	λ : Number of offspring from μ parents	2
	$p_{crossover}$: Crossover probability	0.9
	$p_{mutation}$: Mutation probability	0.1

In the evaluation, 29 professional video game developers participated. The recruitment of developers was carried out by our industrial partner, named Kraken Empire, who randomly selected the developers to participate. Kraken Empire is an independent game development studio, which is specialized in interactive 3D graphics applications, physics based simulations and real time systems. Fig. 2.4 shows the distribution of the developers' work experience. One out of every three has been developing video games for 15+ years, and the other two of them have developed video games for seven years or less. They all participated in the development of Kromaia, either from its inception (the most experienced developers) or creating new content for the game (19 developers). They

were professionals holding a degree in computing and related areas. All of them were from Spain, except for one who was from Latin America.

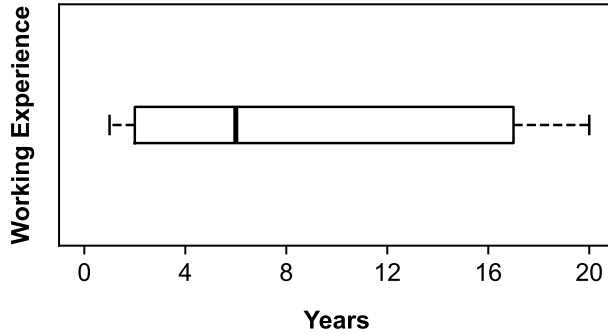


Figure 2.4: Working experience of the participants.

With regard to the time that each developer spends on the evaluation, we chose a slot of two hours since it is the median duration of software engineering experiments [65]. Hence, each developer participates in the localization of four randomly assigned [63] bugs in total (one bug per variant of the approach, plus another one for pure baseline HaFF). Hence, 116 bugs (4×29) were located in total. The assignment was transparent to the developers, so they do not know whether they are locating the bug using one variant of the approach or the baseline HaFF. The decision to only select one bug to locate for each variant of the approach plus baseline was based on the time needed to rate an individual, which could last an average of 15 seconds. Thus, the time that each developer spends on the evaluation of individuals to locate the four bugs is 100 minutes: 10 (individuals for subjective evaluation) $\times 10$ (iterations) = 100 (ratings/bug) $\times 15$ (seconds/rating) = 1500 (seconds/bug) $\times 4$ (bugs) = 6000 seconds (100 minutes).

The remaining 20 minutes are to provide a brief tutorial before the evaluation is started, to conduct a focus group when they finish and, indeed, to add a margin that ensures the completion of the evaluation. The brief tutorial mainly consisted of: 1) describing the objective of locating bugs in the context of Kromaia; 2) showing the main interface aspects to eval-

uate individuals; 3) providing a simple example (not using the bugs that will be located during the evaluation).

The experiment was conducted as scheduled. The session was held on December 14, 2022, at the San Jorge University facilities. None of the subjects in this experiment exceeded the established evaluation time limit. On average, the evaluation lasted 87 minutes, and the minimum and maximum duration were 63 and 100 minutes, respectively.

In order to collect the experimental data we spent about 1 hours and 17 minutes per participant on analyzing the results obtained. We analyzed a total of 29 participants, putting in a total of approximately 37.5 hours. The data related to participant characteristics (e.g., experience) were gathered automatically. The experiment was carried out according to the planned schedule. Therefore, there were no deviations during the execution of the experiment.

The evaluation of SimuHaFF and the baselines was done using identical gaming PCs with the same features. These were three ASUS ROG Strix laptops, using an Intel Core i7-6700HQ processor with 16 GiB of RAM, and running on a 64-bit Windows 10 host operating system. There are two reasons why the PCs were identical: (i) to avoid a potential bias when running the simulations, as technical specifications can influence the number of simulations run, and (ii) since the evaluation by professional video game developers is extremely time-consuming from a computer's perspective [67]. With these specifications, the automatic fitness part converges in 10 seconds and is capable of running more than 470k simulations on average.

The implementation uses the Java(TM) SE Runtime Environment (build 17.0.5), together with Kotlin as the programming language. For purposes of replicability, the implementation source code and the data (software models and oracles) as well as a screenshot of the graphical interface used by humans to evaluate the candidate solutions and the CSV files used as input in the statistical analysis are publicly available at the following URL: <http://www.gamesoftwareengineering.com/tse23/bl-in-mgse>.

2.6 Results

In this section, we present the results obtained in the two baselines (Base1_HaFF and Base2_EMoSim) and in the three variants of our approach (V1-V3_SimuHaFF) in Kromaia.

2.6.1 Research Question 1

Table 3.3 shows the mean values and standard deviations for recall, precision, and F-measure for each baseline and variant. All of the variants and the baseline that includes the human in the fitness function (Base1_HaFF) obtained better results than Base2_EMoSim. Specifically, V3_SimuHaFF (simulations generated by the algorithm first and then the human evaluations) yielded the best results, followed by V2_SimuHaFF and then V1_SimuHaFF.

Table 2.4: Mean values and standard deviations for Recall, Precision, and F-measure for each variant.

	Recall	Precision	F-measure
V1_SimuHaFF	75.38 ± 3.98	65.64 ± 9.62	69.67 ± 5.17
V2_SimuHaFF	73.54 ± 4.58	75.86 ± 5.37	74.53 ± 3.66
V3_SimuHaFF	91.64 ± 4.29	84.84 ± 9.59	87.79 ± 5.74
Base1_HaFF	51.34 ± 7.38	58.38 ± 4.8	54.33 ± 5.03
Base2_EMoSim	67.33 ± 19.84	43.54 ± 19.66	49.86 ± 13.56

RQ₁ answer. The results of all of the variants reveal that hybridizing the fitness functions pays off in the context of BL for GSE. The variants obtained an average value of 80.19 in recall and 75.45 in precision, with V3_SimuHaFF being the variant that obtained the best results (91.64% in recall and 84.84% in precision).

2.6.2 Research Question 2

The p – *Values* obtained in the test are lower than 2.2×10^{-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.

Table 3.4 shows the p – *Values* of the Holm’s post hoc analysis for each pair-wise comparison and performance indicator. All p – *Values* obtained in precision and F-measure were smaller than their corresponding significance threshold value (0.05), indicating that the differences in performance between the three variants and the baselines are significant, except in recall when comparing *V1_SimuHaFF* vs. *Base2_EMoSim*, *V2_SimuHaFF* vs. *Base2_EMoSim* and *V1_SimuHaFF* vs. *V2_SimuHaFF*. Although these comparisons do not indicate significant differences in recall (meaning that the number of elements of the retrieved model fragment is similar), it is important to highlight that these comparisons indicate that there are significant differences in precision. This means that a higher number of model elements were correct according to the ground truth, which is relevant for the solution quality of the model fragment as shown in the corresponding F-measure value (which indicates significant differences).

Table 2.5: Holm’s post hoc p – *Values* for each pair-wise comparison

	Recall	Precision	F-measure
V1 vs Base1	3×10^{-10}	0.00051	3×10^{-10}
V1 vs Base2	0.063	1.1×10^{-6}	2×10^{-8}
V2 vs Base1	3×10^{-10}	3×10^{-10}	3×10^{-10}
V2 vs Base2	0.24	4.8×10^{-8}	8.1×10^{-9}
V3 vs Base1	3×10^{-10}	3×10^{-10}	3×10^{-10}
V3 vs Base2	8.1×10^{-9}	1.5×10^{-9}	3×10^{-10}
V1 vs V2	0.21	2.4×10^{-6}	2.2×10^{-5}
V1 vs V3	3×10^{-10}	3×10^{-9}	3×10^{-10}
V2 vs V3	3×10^{-10}	5.4×10^{-5}	3×10^{-10}

RQ₂ answer. Since the Holm’s post hoc p –*Values* for F-measure that are shown in Table 3.4 are smaller than 0.05, we can state that there are significant differences between the variants and the baselines.

2.6.3 Research Question 3

Table 3.5 shows the values of the effect size statistics of the pair-wise comparisons. Specifically, the table shows Cliff’s Delta [22] values for recall, precision, and F-measure. From the results, we can determine how much the quality of the solution is influenced by using the variants of our approach compared to the baselines (HaFF and EMoSim) as well as the influence among the variants. The magnitude of improvement using any of the variants of our approach instead of the baselines can be interpreted as being large according to the magnitude scales [59] of the Cliff’s Delta values, except for recall when comparing *V1_SimuHaFF vs. Base2_EMoSim*, *V2_SimuHaFF vs. Base2_EMoSim* and *V1_SimuHaFF vs. V2_SimuHaFF*.

Table 2.6: Effect size measures for comparing each pair of algorithms in Kromaia.

	Cliff’s Delta		
	Recall	Precision	F-measure
V1 vs Base1	1 (large)	0.4482759 (medium)	0.9857313 (large)
V1 vs Base2	0.1557669 (small)	0.6575505 (large)	0.7526754 (large)
V2 vs Base1	1 (large)	1 (large)	1 (large)
V2 vs Base2	0.1034483 (negligible)	0.667063 (large)	0.8049941 (large)
V3 vs Base1	1 (large)	1 (large)	1 (large)
V3 vs Base2	0.7812128 (large)	0.8454221 (large)	0.9833532 (large)
V1 vs V2	0.2342449 (small)	-0.6147444 (large)	-0.529132 (large)
V1 vs V3	-1 (large)	-0.8287753 (large)	-0.9928656 (large)
V2 vs V3	-1 (large)	-0.5386445 (large)	-0.9548157 (large)

RQ₃ answer. We can draw conclusions about how much the quality of the solution is influenced each variant of the hybrid fitness compared to each baseline from the results of Table 3.5. The results reveal that the magnitude of improvement in F-measure using any variant is large compared to the baselines according to the magnitude scales [59] of the Cliff’s Delta values.

2.7 Discussion

This section presents: 1) the analysis of the results to understand why the quality of the solutions is influenced by the SimuHaFF variants; 2) the results of a focus group where we acquired feedback from the participants to determine whether the hybrid fitness function is accepted; and 3) the results of a survey where we obtained the opinion of video game developers on which SimuHaFF variant would obtain the best results (in terms of solution quality). Thus, we compare whether the SimuHaFF variant selected by the developers coincides with the SimuHaFF variant that obtains the best results.

We examined the results in order to understand why the human effort in the variants of SimuHaFF significantly influenced the quality of the solutions, especially using V3 (simulations of the algorithm first and then human evaluations). We detected that V3 provided the best candidate solutions that the algorithm locates as input to the human. Thus, the quality of the candidate solutions were even better (in all of the performance indicators) after the human evaluations than the solutions only produced by the algorithm.

In contrast, V2 (which reverses the V3 variant) did not obtain results as good as V3 because the human started the evaluations with candidate solutions that were randomly generated (in contrast to V3 where the human started with the best solutions obtained by the algorithm). Even though the quality of the initial candidate solutions improved after 10 iterations of human evaluations, it was not enough to obtain the best results due to the following: the genetic operations randomly produce new candidate solutions, which due to the randomness make solutions worse in some cases and the execution of the algorithm after the human evaluations produces solutions that are not the ones expected by the human (e.g., including incorrect model elements).

With regard to the results of V1 (interleaving simulations automatically produced by the algorithm with human evaluations), a similar effect occurred as in V2. Even though the algorithm provided the best solution possible and a single human evaluation influenced the selection that the algorithm made to produce new candidate solutions, the results that were

displayed to the human in the next evaluation were not related to the ones that were previously scored with the highest fitness due to the algorithm. This ended up misleading the human and affecting their findings in every evaluation.

We ran a focus group to acquire feedback from the 29 professional video game developers of the industrial partner who participated as fitness function. The focus group was composed of the following open questions: (i) What do you think of the results of the approaches?; (ii) How do you feel about evaluating candidate solutions from simulation traces to locate bugs in video games?; (iii) How do you imagine the use of SimuHaFF in video games of other genres and in more complex video games?

The developers thought that the quality of the solution was better when they participated as fitness function than when they did not. They mentioned that doing the evaluations was intuitive and clear, and they enjoyed participating and using our approach even if it was to detect something that they would not have noticed without the approach. This is especially true in cases where the bugs comprise elements or properties of the models that are highly implicit or at a deep semantic level, since they are extremely complex to locate by a human alone.

The developers stated that the representation of the model fragments on the graphical interface was adequate. The heat map helped them to guide the bug localization. In addition, a potential improvement would be if the model were directly represented graphically in a model viewer that actually showed how it would look in-game, with the added highlighting of the parts that could be the source of a bug. Also, in their opinion, it would be important to always maintain a correlation with the textual model to be able to keep changes afterwards, or better yet, to be interactive to correct the model at run-time.

The developers also mentioned that SimuHaFF could be used for other video game genres and more complex problems if the video game is large (in terms of active assets) and the unit tests are complex or lack sufficient value or usefulness and therefore are not sufficiently representative. This is especially true for cases when a video game is an open game experience or the possibility of combinations between actions and results that the

player performs are almost infinite. Some developers also stressed that regardless of whether or not you have the capacity (budget and effort required), it is better to have this type of assistance to find bugs than not to have it at all. In their opinion, even if the feedback from the approach is not entirely accurate, it can alert a developer to whether something may trigger unexpected behavior. The ideal for them and how they envision our approach in a real working environment is in the form of an integration with the model editor that simulates and tests models as they are built in an assisted manner. They mentioned that it would be worthwhile to have this approach even if the hit rate was low (e.g., it found 3 out of 50 bugs that they would not have to search for). This would be especially helpful for small studios or indie developers who have practically no time to test their games since they invest most of their resources in generating new content. Therefore, it would save them time and allow them to concentrate their efforts and focus on what is really important: creating their game and bringing it to market in a timely manner.

