The Journal of Systems & Software 188 (2022) 111261

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A co-evolutionary genetic algorithms approach to detect video game bugs☆

Aghyad Albaghajati [a], Moataz Ahmed [b,*]

[a] Information and Computer Science Department, King Fahd University of Petroleum & Minerals, Dhahran, Saudi Arabia
[b] Interdisciplinary Research Center of Intelligent Secure Systems (IRC-ISS), King Fahd University of Petroleum & Minerals, Dhahran 31261, Saudi Arabia

ARTICLE INFO

Article history:
Received 19 April 2021
Received in revised form 11 December 2021
Accepted 3 February 2022
Available online 16 February 2022

Keywords:
Video game
Automated software testing
Genetic algorithms
Colored Petri nets
Reachability analysis

ABSTRACT

Video games systems are known for their complexity, concurrency and non-determinism, which makes them prone to challenging tacit bugs. Video games development is costly and the corresponding verification process is tiresome. Testing the nondeterministic and concurrent behaviors of video games systems is not only crucial but also challenging, especially when the game state space is huge. Accordingly, typical software testing approaches are neither suitable nor effective to find related bugs. Novel automated approaches to support video game testing are needed. This problem has caught researchers' attention recently. Approaches found in the literature have tried to address two sub problems: modeling and uncovering bugs. Colored Petri nets is known to support modeling and verifying concurrent and nondeterministic systems. Search approaches have been used in the literature to check the availability of faulty states through exploring state spaces. However, these approaches tend to lack adaptability to test different video games systems due to the limitations of the defined fitness functions, in addition to difficulties in searching huge state spaces due to exhaustive and unguided search. The availability of automated approaches that guide and direct the process of bugs finding is mandatory. Thus, in this study we address this problem as we present a solution for automated software testing using collaborative work of two genetic algorithms (i.e. co-evolutionary) agents, where our approach is applied to colored Petri nets representations of the software workflow. The results of our experiments have shown the potential of the proposed approach in effectively finding bugs automatically.

1. Introduction

A video game system is an immensely complex software product where it is composed of various special aspects like creative design, visual presentations, and nondeterministic behaviors (Blow, 2004; Kanode and Haddad, 2009). In fact, video games systems (for simplicity, we refer to them as "games" throughout this study) are recognized to be intricate software systems, since different system's components are integrated with each other to produce a game that is playable, attractive and enjoyable (Blow, 2004; Aarseth, 2012). This has notably advanced the industry of video games throughout the years (Beattie, 2021).

With the evolution of the video game industry, enormous consumers and fanbase have matured to enjoy games that are built with great quality and provide a stunning user experience. A study by gamesindustry.biz (GamesIndustry.biz, 2021) has shown that the overall revenue value raised in the global video game market of games published on different platforms (i.e. Console, PC, Mobile and Web) was around 174.9 billion US dollars in 2020. Therefore, the verification, validation, and quality assurance processes are considered very important in the video game industry (Murphy-Hill et al., 2014). The verification of game's quality can be achieved via various activities, where playtesting is one of such activities, which is the exercise of playing, evaluating, and reviewing a game (Schultz and Bryant, 2016).

In general, testing software systems and detecting their faults are considered two key aspects in software development (Basili and Selby, 1987). Testing software systems involves verifying, validating, and checking software products to fulfill the needs and specifications of requirements and design, and to ensure that the developed software is meeting the expectations (Klaib, 2015).

With the growing need for more sophisticated software development, testing became not just essential but also more difficult (Kuhn et al., 2004; Bertolino, 2007). Generally in the software industry, testing has been a time-consuming and exhausting

☆ Editor: Raffaela Mirandola.

* Corresponding author.
E-mail addresses: g201703510@kfupm.edu.sa (A. Albaghajati), moataz@kfupm.edu.sa (M. Ahmed).

https://doi.org/10.1016/j.jss.2022.111261
0164-1212/

work. Therefore, to improve quality and reduce costs, automated testing became widely adopted (Bertolino, 2007).

Hiring testers to play and test builds of a game manually at different development stages is a common approach to test games in the industry nowadays (Mozgovoy and Pyshkin, 2018; Ostrowski and Aroudj, 2013). However, due to the complexity of video games (e.g. randomness, nondeterminism, and huge state spaces), automating the testing process has become a necessity. Moreover, in contrast to traditional software development, other dimensions and aspects are considered in video game quality assurance other than performance and correctness, such as testing for game balancing, multiplayer networking, physics, fun factor, level design, etc. (Albaghajati and Ahmed, 2020). Hence, utilizing automated agents can optimize and support the playtesting process (Ortega et al., 2013), where agents can be used to support finding bugs and invalid states, improving the level design and visuals, or enhancing the challenges and fun factor. Thus, using them not only helps in cutting down the costs of testing, but also they help saving time and improving the quality. These automated playtesting agents are recognized to be faster and more efficient than human testers in terms of exploring and checking the state space of a game (Khalifa et al., 2016), in addition to the ability of playing and repeating the game under test multiple times, which helps and supports game developers and designers during the development lifecycle.

Finding automated testing solutions that are generally applicable on games would also make it easier to apply such techniques on other types of software systems. In our previous study (Albaghajati and Ahmed, 2020), we analyzed several automated game testing approaches. Many of these approaches are based on search algorithms (Albaghajati and Ahmed, 2020). In addition, a limited number of studies were found to utilize colored Petri nets to test and find bugs in games, and these testing approaches were based on model checking to reveal bugs using temporal logic and formal verification. However, finding bugs related to nondeterminism and concurrency and in huge search space by exhaustively searching through states is not effective and is time consuming. Moreover, the implications of the literature review (Albaghajati and Ahmed, 2020) have revealed that model based testing approaches and other testing approaches in the literature lack generality of application, where these approaches were applied on specific game genres and using certain game engines. Therefore, the availability of approaches that guide and direct the searching process of such bugs is mandatory (Albaghajati and Ahmed, 2020).

Recognizing the need for automated game testing approaches, we contribute by proposing a novel solution using collaborative work of two genetic algorithms (i.e. co-evolutionary) agents. Our study focuses on the development of a testing approach that is composed of two agents which are based on genetic algorithms, where they work in collaboration to find bugs in games. The first agent is responsible for generating buggy states of the system under test through breaking rules. The other agent takes the generated buggy states from the first agent and applies reachability analysis on colored Petri nets model representation of the system under test to check the availability of such buggy states and to report the scenarios and sequences of events that lead to them. This approach was developed in an attempt to fill some of the gaps found in the literature that were presented in the previous study (Albaghajati and Ahmed, 2020).

In this study we try to find answers to the following research questions, which motivated our work:

- **RQ1:** How can we model game rules?
  - In answering this research question, we would be able to specify a scheme and a way of representing game rules that act as a ground check when searching for buggy states. Game rules are important aspect in games which specify constraints and conditions in games, as described in Section 2.1.
- **RQ2:** What criteria and characteristics of state variables do we have to mutate to find bugs in games?
  - This research question addresses the development of criteria and characteristics of state variables to allow defining bugs. Such criteria and characteristics would consider the state variables' invariants such as state variables' dependencies. The criteria would define the relations between the characteristics and the bugs.
- **RQ3:** How can we reveal game bugs?
  - Answering this research question would investigate using genetic algorithms and reachability analysis to find bugs in games.
- **RQ4:** What types of game bugs can be revealed by our proposed approach?
  - Answering this research question would study the found bugs in the experiments and their types.

The remainder of this paper is organized as follows. Section 2 discusses the necessary background of this research. Related work is presented in Section 3. Section 4 presents our proposed game rules scheme and co-evolutionary game testing approach. Experiments setup and design are discussed in Section 5. Section 6 presents experiments' results, discussion and observations. Finally, Section 7 concludes this study by discussing the limitations of the approach, addressing threats to validity, and proposing future work and further research directions.

2. Background

In this section we discuss important areas and concepts related to our study that would allow for more understanding of the carried on work.

2.1. Games

Video games are complex software systems, where several subsystems work together to make a game well-functioning, attractive and enjoyable (Blow, 2004; Aarseth, 2012). These systems can be formally structured through rules and constraints, where these rules are known to be *game rules* (Salen et al., 2004). Game rules act as abstract guidelines describing how a game system functions (Salen et al., 2004). These rules enforce some limits that define the game bounds, which strict the players to take specific paths to reach certain goals (Salen et al., 2004). Game rules help in determining when a game begins and when it ends, what can and cannot be done, and what should happen when player's actions are received (Togelius and Schmidhuber, 2008). Hence, game rules constrain the game mechanics and actions invoked by the player (Sicart, 2008). Game mechanics provide the capability of interacting with the game state, while game rules provide regulations of transitions between game states (Sicart, 2008).

As a video game is constructed from different subsystems which are constrained by several rules and mechanics, the bugs produced from these systems come in different shapes and forms, where some bugs affect the major part of the game where they can cause a severe crash and stop to the gameplay, others could

be general that they ruin player's experience without stopping player's progression (such as bugs related to non-player characters and environmental elements), and some others might be minor that could cause issues related to game visuals such as game graphics, animations, texts, or user interfaces, in addition to different types of bugs with different levels of complexities (Albaghajati, 2021).

2.2. Colored Petri nets

Petri nets is a formal graphical and mathematical modeling technique for describing and analyzing systems that are characterized as being concurrent, asynchronous distributed, parallel, nondeterministic, and/or stochastic (Murata, 1989). Petri nets can be used for automated model-based testing and model checking to verify nondeterministic behavior of software systems (Offutt and Thummala, 2019). There are several types of Petri nets, and one of them is colored Petri nets (Thong and Ameedeen, 2015).

Colored Petri nets models can be used for system verification by proving that some properties are fulfilled, or undesired properties are absent (Jensen et al., 2007). In addition, a colored Petri nets model is an executable model, where it allows performing simulations interactively or automatically, which enables investigating different behaviors and scenarios (Jensen et al., 2007).

The graphical notation of colored Petri nets consists of several elements: (1) places that represent state variables and they are drawn as ellipses or circles, (2) transitions representing the events/actions and they are drawn as rectangular boxes, (3) directed arcs that connect places to transitions or vise-versa, and (4) textual inscriptions next to places, transitions, and arcs to add more details to models. Moreover, tokens are used in colored Petri nets model to mark places, where each token has a data value or token color attached to it (Jensen and Kristensen, 2009; Jensen et al., 2007).

2.3. Reachability analysis

Reachability analysis is a way of testing concurrent software systems, where it helps in systematically exercising all the behaviors (i.e. nondeterministic and deterministic) of that system (Carver and Lei, 2004).

Reachability analysis provides solutions that allow model checking and verification (Karaçali and Tai, 2000). Thus, reachability analysis requires software executions to be modeled (Carver and Lei, 2004) to derive the reachability graph (Hwang et al., 1994) that enables getting all possible paths and states of the software. The reachability graph is used to verify properties of the software such as livelock, deadlock, or starvation (Hwang et al., 1994; Taylor, 1983), where this graph allows studying the absence of such issues for states that are reachable in a system (Mayr, 1984). Moreover, a reachability graph helps in analyzing whether some arbitrary state can be reached from a certain initial state (Mayr, 1984).

Several approaches were proposed in the literature to tackle reachability problems in Petri nets. Mayr (1984) proposed an algorithm that works iteratively until a specific criterion is met. However, this algorithm has some limitations, where it might get into an EXPSPACE-complete problem making it inefficient for verifying the reachability of a state (Holloway et al., 1997; Ganty and Majumdar, 2012).

In another study, Kostin (2006) proposed an algorithm that is based on linear algebra and transition invariants that are used to detect some special properties of Petri nets like the reproducibility of markings (Kostin, 2003).

Takahashi et al. (1997) applied genetic algorithms and postpone search to approximately tackle the reachability problem, where the goal of the proposed approach was to find the sequences of firing transitions efficiently, which enables reaching a target marking from an initial marking.

2.4. Genetic algorithms

Genetic algorithms are heuristic search-based methods that aim at solving variety of optimization related problems (Kramer, 2017). Genetic Algorithms follow the concept of evolutionary process (Gupta and Rohil, 2008), where the process starts with manually or randomly initialized population of solutions/individuals (Kramer, 2017). After initialization of population, a sequence of evolutionary operations take place, such as, crossover, mutation, fitness evaluation, and selection.

Crossover operation combines the genetic components of two or more solutions (Syswerda, 1993), and it is used to share knowledge between individuals in the population which allows more exploitation. An operator that takes place after crossover in most cases is the mutation operator, which helps in exploring the solution space by randomly changing an individual's solution by disturbing its chromosome values (Kramer, 2017). The strength of disturbing the solutions is called mutation rate (Kramer, 2017). After both crossover and mutation operations, the newly generated population has to be evaluated based on their ability to solve the optimization problem, where fitness function takes place to evaluate the quality of the generated solutions. Selection operation takes place afterwards, which chooses the best generated solutions to be parents in the new cycle, and that helps in achieving the optimum solution (Kramer, 2017). To terminate from the evolutionary cycle, a termination condition gets checked at the beginning of every new generation (Kramer, 2017).

Genetic algorithms has been used in several studies that propose software testing tools and solutions, such as, mutation testing (Emanuelle et al., 2006), functional testing (Last et al., 2005), and model based testing (Doungsa-ard et al., 2007). In addition, genetic algorithms can be used to generate test cases (Emanuelle et al., 2006), functional testing plans (Last et al., 2005), and test data (Doungsa-ard et al., 2007).

3. Related work

In this section we discuss related works that proposed testing games automatically, where we focused on approaches that applied genetic algorithms based solutions, or utilized Petri nets.