Furthermore, we emailed a survey to 20 video game developers to get their opinion on which SimuHaFF variant they thought would achieve the best results (in terms of solution quality). The survey included junior and senior video game developers (4 and 3, respectively) as well as academic experts (13), who are video game developers and also teach in the Design and development of video games degree program of San Jorge University. The survey featured a single-choice question, where the participants were asked to select the most promising variant among the SimuHaFF options. The question itself was accompanied by Fig. 3.5, which was supplemented with a descriptive caption explaining the content of the figure and each variant. Fourteen of the participants opted for interleaving simulations that are automatically produced by the evolutionary algorithm with human expert evaluations (V1), while four others preferred launching simulations that are generated by the algorithm first and then they themselves evaluate the results (V3). Only two thought that the best option for distributing human effort during bug localization would be to rely on several consecutive evaluations of the human expert before launching the simulations that are automatically obtained by the algorithm (V2). However, the results show that V3 performs best,

followed by V2, V1, then B1_HaFF, and finally B2_EMoSim. Hence, the results are counter-intuitive for the video game developers who responded to the survey. Apparently to them, it made more sense for V1 to outperform V3. This is possibly influenced by how they envision a tool-assisted workflow.

2.8 Threats to validity

To acknowledge the threats to the validity of our work, we use the classification suggested by De Oliveira et al. [52].

1) Conclusion Validity threats. We approached the *random variation* threat by considering 30 independent runs only in Baseline 2 (without human effort) for every single bug as suggested in [8]. However, that number of runs could not be contemplated given that the availability of humans is a limited resource, and the result of subsequent runs for the same bug would be influenced by the *learning effect*. For the *lack of a good descriptive analysis* threat, we applied the recall, precision, and F-measure performance metrics to analyze the confusion matrix obtained (although other metrics could be applied). We also applied statistical significance (the Quade test and Holm's post-hoc analysis) and effect size measurements (Cliff's Delta) following accepted guidelines [9]. We tackled the *lack of meaningful comparison baseline* issue by comparing the results obtained from our approach with two baselines: EMoSim and HaFF.

2) Internal Validity threats. To mitigate the threat of *poor parameter settings*, we used values from the SBSE literature. Default ones are adequate to measure performance of localization techniques, as indicated by Arcuri and Fraser [9]. We also used two main metrics (health and victory) to assess the fitness of a simulation as done in [14]. We handled the *lack of real problem instances* by selecting a commercial video game as the case study for the evaluation. Likewise, the problem artifacts were directly obtained from the video game developers and the documentation itself. Then, we randomly selected 29 bugs from the entire documentation. Afterwards, we also randomly assigned four different bugs to each developer in the evaluation. We did not participate in the

selection of the developers to avoid *researcher bias* [27]. In addition, both the order of the variants of the approach and the baseline (which only uses the human as fitness function) and the set of bugs to be localized were randomly assigned to each engineer so that they would not know whether they were using a variant or baseline to locate a certain bug, hence mitigating the *imbalanced group of subjects* threat. We conducted the evaluation following a crossover design: the developers who participated in the oracle provided by the industrial partner were not involved in the evaluation, thus addressing the *learning effect* threat. We gave a briefing before starting the evaluation so that possible *understandability* issues could be minimized. The developers were not allowed to communicate with each other during the evaluation to avoid incurring in the *information exchange* threat. The number of iterations and individuals of the algorithm per bug was set to 10 as recommended in the literature [67] in order to reduce *fatigue impact*. Besides that, we established a total duration of the evaluation of two hours as a whole (including the briefing), in accordance with the average duration of experiments in software engineering [65].

3) Construct Validity threats. To prevent the *lack of assessing the validity of cost measures* threat, we made a fair comparison between the variants of our approach and the two baselines. Furthermore, we used three measurements (recall, precision, and F-measure) for the evaluation, which have been widely adopted by the software engineering research community [60].

4) External Validity threats. We evaluated our approach in a commercial video game, whose instances are collected from real-world problems, to avoid the *lack of a clear object selection strategy* threat. To mitigate the *generalization* threat, we designed our approach to be generic and applicable not only to our industrial case study but also for locating bugs in other different video games. To apply our approach in other video games, three main ingredients are required as in other SBSE approaches: encoding, operations, and fitness function. The operations are widespread crossover and mutation. 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 can be found in well-known gaming genres that include racing, first-person shooter, and real-time strategy. 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 using our approach. Nevertheless, our approach should be replicated with other video games before assuring its generalization.

2.9 Conclusion

Recently, human participation in Search-Based Software Engineering (SBSE) has proven to be useful for obtaining more favorable solutions. However, in order to efficiently explore large complex problem spaces, humans cannot handle the same number of solutions as an algorithm since humans are not immune to fatigue.

In this work, we have proposed a hybrid fitness function that is the first to combine human effort with human simulations for more demanding problems in the context of BL in software models for the video games of GSE. Moreover, we have proposed three variants of the hybrid fitness function that distribute the human effort in different ways in order to study their influence on the quality of the solutions. The evaluation was performed on a commercial video game (Kromaia) where 29 professional video game developers were involved, acting as the human component of the fitness function. As baselines, we replaced the hybrid fitness function with a fitness function that only uses a human and a fitness function that is automatically calculated.

Our proposed hybrid fitness function outperformed the results of the best baseline by 33.46% in F-measure. Furthermore, a focus group confirmed the professional video game developers' acceptance of the hybrid fitness function since it helped them to reduce the amount of manual work and to minimize the number of bugs that go unnoticed. The variant that obtained the best results was not only counter-intuitive with an initial survey that we did with video game developers, but it was also counter-intuitive with previous works.

It is important to highlight that our work has implications from both an academic perspective in software engineering and from a practical standpoint for professional video game developers. From the academic perspective in software engineering, our work shows that the distribution of the human effort (interleaving the algorithm with human evaluations), which is both preferred by video game developers and selected by default in most iSBSE works, does not obtain the best results. This can inspire other academics to explore other distributions of the human effort (such as the distributions of the human effort that are studied in this work) to improve their results. From the practical standpoint of professional video game developers, there is a lack of technical solutions that locate bugs and real-world experience. This is hard to obtain since the majority of related works use academic data and other video game studios do not share the details of their technical solutions to develop and maintain their commercial products. Works like ours may be the path to compensate for this lack, and may motivate other video game developers to reduce the amount of tedious manual work during the location of bugs.

Our results not only help video game developers to locate bugs, but it can also inspire other SBSE researchers to bring hybrid fitness functions to other software engineering tasks instead of using a default approach for distributing the human effort. In fact, part of our future work is to explore hybrid fitness functions in other software engineering tasks and domains.

Acknowledgements

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 S05_20D).

Bibliography

- [1] Vahid Alizadeh et al. “An Interactive and Dynamic Search-Based Approach to Software Refactoring Recommendations”. In: *IEEE Trans. Software Eng.* 46.9 (2020), pp. 932–961. DOI: 10.1109/TSE.2018.2872711 (cit. on pp. 61, 70, 72, 73).
- [2] Bader Alkhazi et al. “Multi-criteria test cases selection for model transformations”. In: *Automated Software Engineering* (2020), pp. 1–28 (cit. on pp. 61, 70, 72).
- [3] Bader Alkhazi et al. “On the value of quality attributes for refactoring ATL model transformations: A multi-objective approach”. In: *Information and Software Technology* 120 (2020), p. 106243. ISSN: 0950-5849. DOI: <https://doi.org/10.1016/j.infsof.2019.106243> (cit. on pp. 61, 70, 72).
- [4] Boukhdhir Amal et al. “On the Use of Machine Learning and Search-Based Software Engineering for Ill-Defined Fitness Function: A Case Study on Software Refactoring”. In: *Search-Based Software Engineering*. Ed. by Claire Le Goues and Shin Yoo. Cham: Springer International Publishing, 2014, pp. 31–45. ISBN: 978-3-319-09940-8 (cit. on pp. 61, 70, 72).
- [5] Apostolos Ampatzoglou and Ioannis Stamelos. “Software engineering research for computer games: A systematic review”. In: *Inf. Softw. Technol.* 52.9 (2010), pp. 888–901. DOI: 10.1016/j.infsof.2010.05.004 (cit. on p. 62).
- [6] Allysson Alex Araújo et al. “An Architecture Based on Interactive Optimization and Machine Learning Applied to the next Release Problem”. In: *Automated Software Engg.* 24.3 (Sept. 2017), 623–671. ISSN: 0928-8910. DOI: 10.1007/s10515-016-0200-3 (cit. on pp. 61, 70, 72, 73).
- [7] Lorena Arcega et al. “Bug Localization in Model-Based Systems in the Wild”. In: *ACM Trans. Softw. Eng. Methodol.* 31.1 (2021), 10:1–10:32. ISSN: 1049-331X. DOI: 10.1145/3472616 (cit. on pp. 71, 72).
- [8] Andrea Arcuri and Lionel Briand. “A Hitchhiker’s guide to statistical tests for assessing randomized algorithms in software engineering”. In: *Software*

Testing, Verification and Reliability 24.3 (May 2014), pp. 219–250. ISSN: 0960-0833. DOI: 10.1002/stvr.1486 (cit. on pp. 63, 80, 81, 91).

- [9] Andrea Arcuri and Gordon Fraser. “Parameter tuning or default values? An empirical investigation in search-based software engineering”. In: *Empirical Software Engineering* 18.3 (2013), pp. 594–623. ISSN: 1573-7616. DOI: 10.1007/s10664-013-9249-9 (cit. on pp. 63, 91).
- [10] Sinan Ariyurek, Aysu Betin-Can, and Elif Surer. “Automated Video Game Testing Using Synthetic and Humanlike Agents”. In: *IEEE Transactions on Games* 13.1 (2021), pp. 50–67. DOI: 10.1109/TG.2019.2947597 (cit. on p. 72).
- [11] Sinan Ariyurek, Aysu Betin-Can, and Elif Sürer. “Enhancing the Monte Carlo Tree Search Algorithm for Video Game Testing”. In: *IEEE Conference on Games, CoG 2020, Osaka, Japan, August 24-27, 2020*. IEEE, 2020, pp. 25–32. DOI: 10.1109/CoG47356.2020.9231670 (cit. on p. 72).
- [12] Edouard Batot and Houari Sahraoui. “A Generic Framework for Model-Set Selection for the Unification of Testing and Learning MDE Tasks”. In: *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. MODELS ’16. Saint-malo, France: Association for Computing Machinery, 2016, 374–384. ISBN: 9781450343213. DOI: 10.1145/2976767.2976785 (cit. on pp. 61, 70, 72).
- [13] Carlos Vinicius Bindewald et al. “Towards the Support of User Preferences in Search-Based Product Line Architecture Design: An Exploratory Study”. In: *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*. SBES 2019. Salvador, Brazil: Association for Computing Machinery, 2019, 387–396. ISBN: 9781450376518. DOI: 10.1145/3350768.3351993 (cit. on pp. 61, 70, 72, 73).
- [14] Daniel Blasco et al. “An evolutionary approach for generating software models: The case of Kromaia in Game Software Engineering”. In: *J. Syst. Softw.* 171 (2021), p. 110804. DOI: 10.1016/j.jss.2020.110804 (cit. on pp. 65, 91).
- [15] Ilhem Boussaïd, Patrick Siarry, and Mohamed Ahmed-Nacer. “A survey on search-based model-driven engineering”. In: *Automated Software Engineer-*

- ing 24.2 (2017), pp. 233–294. ISSN: 1573-7535. DOI: 10.1007/s10515-017-0215-4 (cit. on p. 71).
- [16] Loli Burgueño et al. “Static Fault Localization in Model Transformations”. In: *IEEE Transactions on Software Engineering* 41.5 (2015), pp. 490–506. ISSN: 0098-5589. DOI: 10.1109/TSE.2014.2375201 (cit. on p. 72).
- [17] R. Calinescu et al. “Designing Robust Software Systems through Parametric Markov Chain Synthesis”. In: *2017 IEEE International Conference on Software Architecture (ICSA)*. 2017, pp. 131–140 (cit. on pp. 61, 70, 72).
- [18] Théo Le Calvar et al. “Toward a declarative language to generate explorable sets of models”. In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC 2019, Limassol, Cyprus, April 8-12, 2019*. Ed. by Chih-Cheng Hung and George A. Papadopoulos. ACM, 2019, pp. 1837–1844. DOI: 10.1145/3297280.3297461 (cit. on pp. 61, 70, 72).
- [19] Rodrigo Casamayor et al. “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*. Ed. by Eugene Syriani et al. ACM, 2022, pp. 356–366. DOI: 10.1145/3550355.3552440 (cit. on pp. 63, 65, 66, 68, 71, 72, 82).
- [20] Xiaoyun Cheng et al. “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. DOI: 10.1109/COMPSAC48688.2020.0-152 (cit. on p. 72).
- [21] Agnieszka Ciborowska and Kostadin 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. DOI: 10.1145/3510003.3510042 (cit. on p. 72).
- [22] Norman Cliff. *Ordinal methods for behavioral data analysis*. Lawrence Erlbaum Associates, Inc, 1996, p. 212. DOI: <https://doi.org/10.4324/9781315806730> (cit. on pp. 81, 87).

- [23] William Jay Conover. *Practical Nonparametric Statistics, 3rd Edition*. 3. ed. Wiley series in probability and statistics. USA: Wiley, 1999. VIII, 584. ISBN: 0471160687 (cit. on p. 80).
- [24] Csaba Debreceni et al. “Automated Model Merge by Design Space Exploration”. In: *Fundamental Approaches to Software Engineering*. Ed. by Perdita Stevens and Andrzej Wasowski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 104–121 (cit. on pp. 61, 70, 72).
- [25] Pedro Delgado-Pérez et al. “InterEvo-TR: Interactive Evolutionary Test Generation with Readability Assessment”. In: *IEEE Transactions on Software Engineering* (2022), pp. 1–17. DOI: 10.1109/TSE.2022.3227418 (cit. on pp. 61, 70, 72).
- [26] Robert Feldt. *An Interactive Software Development Workbench based on Biomimetic Algorithms*. Tech. rep. Tech. report 02-16. Dept. of Computer Engineering, Chalmers University of Technology, Gothenburg, 2002 (cit. on pp. 61, 68, 69).
- [27] Robert Feldt and Ana Magazinius. “Validity Threats in Empirical Software Engineering Research-An Initial Survey.” In: *Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering (SEKE’2010)*. 2010, pp. 374–379 (cit. on p. 92).
- [28] Raihana Ferdous et al. “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*. Ed. by Una-May O’Reilly and Xavier Devroey. Vol. 12914. Lecture Notes in Computer Science. Springer, 2021, pp. 56–71. DOI: 10.1007/978-3-030-88106-1_5 (cit. on p. 72).
- [29] Martin Fleck, Javier Troya, and Manuel Wimmer. “Search-Based Model Transformations”. In: *J. Softw. Evol. Process* 28.12 (Dec. 2016), 1081–1117. ISSN: 2047-7473. DOI: 10.1002/smr.1804 (cit. on pp. 61, 70, 72).
- [30] J. Font et al. “Achieving Feature Location in Families of Models through the use of Search-Based Software Engineering”. In: *IEEE Transactions on Evolutionary Computation* 22.3 (2018), pp. 363–377. ISSN: 1089-778X. DOI: 10.1109/TEVC.2017.2751100 (cit. on p. 61).

- [31] Willian Marques Freire et al. “Validating an Interactive Ranking Operator for NSGA-II to Support the Optimization of Software Engineering Problems”. In: *SBES 2022: XXXVI Brazilian Symposium on Software Engineering, Virtual Event Brazil, October 5 - 7, 2022*. Ed. by Marcelo de Almeida Maia et al. ACM, 2022, pp. 337–346. DOI: 10.1145/3555228.3555232 (cit. on pp. 61, 70, 72, 73).
- [32] Salvador García et al. “Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: Experimental analysis of power”. In: *Information Sciences* 180.10 (May 2010). Special Issue on Intelligent Distributed Information Systems, pp. 2044–2064. ISSN: 0020-0255. DOI: <http://dx.doi.org/10.1016/j.ins.2009.12.010> (cit. on p. 80).
- [33] Adnane Ghannem, Ghizlane El Boussaidi, and Marouane Kessentini. “Model Refactoring Using Interactive Genetic Algorithm”. In: *Search Based Software Engineering*. Ed. by Günther Ruhe and Yuanyuan Zhang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 96–110 (cit. on pp. 61, 70, 72, 73).
- [34] Pablo Gómez-Abajo, Esther Guerra, and Juan de Lara. “A domain-specific language for model mutation and its application to the automated generation of exercises”. In: *Computer Languages, Systems & Structures* 49 (2017), pp. 152–173. ISSN: 1477-8424. DOI: <https://doi.org/10.1016/j.cl.2016.11.001> (cit. on pp. 61, 70, 72).
- [35] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. “Search-based software engineering: Trends, techniques and applications”. In: *ACM Comput. Surv.* 45.1 (2012), 11:1–11:61. DOI: 10.1145/2379776.2379787 (cit. on p. 61).
- [36] Sidra Iftikhar et al. “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*. Ed. by Timothy Lethbridge, Jordi Cabot, and Alexander Egyed. IEEE Computer Society, 2015, pp. 426–435. DOI: 10.1109/MODELS.2015.7338274 (cit. on p. 72).

- [37] Hisao Ishibuchi, Yusuke 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. DOI: 10.1109/CEC.2006.1688438 (cit. on p. 79).
- [38] Helson Luiz Jakubovski Filho, Thiago Nascimento Ferreira, and Silvia Regina Vergilio. “Preference based multi-objective algorithms applied to the variability testing of software product lines”. In: *Journal of Systems and Software* 151 (2019), pp. 194–209. ISSN: 0164-1212. DOI: <https://doi.org/10.1016/j.jss.2019.02.028> (cit. on pp. 61, 70, 72).
- [39] Wael Kessentini and Vahid Alizadeh. “Interactive metamodel/model co-evolution using unsupervised learning and multi-objective search”. In: *MoDELS ’20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020*. Ed. by Eugene Syriani et al. ACM, 2020, pp. 68–78. DOI: 10.1145/3365438.3410966 (cit. on pp. 61, 70, 72, 73).
- [40] Wael Kessentini and Vahid Alizadeh. “Semi-automated metamodel/model co-evolution: a multi-level interactive approach”. In: *Softw. Syst. Model.* 21.5 (2022), pp. 1853–1876. DOI: 10.1007/s10270-022-00978-2 (cit. on pp. 61, 64, 70, 72, 73).
- [41] Wael Kessentini, Manuel Wimmer, and Houari Sahraoui. “Integrating the Designer In-the-Loop for Metamodel/Model Co-Evolution via Interactive Computational Search”. In: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. MODELS ’18*. Copenhagen, Denmark: Association for Computing Machinery, 2018, 101–111. ISBN: 9781450349499. DOI: 10.1145/3239372.3239375 (cit. on pp. 61, 70, 72).
- [42] Ahmed 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*. Ed. by Abhik Roychoudhury, Cristian Cadar, and Miryung Kim. ACM, 2022, pp. 1790–1794. DOI: 10.1145/3540250.3558907 (cit. on p. 72).

- [43] AV Kolchin. “Interactive method for cumulative analysis of software formal models behavior”. In: *PROBLEMS IN PROGRAMMING* 2-3 (2018), pp. 115–123 (cit. on pp. 61, 70, 72).
- [44] Fernando H. Kuviatkovski et al. “Evaluating Machine Learning Algorithms in Representing Decision Makers in search-based PLA”. In: *IEEE 19th International Conference on Software Architecture Companion, ICSA Companion 2022, Honolulu, HI, USA, March 12-15, 2022*. IEEE, 2022, pp. 68–75. DOI: 10.1109/ICSA-C54293.2022.00057 (cit. on pp. 61, 70, 72).
- [45] Hongliang Liang, Dengji Hang, and Xiangyu Li. “Modeling function-level interactions for file-level bug localization”. In: *Empir. Softw. Eng.* 27.7 (2022), p. 186. DOI: 10.1007/s10664-022-10237-z (cit. on p. 72).
- [46] Yun Lin et al. “Interactive and Guided Architectural Refactoring with Search-Based Recommendation”. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. Seattle, WA, USA: Association for Computing Machinery, 2016, 535–546. ISBN: 9781450342186. DOI: 10.1145/2950290.2950317 (cit. on pp. 61, 70, 72).
- [47] H. Lu et al. “Nonconformity Resolving Recommendations for Product Line Configuration”. In: *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 2016, pp. 57–68 (cit. on pp. 61, 70, 72).
- [48] Bogdan Marculescu et al. “An initial industrial evaluation of interactive search-based testing for embedded software”. In: *Appl. Soft Comput.* 29 (2015), pp. 26–39. DOI: 10.1016/j.asoc.2014.12.025 (cit. on pp. 61, 68, 69).
- [49] Bogdan Marculescu et al. “Transferring interactive search-based software testing to industry”. In: *Journal of Systems and Software* 142 (2018), pp. 156–170. ISSN: 0164-1212. DOI: <https://doi.org/10.1016/j.jss.2018.04.061> (cit. on pp. 61, 70, 72, 73).
- [50] Jabier Martinez et al. “Towards Estimating and Predicting User Perception on Software Product Variants”. In: *New Opportunities for Software Reuse*. Ed. by Rafael Capilla, Barbara Gallina, and Carlos Cetina. Cham: Springer

- International Publishing, 2018, pp. 23–40. ISBN: 978-3-319-90421-4 (cit. on pp. 61, 70, 72).
- [51] Jabier Martinez et al. “Variability Management and Assessment for User Interface Design”. In: *Human Centered Software Product Lines*. Ed. by Jean-Sébastien Sottet, Alfonso García Frey, and Jean Vanderdonckt. Cham: Springer International Publishing, 2017, pp. 81–106. ISBN: 978-3-319-60947-8. DOI: 10.1007/978-3-319-60947-8_3 (cit. on pp. 61, 70, 72, 73).
- [52] Márcio de Oliveira Barros and Arilo Claudio Dias Neto. *Threats to Validity in Search-based Software Engineering Empirical Studies*. Tech. rep. 0006/2011. Jan. 2011 (cit. on p. 91).
- [53] F. Pérez, T. Ziadi, and C. Cetina. “Utilizing Automatic Query Reformulations as Genetic Operations to Improve Feature Location in Software Models”. In: *IEEE Transactions on Software Engineering* 2 (2020), pp. 713–731. ISSN: 1939-3520. DOI: 10.1109/TSE.2020.3000520 (cit. on p. 76).
- [54] Francisca Pérez et al. “Empowering the Human as the Fitness Function in Search-Based Model-Driven Engineering”. In: *IEEE Transactions on Software Engineering* 01 (2021), pp. 1–1. ISSN: 1939-3520. DOI: 10.1109/TSE.2021.3121253 (cit. on pp. 61–63, 68–70, 72, 73, 82).
- [55] S. Procter and L. Wrage. “Guided Architecture Trade Space Exploration: Fusing Model Based Engineering Design by Shopping”. In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2019, pp. 117–127 (cit. on pp. 61, 70, 72).
- [56] Sophia Quach et al. “Evaluating the impact of falsely detected performance bug-inducing changes in JIT models”. In: *Empir. Softw. Eng.* 26.5 (2021), p. 97. DOI: 10.1007/s10664-021-10004-6 (cit. on p. 72).
- [57] A. Ramírez, J. R. Romero, and C. L. Simons. “A Systematic Review of Interaction in Search-Based Software Engineering”. In: *IEEE Transactions on Software Engineering* 45.8 (Jan. 1, 2019), pp. 760–781. ISSN: 0098-5589. DOI: 10.1109/TSE.2018.2803055. published (cit. on pp. 61, 68, 69, 71).
- [58] Aurora Ramírez et al. “Interactivity in the Generation of Test Cases with Evolutionary Computation”. In: *IEEE Congress on Evolutionary Compu-*

- tation, *CEC 2021, Kraków, Poland, June 28 - July 1, 2021*. IEEE, 2021, pp. 2395–2402. DOI: 10.1109/CEC45853.2021.9504786 (cit. on pp. 61, 68, 70, 72).
- [59] Jeanine Romano et al. “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. 1–33 (cit. on pp. 64, 81, 87).
- [60] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. New York, NY, USA: McGraw-Hill, Inc., 1986. ISBN: 0070544840 (cit. on pp. 79, 92).
- [61] Jesús Sánchez Cuadrado, Esther Guerra, and Juan Lara. “Quick Fixing ATL Transformations with Speculative Analysis”. In: *Softw. Syst. Model.* 17.3 (July 2018), 779–813. ISSN: 1619-1366. DOI: 10.1007/s10270-016-0541-1 (cit. on p. 72).
- [62] Jesús Sánchez Cuadrado, Esther Guerra, and Juan Lara. “Static Analysis of Model Transformations”. In: *IEEE Transactions on Software Engineering* 43.9 (2017), pp. 868–897. DOI: 10.1109/TSE.2016.2635137 (cit. on p. 72).
- [63] Howard J Seltman. “Experimental design and analysis”. In: *Online at: <http://www.stat.cmu.edu/~hseltman/309/Book/Book.pdf>* (2012) (cit. on p. 83).
- [64] D. N. A. d. Silva. “Adaptation oriented test data generation for Adaptive Systems”. In: *2020 15th Iberian Conference on Information Systems and Technologies (CISTI)*. 2020, pp. 1–7 (cit. on pp. 61, 70, 72).
- [65] D. I. K. Sjoeborg et al. “A survey of controlled experiments in software engineering”. In: *IEEE Transactions on Software Engineering* 31.9 (2005), pp. 733–753 (cit. on pp. 83, 92).
- [66] Hideyuki Takagi. “Interactive evolutionary computation: fusion of the capabilities of EC optimization and human evaluation”. In: *Proc. IEEE* 89.9 (2001), pp. 1275–1296. DOI: 10.1109/5.949485 (cit. on p. 62).

- [67] Hideyuki Takagi. “Interactive Evolutionary Computation: Fusion of the Capabilities of EC Optimization and Human Evaluation”. In: *Proceedings of the IEEE* 89.9 (2001), pp. 1275–1296. ISSN: 0018-9219 (cit. on pp. 62, 68, 69, 76, 84, 92).
- [68] Javier Troya et al. “Spectrum-Based Fault Localization in Model Transformations”. In: *ACM Trans. Softw. Eng. Methodol.* 27.3 (Sept. 2018). ISSN: 1049-331X. DOI: 10.1145/3241744 (cit. on pp. 71, 72).
- [69] Rosalia Tufano et al. “Using Reinforcement Learning for Load Testing of Video Games”. In: *Proceedings of the 44th International Conference on Software Engineering (ICSE 2022), Pittsburgh, USA*. 2022 (cit. on p. 72).
- [70] L. Van Rooijen and H. Hamann. “Requirements Specification-by-Example Using a Multi-objective Evolutionary Algorithm”. In: *2016 IEEE 24th International Requirements Engineering Conference Workshops (REW)*. 2016, pp. 3–9 (cit. on pp. 61, 70, 72).
- [71] Zan Wang et al. “Genetic algorithm based multiple faults localization technique”. In: *Ruan Jian Xue Bao/Journal of Software* 27.4 (2016), 879 – 900. DOI: 10.13328/j.cnki.jos.004970 (cit. on pp. 61, 68, 69).
- [72] T. Yue et al. “Search-based decision ordering to facilitate product line engineering of Cyber-Physical System”. In: *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. 2016, pp. 691–703 (cit. on pp. 61, 70, 72).
- [73] Jinglei Zhang et al. “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. DOI: 10.1145/3387904.3389281 (cit. on p. 72).
- [74] Yan Zheng et al. “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. DOI: 10.1109/ASE.2019.00077 (cit. on p. 72).