Several studies proposed genetic algorithms based game testing solutions. A Genetic Algorithms based approach was proposed by Chan et al. (2004). This approach was implemented to verify, detect and reach unexpected game states that were unnoticeable during the game design phase. In another study, Salge et al. (2008) presented a Genetic Algorithms agent which supports the verification process of a game by detecting gameplay bugs and flaws. In another study, Zheng et al. (2019) approached automated game testing with an agent they named *Wuji*, which explores and verifies game states space by applying Evolutionary Algorithms and multi-objective optimization. A study by Ahumada and Bergel (2020) presented a Genetic Algorithms based approach that can be used to find input command sequences that could lead to an already known faulty state in a game (i.e. the faulty state is given by the users). As stated in the study, this approach can be applied to deterministic games.

Although Genetic Algorithms based approaches in the literature were proven to be effective in automatically testing games through searching and exploring game state spaces, such approaches might have some limitations and drawbacks, where choosing the best fitness function for such agents could be time consuming (Chan et al., 2004). In addition, Chan et al. (2004) noted that applying Evolutionary Algorithms relies on off-line learning which might occasionally fail to cover all game's sequences. Moreover, one of the drawbacks of such techniques is that they might be limited and not generally applicable due

to the applicability to deterministic games only (Ahumada and Bergel, 2020), in addition to creating new fitness function for every different known bug in the tested game (Ahumada and Bergel, 2020). Hence, to avoid such drawbacks, Salge et al. (2008) integrated their approach with three AI paradigms which are Councilor AI, Swarm AI, and Reactive AI to make their approach more beneficial and support reaching more states. Zheng et al. (2019) supported their approach with Reinforcement Learning to guide and direct the agent while investigating the state space.

On the other hand, few studies were found to propose colored Petri nets based solutions to test games. Yessad et al. (2014) presented a Petri nets based framework that helps designers in modeling and automatically checking and verifying games at early design stages and before the implementation starts. In another study, Reuter et al. (2015) proposed a colored Petri nets approach that detects and finds structural errors in a scene-based game. The process of verifying the game is achieved by running reachability analysis on the colored Petri nets. However, one of the limitations of this approach is that it can test scene-based games only, since the approach was integrated with a specific game engine.

After studying and analyzing the found related work from the literature, we observe that although different approaches were proposed in the literature to automatically test games using either genetic algorithms or colored Petri nets, there is still lack of general applicability of such solutions, where these approaches were applied on specific game genres and using certain game engines, or to deterministic games only. Furthermore, we could not find from the existing studies any approach that applies collaborative work between genetic algorithms agents to test a game and find bugs. Moreover, no approaches were found to combine between genetic algorithms and colored Petri nets. Therefore, our study focuses on addressing these gaps through a co-evolutionary approach, where to the best of our knowledge, this approach is the first in the literature to find bugs using two collaborative genetic algorithms agents, and utilizes colored Petri nets to represent systems and apply reachability analysis on them.

4. Co-evolutionary based video game bug detection

We propose an automated solution to test video games using two co-evolutionary genetic algorithms agents and colored Petri nets model that represents a game's workflow. We have chosen colored Petri nets because this modeling approach supports the verification of concurrent and nondeterministic systems, and games are known to be complex nondeterministic type of systems as discussed in Section 2.1.

The approach's cycle begins from generating faulty/buggy states using the first genetic algorithms agent which is called Buggy States Generation Agent (BSGA). This agent generates buggy states by using game rules that are created using our developed game rules scheme which is discussed in Section 4.1, and by using the targeted game's colored Petri nets model. The generated states are potential invalid states that break game's rules, however, these states are not counted as valid broken states until they are proven to be reachable and available in the game under test. To do so, BSGA passes the generated buggy states to the second genetic algorithms agent, which is called Reachability Agent (RA), to perform reachability analysis and checking using the game's colored Petri nets model to check the availability of scenarios and paths that lead to the provided buggy states. Once the reachability run ends, RA informs BSGA whether a state is reachable or not, and if a state was reachable it provides the scenarios and paths to that buggy state.

The goal of our approach is to find and point at invalid states in a game based on breaking game rules fed to BSGA, and then, with RA we try to justify the ability of reaching these buggy states and reporting their reachability scenarios. The details of our approach and its building blocks, namely, the game rules scheme, BSGA, and RA are discussed in the next subsections. The process of the proposed approach (depicted in Fig. 1) consists of several steps that are described in the following points:

1. Define game rules manually using the proposed game rules scheme.
2. Feed manually created colored Petri nets models of a game and the game rules to BSGA.
3. BSGA generates multiple potential buggy states, where each state breaks at least one rule from game rules.
4. BSGA passes the generated buggy states to RA.
5. RA performs reachability checks on colored Petri nets model to find a scenario/path that leads to a given potential buggy state.
6. RA reports back to BSGA whether a buggy states was reached or not, in addition to providing the details of the scenarios if found.
7. BGSA evaluates the generated bugs based on RA's reporting, and regenerates new potential buggy states.
8. Repeat steps 4–7 until no faulty markings found, or a stopping criteria is matched.
9. Report the found bugs and their scenarios.
10. Test the found bugs' scenarios on the game manually.

4.1. Game rules scheme

In this section we describe a game rules scheme that was developed to be used within our approach. This scheme extends colored Petri nets meta-model to reflect the rules on game's colored Petri nets model to allow our testing approach to work properly and effectively.

4.1.1. Extending colored Petri nets meta-model

Game rules play a major part in our approach, where we use them to check generated states for rules violations. There are several approaches available in the literature for representing rules and constraints of software systems. One of such approaches is linear temporal logic which can be used to model rules over time (Rozier, 2011). Another example is object constraint language (OCL) which is based on set theory and first order logic (Ali et al., 2013). OCL is a widely accepted language for writing constraints on UML models (Soeken et al., 2010), where as mentioned by Correa and Werner (2004), OCL extends UML by extending the ModelElement component which represents UML modeling elements such as class diagram. However, OCL is not only able to extend ModelElement, but it can also be used to extend UML meta-model to serve the purpose of the approach utilizing it. Extending UML meta-model was discussed by Spanoudakis and Kasis (2000), where the authors have shown an approach of extending UML to add consistency rules and significance criterion using OCL.

Due to the fact that OCL is known for allowing modelers in writing rules and constraints at different levels of abstraction and serves various types of models (Ali et al., 2013), we focused on using OCL in our approach to represent game rules. However, our approach is based on colored Petri nets models to represent games, thus we extended colored Petri nets meta-model with OCL to represent game rules and to reflect those rules on colored Petri nets models of games.

Rodríguez et al. (2019) have discussed creating domain specific languages, and specifically they have shown a meta-model of colored Petri nets and presented how to extend it. The extension process goes simply by adding components, attributes, and relationships in the meta-model.

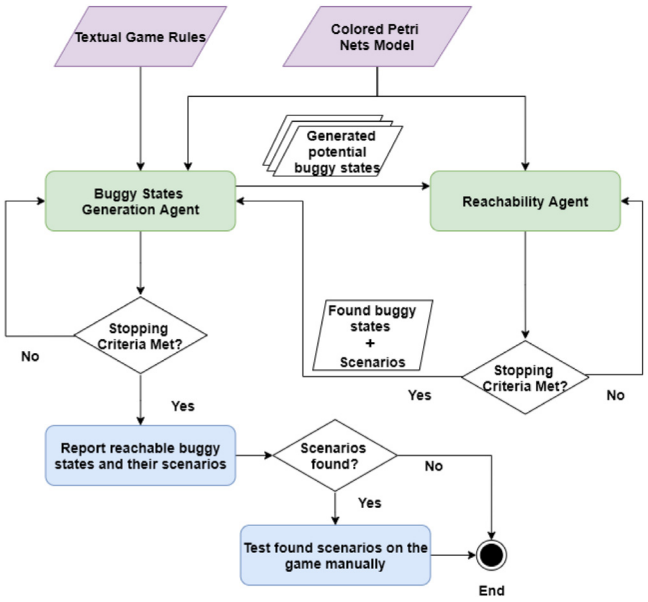

Fig. 1. Overview of the proposed approach's cycle.

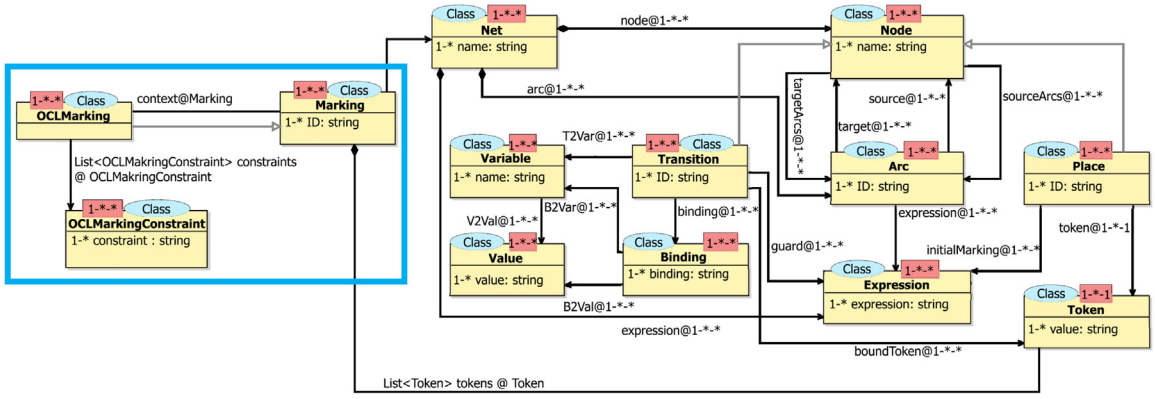

Fig. 2. Colored Petri nets extension: CPN-MCL. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

By using the meta-model of colored Petri nets shown in Rodríguez et al. (2019), we have extended colored Petri nets with OCL constraints on top of them. The proposed extension is called Colored Petri Nets Marking Constraint Language (CPN-MCL), where the purpose of this extension is to represent and support adding rules and constraints on colored Petri nets markings (states) for checking purposes. Fig. 2 shows our proposed extension of colored Petri nets to support OCL constraints, where the extensions are highlighted within a blue rectangle in the figure.

The extension is composed of three newly added components to the colored Petri nets meta-model which are, Marking, OCLMarking, and OCLMarkingConstraint. The first extension is the Marking component which is connected with Net component in the meta-model through a directed association. Moreover, Marking component has also a 'use' relationship of type composition with Token component, where a marking is a representation of an instance of a Net with specific set of tokens on each place that a net has. The second extension is OCLMarkingConstraint which represents rules and constraints. The third extension is OCLMarking component which represents mapping an instance of a Marking with the constraints, where OCLMarking is connected to Marking component through an extension/inheritance relationship and with OCLMarkingConstraint through directed association and use.

4.1.2. Representing game rules using CPN-MCL

By applying our extension CPN-MCL, we can represent rules using OCL invariants, and apply validations and checks on markings (states). The rules constructed using CPN-MCL are based on OCL invariants which can be represented through logical expressions that contain comparison operators (==, !=, >, >=, <, <=). Moreover, to connect these logical expressions in a valid way, logical operators (and, or, and not) are used. Rules are created to show constraints on markings (states) of colored Petri nets and the values of the places they contain through combining and connecting logical expressions.

To add more understanding of the rules' context, we introduced a syntax to the places involved in the rules to differentiate between their types. These types can be described as follows:

- **Active place:** a place syntax that when involved in an expression or invariant means that we care only about the place's presence within the rule (i.e. place is active and has token). To represent such places in rules, place's name is prefixed with '@' character (for example, @health).
- **Read-Only place:** a place syntax that always has one token which's value never changes. This type of places is used to be involved in guards and checks in the colored Petri nets model. This type of places is represented in the rule using

the place's name prefixed with '#' character (for example, #boardBounds).

- **Inactive place:** a place syntax that when used in an invariant, then we care about place's absence (i.e. place is inactive and empty with no tokens). The places of this type do not have a prefix in their names when used in a rule.

Moreover, we have added another syntax to enhance the understandability of the rules. This syntax is used when a place's color is vector/list/tuple/array, and it is involved in a rule with a specific index of that place. The representation of this syntax is achieved by adding a postfix to the place's name which is represented with '_' underscore character followed by the index number (for example, @playerPosition_0 represents the first index of "playerPosition" place which is of type vector). The following Listing 4.1 shows an example of a game rule represented using CPN-MCL.

Listing 4.1 (CPN-MCL Rules Example)

```
Context Game{
inv healthRule: (@health >= 0 and @health <= 6)
}
```

Later in Section 6.6.1 we discuss the steps of constructing rules using CPN-MCL.

4.2. Buggy states generation agent

This agent (BSGA) is the first in our approach, and it is meant to generate buggy states automatically of a given colored Petri nets model of a game. Generating values of variables that violate constraints or rules is a type of combinatorial and constraint satisfaction problems which are known to be NP-Hard (Salcedo-Sanz, 2009). However, one of the solutions to solve this problem is to utilize genetic algorithms, and specifically hybrid genetic algorithms (Salcedo-Sanz, 2009). Hybrid genetic algorithms introduce operations to the GA lifecycle other than the traditional ones (Orvosh and Davis, 1994). One of these operations is the repair mechanism. BSGA is based on hybrid genetic algorithms, thus the agent is composed of several operations that work together to form GA lifecycle. One of the main constraints in BSGA is that all individuals in the population during the GA lifecycle have to be feasible solutions (i.e. faulty states), meaning that each individual in the population should violate one rule at least. To maintain this constraint, a repair mechanism is applied in two stages of the GA which are Initial population generation and Mutation. In the following we describe each one of BSGA's operations.

4.2.1. Chromosome design

An individual in BSGA is composed of state places along with their values. A state consists of multiple places (variables) each of which has its own type and value, thus we represent an individual with a dictionary, where the dictionary keys are places' names, and each key has a value corresponding to the place's value. The places that an individual contains are the unique places involved in game rules. If the combination of the places' values break some of the game rules, the identifiers of the rules that were broken are assigned to an individual to mark this individual as breaking these rules. The following Fig. 3 depicts the chromosome design of an individual in BSGA.

4.2.2. Initial population generation

Our approach of generating individuals for this agent is to force feasible solutions (i.e. buggy states), where each individual violates a randomly selected rule. An individual is composed of places involved in all game rules with random values. BSGA starts by selecting a random rule, and then it generates random values for each one of the places in the individual with the goal of breaking that rule, where BSGA applies "do not care condition", which means it generates individuals that have places with values that violate some of the given game rules without caring about the whole active places of a marking/state and their values, where they do not affect the violation of a rule. BSGA forces an individual to be faulty/buggy by going into mini iterations checking and repairing, where the created state of the individual is checked if it violates the selected rule. In case the generated state was not breaking the selected rule, the state gets replaced by a new one until a feasible solution is created. Using this repair approach will boost and speed up generating buggy states, where it will ensure that each one of the created individuals breaks exactly one rule. Algorithm 1 presents the steps taken to generate initial population.

Algorithm 1: BSGA initial population generation

```
   input : popSize, rulesPlaces, JLR, placesRanges
1  population = [];
2  for i in popSize do
3  |  index = getRandomRuleIndex();
4  |  indv = createIndv(rulesPlaces, placesRanges);
5  |  while notViolated(indv, JLR[index]) do
6  |  |  indv = createIndv(rulesPlaces, placesRanges);
7  |  end
8  |  population.append(indv);
9  end
   output: population
```

4.2.3. Fitness evaluation

The process of evaluating the fitness of an individual starts by constructing the goal state information to be sent to RA for reachability check. Constructing goal state information is done through the following steps:

1. Selection of broken rules to construct the goal state. As we mentioned in the previous subsection, an individual is composed of places involved in all rules with each place having a value. Thus, if an individual breaks more than one rule, and if it was sent to RA for reachability check, RA might get distracted because of the availability of different places in the goal state that cannot be together in reality. Thus, to limit this problem, the goal state construction starts by randomly selecting one to three of the broken rules, where at least one rule is selected and at max three rules can be selected.
2. Places that are of type Read-only are neglected when constructing the goal state, as by their nature, their values do not change over colored Petri nets cycle.
3. Construction of goal state to be sent to RA, which is composed of places of the selected rules along with their values.

After constructing the goal state, the process of evaluating the fitness of an individual in BSGA involves interacting with RA, where BSGA sends the constructed goal state to RA to check if there exists a path/scenario that leads to the given goal state, and based on the reachability run, RA replies back to BSGA with the following:

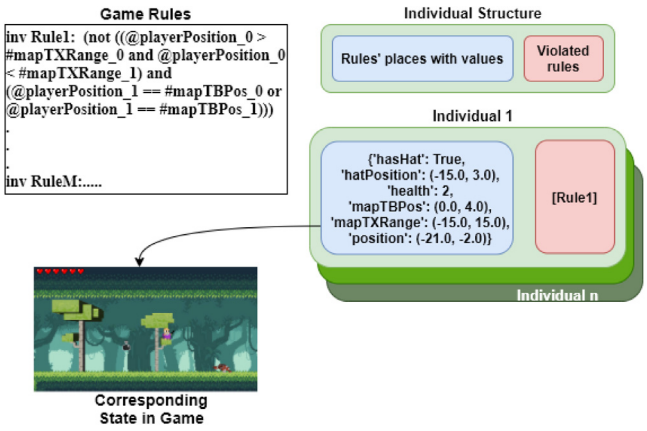

Fig. 3. BSGA chromosome design.

- **Reachability result:** which reflects whether the goal state was found or not.
- **Scenario markings:** a list of markings showing the state of the Petri nets model and change of markings from the beginning of the simulation until finding the goal state. This is used for reporting purposes when BSGA is terminated.
- **Scenario actions:** a list of sequence of transitions that were taken to be able to reach the goal state. This is used for reporting purposes when BSGA is terminated.
- **Reached states list:** a list containing all states that were reached by RA during the reachability run. These states are composed of places involved in the goal state only, where RA does not care about other states or other state variables within the same state while searching for the goal state. This is used during the mutation process, as we will describe later.
- **Reached places list:** a list containing all places involved in the goal state only and that were reached during RA reachability run. This is used during the mutation process, as we will describe later.
- **Nearest state scores:** nearest state is a state that got the best score of distance when searching the goal state, where the distance is calculated per place, which is the hamming distance between a place in the goal state and another state during the reachability run. This distance is considered if a place in the goal state is active in the RA's current marking, and it is calculated using the following formula:

Listing 4.2 (Nearest state scores)

$$formula = \frac{1}{1 + hammingDistance\,(goalPlace, RAMarkingPlace\,)}$$

The result from this formula is used to calculate the fitness of an individual of BSGA as we will show later.

When RA finishes the reachability run, the fitness of an individual is calculated. The calculation of the fitness is done through this formula:

Listing 4.3 (BSGA fitness)

$$fitness = Max_{AllBrokenRules}\,((pow\,(((sum\,(\,\mathrm{nss})/RPC) + nobBSGA)\,, 2))^{*}\,\mathrm{rw})$$

where:
nss = Nearest state scores,
rw = Rule's weight,
$nobBSGA$ = Number of overall found bugs by BSGA,
and RPC = Rule's places count

This calculation reflects the most representative broken rule of the goal state/individual. Rules weights are very important when calculating the fitness value, where they allow prioritizing rules that were not broken before over those that were broken.

While calculating the fitness function, if a goal state was constructed based on several broken rules and it was found that the goal state was partially found (i.e. the whole goal state is not reachable however a situation, which is a subset of it, that breaks one or more of the rules was found during the reachability run), then these rules with the corresponding places are considered broken. This process is important to prevent ignoring rules that were broken and found broken during the RA run. On the other hand, to make sure that the individual will benefit the population in later generations, all rules that are found to be fully reachable after finishing the RA run are considered as the main representative rules of the individual and they replace the set of broken rules of that individual.

After returning from the evaluation operation, two small processes occur before proceeding with crossover, the first is updating the rule weights, which goes as follows, if a rule was broken, its weight gets decreased by 75% and this decreased amount gets distributed and shared among rules that were not broken. The second process is related to updating reached states and reached places buckets that are shared among the population, where each individual dumps down the unique reached states and unique reached places reported by RA, these buckets are used later during mutation.

The fitness function is inherently multi-objective function. However, we opted not to formulate it as such, with relative weights, to avoid distracting the individuals from targeting their goals of creating buggy states that are more potential to be found by RA. In fact, each individual has its different set of places and their values that break different rules. Therefore, each individual has its own objective in each GA run in the cycle. However, knowledge gets shared between individuals with the help of cross-over operation, in addition to learning from the history of generated states (reached places and reached states buckets) in the mutation operation as we will discuss in the next subsections.

4.2.4. Crossover operation

The operation of crossover carried out by BSGA goes as follows:

1. Two parents are selected to produce two children.
2. Each parent picks the most representative broken rule based on the rules scores calculated during fitness evaluation.

3. Parent-1 swaps the values of places involved in the selected rule of Parent-2.
4. Parent-2 swaps the values of places involved in the selected rule of Parent-1.
5. Re-evaluate and update the broken rules of each resulting child.

4.2.5. Mutation operation

The mutation process starts by selecting a rule randomly to have its involved places' values mutated, where mutation applies repair procedures to force breaking the randomly selected rule. To select a random rule, mutation operation uses roulette wheel approach to select a rule randomly based on rules' weights. After selecting a rule to be broken, the mutation operation is composed of two steps, the first is a repair operation that searches the reached states bucket and selects a state that fulfills violating the randomly selected rule, and then replaces the rule places in the individual with those in the found state. The repair operation is guarded with a rate, if this rate was met, the repair occurs. Moreover, if the repair operation was triggered and no states were found that fulfill the selected random rule, the second process is considered.

The second process is similar to that in the initial generation, where there is a mini iteration loop that forces the individual to break the selected rule by mutating the places of the selected rule only. In this process, places' values from the reached places bucket are used, in addition, predefined places ranges that are used in the initialization operation are also used in case no places were found in the reached places bucket.

When the mutation operation finishes, the individual's list of broken rules is updated with the selected rule to make the individual focused on that rule. Algorithm 2 shows the mutation procedure.

Algorithm 2: BSGA Mutation Operation

```
input : rw, rulesPlaces, JLR, placesRanges,
        reachedStatesBucket, placesReachedBucket, rr, indv
rule = getRandomRuleIndex(rw);
brokenRules = [];
if random() <= rr then
    foundState = findState(rule, JLR[rule], rulesPlaces[rule],
     reachedStatesBucket);
    if foundState != Null then
        replacePlacesValues(foundState, indv,
         rulesPlaces[rule]);
        brokenRules = [rule];
    end
end
while empty(brokenRules) or not (rule in brokenRules) do
    updateIndv(indv, rulesPlaces[rule],
     placesReachedBucket, placesRanges);
    brokenRules = getBrokenRules(indv, JLR[rule]);
end
indv.brokenRules = [rule]
output: indv
```

4.3. Reachability agent

This agent utilizes genetic algorithms to find a sequence of firing transitions to reach a specified goal state using colored Petri nets simulations with a predefined initial state. RA communicates with BSGA to try to reach buggy states and report the scenarios if found. This agent consists of several operations that are discussed in the sequel.

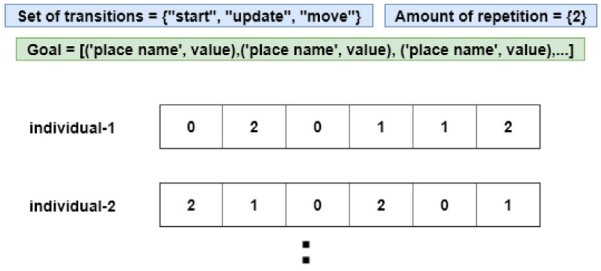

Fig. 4. RA chromosome design.

4.3.1. Chromosome design and initial population generation

The created population consists of multiple individuals, where each individual is composed out of vector of integers that represent transitions' identifiers/indices according to the colored Petri nets model of the game under test. This vector is randomly generated where each element x in the vector of an individual is repeated n times based on a parameter we call *RAMultiplier*, this repetition is done according to Parikh vector theory, where the vector of an individual shows the repetition of each transition. In our study, we give all transitions in an individual the same amount of repetition. This vector represents sequence of transitions that are placed randomly, thus these sequences might be invalid at the beginning of RA cycle, however they get organized through applying postpone search as we describe in the fitness evaluation. The following Fig. 4 depicts an example of initialization.

4.3.2. Fitness evaluation

The fitness function algorithm was inspired by the forward postpone search algorithm (Takahashi et al., 1997). Our algorithm is composed of the following steps, which is also depicted in Fig. 5:

1. Given a vector of integers (i.e. transitions indices) and an initial marking m, firing pointer p is initially placed on the first index of the vector as a starting point, where p equals to 0. In addition to that, another pointer called nonFiring pointer q is initialized as 0.
2. If the transition t pointed by p satisfies (canFire) condition at a marking m, then update the marking m after firing the transition.
   (a) Add 1 to p, which means pointing at the next transition in the vector.
   (b) Check if the goal state was found, if yes, stop the search and go to step 6.
   (c) If the goal state was not found, go to step 4.
3. Otherwise, if transition t was not fired, move the pointed transition t (which could not fire) to the end of the individual's vector, shift each transition after p to a lefthand index respectively, and add 1 to q.
4. If (p+q) equals to the length of the vector, stop this procedure and calculate the fitness of the individual. Otherwise go to step 2.
5. If (p+q) equals to the length of the vector, and the postpone search stopped, check if the current marking activates any transition, if no active transitions were found, report deadlock and the scenario/sequence of actions that led to it.
6. If goal state was found, stop the procedure and calculate the fitness.

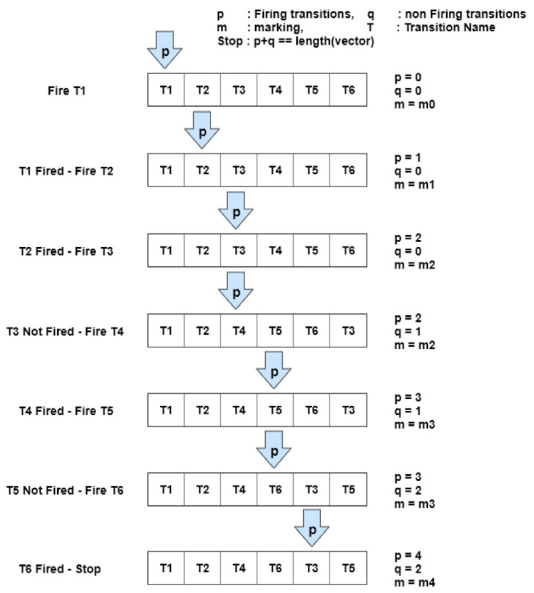

Fig. 5. RA postpone search example.