- [75] Jose Zubcoff et al. “Evaluating different i*-based approaches for selecting functional requirements while balancing and optimizing non-functional requirements: A controlled experiment”. In: *Information and Software Technology* 106 (2019), pp. 68–84. ISSN: 0950-5849. DOI: <https://doi.org/10.1016/j.infsof.2018.09.004> (cit. on pp. 61, 70, 72).

Chapter 3

Boosting bug localization in software models of video games with simulations and component-specific genetic operations

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.

3.1 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 [46]. 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 [21]. This might suggest that the number of video game developers will continue to grow in the future as the Metaverse is developed [53].

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 [35].

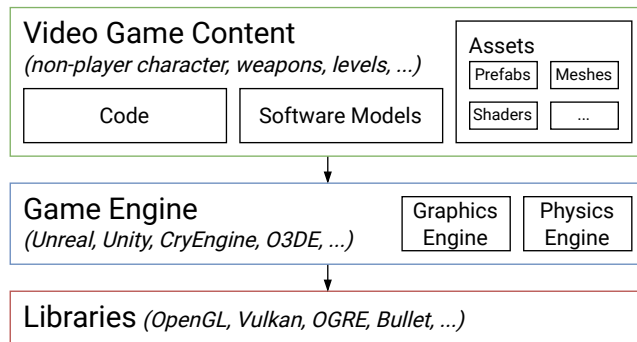


Figure 3.1: Overview of video game artifacts.

Nowadays, most video games are developed by means of so-called *game engines*. A *game engine* refers to a development environment that in-

tegrates 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 [51] and Unreal Engine [22], but it is also possible for a studio to make its own specific engine (e.g., CryEngine [19]).

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. 3.1). 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 [58] 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 [35]. Nevertheless, a fault localization survey [54] 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 [38].

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* [24]. 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 [47].

There are significant differences between Classic Software Engineering and Game Software Engineering [2, 38]. 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 [13], 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 [3, 4], which is specifically designed to locate bugs in software models and uses bug reports and the defect localization principle [27] (as many bug localization approaches do [54]). The “defect localization principle” states that the most recent modifications to a project are the most relevant for certain Information Retrieval purposes [27, 59, 45]. 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.

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 [49]. 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

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

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 [13]. In addition, we perform a statistical analysis (following the guidelines by Arcuri and Briand [5]) 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 3.2, we present the case study (Kromaia) and the types of bugs. In Section 3.3, we carry out the classification of the types of bugs in the categories of the taxonomy. In Section 3.4, we describe our approach, EMoSim. In Section 3.5, we evaluate our approach in Kromaia, and we present the results, which are discussed in Section 3.6. In Section 3.7, we review the threats to validity. In Section 3.8, we examine the related work on the topics covered in this paper. Finally, we conclude the paper in Section 3.9.

3.2 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. 3.2 shows the content for a gameplay of Kromaia. Each of the levels involves a player's spaceship (Fig. 3.2.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. 3.2.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. 3.2.G). These entities, including enemy bosses, derive their structure from rigid bodies or solid objects known as hulls (Fig. 3.2.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.

The bosses are specified with the Shooter Definition Model Language (SDML). SDML is a custom DSL model for the video game domain of

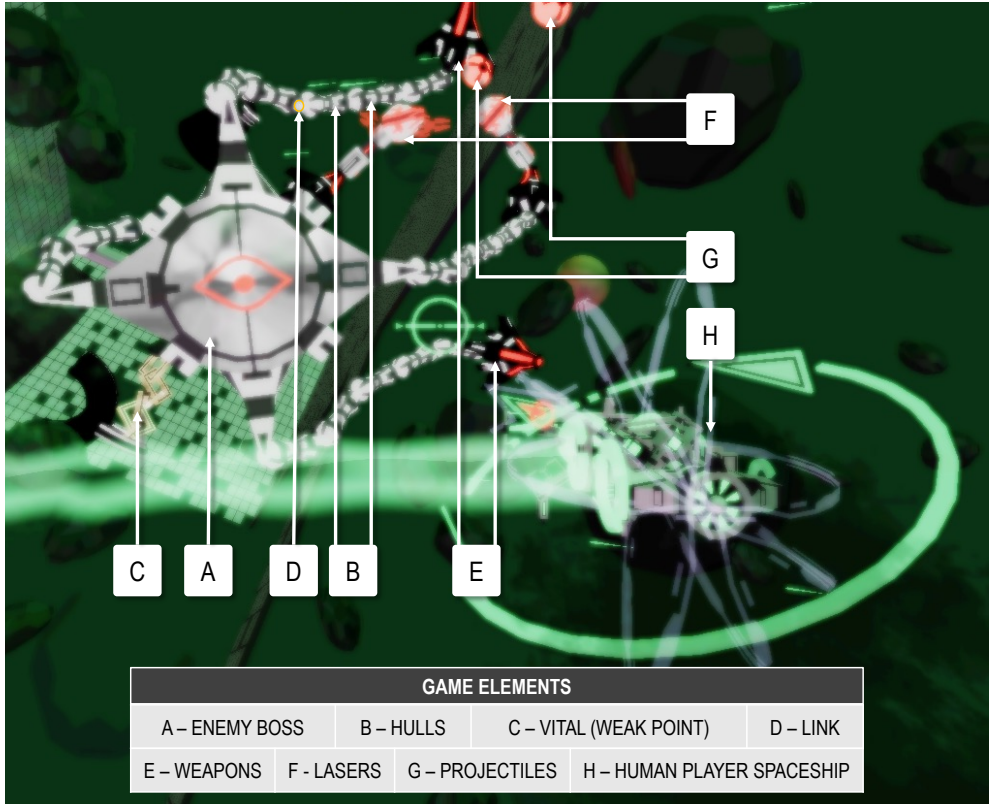


Figure 3.2: Screenshot showing the game content in Kromaia.

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. 3.3 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.

3.2.1 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 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

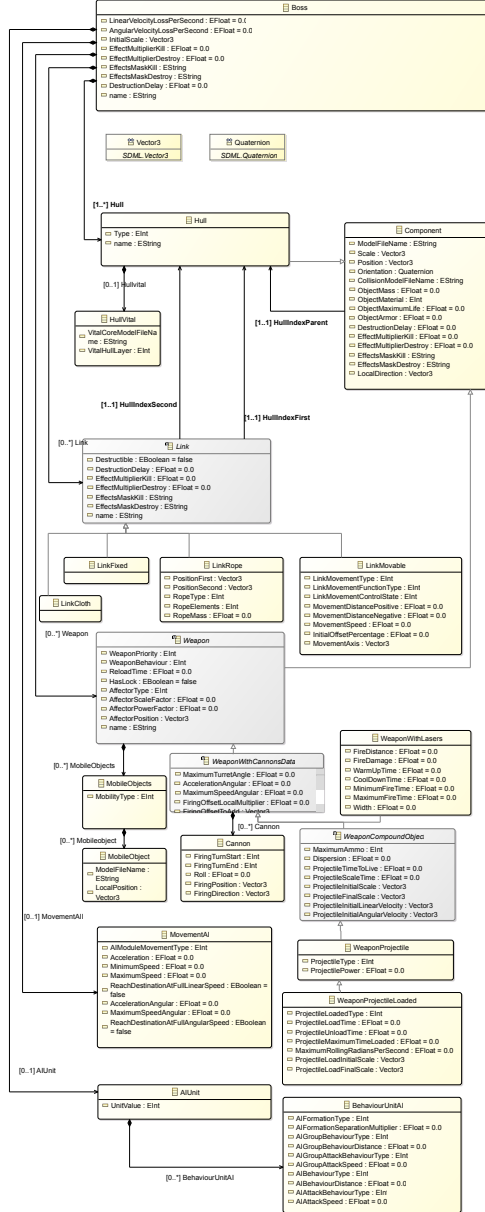


Figure 3.3: SDML metamodel of a Kromaia boss.

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.

3.2.2 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 [20]. 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.

3.3 Classifying Bug Types in Taxonomy Categories

Truelove et al. [49] 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 3.2. 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 [49], we involved multiple people by employing a methodology similar to those used in systematic literature reviews (SLRs) that involve inter-rater reliability (IRR) [31]. 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 [49], we want to emphasize that the presented mappings between our bug types and the categories from [49] 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 [49]. We aligned each bug type with the category in [49] 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* $\rightarrow$ *category of bug in the taxonomy*) is as described below.

WPH $\rightarrow$ 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”

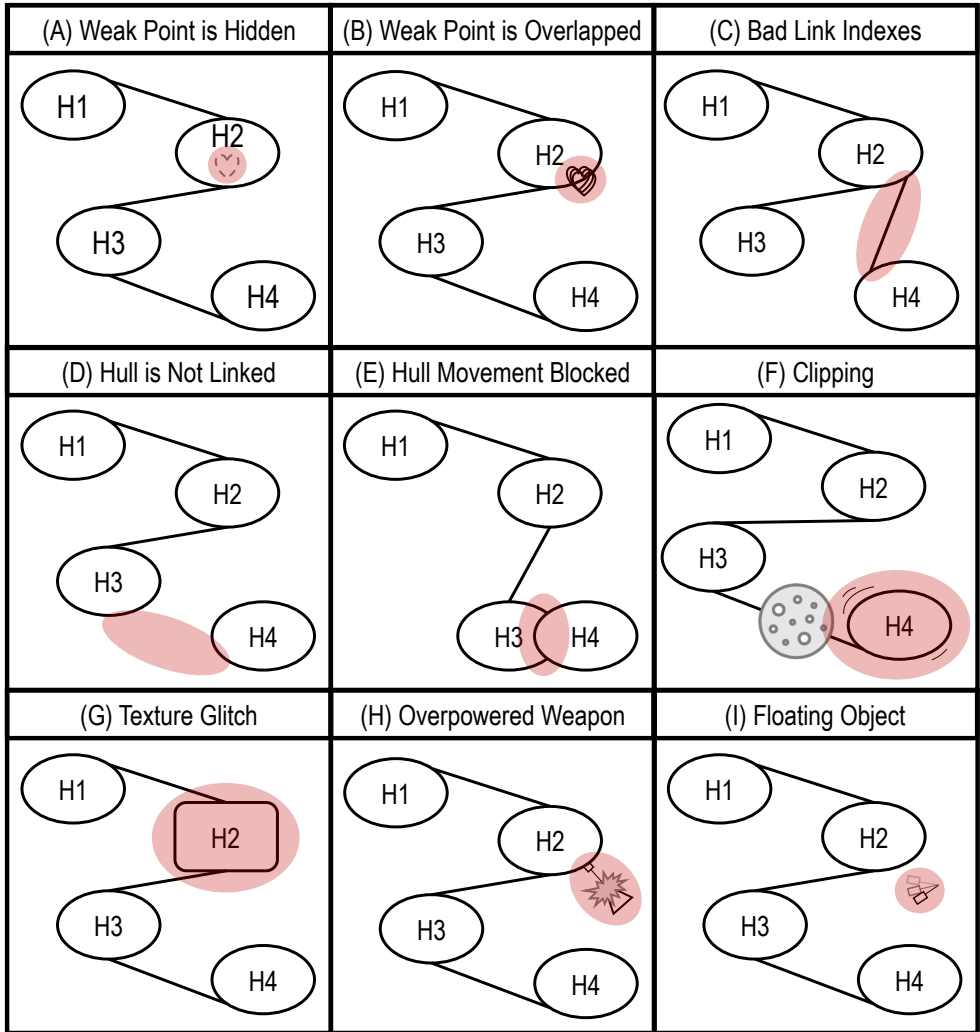


Figure 3.4: Graphical representation of the bug types.

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. 3.4.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.

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. 3.4.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. 3.4.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. 3.4.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. 3.4.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 [13], 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. 3.4.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. 3.4.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. 3.4.H shows an

Table 3.1: 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)

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. 3.4.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 3.1 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 [49] 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.

3.4 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.

3.4.1 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. 3.5. 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 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

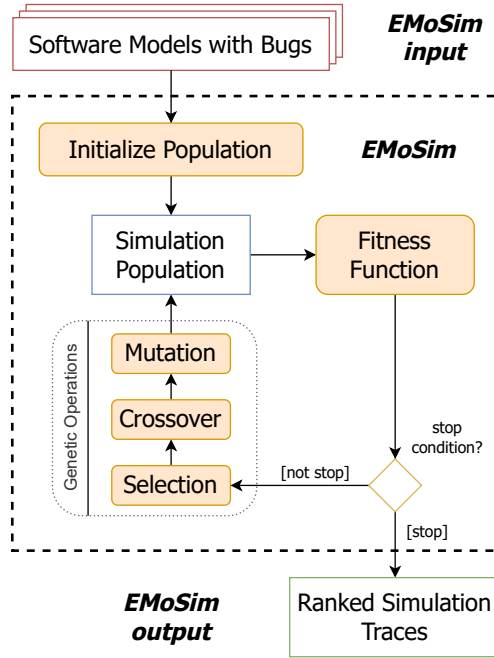


Figure 3.5: EMOsim approach.