The fitness function of this agent is calculated with the following equation:

Listing 4.4 (RA fitness)

```
fitness = (p * (1 / (1 + nsd)))
```

where,

p = number of fired transitions in sequence,

and nsd = Nearest State Hamming Distance between a state in the reachability run and the goal state, which is calculated using the following equation:

Listing 4.5 (Nearest state Distance)

```
nsd = sum(hammingDistance(sp, gp))
```

where,

sp = state place in reachability run, and gp = goal place in the goal state.

The fitness function reflects the importance of distance and its relation with the sequence of transitions.

4.3.3. Crossover operation

In this step, children are generated by applying subsequence exchange crossover algorithm to the chosen pairs of individuals (parents). The subsequence exchange crossover was inspired by Takahashi et al. (1997), which is a natural extension of subtour exchange crossover, that allows exchanging a subsequence of a parent for a subsequence of another parent, where the number of transitions in one of the subsequences is similar to the other, and the number of replications of a transition in one subsequence is equal to that in the other one by exchanging permutations of subsequences from both parents. Thus, we can maintain the number of transitions and their occurrences in the vector. The subsequences are generated from the fired part of the parent, which is from index 0 to p (last fired index). The subsequences are created using an interval of lengths provided as parameter to the agent, where the first length in the interval is the smallest possible subsequence length, and the process of searching for a length that results into a valid subsequence in both parents goes gradually through the interval of lengths until a subsequence is found or all lengths are checked from the passed interval. If no subsequences were found, the new children will be equal to the parents without any changes. This type of crossover is used for inheriting characteristics from parents to children, where learnt successful subsequences are kept through generations. The following Fig. 6 shows an illustration of the crossover operation.

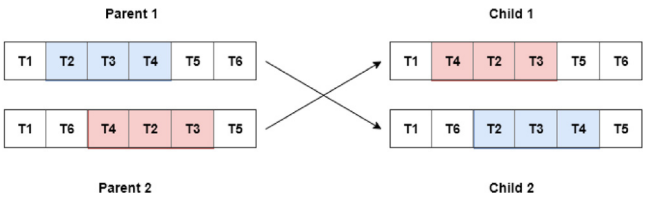

Fig. 6. RA subsequence exchange crossover example.

4.3.4. Mutation operation

The mutation operation consists of two steps:

1. Selecting ten random transitions from the individual.
2. Swapping the indices of the selected transitions to form a new individual (sequence of transitions).

5. Experiments setup and design

To be able to evaluate the approach, we have implemented its components (BSGA and RA) using Python 3.8. Moreover, several libraries were integrated and used as part of the implementation, such as ZINC (Pommereau, 2018) to create and simulate colored Petri nets, Deap (Fortin et al., 2012) which provides a skeleton implementation that makes creating GA solutions easier and faster to allow rapid prototyping of evolutionary algorithms, Concurrent which is a python library that allows running and creating multiprocessing and concurrent executions, and JsonLogic[1] which is an open-source json based data checking library, which we utilized within the rule checker module that was used to check the feasibility of the generated buggy states.

We implemented our approach to be executing individuals' fitness evaluations in parallel to speed up this operation, as it is a time consuming process, where parallel buggy states in a BSGA population get evaluated by several parallel RA individuals. Fig. 7 shows an illustration of applying parallelism in our implementation.

We applied our approach to different systems which are described in the sequel.

- **Platformer game:** This game is an open-source 2D platformer game[2]. This game is challenging from two aspects, first, the parallel behaviors of enemies and player, and second, the placement of enemies and collectibles in randomly selected spawn points. The colored Petri nets model of this game consists of 110 places and 58 transitions. Moreover, this game has 25 rules.
- **Rogue-like game:** One of the very popular Unity[3] full games that are freely available for developers[4], which is a 2D procedurally generated turn-based game with an 8 × 8 grid including start and exit points. This game was considered in our experiments due to the non deterministic behavior it has, where levels are procedurally generated, which makes

[1] JsonLogic https://jsonlogic.com/.

[2] Platformer game https://github.com/MitchellSturba/The-Magic-Hat.

[3] Unity Technologies https://unity.com/.

[4] Unity's roguelike game https://assetstore.unity.com/packages/essentials/tutorial-projects/2d-roguelike-29825.

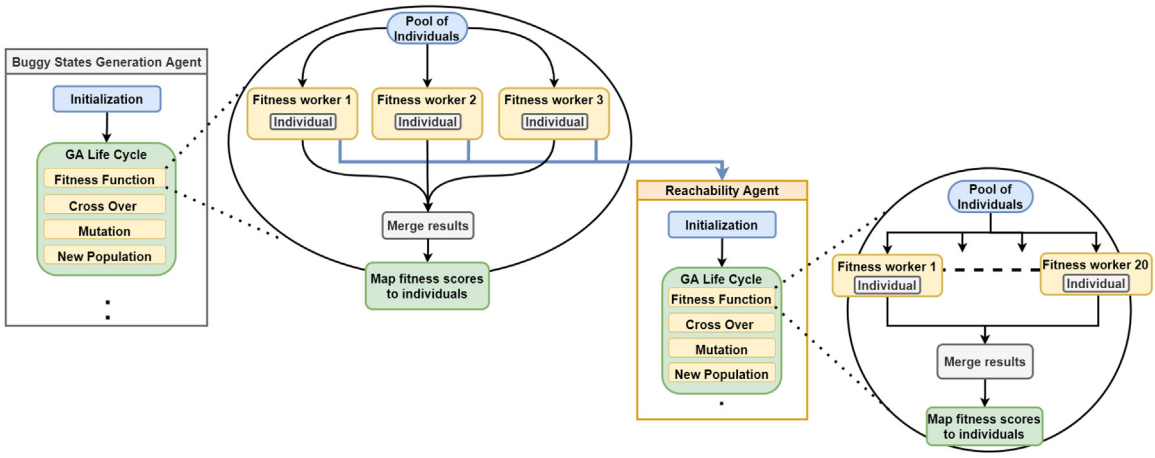

Fig. 7. Multiprocessing implementation illustration.

it challenging for RA to find and reach bugs since levels are changing randomly. It is also challenging for BSGA to generate feasible buggy states that can be reached. The colored Petri nets model of this game consists of 30 places and 19 transitions. Moreover, this game has 6 rules.

- **Infinite runner game:** This game is an open-source 2D infinite runner game[5]. The goal of the game is to reach the highest possible score by going through an infinite level of regenerated platforms. The colored Petri nets model of this game consists of 37 places and 21 transitions. Moreover, this game consists of 4 rules.
- **Software System:** A shutdown system for a Korean nuclear power plant (Cho et al., 1996). This system is considered nondeterministic due to the parallel executions and randomization it contains. This system's representation has been used in the literature as a case study for reachability analysis of colored Petri nets and finding hazard situations (Cho et al., 1996; Bouali et al., 2009). The colored Petri nets model of the system consists of 6 places and 4 transitions, in addition, the system consists of two rules which were discussed by Cho et al. (1996). This system was selected to further test and push the applicability of the proposed approach. Also, to verify whether the proposed approach can be applied to systems other than games.

Before diving into the main experiments that were conducted to test our approach's effectiveness in finding bugs, we conducted a set of experiments to select suitable parameters to be used for RA, we designed and ran 16 experiments with 10 runs each, to test 16 parameters settings that are shown in Table 1, where we investigated four parameters that can influence and affect the performance of RA. To run the experiments we used a faulty state from platformer game. The faulty state used in the experiments is considered complex, where it consists of six places.

Listing 5.1 shows the state that was used during the RA parameters selection experiments. This state reflects a bug that occurs when the player is continuously colliding with the enemy and player's health is not getting decremented.

Listing 5.1 (RA Experiments State)

State : ('enemy0Active', True), ('enemy0Position', (-3.0,1.0)), ('collisionEnterEnemy', 2), ('healthWithEnemy', 5), ('enemyChecked', 4), ('playerPosition', (-3.0,1.0))

[5] Infinite runner game https://github.com/alexFiorenza/SpaceMan-Game.

Table 1
Experiments to select suitable RA parameters.

Mutation rate	Crossover rate	Generations	Population	Setting ID
0.1	0.3	50	20	1
			40	2
		70	20	3
			40	4
	0.5	50	20	5
			40	6
		70	20	7
			40	8
0.3	0.3	50	20	9
			40	10
		70	20	11
			40	12
	0.5	50	20	13
			40	14
		70	20	15
			40	16

To make a decision on which parameter setting shall be used in RA, we applied mean with 95% confidence interval analysis on the results of the experiment based on the percentage of the average found bugs through the whole 10 runs. The results of the experiments are shown in Fig. 8. As it can be clearly seen in the figure, the parameter setting 16 was the best among all other parameters used during the experiments, where its mean is 0.8 (i.e. it can find the bug 80% of the time) and its confidence interval is small compared to others. Most of the other parameter settings had a mean that is below 0.5. Hence, based on the results we selected parameter setting 16 to be used for RA during the main experiments discussed later in Section 6.

6. Experiments results and discussion

In this section, we present the conducted experiments. Each experiment is applied to a single game/system by using a testing approach. Moreover, for RA, the parameters we selected were based on the results shown in the parameter selection experiments, which were discussed in Section 5. In addition, Roulette wheel was selected as selection algorithm for both BSGA and RA in all experiments. Moreover, all rules in the experiments were created using our game rules scheme CPN-MCL. Furthermore, in each experiment a set of parameters were defined and used for BSGA, which are presented in Table 2.

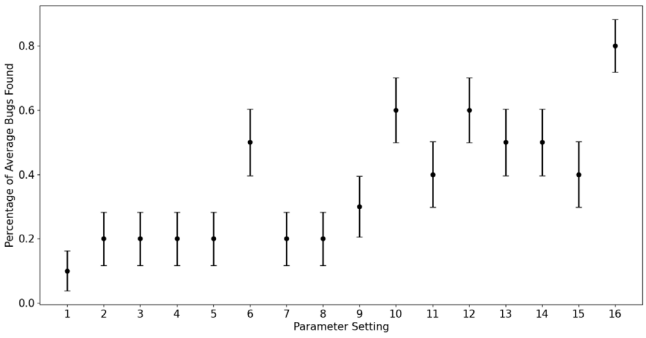

Fig. 8. Confidence interval analysis of the reachability parameter setting experiments based on the percentage of average bugs found.

Table 2
BSGA's parameters in all experiments.

BSGA parameter/Exp	A.1	A.2	B.1	B.2	C	D	E
Number of generations	30	30	50	50	30	30	30
Population size	25	25	25	25	25	50	15
Crossover rate	0.5	0.5	0.5	0.5	0.5	–	0.5
Mutation rate	0.1	0.4	0.1	0.4	0.4	–	0.4
Repair rate	0.3	0.3	0.3	0.3	0.3	–	0.3

6.1. Experiment A — platformer game: BSGA and RA

Two sub-experiments were conducted within this experiment and both were run ten times, where each sub-experiment has different mutation rate for BSGA. Moreover, we set the RAMultiplier in this experiment to be 30.

6.1.1. Experiment A.1

The first sub-experiment was conducted to show how effective our approach is in generating and finding buggy states of the game under test, where the focus of this sub-experiment is to study the effects of low mutation over BSGA and its ability of generating reachable buggy states that violate different rules.

The most important thing when using our approach is to break rules by generating buggy states and reaching those generated states, which proves the availability of such bugs in the system under test. Fig. 10.A.1 shows the fully broken rules and the number of bugs found in each rule during each of the ten runs of this sub-experiment. The analysis of this figure shows that different types of bugs were found during different runs, where our approach (BSGA-RA) was able to fully break 15 rules out of 25.

Fig. 11 shows the maximum fitness of BSGA during its lifetime throughout the ten runs. In addition, the figure depicts the average of the maximum fitness among the ten runs. As it appears in the figure, BSGA was able to gradually evolve, where the fitness increases over the generations, which indicates the ability of generating buggy states that are reachable by RA. Although the fitness scores were close in all runs, some runs had lower fitness compared to others, such as **Run 0**, where in such runs less variations of broken rules were found, and this can be seen in Fig. 10.

As we discussed in the definition of BSGA in Section 4.2, BSGA utilizes mini iterations to repair the generated state and to ensure that it is being faulty, where having these mini iterations of repairing speeds up the generation of buggy states and boosts the quality of the generated states while maintaining their feasibility of being buggy.

Fig. 12 depicts a plot of the maximum, minimum, and average number of mini iterations that occurred per generation during BSGA's lifetime, where each run is depicted. As the figure shows, a good proportion of the maximum number of mini iterations per generations are outliers, especially in runs **Run 1** and **Run 3**, where in these two runs the agent was able to find more variations of buggy states that break different rules. In addition, the number of outliers of these two runs indicates that BSGA was targeting new rules that were not broken before. For example, in the 28th generation of the 4th run (i.e. **Run 3**), the maximum number of mini iterations is about 2920. Nonetheless, although there are some outliers with huge number of iterations, based on the analysis of Fig. 12, the overall average number of mini iterations per generation is considerably sufficient and non-exhaustive, where for instance in **Run 3** the total average number of mini iterations required during BSGA's lifetime is about 900 (i.e. 30 mini iterations in average per generation in 30 generations), which indicates the effort BSGA requires to generate feasible buggy states.

6.1.2. Experiment A.2

In this run of Experiment A, we study the effects of increasing the mutation rate to 0.4 in finding more bugs and breaking more variations of rules, especially those that are difficult to break or were not broken during Experiment A.1.

Fig. 10 shows the number of found bugs per rule during Experiment A.2. By analyzing the figure we can see that BSGA was able to find and fully break rules in most of the runs. In this experiment more rules were broken compared to Experiment A.1, where 16 different rules out of 25 were violated by the approach in this experiment. Some of these rules were pretty complex, as they include multiple places with different colors and values, and they were found by several runs such as rules 7 and 20 which were found in less number of times in Experiment A.1. Moreover, **Rule 16**, which is considered complex, was broken in this experiment three times, whereas in Experiment A.1, it was not broken in any run. From both sub-experiments of Experiment A we can notice that there are some rules that were broken more often than others, such as **Rule 2**, **Rule 5**, and **Rule 11**, which indicates that these rules can be violated easily, and violating them happens quite often in the game as some of them are composed of fewer number of places.

In another side of the analysis, Fig. 11 shows that increasing the mutation rate affected the performance of BSGA positively, where the quality of the generated buggy states that are reachable got improved over time. Compared to the fitness of the approach in Experiment A.1, the maximum value of the average fitness through all runs was about 700, whereas in Experiment A.2 the maximum average fitness of all runs was about 1800.

Fig. 11 also shows some spikes and decrements in the fitness score in some generations during different runs, and this is due to two reasons: (1) falling into local minima and escaping it by finding buggy states that are reachable of rules that are already

broken, which resulted into changing the priorities of rules by updating rules' weights. (2) shifting the focus of BSGA from generating buggy states that potentially break rules which were already fully broken (i.e. previous states that break such rules were found and reached) to rules that were not broken before by generating potential buggy states and are yet to be discovered by RA. This figure in fact explains the fast convergence during the first ten generations in some of the runs, such as **Run 1** and **Run 6**. In addition, it provides an insight of the reasons that caused the spikes and low curves in the fitness scores at late generations, where the first was caused by finding high number of buggy states that violate variations of rules, and the later is because of shifting the focus from generating states that violate already broken rules to focus on rules that were not yet broken.

Fig. 12 shows the analysis of mini iterations during all ten runs of Experiment A.2. Comparing the results depicted in this figure with the results studied in Experiment A.1, we can see that in Experiment A.2 there are more outliers, and this in fact is because of increasing the mutation rate in Experiment A.2, which caused BSGA to perform more repair processes during this experiment than in Experiment A.1. Nevertheless, introducing more outliers through generations in Experiment A.2 did not affect the overall total average number of mini iterations in all runs.

6.1.3. Discussion of Experiment A

From the results and discussion above we can see that increasing the mutation rate of BSGA improved the performance of our approach, where it allowed BSGA to generate more buggy states that are reachable by RA and which break more variations of rules compared to lower mutation rate.

Moreover, when we went back to the logs of the experiment's runs, we noticed that most of the found bugs were breaking at maximum one rule at a time when the mutation rate is low (except for a case that was found in **Run 7**, where a state breaks two rules, 2 and 3, which are considered close and pretty simple and easy to find and violate). However, increasing the mutation rate allowed BSGA to generate states in different occurrences in several runs that violate multiple rules at the same time and which are reachable and found by RA. In addition, increasing the mutation rate allowed BSGA to break more sophisticated rules that include different places, where in Experiment A.2 more rules were broken compared to Experiment A.1.

On the other hand, studying the number of mini iterations per generation in both sub-experiments, we observed that there is a little increase in the average number of mini iterations when we increased the mutation rate. However, this growth of the average number of mini iterations did not affect the performance of the approach.

6.2. Experiment B — Rogue-like game: BSGA and RA

This experiment consists of two sub-experiments similar to those done in Experiment A, where different mutation rates are used. Both sub-experiments were repeated ten times. In addition, we set the RAMultiplier in this experiment to be 64.

6.2.1. Experiment B.1

In this sub-experiment of Experiment B, we study the effects of low mutation rate (0.1) in finding bugs and breaking rules.

Only 2 rules out of 6 were broken in all runs of this experiment. However, our approach with low mutation rate was not able to generate and find states that break both rules in most of the runs, where the majority of the runs either broke one rule or no broken rules were detected, and this is due to the nature of procedural generation in the game, in addition to the low mutation rate which limited the exploration of new states that could potentially be reachable. Nonetheless, only one run (**Run 7**) was found to be able to generate and find buggy states that violate both rules (i.e. **Rule 0** and **Rule 2**). Fig. 10 shows that in some runs (i.e. **Run 0** and **Run 4**) our agent was not able to find any buggy state.

In another note, RA was able to find several deadlocks during the whole 50 generations of BSGA in all runs. The root cause of the reported deadlocks was found to be the same for all of them.

We elaborate more on the found bugs and deadlocks later in the discussion section of Experiment B.

Fig. 11 shows that the approach performed poorly, where the average fitness score of all runs was about 4. This indicates that the generated buggy states were not reachable and they were giving similar fitness all the time in most of the runs. Perhaps this was due to the complexity of randomness and non deterministic behavior of procedural generation in the game which happens in each level or every time the game restarts, and this resulted into making RA unable to reach valid buggy states that were supposed to be reachable. In addition, low mutation might have affected the results, where the process of exploring more feasible solutions was limited by the low mutation rate.

Due to the small state space and small number of places involved in the rules, the average number of mini iterations per generation was small, which makes the total average number of mini iterations' in all runs narrow. Moreover, as depicted in the plot in Fig. 12, the amount of mini iterations in this sub-experiment was also low due to the small mutation rate, which limits the number of mutations per generation, thus the number of iterations decreases.

6.2.2. Experiment B.2

When we increased the mutation rate to 0.4 in this sub-experiment, BSGA was able to generate buggy states that are reachable by RA, where in this sub-experiment the approach was able to find generated buggy states that break rules 0 and 2 in most experiment's runs, which gives an advantage of our approach with high mutation over that with low mutation in Experiment B.1.

In another note, RA reported several deadlocks in this sub-experiment as well, which have the same root cause of those reported in Experiment B.1.

As Fig. 11 shows, increasing the mutation rate boosted the performance and allowed the agent to achieve higher fitness score by generating buggy states that are reachable. Also, from the figure we can deduce that the changes that happened to the fitness score in different runs were due to finding bugs, changing rules' weights, and shifting the interest to other rules that are not yet broken.

Because of the high mutation rate, as in Experiment A.2, the average number of mini iterations per generation raised in most of the runs. However, the amount of increment in the number of mini iterations was not that high considering that the number of generations was set to 50, and the mutation rate of this sub-experiment was high, such in **Run 2**, where the overall highest number of mini iterations was about 5400, however, the overall average number of mini iterations in that run was about 72 per generation. Mini iterations analysis is depicted in Fig. 12.

6.2.3. Discussion of Experiment B

Increasing the mutation rate to 0.4 made our approach perform better in terms of generating buggy states that break different rules, despite the non deterministic nature of the game under test.

In another note, several deadlocks were found during this experiment, and after investigating the root causes of the deadlocks from the reported sequences of transitions, it turns out that these

deadlocks happened after the game went into a state where the player wins and loses at the same time, and this state actually breaks **Rule 2** of the game. Thus, if this bug was fixed in the game, deadlocks might be prevented.

During Experiment B.1, rules were not broken together in one run, where in the majority of the runs at maximum one rule was broken, which provides an insight that lowering mutation rate of the generation agent BSGA limits its abilities of targeting different rules with more feasible buggy states. However, increasing the mutation rate has to be limited to some extent, where making the mutation 100% will result into breaking the genetic algorithms nature of self-learning, adapting, and growing towards goals. The results of Experiment B.2 reflect that increasing the mutation rate to 0.4 has given the agent more ability to explore new states since in most of the runs.

6.3. Experiment C — Infinite runner game: BSGA and RA

This experiment was conducted with mutation rate set to 0.4 for BSGA. In addition, we set the RAMultiplier in this experiment to be 30. The experiment was repeated ten times.

In this experiment, all 4 rules were broken in all runs and within few generations, where BSGA was able to quit the GA lifecycle due to the stopping criteria which is breaking all rules. As Fig. 10 shows, some rules were broken more frequently than others, such as **Rule 1**, which indicates how often this bug occurs within the game. On the other hand, **Rule 0** was not repeatedly broken as other rules, though it was broken in all the runs.

Fig. 11 shows the fitness score of BSGA for all runs in this experiment. As depicted in the figure, our approach was able to fully break rules and find buggy states in early generations. On the other hand, some runs required more generations to find states that are reachable and violate all the rules. This perhaps reflects that either the game was totally buggy which made the rules easily broken and the faulty states were easy to reach, or that the state space is simple to search through and find the generated buggy states.

Fig. 12 shows the analysis of mini iterations. Due to the small state space, small number of places involved in rules, and finding all bugs in early generations, the average number of mini iterations per generation was very small in all runs, although the mutation rate was set to 0.4.

6.4. Experiment D — Platformer game: Random and RA

In the previous experiments we studied the proposed approach and the effects of high and low mutations. The results have shown that BSGA and RA were able to work collaboratively to generate buggy states by violating rules and finding those buggy states. However, in this experiment, we focus on studying and comparing the performance of random buggy states generator and its effectiveness in finding different bugs breaking various rules. Thus, in this experiment we used a random states generator in place of BSGA. The random generator is not based on genetic algorithms, but rather it was built to generate random values for places in the generated states using brute force via mini iterations, where a random rule is selected and the random generator generates values of the involved places until feasible solution is found, hence to ensure that the generated state breaks at least one rule. In every new generation, the whole population gets replaced by newly generated one. The reason behind forcing the random generator to generate feasible solutions is to make the comparison fair between it and BSGA, and also the goal of our approach is to find and reach feasible buggy states. On the other hand, as rules' prioritization was part of BSGA, the random generator does not prioritize rules. Thus, the fitness equation is similar to BSGA's, however rules' weights are always substituted with 1 to neglect their influence on the fitness score of the random agent's individuals.

To give the random generator more advantage and to make the experiment challenging, we increased the population size for the random generator in this experiment to be the double of number of individuals in Experiment A (50 individuals). Furthermore, we kept the maximum number of generations to 30. In addition, as the random generator is not based on GA, there are no selection, crossover, nor mutation operations.

Later in this experiment we compare the results obtained from this experiment with the results in Experiment A, where we applied this approach (random generation — RA) on the same platformer game. In addition, we supplied the same places ranges used in Experiment A. Moreover, in another sub-experiment, we use larger ranges of places' values to check the effects of using larger ranges on finding more bugs and breaking different rules. Thus, this experiment is composed of two sub-experiments, and each sub-experiment was repeated ten times.

6.4.1. Experiment D.1

This sub-experiment uses random generator supplied with small places' ranges used in Experiment A.

Looking at the details of the broken rules in this sub-experiment, Fig. 10 shows that the number of broken rules was only 4 rules out of 25. In fact, these rules are considered simple due to the number of involved places, and because of violating them repeatedly during multiple runs. Although the number of found bugs is high, but not breaking more variations of rules indicates poor performance by this agent.

Fig. 11 shows the fitness of the random generator. Although the fitness was increasing gradually over generations in all runs, the average fitness of all runs' scores was close to that in Experiment A.2, and it is note worthy that the fitness score in this experiment (Experiment D.1) is not affected by rules' weights, which reflects the poor performance of this agent where its score was still low even when rules' weights are not considered. This is due to the fact that similar buggy states that violate similar rules were found during most of the generations in different runs. In addition, despite the fact that this agent has more individuals (double) than in Experiment A, this agent was not able to generate and find different buggy states that break various rules. Having more individuals allowed the agent to generate more buggy states with different values, however, some of these values were repeated since the agent in this experiment was supplied with small places' ranges, which allowed the random agent to generate the same buggy states multiple times.

6.4.2. Experiment D.2

This sub-experiment uses random generator supplied with large places' ranges.

As Fig. 10 shows, it was surprising that increasing the ranges of values have resulted in worse results compared to those obtained from Experiment D.1, where rules were broken less times during the runs of this sub-experiment. Again, these found bugs are easily reachable and the rules of these buggy states occur more often than other rules which made them easy to find.

Although the number of found bugs is high in both sub-experiments of Experiment D, failing to break sophisticated rules and other variations of rules indicates poor performance by this random agent, even when the places' ranges are either large or small.

Fig. 11 shows the fitness of the random generator in Experiment D.2 where large places ranges were given. The performance was similar to Experiment D.1, the fitness value was increasing gradually over generations. However, comparing the average of

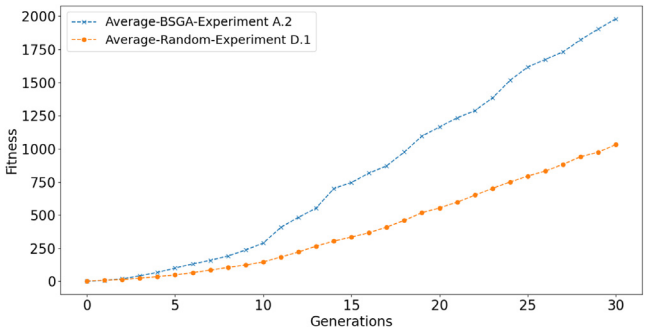

Fig. 9. Comparison between BSGA's Performance in Experiment A.2 and Random Generator's Performance in Experiment D.1.

fitness scores of all runs in this experiment with that in Experiment D.1, we can see that Experiment D.1 with smaller ranges of values performed way better. Though, both sub-experiments' performances were poor compared to our approach BSGA-RA in Experiment A.