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.

3.4.2 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 [11]. 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.

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 3.2 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.

Table 3.2: 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	$(-\infty, -1]$
hullRouteDirection	Direction of the route followed by the simulated player: 1 forward and -1 backward.	Int	1	1, -1

^aBased on the value specified in the model.

Fig. 3.6 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

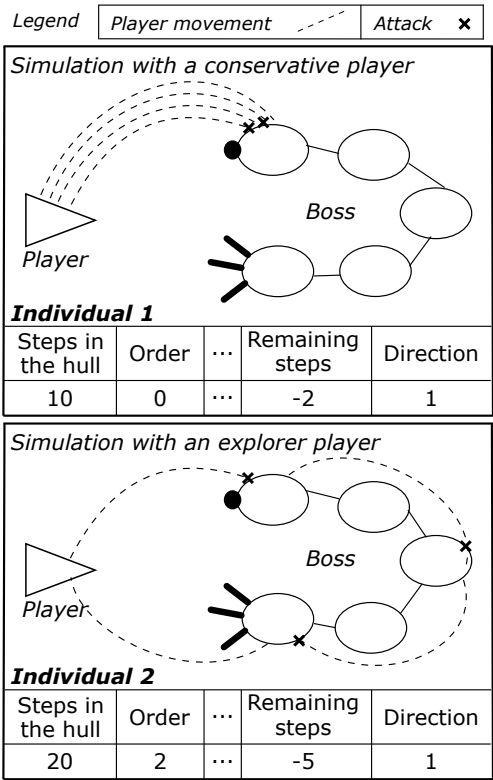


Figure 3.6: Representation of a simulation as an individual.

unit of the time spent by the simulated player on a certain element of the model.

Each example in Fig. 3.6 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.

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 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 [10]. This fitness rewards simulations that have behaved as the developers intended for their game. In [10], 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 [10]:

- *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):

$$victory = 1 - \frac{|V_O - V_P|}{V_O} \quad (3.1)$$

- *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):

$$health = 1 - \frac{\sum_{d=1}^{V_P} \frac{|H_O - H_P|}{H_O}}{V_P} \quad (3.2)$$

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.

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. 3.7 depicts an example of a model of a Kromaia boss and a trace of an individual.

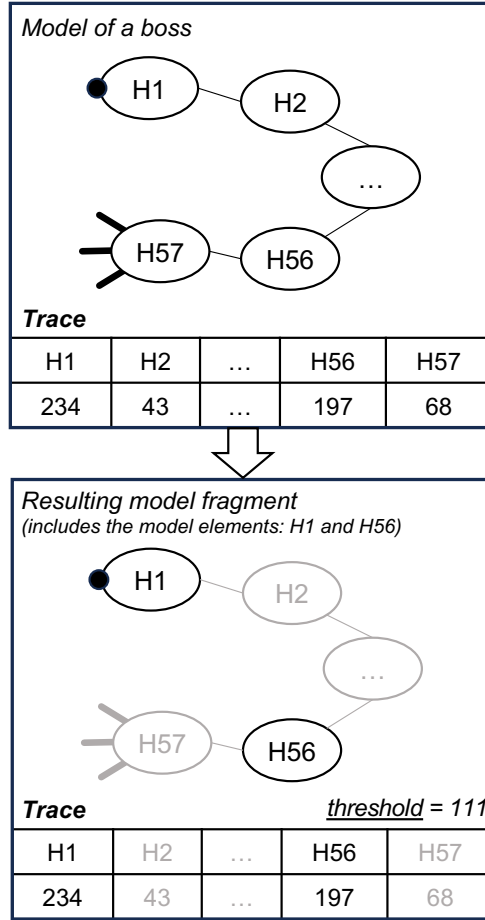


Figure 3.7: Example of a simulation trace and the model fragment mapping.

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. 3.7 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.

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.

The Classic Genetic Operations

Most of the works use the classic genetic operations (single-point crossover plus random mutation) [36] 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. 3.8-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. 3.8-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 ap-

proach 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.

The Component-Specific Genetic Operations

Most previous works choose the classic genetic operations [36] 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 [13], 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 pa-

rameters 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. 3.8 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. 3.8-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. 3.8-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. 3.8-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. 3.8). 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.

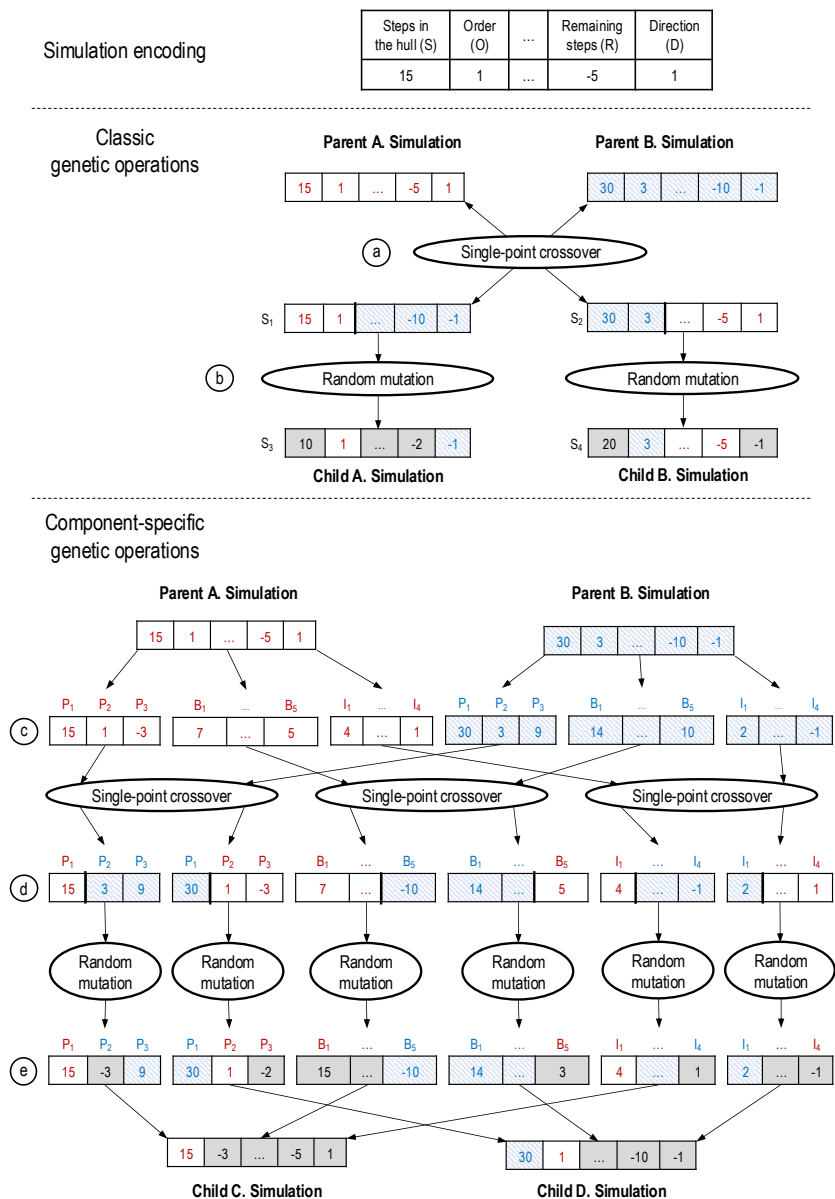


Figure 3.8: The component-specific genetic operations using simulation encoding.

3.5 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).

3.5.1 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. 3.9 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 [4]), 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 [54] 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 [13] 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.

3.5.2 Experimental Setup

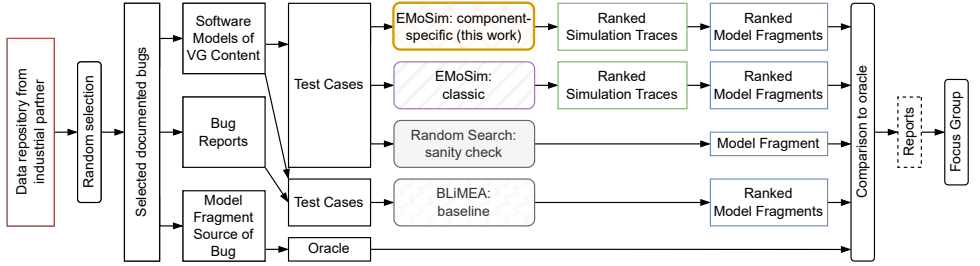


Figure 3.9: Evaluation process.

Fig. 3.9 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 3.4) with the component-specific genetic operations (as described in Section 3.4.2). 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 3.4.2). 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 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 [29]. 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. 3.10 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.

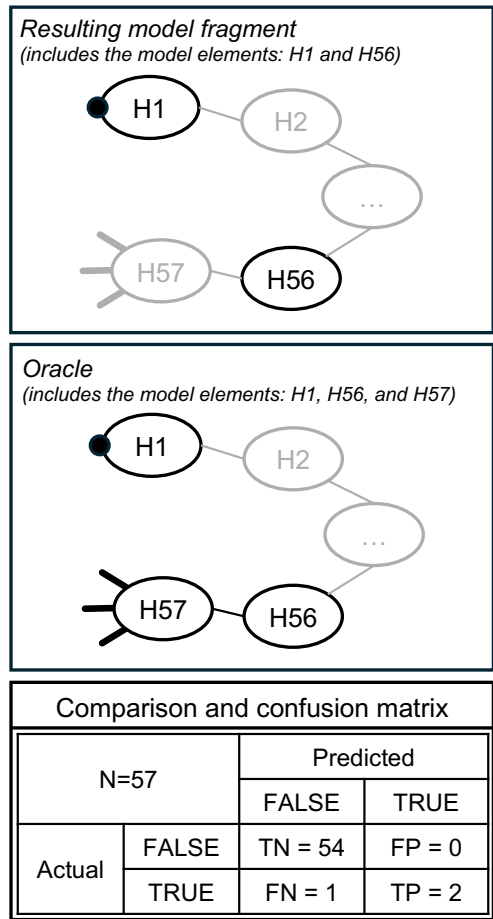


Figure 3.10: Example of the calculation of the confusion matrix.

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

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

F-measure ($2 * \frac{Precision * Recall}{Precision + Recall}$) 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.

3.5.3 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 [5]. For BLiMEA, we used the same parameters as reported in [3]. For EMoSim, we started from those reported in [10] (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>.

Table 3.3: 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 $\pm (\sigma)$	93.15 $\pm$ 16.78	96.48 $\pm$ 8.62	98.33 $\pm$ 4.08	97.64 $\pm$ 5.78	100	97.78 $\pm$ 5.44	97.55 $\pm$ 8.04
Precision $\pm (\sigma)$	59.23 $\pm$ 12.99	40.96 $\pm$ 9.88	36.37 $\pm$ 3.65	36.92 $\pm$ 2.59	36.58 $\pm$ 6.67	34.94 $\pm$ 3.88	40.82 $\pm$ 16.43
F-measure $\pm (\sigma)$	70.92 $\pm$ 12.8	57.03 $\pm$ 9.73	52.94 $\pm$ 3.32	53.52 $\pm$ 3.16	53.29 $\pm$ 6.67	51.38 $\pm$ 4.29	55.73 $\pm$ 12.59
EMoSim (classic genetic operations)							
	Action	Artificial Intelligence	Collision of Objects	Game Graphics	Interaction Between Object Properties	Position of Object	Overall
Recall $\pm (\sigma)$	59.17 $\pm$ 23.82	68.34 $\pm$ 20.06	74.34 $\pm$ 12.28	65.83 $\pm$ 20.63	80 $\pm$ 23.29	72.59 $\pm$ 18.29	69.21 $\pm$ 22.99
Precision $\pm (\sigma)$	64.03 $\pm$ 6.98	36.33 $\pm$ 4.53	37.53 $\pm$ 6.13	35.95 $\pm$ 2.51	35.59 $\pm$ 9.87	34.14 $\pm$ 1.57	39.5 $\pm$ 15.66
F-measure $\pm (\sigma)$	59.88 $\pm$ 17.09	46.97 $\pm$ 7.96	49.37 $\pm$ 5.6	45.86 $\pm$ 6.81	48.66 $\pm$ 12.47	45.96 $\pm$ 4.69	47.8 $\pm$ 13.09
BLiMEA (baseline)							
	Action	Artificial Intelligence	Collision of Objects	Game Graphics	Interaction Between Object Properties	Position of Object	Overall
Recall $\pm (\sigma)$	46.11 $\pm$ 16.26	35.47 $\pm$ 9.31	35.14 $\pm$ 8.21	34.35 $\pm$ 9.11	36.95 $\pm$ 10.3	38.87 $\pm$ 10.38	41.15 $\pm$ 19.98
Precision $\pm (\sigma)$	41.08 $\pm$ 7.28	23.52 $\pm$ 6.75	22.05 $\pm$ 7.63	20.79 $\pm$ 3.08	22.58 $\pm$ 6.6	18.3 $\pm$ 3.47	25.64 $\pm$ 15.62
F-measure $\pm (\sigma)$	41.43 $\pm$ 3.87	27.81 $\pm$ 6.22	27.78 $\pm$ 6.15	25.56 $\pm$ 3.85	27.56 $\pm$ 6.62	24.62 $\pm$ 4.68	28.56 $\pm$ 11.62
Random Search (sanity check)							
	Action	Artificial Intelligence	Collision of Objects	Game Graphics	Interaction Between Object Properties	Position of Object	Overall
Recall $\pm (\sigma)$	56.67 $\pm$ 18.23	70.93 $\pm$ 16.53	72.09 $\pm$ 13.72	70.42 $\pm$ 16.11	81.67 $\pm$ 17.48	68.7 $\pm$ 10.17	68.79 $\pm$ 18.24
Precision $\pm (\sigma)$	7.13 $\pm$ 3.95	4.84 $\pm$ 2.3	9.77 $\pm$ 4.99	4.25 $\pm$ 1.88	3.78 $\pm$ 1.48	3.16 $\pm$ 1.33	5.23 $\pm$ 4.19
F-measure $\pm (\sigma)$	12.57 $\pm$ 6.73	9.03 $\pm$ 4.16	16.84 $\pm$ 8.05	8 $\pm$ 3.45	7.2 $\pm$ 2.75	6.03 $\pm$ 2.47	9.39 $\pm$ 6.78

The specific code that is executed to obtain the traces is in the file “Source Code/EMoSim/approach/simulation/EAlgorithm.kt” (lines 31-107).