6.4.3. Discussion of Experiment D

Based on the results of this experiment, we can see that compared to the sub-experiments of Experiment A, and although the random generator approach was given more advantage with the huge population size and also in Experiment D.2 it was given more advantage with huge places' ranges, our approach BSGA-RA was more efficient even with smaller population size. Analyzing the fitness of both approaches, we can see that BSGA-RA was able to fastly evolve in less number of generations. Moreover, BSGA-RA allows distinguishing solutions in the population where we can see that there is an obvious difference between the average score of fitness in all runs during BSGA's lifetime compared to the random agent. Fig. 9 shows an analysis that compares between the averages of BSGA during Experiment A.2 and the random generator in Experiment D.1. These two sub-experiments were selected in this analysis because of their high scores in comparison to the other sub-experiments during the experiments they belong to.

On the other hand, although the number of found bugs by the approach implemented in Experiment D is high, not being able to find variations of states that break different rules reflects bad performance by this approach. In contrast, our approach BSGA-RA in both sub-experiments of Experiment A was able to generate and reach buggy states that break different rules which vary from simple to complex.

As we mentioned, places' ranges that were provided in Experiment D.1 might have limited the random generator from generating buggy states that need more ranges to be reachable. However, the same ranges were also provided to our approach BSGA-RA in Experiment A, and the results were way better, and this in fact is because of the ability that BSGA has, which is learning and using values of places reached during the reachability runs by RA, where these values are kept in the reached places and reached states buckets (as discussed in Section 4.2) and used during the mutation operation, which allowed BSGA to use more focused ranges of values in addition to the supplied ones. Moreover, using crossover operation, knowledge gets shared among individuals, which could allow in generating states that are potentially reachable.

6.5. Experiment E — Software system: BSGA and RA

The goal of this experiment is to push our approach further by applying it to a software system rather than a game. The experiment was conducted with mutation rate set to 0.4 for BSGA. In addition, the RAMultiplier was set to 30 in this experiment. The experiment was repeated ten times.

BSGA was able to generate buggy states that are reachable and break both system's rules in most of the runs. Fig. 10 shows that not all runs were able to break all 2 rules, where runs (**Run 6** and **Run 8**) were just able to break one of the 2 rules. This is because of the high randomization of the system, in addition to the parallel executions which prevented the agent from generating buggy states that are reachable. However, in all rest of the runs (i.e. all runs except 6 and 8) the approach was able to break all the 2 rules and find hazard situations in the system.

Fig. 11 shows the fitness score of BSGA for all runs during the experiment. Our approach was able to generate buggy states that break rules in early generations, such in **Run 3**. Other runs varied in their GA cycle duration. The amount of spikes in the figure reflect the difficulty of finding the generated buggy states that violate the rules, where the obstacle that BSGA and RA faced was caused by the nondeterministic behavior of the system.

6.6. Discussion

In this section we discuss the observations of the experiments results through answering the research questions.

6.6.1. RQ1-modeling game rules

It is widely known that several artifacts are developed and produced during the game development lifecycle: Requirements, Design, and Source Code, etc. (McDonald et al., 2020). Indeed, the source code is the core of the game and it represents the implementation of game's functionality and behavior. However, game requirements and design artifacts hold the details related to game's rules, constraints, functional requirements, game mechanics, gameplay details, puzzles, concept arts, storyline, and more (Colby and Colby, 2019). Utilizing these artifacts along with colored Petri nets representation of the game's workflow is mandatory in our approach to construct game rules, where these rules are mapped to the game under test.

Game rules can be created and modeled using our proposed CPN-MCL discussed in Section 4.1.2. The rules are basically constructed using OCL invariants which are represented through logical expressions that involve places from colored Petri nets model of the game, in addition to logical operators to create compound statements.

Moreover, to have better understanding of the rules, we also introduced a syntax that helps in distinguishing between different types of places' values based on the nature of that place in the colored Petri nets model, where the aim of CPN-MCL is to reflect the game's rules on the colored Petri nets model of that game to allow testing it.

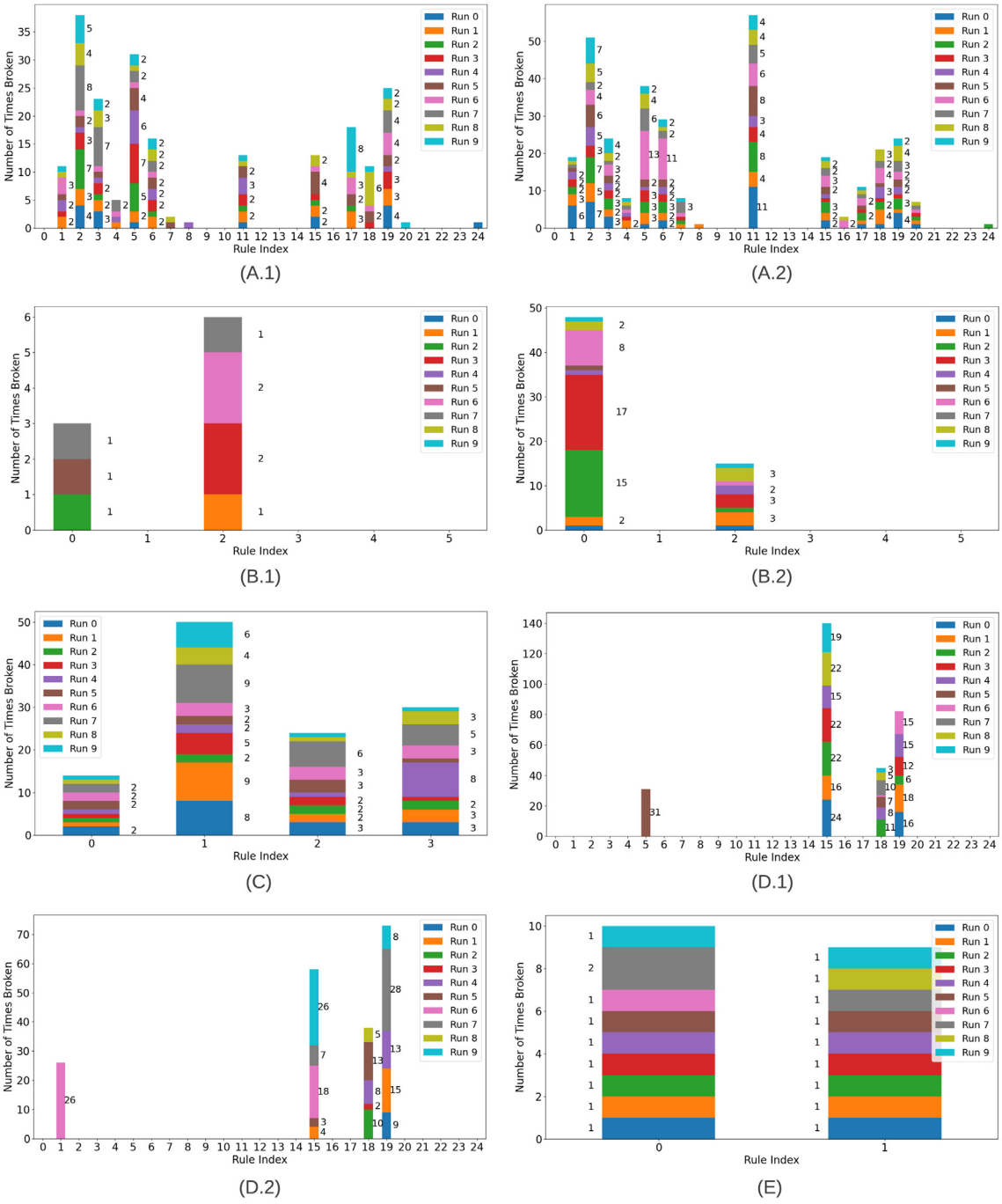

Fig. 10. Bugs found per rule during all experiments.

The following is an example that discusses the steps of rule construction starting from game requirements. This requirement is part of the platformer game.

Listing 6.1 (Requirement)

Player shall be able to navigate anywhere in the map except tiles' (x,y) points.

This requirement can be translated into a rule by following the **steps** in the sequel:

1. Extracting the state variables from the requirement and which are mandatory in the rule. These state variables are also represented as places in the game's colored Petri nets model.
2. Representing the extracted variables using the CPN-MCL's places syntax (discussed in Section 4.1.2).
3. Composing the rule through logical relations and expressions using the extracted variables. The rule shall reflect the intent of the requirement.

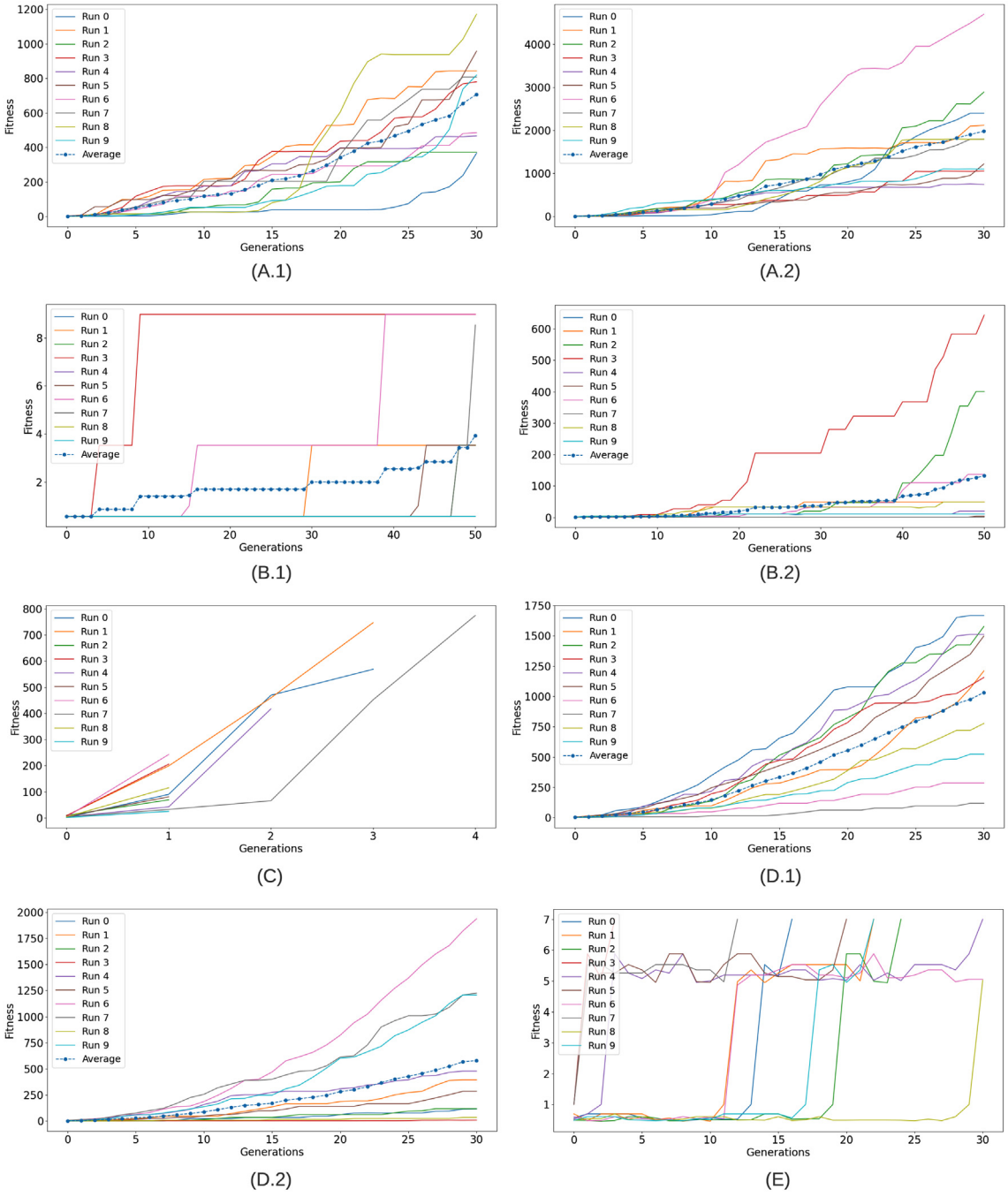

Fig. 11. Fitness performance during all experiments.

Starting from step (1), we can see that there are three state variables contributing in this requirement, which are *playerPosition*, *mapTXRange* (map *x* range), and *mapTBRange* (map top bottom range).

After extracting the variables, step (2) takes place, where variables are represented using CPN-MCL syntax. In this example, the map is predefined, hence mapTXRange and mapTBRange never change, therefore both can be considered of type **Read-Only** places. On the other hand, as the player can interact with the environment (i.e. playerPosition's value changes during game's lifetime), and the presence of player's position is important when evaluating a state with the rule, then we consider it of type **Active** place. Moreover, all of these variables can be thought of as vectors, where they represent coordinates or ranges. Therefore, the resulting representations of the variables from step (2) are, @playerPosition, #mapTXRange, and #mapTBPos.

Lastly, in step (3) the rule is composed using the extracted variables. This rule has to reflect the requirement. Based on the requirement shown in Listing 6.1, the player is allowed to navigate anywhere in the level except for the exact coordinate points of the tiles. In the game and the Petri nets representation, the tiles are located in a linear style and their positions are defined in the #mapTBPos vector, where the first index (0) represents the lowest *y* point (i.e. ground/bottom of the map) and the second index (1) represents the highest *y* point (i.e. top of the map). These tiles are spread among the level's *x* range defined in #mapTXRange.

Based on the previous steps, the requirement shown in Listing 6.1 can be represented as the following rule in Listing 6.2:

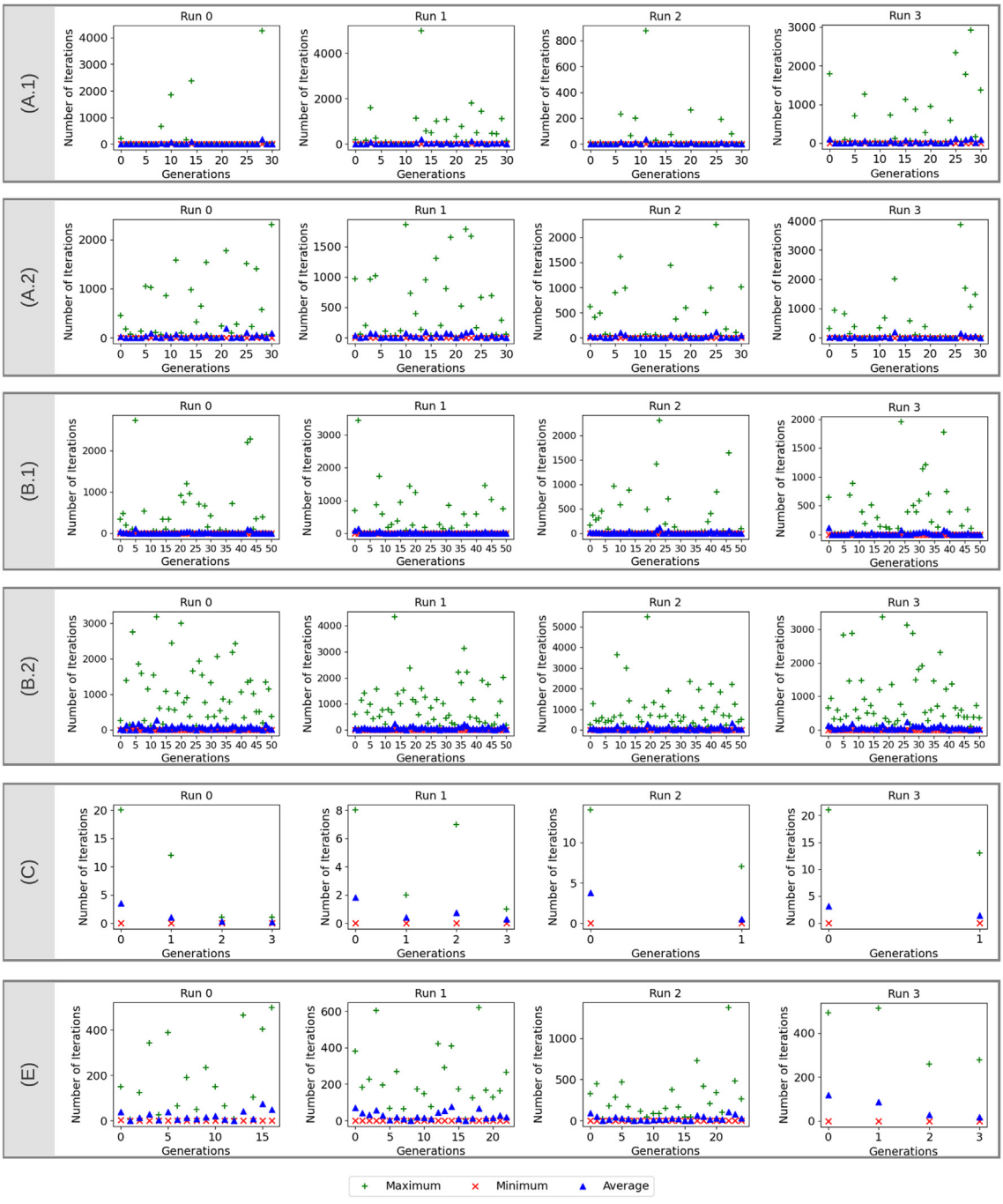

Fig. 12. Analysis of mini iterations during the first 4 runs in each experiment of BSGA.

Listing 6.2 (Extracted Rule)

```
inv rule: not ((@playerPosition_0 > #mapTXRange_0 and @play-
erPosition_0 < #mapTXRange_1) and (@playerPosition_1 ==
#mapTBPos_0 or @playerPosition_1 == #mapTBPos_1))
```

Constructing games' rules with the proposed CPN-MCL scheme allowed our BSGA in generating buggy states that violate those rules. We have shared the rules that were created and used during the experiments of this study publicly on GitHub.[6]

[6] Available via https://github.com/MrAghyad/BSGA-RA.

6.6.2. RQ2-characteristics of state variables to allow finding bugs

In Section 4.1.2 we proposed a place syntax to represent places within rules, where we described the three place syntax types. Places that we care about mutating their values are active places and inactive places. The nature of the active place implies having token with value, thus we can mutate the value of this place, whereas an inactive place implies not having tokens which we can mutate to add tokens with values or remove tokens. On the other hand, we also defined previously a third type which is read-only places, however these places' tokens never change during colored Petri nets simulation, thus we do not care about mutating them.

The relationships between places differ, some places are used to guard and bound the values of other places, such as read-only places, for example in the platformer game, rule 15 implies that player's position should be anywhere other than ground and ceiling tiles, which is guarded by map bounds (#mapTXRange and #mapTBPos), mutating the values of player's position to break this rule allowed our approach in finding such bug in the game. Some other places depend on other places in previous transitions, where firing a transition changes values of output places based on input places, such as in the platformer game, one of the discovered bugs was caused by resetting player's jump points after he collides with the ceiling, which gave him the ability of jumping infinitely as long as he is near the ceiling, this happened after transition of checking tile collision is fired.

In our approach, we mutate the values of places involved in a rule to violate that rule. Mutating these values happens in random manner. However, sometimes we have to observe the relationships between places within a state to be able to fully break a rule (i.e. generate buggy state and reach it), thus our approach utilizes reached states bucket during mutation to allow BSGA to generate buggy states that maintain the relationships between places within a state. Moreover, RA can observe if a path is going to lead to a buggy state, where RA's fitness function favors paths that go through states that involve places of interest (i.e. places involved in goal state) by using distance calculations as described in Section 4.3.

6.6.3. RQ3-revealing game bugs

As show in the experiments in Section 6, our approach was able to break game's rules, generate buggy states, and find bugs in games. Our approach is composed of two genetic algorithms agents that work collaboratively to reveal bugs in games, where the first is BSGA which generates potential buggy states, and the other is RA that performs reachability check using colored Petri nets model of the game under test trying to reach the buggy state. Both agents work together where BSGA sends generated buggy states to RA so it can search through simulations of colored Petri nets model and check the existence of the provided buggy state. RA then replies back to BSGA on whether the buggy state was found or not. BSGA takes the information reported by RA and tries to generate more feasible buggy states, in addition if a buggy state was found by RA, BSGA will shift its interest to violating rules that were not violated before. This cycle continues until either all rules are broken, or the cycle ends by stopping criteria. When the generation cycle ends, the found bugs are reported along with the sequences of actions and transitions that lead to them. These sequences of actions are then manually applied on the game, and were traceable to the code to reveal the erroneous code part. However, sometimes in general, the tester might face some cases where it is difficult to reproduce the bug manually on the game due to uncontrolled and non-deterministic actions taken by other entities in the systems. In such cases, bugs can be revealed on the code level directly by checking the sequences of transitions/actions with the code. We described our approach in detail in Section 4.

Although our approach is able to find bugs by violating rules and reaching generated buggy states, but some rules might not be fully breakable, where the generated buggy states that violate such rules are not reachable where these states do not occur in the game under test. This was observed from the experiments where not all rules were fully broken by reachable buggy states.

6.6.4. RQ4-types of revealed game bugs

Based on the results of the experiments we observe that our approach can reveal the following bug types:

- **Invalid States:** This type of bugs was found in several occurrences. In fact, any state found that breaks a rule can be generally considered an invalid state. For instance, in the platformer game, one of the found invalid states is that the player dies when an enemy falls into lava, which is also another invalid state where an enemy should stay on the ground within the level bounds.
- **Conflicting Overlapping Goal States:** A goal state can be thought of as the set of conditions that define a partial description of a state. This partial description is called a situation, where state variables satisfy a set of conditions. Conflicting overlapping goal states are two goal states that occur at the same time because their state variables satisfied a situation and its conditions. However, these goal states should not happen at the same time. To be able to find such bug, we introduced two rules that describe the goal state (e.g. win and game over), in addition, a third rule was added to prevent the conflict of both happening at the same time. In the rouge-like game our approach was able to find a state where the player was winning and loosing at the same time. This bug happens when the player has one food point left and he is one step to exit the level. If he goes to exit, he looses a point (i.e. food points will become zero) which fires game over, but at the same time he reaches the exit which activates the next level.
- **Deadlocks:** Our approach was also able to find deadlocks and report the sequences of actions that lead to them. Detecting deadlocks is done by RA, where during searching for the provided buggy state, if a state (marking) was found to not activate any transition in the net, then a deadlock gets reported. Deadlocks might not be associated with a game rule, thus we embedded finding them in RA. In Section 6.2 we discussed deadlocks that occurred during Experiment B and their causes.
- **Level Design Issues:** From the experiments with platformer game, our approach was able to find a bug related to wrongly placing spawn points of enemies near the bounds of the level, which resulted later into violating several rules with different buggy states, where the enemy fell into lava. This is considered as level design issue because the placement of the spawn points in the first place are related to level design. On the other hand, in another example from platformer game our approach was able to find stuck spots in the game, where the player got stuck in the ground or ceiling tiles. This bug occurs when player teleports to the tiles directly which should not happen.

7. Concluding discussion

In this study, we have proposed and presented an automated game testing approach that tests and verifies games through collaborative work between two genetic algorithms based agents. One agent generates buggy states, hence BSGA. The other agent we call RA, that receives buggy states from BSGA and performs reachability analysis and checking on them using colored Petri nets representation of the game. Moreover, we established a scheme to define game rules that is based on extending colored Petri nets with OCL, which is used to enable generating buggy states though checking rules violations.

The proposed approach was validated through experimentation using three different games from different genres. In addition, the approach was validated by experimenting and applying it to a colored Petri nets representation of a software system. Results of the experiments revealed the ability of the approach to generate and find bugs.

In this section we discuss the limitations of the approach, threats to validity of the work, and we propose future work in the field of automated video game testing.

7.1. Limitations

The proposed approach has the following limitations:

1. In our approach we used colored Petri nets as the base representation of games under test. Colored Petri nets is known to be high level modeling language, which might limit the ability of capturing different types of bugs, such as those related to physics, animations, level design, and visual effects.
2. Our approach focuses mainly on testing functional aspects of games, where games' functionalities are represented using colored Petri nets. However, one limitation of the proposed approach is that it does not capture time-related events.
3. Our established CPN-MCL game rules scheme uses OCL invariants that are based on logical expressions. However, other aspects of game rules might require more details, hence logical expressions might not be sufficient to represent such rules, and this suggests a limitation in the game rules scheme.

7.2. Threats to validity

We applied and tested our approach on different games, and our approach was able to find bugs in those games. However, generalizing the results among all genres of games might be a threat to this work. Though, this threat was tackled by using colored Petri nets to represent games, which abstracts games and removes dependency to certain genre or game engine. As another note, we applied our approach on one (non-game) software system in Experiment-E (Section 6.5), which have shown that the approach could be usable for non-game systems as well. However, a threat to validity can be imposed here which is related to the general applicability of the approach on different non-game software systems. This threat is further discussed as future work in Section 7.3.

We manually created colored Petri nets to model the games' code we used in the experiments we conducted to validate the proposed approach. This might have caused missing some games' details during the modeling process, or it might have introduced aspects that are not present in the games due to the biased modeling. This could pose threats to the validity of our conclusions. To mitigate this threat, we validated the Petri nets models we developed against observations collected from playing through the games. Moreover, found bugs and their scenarios were tested manually on the games to prove their existence.

Similarly, games' rules were manually created for the purpose of the experiments. Potential bias in the manual construction of these rules might pose a threat too. However, we tackled this by creating these rules based on the tested games' context, the authors' background in game development, the types of bugs studied and discussed in the literature (Albaghajati and Ahmed, 2020; Albaghajati, 2021). Since game rules are established to constrain and limit the player's actions and define the bounds of the game, we created them using the tested games' state variables that correspond to winning, losing, game objects' placement in the environment (including player's position and non-player characters' positions), bounds of levels, bounds of power-ups and health meters, and interactions and collisions between game objects. These state variables could contribute to different types of bugs, where invalid state bugs could occur due to inappropriate situation related to state variables, level design issues could be found because of wrong placement of game objects, conflicting overlapping goal states could be triggered when two or more rules are being met while they should not occur at the same time, and physics related bugs could happen due to collision issues.

As stated in this study, the current state of our approach requires some manual work, this could occur during the construction of Petri nets models, the extraction of game rules, or when checking the reported bugs on the game. This act of manual work can be thought of as a threat to validity for this work due to the possible bias during the manual activities. It is also understood that, in general, developers tend to prefer fully automated testing tools over the manual efforts in the development cycle (Ziftci and Cavalcanti, 2020). Although in our approach we automate the process of finding bugs and reporting their causes, fully automating the process from generating rules and Petri nets models to testing the reported bugs and their sequences on the game directly is considered a future work as we discuss in Section 7.3.

In another note, using repair methods in BSGA in our approach might cause the agent to deviate from the nature of GA. However, we mitigated that by guarding the repair operation with a reasonably small rate. In a similar way, the mini iterations used to force feasible solutions might introduce an overhead and exhaustive number of iterations and performance issues. Nonetheless, we analyzed their effect during the experiments and the results have shown that the average total number of mini iterations during the lifetime of BSGA is reasonable compared to the complexity of the game's states.