3.5.4 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 3.3 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 3.3. Fig. 3.11 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.

RQ₁ 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.

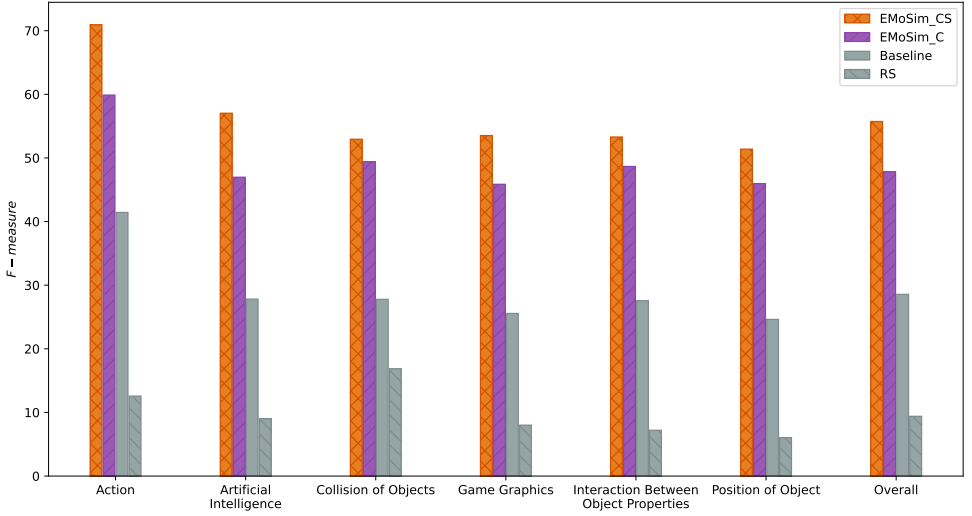


Figure 3.11: Bar chart that represents the F-measure values for each category of bugs.

RQ₂ answer. There are differences in performance between the classic genetic operations and the component-specific genetic operations. As Table 3.3 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.

3.5.5 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 [5].

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 [25]. In addition, according to Conover [18], 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 $< 2.2 \times 10^{-16}$ 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 algo-

rithm 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 3.4 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 3.4: Holm’s post hoc p – *Values* for each pair of algorithms in Kromaia.

	Recall	Precision	F-measure
EMoSim_CS vs EMoSim_C	$< 2 \times 10^{-16}$	0.83	2.1×10^{-8}
EMoSim_CS vs Baseline	$< 2 \times 10^{-16}$	3×10^{-5}	$< 2 \times 10^{-16}$
EMoSim_CS vs RS	$< 2 \times 10^{-16}$	$< 2 \times 10^{-16}$	$< 2 \times 10^{-16}$
EMoSim_C vs Baseline	9.7×10^{-8}	2.7×10^{-5}	2.8×10^{-12}
EMoSim_C vs RS	0.89	$< 2 \times 10^{-16}$	$< 2 \times 10^{-16}$
Baseline vs RS	2×10^{-9}	2.6×10^{-16}	3.2×10^{-14}

RQ₃ 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).

Table 3.5: 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)

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 [5]. 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 $\hat{A}_{12}$ [52, 26]. $\hat{A}_{12}$ 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 3.5 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 [17] values for recall, precision, and F-measure.

RQ₄ 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 [42] 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.

3.6 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 [13] (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 [36].

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.

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. [35] and Politowski et al. [38]. 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 [54, 59]. 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 [3] (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. 3.1). 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.

3.7 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 [5]. We

used measurements that are widely accepted in the software engineering research community (recall, precision, and F-measure) [6] 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 [6]. 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 [6], 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 [10]. 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 [6].

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.

3.8 Related Work

Table 3.6 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 3.6 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 [13, 14] 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

Table 3.6: 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. [12]	2015	✗	✓	✗	–	✗	✗
Ifrikhar et al. [28]	2015	✗	✗	✓	Context State, Position of Object	✗	✓
Sánchez-Cuadrado et al. [44]	2017	✗	✓	✗	–	✗	✗
Sánchez-Cuadrado et al. [43]	2018	✗	✓	✗	–	✗	✗
Troya et al. [48]	2018	✗	✓	✗	–	✗	✗
Ariyurek et al. [7]	2019	✓	✗	✗	Action, Crash, Event Occurrence	✗	✗
Zheng et al. [57]	2019	✓	✗	✗	Action, Audio, Crash, Game Graphics, Value	✗	✓
Ahumada and Bergel [1]	2020	✓	✗	✗	Action, Triggered Event	✗	✗
Ariyurek et al. [8]	2020	✓	✗	✗	Action, Crash, Event Occurrence	✗	✗
Bergdahl et al. [9]	2020	✓	✗	✗	Action, Exploit	✗	✗
Cheng et al. [15]	2020	✗	✓	✗	–	✗	✗
Wu et al. [55]	2020	✓	✗	✗	Context State, Crash	✗	✓
Zhang et al. [56]	2020	✗	✓	✗	–	✗	✗
Ferdous et al. [23]	2021	✗	✗	✓	Action, Triggered Event	✗	✗
Quach et al. [40]	2021	✗	✓	✗	–	✗	✗
Arcega et al. [4]	2022	✗	✓	✗	–	✗	✓
Ciborowska et al. [16]	2022	✗	✓	✗	–	✗	✓
Khanfir [30]	2022	✗	✓	✗	–	✗	✓
Liang et al. [32]	2022	✗	✓	✗	–	✗	✓
Tufano et al. [50]	2022	✓	✗	✗	Implementation Response	✗	✗
Liu et al. [33]	2023	✗	✓	✗	–	✗	✓
Politowski et al. [39]	2023	✓	✗	✗	Action, Context State, Interaction Between Object Properties	✗	✗
Pérez et al. [37]	2023	✗	✓	✗	–	✗	✓
Roca et al. [41]	2024	✗	✗	✓	Action, Artificial Intelligence, Game Graphics, Position of Object	✓	✓
Our previous work [13]	2022	✗	✗	✓	Action, Artificial Intelligence, Game Graphics, Position of Object	✓	✓
Our previous work [14]	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	✓	✓

of the taxonomy of bugs in video games [49] (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.

3.8.1 Bug Localization in Games

Some papers do bug localization in games as shown in Table 3.6. For instance, 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 [8], 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. [57] 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. [50] 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. [39] 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.

3.8.2 Bug Localization in Models

Outside the game domain, several techniques are used for bug localization in models. For example, Arcega et al. [4] 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. [48] 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. [44] 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 [43]. 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. [12] 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. [37] 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 3.6 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.

3.8.3 Bug Localization in Games That Use Models

Others do bug localization in games that use models as Table 3.6 shows. Iftikhar et al. [28] 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. [23] 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 3.6, 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. [28] and Ferdous et al. [23]) 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 [13] includes the novelty of using a fitness function that collects information about the simulations. It covers more categories of bugs than the previous works [28, 23] (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. [41] 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 [14] uses our EMoSim approach with the classic genetic operations as baseline [13] 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 [13] 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.

3.9 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.

Acknowledgements

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).

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Bibliography

- [1] Tomás Ahumada and Alexandre Bergel. “Reproducing Bugs in Video Games using Genetic Algorithms”. In: *2020 IEEE Games, Multimedia, Animation and Multiple Realities Conference (GMAX)*. 2020, pp. 1–6. DOI: 10.1109/GMAX49668.2020.9256837 (cit. on p. 157).

- [2] Apostolos Ampatzoglou and Ioannis Stamelos. “Software engineering research for computer games: A systematic review”. In: *Information and Software Technology* 52.9 (2010), pp. 888–901. ISSN: 0950-5849. DOI: <https://doi.org/10.1016/j.infsof.2010.05.004> (cit. on p. 111).
- [3] Lorena Arcega et al. “An approach for bug localization in models using two levels: model and metamodel”. In: *Software and Systems Modeling* 18.6 (2019), pp. 3551–3576. DOI: 10.1007/s10270-019-00727-y (cit. on pp. 111, 145, 152).
- [4] Lorena Arcega et al. “Bug Localization in Model-Based Systems in the Wild”. In: *ACM Trans. Softw. Eng. Methodol.* 31.1 (2021), 10:1–10:32. ISSN: 1049-331X. DOI: 10.1145/3472616 (cit. on pp. 111, 140, 157, 159).
- [5] Andrea Arcuri and Lionel Briand. “A Hitchhiker’s guide to statistical tests for assessing randomized algorithms in software engineering”. In: *Software Testing, Verification and Reliability* 24.3 (May 2014), pp. 219–250. ISSN: 0960-0833. DOI: 10.1002/stvr.1486 (cit. on pp. 113, 145, 148, 150, 154).
- [6] Andrea Arcuri and Gordon Fraser. “Parameter tuning or default values? An empirical investigation in search-based software engineering”. In: *Empirical Software Engineering* 18.3 (2013), pp. 594–623. ISSN: 1573-7616. DOI: 10.1007/s10664-013-9249-9 (cit. on p. 155).
- [7] Sinan Ariyurek, Aysu Betin-Can, and Elif Surer. “Automated Video Game Testing Using Synthetic and Humanlike Agents”. In: *IEEE Transactions on Games* 13.1 (2021), pp. 50–67. DOI: 10.1109/TG.2019.2947597 (cit. on p. 157).
- [8] Sinan Ariyurek, Aysu Betin-Can, and Elif Sürer. “Enhancing the Monte Carlo Tree Search Algorithm for Video Game Testing”. In: *IEEE Conference on Games, CoG 2020, Osaka, Japan, August 24-27, 2020*. IEEE, 2020, pp. 25–32. DOI: 10.1109/CoG47356.2020.9231670 (cit. on p. 157).
- [9] Joakim Bergdahl et al. “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. DOI: 10.1109/CoG47356.2020.9231552 (cit. on p. 157).

- [10] Daniel Blasco et al. “An evolutionary approach for generating software models: The case of Kromaia in Game Software Engineering”. In: *J. Syst. Softw.* 171 (2021), p. 110804. DOI: 10.1016/j.jss.2020.110804 (cit. on pp. 131, 145, 155).
- [11] Ilhem Boussaïd, Patrick Siarry, and Mohamed Ahmed-Nacer. “A survey on search-based model-driven engineering”. In: *Automated Software Engineering* 24.2 (2017), pp. 233–294. ISSN: 1573-7535. DOI: 10.1007/s10515-017-0215-4 (cit. on p. 128).
- [12] Loli Burgueño et al. “Static Fault Localization in Model Transformations”. In: *IEEE Transactions on Software Engineering* 41.5 (2015), pp. 490–506. ISSN: 0098-5589. DOI: 10.1109/TSE.2014.2375201 (cit. on pp. 157, 159).
- [13] Rodrigo Casamayor et al. “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*. Ed. by Eugene Syriani et al. ACM, 2022, pp. 356–366. DOI: 10.1145/3550355.3552440 (cit. on pp. 111, 113, 124, 136, 140, 151, 156, 157, 161).
- [14] Rodrigo Casamayor et al. “Studying the Influence and Distribution of the Human Effort in a Hybrid Fitness Function for Search-Based Model-Driven Engineering”. In: *IEEE Trans. Software Eng.* 49.12 (2023), pp. 5189–5202. DOI: 10.1109/TSE.2023.3329730 (cit. on pp. 156, 157, 161).
- [15] Xiaoyun Cheng et al. “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. DOI: 10.1109/COMPSAC48688.2020.0-152 (cit. on p. 157).
- [16] Agnieszka Ciborowska and Kostadin 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. DOI: 10.1145/3510003.3510042 (cit. on p. 157).

- [17] Norman Cliff. *Ordinal methods for behavioral data analysis*. Lawrence Erlbaum Associates, Inc, 1996, p. 212. DOI: <https://doi.org/10.4324/9781315806730> (cit. on p. 150).
- [18] William Jay Conover. *Practical Nonparametric Statistics, 3rd Edition*. 3. ed. Wiley series in probability and statistics. USA: Wiley, 1999. VIII, 584. ISBN: 0471160687 (cit. on p. 148).
- [19] Crytek. *CryEngine*. <https://www.cryengine.com>. [Online; accessed 21-November-2021]. 2002 (cit. on p. 110).
- [20] África Domingo et al. “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*. Ed. by Schahram Dustdar et al. Vol. 12127. Lecture Notes in Computer Science. Springer, 2020, pp. 353–367 (cit. on p. 119).
- [21] Yogesh K. Dwivedi et al. “Metaverse beyond the hype: Multidisciplinary perspectives on emerging challenges, opportunities, and agenda for research, practice and policy”. In: *Int. J. Inf. Manag.* 66 (2022), p. 102542. DOI: 10.1016/J.IJINFOMGT.2022.102542 (cit. on p. 109).
- [22] Epic Games. *Unreal Engine*. <https://www.unrealengine.com>. [Online; accessed 21-November-2021]. 1998 (cit. on p. 110).
- [23] Raihana Ferdous et al. “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*. Ed. by Una-May O’Reilly and Xavier Devroey. Vol. 12914. Lecture Notes in Computer Science. Springer, 2021, pp. 56–71. DOI: 10.1007/978-3-030-88106-1_5 (cit. on pp. 157, 160, 161).
- [24] PC Gamer. *How buggy is Cyberpunk 2077, really?* <https://www.pcgamer.com/how-buggy-is-cyberpunk-2077-really/>. [Online; accessed 21-November-2021]. 2020 (cit. on p. 110).
- [25] Salvador García et al. “Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data

- mining: Experimental analysis of power”. In: *Information Sciences* 180.10 (May 2010). Special Issue on Intelligent Distributed Information Systems, pp. 2044–2064. ISSN: 0020-0255. DOI: <http://dx.doi.org/10.1016/j.ins.2009.12.010> (cit. on p. 148).
- [26] Robert J. Grissom and John J. Kim. *Effect sizes for research: A broad practical approach*. New Jersey, United States: Lawrence Erlbaum Associates Publishers, 2005 (cit. on p. 150).
- [27] Ahmed E. Hassan and Richard C. Holt. “The Top Ten List: Dynamic Fault Prediction”. In: *Proceedings of the 21st IEEE International Conference on Software Maintenance*. ICSM ’05. USA: IEEE Computer Society, 2005, 263–272. ISBN: 0769523684. DOI: 10.1109/ICSM.2005.91 (cit. on p. 111).
- [28] Sidra Iftikhar et al. “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*. Ed. by Timothy Lethbridge, Jordi Cabot, and Alexander Egyed. IEEE Computer Society, 2015, pp. 426–435. DOI: 10.1109/MODELS.2015.7338274 (cit. on pp. 157, 160, 161).
- [29] Hisao Ishibuchi, Yusuke 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. DOI: 10.1109/CEC.2006.1688438 (cit. on p. 142).
- [30] Ahmed 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*. Ed. by Abhik Roychoudhury, Cristian Cadar, and Miryung Kim. ACM, 2022, pp. 1790–1794. DOI: 10.1145/3540250.3558907 (cit. on p. 157).
- [31] J. Richard Landis and Gary G. Koch. “The Measurement of Observer Agreement for Categorical Data”. In: *Biometrics* 33.1 (1977) (cit. on p. 120).

- [32] Hongliang Liang, Dengji Hang, and Xiangyu Li. “Modeling function-level interactions for file-level bug localization”. In: *Empir. Softw. Eng.* 27.7 (2022), p. 186. DOI: 10.1007/s10664-022-10237-z (cit. on p. 157).
- [33] Di Liu et al. “Towards understanding bugs in Python interpreters”. In: *Empir. Softw. Eng.* 28.1 (2023), p. 19. DOI: 10.1007/s10664-022-10239-x (cit. on p. 157).
- [34] Márcio de Oliveira Barros and Arilo Claudio Dias Neto. *Threats to Validity in Search-based Software Engineering Empirical Studies*. Tech. rep. 0006/2011. Jan. 2011 (cit. on p. 154).
- [35] Luca Pascarella et al. “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*. Ed. by Andy Zaidman, Yasutaka Kamei, and Emily Hill. ACM, 2018, pp. 392–402. DOI: 10.1145/3196398.3196418 (cit. on pp. 109, 110, 152).
- [36] F. Pérez, T. Ziadi, and C. Cetina. “Utilizing Automatic Query Reformulations as Genetic Operations to Improve Feature Location in Software Models”. In: *IEEE Transactions on Software Engineering* 2 (2020), pp. 713–731. ISSN: 1939-3520. DOI: 10.1109/TSE.2020.3000520 (cit. on pp. 135, 136, 151).
- [37] Francisca Pérez et al. “How the Quality of Maintenance Tasks is Affected by Criteria for Selecting Engineers for Collaboration”. In: *ACM Trans. Softw. Eng. Methodol.* 32.3 (2023), 63:1–63:22. DOI: 10.1145/3561384 (cit. on pp. 157, 160).
- [38] Cristiano Politowski, Fábio Petrillo, and Yann-Gaël 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. DOI: 10.1109/AST52587.2021.00018 (cit. on pp. 110, 111, 152).
- [39] Cristiano Politowski et al. “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. DOI: 10.1109/GAS59301.2023.00011 (cit. on pp. 157, 158).
- [40] Sophia Quach et al. “Evaluating the impact of falsely detected performance bug-inducing changes in JIT models”. In: *Empir. Softw. Eng.* 26.5 (2021), p. 97. DOI: 10.1007/s10664-021-10004-6 (cit. on p. 157).
- [41] Isis Roca et al. “Co-evolving scenarios and simulated players to locate bugs that arise from the interaction of software models of video games”. In: *Inf. Softw. Technol.* 169 (2024), p. 107412. DOI: 10.1016/J.INFSOF.2024.107412 (cit. on pp. 157, 161).
- [42] Jeanine Romano et al. “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 (cit. on p. 151).
- [43] Jesús Sánchez Cuadrado, Esther Guerra, and Juan Lara. “Quick Fixing ATL Transformations with Speculative Analysis”. In: *Softw. Syst. Model.* 17.3 (July 2018), 779–813. ISSN: 1619-1366. DOI: 10.1007/s10270-016-0541-1 (cit. on pp. 157, 159).
- [44] Jesús Sánchez Cuadrado, Esther Guerra, and Juan Lara. “Static Analysis of Model Transformations”. In: *IEEE Transactions on Software Engineering* 43.9 (2017), pp. 868–897. DOI: 10.1109/TSE.2016.2635137 (cit. on pp. 157, 159).
- [45] 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 (cit. on p. 111).
- [46] SlashData. *Global developer population report 2019*. <https://sdata.me/GlobalDevPop19>. [Online; accessed 21-November-2021]. 2019 (cit. on p. 109).
- [47] Steam Community. *Cyberpunk 2077 Bug Reports and Discussions*. Accessed: 2024-06-20. 2020 (cit. on p. 111).

- [48] Javier Troya et al. “Spectrum-Based Fault Localization in Model Transformations”. In: *ACM Trans. Softw. Eng. Methodol.* 27.3 (Sept. 2018). ISSN: 1049-331X. DOI: 10.1145/3241744 (cit. on pp. 157, 159).
- [49] Andrew Truelove, Eduardo Santana de Almeida, and Iftekhhar 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. DOI: 10.1109/ICSE43902.2021.00073 (cit. on pp. 112, 119, 120, 126, 157).
- [50] Rosalia Tufano et al. “Using Reinforcement Learning for Load Testing of Video Games”. In: *Proceedings of the 44th International Conference on Software Engineering (ICSE 2022), Pittsburgh, USA, 2022* (cit. on pp. 157, 158).
- [51] Unity Technologies. *Unity*. <https://unity.com>. [Online; accessed 21-November-2021]. 2005 (cit. on p. 110).
- [52] András Vargha and Harold D. Delaney. “A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong”. In: *Journal of Educational and Behavioral Statistics* 25.2 (2000), pp. 101–132. DOI: 10.3102/10769986025002101. eprint: <http://jeb.sagepub.com/content/25/2/101.full.pdf+html> (cit. on p. 150).
- [53] Hang Wang et al. “A Survey on the Metaverse: The State-of-the-Art, Technologies, Applications, and Challenges”. In: *IEEE Internet Things J.* 10.16 (2023), pp. 14671–14688. DOI: 10.1109/JIOT.2023.3278329 (cit. on p. 109).
- [54] W. Eric Wong et al. “A Survey on Software Fault Localization”. In: *IEEE Trans. Software Eng.* 42.8 (Aug. 2016), pp. 707–740. ISSN: 0098-5589. DOI: 10.1109/TSE.2016.2521368 (cit. on pp. 110, 111, 140, 152).
- [55] Yuechen Wu et al. “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. DOI: 10.1109/ICSME46990.2020.00074 (cit. on p. 157).

- [56] Jinglei Zhang et al. “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. DOI: 10.1145/3387904.3389281 (cit. on p. 157).
- [57] Yan Zheng et al. “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. DOI: 10.1109/ASE.2019.00077 (cit. on pp. 157, 158).
- [58] Meng Zhu and Alf Inge Wang. “Model-driven Game Development: A Literature Review”. In: *ACM Comput. Surv.* 52.6 (2020), 123:1–123:32. DOI: 10.1145/3365000 (cit. on p. 110).
- [59] Thomas Zimmermann et al. “Mining Version Histories to Guide Software Changes”. In: *Proceedings of the 26th International Conference on Software Engineering*. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 563–572. ISBN: 0-7695-2163-0 (cit. on pp. 111, 152).

Part III

Discussion

Discussion

The present part discusses this work, taking into consideration the motivation, research questions, and results of the thesis, given the outcomes of the research articles presented in Part I and in Part II. Additionally, the discussion introduces the research lines of the new works started as a continuation of this research.

Research Questions

At the outset of this research, this part aims to discuss the answer to each of the research questions that were presented in Part I. Next, each research question is now discussed in turn, along with a summary of the findings and insights gained:

RQ₁: *How can video game simulations be used in search-based techniques to locate bugs in software models, and is there any difference in performance in terms of solution quality compared to traditional bug localization approaches?*

The first work demonstrated how video game simulations can be used in an evolutionary search-based approach to locate bugs in software models of video games. The EMoSim approach exemplified how formulating bug localization as a search problem (where each individual is a game simulation scenario) allows the exploration of a vast space of behaviors to uncover failure-causing situations.

The results show that EMoSim was able to locate buggy model elements and it significantly outperformed the baseline technique (which relied on static analysis of bug reports) in terms of accuracy (for example, achieving roughly a 20% higher F-measure in the case study). This indicates that simulations are a potent strategy for bug localization in the context of game software engineering.

By automatically evolving gameplay scenarios that trigger bugs, the approach reduced the manual effort needed from developers to reproduce and identify defects. Hence, video game simulations in search-based techniques can be used to locate bugs in software models, and can surpass traditional methods, which do not use video game simulations.

This work was published in MODELS'22 [2] and ranked as the second best paper of the conference. The novelty and relevance of this work made that ACM SIGSOFT Distinguished Paper Award was received. Moreover, it was later invited to be presented in the Showcase Track at ICSE'23.

RQ₂: *What is the influence of the human effort for assessing candidate solutions in search-based bug localization?*

The second work addressed this question by introducing human expertise into the evaluation loop of candidate solutions during the location of bugs. The SimuHaFF approach introduces the first hybrid fitness function that combines human evaluations with simulated gameplay traces, where simulations handle large-scale exploration and human experts assess selected candidates to avoid fatigue.

The results indicated that incorporating human for evaluating candidate solutions as fitness function can improve the search performance in terms of solution quality. We propose three variants for the hybrid fitness function that vary in the distribution of human effort in order to study whether the variants influence the performance in terms of solution quality. Specifically, the experiments showed that a hybrid fitness function (where a human evaluator assesses certain intermediate results or final candidates) can improve the quality of the solution (e.g., reducing false positives) compared to only using an algorithmic fitness function. However, the timing and frequency of human involvement are critical. If a human is asked to evaluate too many candidates or too often, the effort may become impractically high, leading to evaluator fatigue and thus negating the benefits of automation. Results indicated that concentrating the human input at a strategic point (for instance, allowing the algorithm to run largely autonomously and then having the human validate a small set of top-ranked results) yields a significant performance boost with minimal manual effort. In one variant, the approach achieved nearly the same bug localization accuracy as scenarios involving continuous human oversight, while substantially reducing the required human workload.

Thus, in answer to RQ₂, it can be concluded that a hybrid fitness function that combines human effort with human simulations for SBSE can improve bug localization performance, and that the distribution of human effort should be optimized (for example, using the human as a final oracle or periodic guide rather than at every iteration to avoid human fatigue). The synergy between human insight and automated search leads to better results than either alone: the human corrects or guides the

search on aspects that are hard to quantify, and the algorithm relieves the human from brute-force exploration.

This work was published in TSE'23 journal [3], presented at CEDI'24 and also discussed with a focus group of professional developers.

RQ₃: *To what extent can component-specific genetic operations improve the effectiveness of search-based bug localization in video games?*

The third research question probed whether infusing domain-specific knowledge into the evolutionary search (via custom genetic operations) would yield tangible improvements. The results from the third work indicate that using component-specific genetic operations significantly boosts bug localization performance, rather than simply combining all components as a whole. Designing crossover and mutation operators that respect the structure of game models (e.g., recombining scenarios at the level of individual game elements, and maintaining model validity), led to improvements in performance (in terms of recall, precision, and F-measure).

The component-specific genetic operations enabled the search to navigate the solution space by selectively combining simulation components instead of applying generic operators to all elements at once. In the Kromaia case, this translated into improved bug localization performance in terms of solution quality. Quantitatively, the approach with component-specific operations achieved up to a 27% higher F-measure than the baseline approach that focuses on CSE and utilizes bug reports, and it even outperformed the baseline that uses the widespread genetic operations that randomly combine all of the components by around 8%. These are significant gains in the context of bug localization.

Therefore, customizing the genetic operations to game model components boosts the performance of the bug localization. Moreover, the results show whether there is any difference in performance among the different categories of bugs in video games. This result confirms that one-size-fits-all metaheuristics can be suboptimal, and that incorporating knowledge of the problem domain (here, the game's model/metamodel) is a powerful way to enhance SBSE techniques. It also suggests that future

search-based approaches in GSE should consider similar customizations for other tasks, as the effort to craft domain-aware operators can pay off in performance.

This work was accepted for its publication in a journal (SOSYM'24 [1]).

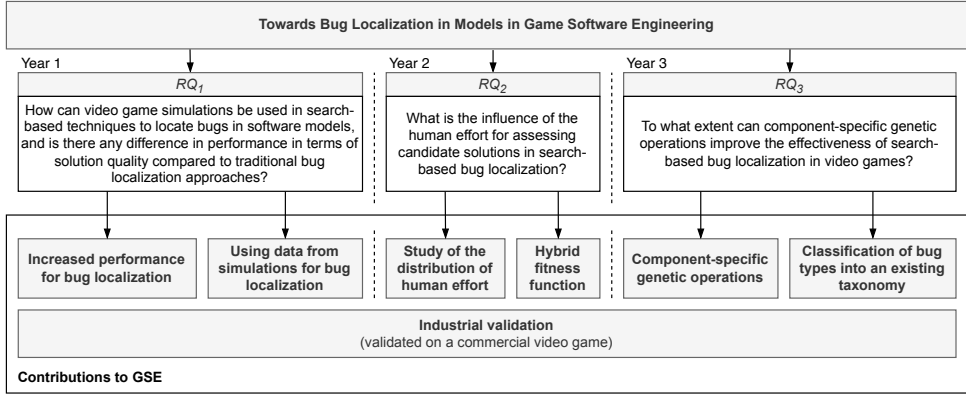


Figure 12: Research question outcome overview.

The findings demonstrate that Search-Based Game Software Engineering approaches are not only feasible but also beneficial for addressing a game development challenge like bug localization. Figure 12 summarizes the outcomes of this thesis with regard to each of the research questions formulated. By combining automated evolutionary search with game simulations (RQ₁), human effort for assessing candidate solutions (RQ₂), and component-specific genetic operations (RQ₃), this research advances the state-of-the-art beyond previous approaches. These findings reinforce the initial motivation: game software engineering problems, though complex, can be effectively tackled by bridging domain knowledge with intelligent search techniques. Moreover, the positive outcomes obtained in an industrial setting (the Kromaia game) give confidence that these approaches can have real-world impact. In the broader perspective, this work provides a concrete example of how tailoring software engineering research to the needs of the video game industry (and vice versa, leveraging game features in CSE) can yield fruitful results. The following section discusses how this work can be extended and what new questions have emerged, outlining future research directions.

Future Lines of Research

While this thesis has answered its primary research questions, it also opens up several new avenues for exploration. The following are two prominent future lines of research that emerge from the findings and have already begun to be pursued as a continuation of this work:

Generalization to Other Domains and Scenarios

One immediate direction is to generalize the approaches developed here to other contexts beyond the specific scope of this thesis. First, within the domain of video games, applying search-based bug localization approaches to different game genres and engines would be valuable. For example, one could evaluate the simulation-driven bug search on a larger open-world game or a networked multiplayer game, where the types of bugs and model structures differ from Kromaia’s scenario. Such studies would test the versatility of EMoSim and its variants, potentially revealing any adjustments needed for different kinds of game logic (e.g., physics-heavy simulations or AI-driven emergent gameplay). Second, an exciting possibility is to transfer the concept to classic software engineering (CSE) domains. The success of evolving game simulations to find bugs suggests that similar search-based techniques might assist in traditional software debugging: for instance, by simulating user interactions in interactive applications or by generating sequences of API calls that lead to a failure in enterprise systems. This could help bridge GSE and CSE, as the results already suggest that techniques effective for games may also benefit non-game software. Investigating this could involve creating “simulation” models for non-game software (such as using UI testing frameworks or environment models for cyber-physical systems) and then employing evolutionary search to find failing scenarios. Additionally, beyond bug localization, these approaches could inspire new SBSE applications in game development. For instance, the idea of evolving simulations might be extended to automated game testing more broadly, or even to content generation (where the “fitness” could be fun or difficulty rather than bug manifestation).

Another future line of work focuses on refining and improving the SBSE approaches using the latest cutting-edge practices in technology and methodology at the time of this writing. One promising avenue is the integration of machine learning or intelligent heuristics to complement the evolutionary search. For example, one could train a predictive model (using data from past evolutionary runs or developer feedback) to guide the search more intelligently, essentially a form of learned fitness approximation or directing mutations towards likely bug-revealing changes. This might reduce the computational cost of simulations by avoiding clearly unpromising regions of the search space. Similarly, there is potential in exploring co-evolutionary algorithms. Instead of evolving only game scenarios, a co-evolution approach could simultaneously evolve two populations (e.g., game scenarios and AI player behaviors) that compete or collaborate, as a means to discover even more subtle bugs that involve complex player-game interactions. Co-evolution could push the system under test in creative ways, potentially uncovering issues that a single-population genetic algorithm might miss. Another refinement is to further decrease reliance on human input by using surrogate models for human judgment: for instance, if SimuHaFF showed that human feedback at certain points is crucial, future work could attempt to replicate those judgments with a trained model (like a neural network classifier that predicts a human's assessment). This would align with the trend of human-augmented AI being eventually automated.

Regarding genetic operations, the work on component-specific crossover and mutation represents just one example of domain tailoring. Future research can design new genetic operations or evolutionary strategies for game models, perhaps targeting specific categories of bugs. For instance, if certain bugs involve timing issues, one could introduce a mutation that specifically tweaks time-related parameters in simulations. The significant improvement observed through the use of specialized operators indicates that investment in customized heuristics is worthwhile. The community is encouraged to explore a variety of alternative genetic operations applied to game model debugging, rather than defaulting to standard implementations.

Beyond algorithmic improvements, another line of refinement is tool support and developer integration. Future work envisions the creation of developer-friendly tools (such as plugins for game engines or model editors) that integrate these bug localization techniques behind the scenes. Such tools could run evolutionary bug searches during nightly builds or continuous integration cycles, alerting developers to problematic model elements before they become expensive issues. Integrating the research into practical workflows not only validates the approach in real development settings but could also provide feedback for further refinement (e.g., logging developer actions to improve the fitness function or operator design).

Finally, studying the effect of this 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 its success. However, most APR tools require the location to be performed on code statements instead of considering other software artifacts such as model elements. Moreover, taking advantage of evolving simulations can be relevant not only for locating bugs in software models but also for automatically fixing the located bugs.

These efforts represent the next chapter of Game Software Engineering, building directly on the contributions of this work. There is reason for optimism that future work will further consolidate the role of SBSE in game development practice and uncover new insights into both software engineering and game design processes.

Bibliography

- [1] Rodrigo Casamayor et al. “Boosting bug localization in software models of video games with simulations and component-specific genetic operations”. In: *Software and Systems Modeling* (2025). ISSN: 1619-1374. DOI: 10.1007/s10270-024-01253-2 (cit. on p. 179).

- [2] Rodrigo Casamayor et al. “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*. Ed. by Eugene Syriani et al. ACM, 2022, pp. 356–366. DOI: 10.1145/3550355.3552440 (cit. on p. 176).

- [3] Rodrigo Casamayor et al. “Studying the Influence and Distribution of the Human Effort in a Hybrid Fitness Function for Search-Based Model-Driven Engineering”. In: *IEEE Trans. Software Eng.* 49.12 (2023), pp. 5189–5202. DOI: 10.1109/TSE.2023.3329730 (cit. on p. 178).

Part IV

Conclusions

Conclusions

The last chapter describes the final conclusions of this thesis with regard to the objectives, results, and impact achieved by the research in the context studied through the development of the work.

The primary aim of this thesis was to develop and evaluate novel approaches for automating bug localization in video game software models using search-based techniques. Over the course of the research, three complementary approaches were implemented, each addressing a facet of this aim, and validated them on an industrial game. The work followed a design science methodology, ensuring that each solution was evaluated and iteratively improved. Overall, the thesis succeeded in demonstrating that Search-Based Software Engineering can be effectively applied to Game Software Engineering problems: the proposed approaches can locate bugs in game models more efficiently and accurately than traditional methods, thereby contributing to improved maintenance of game software. The research has culminated in a compendium of three peer-reviewed publications (one in a specialized international conference and two in high-impact journals), which attests to the novelty and relevance of the contributions.

Concretely, the research produced the following published works (included in Section II) with their key outcomes:

- **The first work:** Introduced the EMoSim approach, an evolutionary algorithm using game simulations to localize bugs in software models. This work demonstrated the viability of evolving in-game scenarios to detect model faults, and it showed significant improvements over a traditional bug localization baseline in terms of accuracy and thoroughness.
- **The second work:** Presented SimuHaFF, a hybrid approach that combines simulation-based search with human effort in the fitness function. This study provided insights into how human expertise can be injected into an automated process and quantified the trade-offs. It confirmed that a well-balanced hybrid fitness function that combines human effort with human simulations for SBSE can enhance bug localization results while limiting the required human effort.
- **The third work:** Developed an enhanced version of the evolutionary approach by incorporating component-specific genetic operations tailored to game models. This work achieved further perfor-

mance gains, boosting bug localization success rates across different bug types. It underscored the benefit of customizing search algorithms with domain knowledge, as the specialized operators outperformed generic ones.

These publications collectively capture the research trajectory and contributions of the thesis. The positive outcomes reported (improvements in bug localization precision, reduced manual debugging effort, and successful application in an industrial game) validate the thesis objectives and demonstrate a meaningful advancement in the field of Game Software Engineering.

Based on the results of the research, several important conclusions can be drawn:

- 1. Feasibility and Effectiveness of Simulation-Based SBGSE for Bug Localization:** It is demonstrated that Search-Based Game Software Engineering techniques, by leveraging video game simulations, can solve real-world problems in game development, such as automatically localizing bugs in software models. By casting bug-finding as an optimization problem and evolving gameplay scenarios, higher bug detection accuracy was achieved compared to conventional methods. This confirms that the dynamic and complex nature of video games, which traditionally made debugging difficult, can be turned into an advantage by using in-game simulations as a testing ground for an evolutionary search. In short, automated search can navigate the enormous space of game behaviors more efficiently than manual testing, making it a powerful tool for game maintenance.
- 2. Value of Hybrid Human-AI Approaches:** This research shows that combining human insight with automated search yields better outcomes than either alone, when the combination is done in a carefully managed way. Human expertise remains crucial for aspects of quality that are hard to quantify (e.g., distinguishing a true bug from a benign anomaly), but the results demonstrate that human involvement is not required at every step. A strategic infusion of human evaluation (for example, validating the final results of an

algorithm or providing feedback at key decision points) can guide the search algorithm to avoid false paths and improve precision, all while keeping the human effort at a minimal and acceptable level. This conclusion reinforces a broader principle in software engineering: intelligent tools should augment developers, not replace them entirely, allowing humans to focus on judgment-intensive tasks and machines to handle brute-force computation.

- 3. Impact of Domain-Specific Knowledge on SBSE Performance:** A significant finding of this thesis is that incorporating domain-specific knowledge into SBSE techniques leads to substantial performance improvements. By tailoring genetic operators to respect the structure of game models and the nature of game bugs, the evolutionary algorithm became more effective at its task: converging faster and finding bugs that a generic approach might miss. This highlights that the success of SBSE is not only about applying generic metaheuristics, but also about understanding the problem domain and encoding that understanding into the search process. For game software engineering (and likely other domains), this means that researchers and practitioners should strive to customize their optimization techniques (for instance, using problem-specific solution encodings, operators, or fitness metrics) to get the best results. This work serves as evidence that such customization is worth the effort, as it can yield better outcomes than out-of-the-box algorithms.

Beyond these specific conclusions, the overall message of the thesis is that game development can benefit from automated, search-based solutions to traditionally hard problems. In particular, this work demonstrates the potential of exploiting information from evolved simulations to automatically locate bugs in game models. The case study on Kromaia demonstrated that even a complex, real-world game can be used to automatically localize bugs, something that would have otherwise required extensive manual debugging by the developers. This implies that game companies could adopt similar approaches in their development pipelines to catch bugs earlier and more reliably, potentially improving game quality and reducing time spent on tedious testing.

The contributions of this thesis have both scientific and practical impact. Scientifically, they add to the growing body of knowledge in Search-Based Software Engineering by extending it into the gaming domain, and they contribute to Game Software Engineering by providing new approaches specifically tailored for game maintenance. The publications arising from this work have introduced new concepts (such as simulation-driven bug localization and hybrid fitness functions with human effort) that other researchers can build upon. Practically, the approaches developed here take a step toward automated debugging tools for games. In an industry increasingly characterized by complex software and the need for rapid updates, such tools can be very valuable. By validating the approaches on a commercial video game, progress has been made toward bridging the gap between academic research and industry practice in this area. It is anticipated that the ideas presented (for example, using AI agents to stress-test game scenarios and find bugs) could inspire future debugging features in game engines or be integrated into quality assurance processes.

In conclusion, this thesis has achieved its goals of improving bug localization in video game models and has demonstrated the promise of search-based techniques in game development contexts. The work stands at the intersection of software engineering and game development, illustrating how innovations in one domain (SBSE) can effectively address challenges in another (GSE). By evolving simulations, incorporating human expertise, and customizing algorithms for the domain, this research advances the state-of-the-art in automated bug location for games. It is expected that these contributions will not only be viewed as academic results, but also as practical foundations for next-generation tools that make game development more robust and efficient. Ultimately, by reducing the burden of debugging, developers can focus more on creativity and design, leading to better games and a more enjoyable experience for both creators and players. The research “towards bug localization in models in game software engineering” does not end here, it opens new pathways for future research and development, with anticipation for the continued evolution of this exciting interdisciplinary field.