Moreover, due to the probabilistic nature of genetic algorithms, results obtained from experiments might not reflect all cases of the tested games. However, we mitigated this by conducting several runs of the experiments and analyzed the results based on the averages of the experiments' runs.

To the best of our knowledge, our approach is the first in the literature to find bugs using co-evolutionary approach that uses two genetic algorithms agents and utilizes colored Petri nets to represent systems and apply reachability analysis on them. Therefore, we could not benchmark our approach against any other approach from the literature. Not being able to benchmark our approach against other techniques could be a threat to validity to our work.

7.3. Future work

As models and rules of games were created manually in this study, a future work should study and experiment with creating an *Extraction Module* to automate the process of creating colored Petri nets that reflect the workflow of game's source code, in addition to extracting game's rules from game requirements and game design documents. On the other hand, since in this work we manually checked the found bugs on games to verify their existence, a future work should look at developing a *Testing Module* that simulates the found bugs directly and automatically on the game under test using the found sequences of transitions that led to a buggy state by connecting game's colored Petri nets model to the game.

In another note, our approach is composed of two co-evolutionary genetic algorithms agents, BSGA that generates buggy states and RA which applies reachability analysis on the generated buggy states using colored Petri nets model of the game under test. A future work might investigate other algorithms to be used in testing and finding bugs in games other than genetic algorithms, such as reinforcement learning (Zheng et al., 2019).

We used colored Petri nets in our approach to represent the functional aspects of games under test. However, future work should consider other languages, such as Timed Petri Nets to allow representing and verifying time related events and states in games (Felder et al., 1993). Moreover, our established game rules scheme uses OCL invariants that are based on logical expressions, as discussed in Section 4.1.2. Hence, a future work

could study adding further OCL invariants types, such as pre and post conditions which would allow checking change of states for an executed event (Gogolla et al., 2007). In addition, the integration of other formalization languages could be studied in future work, such as linear temporal logic which is a detailed specification language that allows checking temporal and safety properties (Varvaressos et al., 2014), and Promela to verify properties related to safety, correctness, and liveness (Abdulhameed et al., 2015).

In another note, the reachability agent in our approach (RA) is able to find bugs by checking and matching the reached states during colored Petri nets simulation with the buggy goal state it is trying to reach. A buggy state breaks at least one rule and it might break more. These buggy states might occur at some point during the system's cycle. Once a buggy state is found by RA, the reachability run terminates and results from this run are reported to BSGA. RA does not report sequences of bugs though, where it only reports the found buggy state and the sequence that led to it. However, sometimes what caused a bug to appear might be another bug that propagates its smelly state variables to other states through firing transitions, where this root bug was not detected in the first place. Thus, in a future work we might give RA the ability of detecting several bugs that occurred during one sequence of fired transitions.

During this work several experiments were conducted. Nonetheless, more games and game genres shall be tested using our approach. In addition, although our approach was applied to different games, however these games' state space might not be that large compared to other games produced by the industry. Thus, it would be interesting to study testing such games using our approach in the future.

Games are considered sophisticated when it comes to rules, concurrent actions/executions, and state space of possibilities. Applying our approach on games will mostly allow targeting other systems that are less complicated. Therefore, applying this approach to software systems could be another potential future work.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

The authors wish to acknowledge and thank King Fahd University of Petroleum and Minerals for the support provided to conduct this study.

References

Aarseth, E., 2012. A narrative theory of games. In: Proceedings of the International Conference on the Foundations of Digital Games. pp. 129–133.
Abdulhameed, A., Hammad, A., Mountassir, H., Tatibouet, B., 2015. An approach to verify sysml functional requirements using promela/SPIN. In: 2015 12th International Symposium on Programming and Systems. ISPS, pp. 1–9.
Ahumada, T., Bergel, A., 2020. Reproducing bugs in video games using genetic algorithms. In: 2020 IEEE Games, Multimedia, Animation and Multiple Realities Conference (GMAX). IEEE, pp. 1–6.
Albaghajati, A., 2021. Detecting game bugs using genetic algorithms and reachability analysis of colored Petri nets models. In: King Fahd University of Petroleum and Minerals (Master Thesis). [Online]. Available: https://eprints.kfupm.edu.sa/id/eprint/141862.
Albaghajati, A.M., Ahmed, M.A.K., 2020. Video game automated testing approaches: An assessment framework. IEEE Trans. Games 1.
Ali, S., Iqbal, M.Z., Arcuri, A., Briand, L.C., 2013. Generating test data from OCL constraints with search techniques. IEEE Trans. Softw. Eng. 39 (10), 1376–1402.
Basili, V.R., Selby, R.W., 1987. Comparing the effectiveness of software testing strategies. IEEE Trans. Softw. Eng. (12), 1278–1296.
Beattie, A., 2021. How the video game industry is changing. Investopedia, [Online]. Available: https://www.investopedia.com/articles/investing/053115/how-video-game-industry-changing.asp.
Bertolino, A., 2007. Software testing research: Achievements, challenges, dreams. In: Future of Software Engineering. FOSE '07, pp. 85–103.
Blow, J., 2004. Game development: Harder than you think. Queue 1 (10), 28–37.
Bouali, M., Rocheteau, J., Barger, P., 2009. Backward reachability analysis of colored petri nets.
Carver, R.H., Lei, Y., 2004. A general model for reachability testing of concurrent programs. In: International Conference on Formal Engineering Methods. Springer, pp. 76–98.
Chan, B., Denzinger, J., Gates, D., Loose, K., Buchanan, J., 2004. Evolutionary behavior testing of commercial computer games. In: Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No. 04TH8753), Vol. 1. IEEE, pp. 125–132.
Cho, S.M., Hong, H.S., Cha, S.D., 1996. Safety analysis using coloured Petri nets. In: Proceedings 1996 Asia-Pacific Software Engineering Conference. IEEE, pp. 176–183.
Colby, R., Colby, R.S., 2019. Game design documentation: four perspectives from independent game studios. Commun. Des. Quart. Rev. 7 (3), 5–15.
Correa, A., Werner, C., 2004. Applying refactoring techniques to UML/OCL models. In: International Conference on the Unified Modeling Language. Springer, pp. 173–187.
Doungsa-ard, C., Dahal, K.P., Hossain, M.A., Suwannasart, T., 2007. An automatic test data generation from UML state diagram using genetic algorithm. IEEE.
Emanuelle, F., Menezes, R., Braga, M., 2006. Using genetic algorithms for test plans for functional testing. In: 44th ACM SE Proceeding. pp. 140–145.
Felder, M., Ghezzi, C., Pezzé, M., 1993. High-level timed Petri nets as a kernel for executable specifications. Real-Time Syst. 5 (2), 235–248.
Fortin, F.-A., De Rainville, F.-M., Gardner, M.-A., Parizeau, M., Gagné, C., 2012. DEAP: Evolutionary algorithms made easy. J. Mach. Learn. Res. 13, 2171–2175.
GamesIndustry.biz, 2021. Gamesindustry.biz presents... The year in numbers 2020. https://www.gamesindustry.biz/articles/2020-12-21-gamesindustry-biz-presents-the-year-in-numbers-2020. (Accessed April 2021).
Ganty, P., Majumdar, R., 2012. Algorithmic verification of asynchronous programs. ACM Trans. Program. Lang. Syst. (TOPLAS) 34 (1), 6.
Gogolla, M., Büttner, F., Richters, M., 2007. USE: A UML-based specification environment for validating UML and OCL. Sci. Comput. Program. 69 (1–3), 27–34.
Gupta, N.K., Rohil, M.K., 2008. Using genetic algorithm for unit testing of object oriented software. In: 2008 First International Conference on Emerging Trends in Engineering and Technology. IEEE, pp. 308–313.
Holloway, L.E., Krogh, B.H., Giua, A., 1997. A survey of Petri net methods for controlled discrete event systems. Discrete Event Dyn. Syst. 7 (2), 151–190.
Hwang, G.-H., Tai, K.-C., Huang, T.-L., 1994. Reachability testing: An approach to testing concurrent software. In: Proceedings of 1st Asia-Pacific Software Engineering Conference. IEEE, pp. 246–255.
Jensen, K., Kristensen, L.M., 2009. Coloured Petri Nets: Modelling and Validation of Concurrent Systems. Springer Science & Business Media.
Jensen, K., Kristensen, L.M., Wells, L., 2007. Coloured Petri nets and CPN tools for modelling and validation of concurrent systems. Int. J. Softw. Tools Technol. Transf. 9 (3–4), 213–254.
Kanode, C.M., Haddad, H.M., 2009. Software engineering challenges in game development. In: 2009 Sixth International Conference on Information Technology: New Generations. IEEE, pp. 260–265.
Karaçali, B., Tai, K.-C., 2000. Model checking based on simultaneous reachability analysis. In: International SPIN Workshop on Model Checking of Software. Springer, pp. 34–53.
Khalifa, A., Isaksen, A., Togelius, J., Nealen, A., 2016. Modifying MCTS for human-like general video game playing. In: IJCAI. pp. 2514–2520.
Klaib, M.F., 2015. A parallel tree based strategy for 3-way interaction testing. Procedia Comput. Sci. 65, 377–384.
Kostin, A.E., 2003. Reachability analysis in T-invariant-less Petri nets. IEEE Trans. Autom. Control 48 (6), 1019–1024.
Kostin, A.E., 2006. A reachability algorithm for general Petri nets based on transition invariants. In: International Symposium on Mathematical Foundations of Computer Science. Springer, pp. 608–621.
Kramer, O., 2017. Genetic Algorithm Essentials, vol. 679. Springer.
Kuhn, D.R., Wallace, D.R., Gallo, A.M., 2004. Software fault interactions and implications for software testing. IEEE Trans. Softw. Eng. 30 (6), 418–421.
Last, M., Eyal, S., Kandel, A., 2005. Effective black-box testing with genetic algorithms. In: Haifa Verification Conference. Springer, pp. 134–148.
Mayr, E.W., 1984. An algorithm for the general Petri net reachability problem. SIAM J. Comput. 13 (3), 441–460.
McDonald, C., Schmalz, M., Monheim, A., Keating, S., Lewin, K., Cifaldi, F., Lee, J.H., 2020. Describing, organizing, and maintaining video game development artifacts. J. Assoc. Inf. Sci. Technol..

Mozgovoy, M., Pyshkin, E., 2018. A comprehensive approach to quality assurance in a mobile game project. In: Proceedings of the 14th Central and Eastern European Software Engineering Conference Russia. In: CEE-SECR '18, Association for Computing Machinery, New York, NY, USA.
Murata, T., 1989. Petri nets: Properties, analysis and applications. Proc. IEEE 77 (4), 541–580.
Murphy-Hill, E., Zimmermann, T., Nagappan, N., 2014. Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development? In: Proceedings of the 36th International Conference on Software Engineering. pp. 1–11.
Offutt, J., Thummala, S., 2019. Testing concurrent user behavior of synchronous web applications with Petri nets. Softw. Syst. Model. 18 (2), 913–936.
Ortega, J., Shaker, N., Togelius, J., Yannakakis, G.N., 2013. Imitating human playing styles in super mario bros. Entertain. Comput. 4 (2), 93–104.
Orvosh, D., Davis, L., 1994. Using a genetic algorithm to optimize problems with feasibility constraints. In: Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence. IEEE, pp. 548–553.
Ostrowski, M., Aroudj, S., 2013. Automated regression testing within video game development. GSTF J. Comput. (JoC) 3 (2), 1–5.
Pommereau, F., 2018. ZINC: a compiler for" any language"-coloured Petri nets.
Reuter, C., Göbel, S., Steinmetz, R., 2015. Detecting structural errors in scene-based multiplayer games using automatically generated Petri nets. In: Foundations of Digital Games, Pacific Grove, USA.
Rodríguez, A., Rutle, A., Kristensen, L.M., Durán, F., 2019. A foundation for the composition of multilevel domain-specific languages. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). IEEE, pp. 88–97.
Rozier, K.Y., 2011. Linear temporal logic symbolic model checking. Comp. Sci. Rev. 5 (2), 163–203.
Salcedo-Sanz, S., 2009. A survey of repair methods used as constraint handling techniques in evolutionary algorithms. Comp. Sci. Rev. 3 (3), 175–192.
Salen, K., Tekinbaş, K.S., Zimmerman, E., 2004. Rules of Play: Game Design Fundamentals. MIT Press.
Salge, C., Lipski, C., Mahlmann, T., Mathiak, B., 2008. Using genetically optimized artificial intelligence to improve gameplaying fun for strategical games. In: Proceedings of the 2008 ACM SIGGRAPH Symposium on Video Games. pp. 7–14.
Schultz, C.P., Bryant, R.D., 2016. Game Testing: All in One. Stylus Publishing, LLC.
Sicart, M., 2008. Defining game mechanics. Game Stud. 8 (2), n.
Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R., 2010. Verifying UML/OCL models using boolean satisfiability. In: 2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010). IEEE, pp. 1341–1344.
Spanoudakis, G., Kasis, K., 2000. Significance of inconsistencies in UML models. Interaction 1, 1.
Syswerda, G., 1993. Simulated crossover in genetic algorithms. In: Foundations of Genetic Algorithms, vol. 2. Elsevier, pp. 239–255.
Takahashi, K., Ono, I., Satoh, H., Kobayashi, O., 1997. An efficient genetic algorithm for reachability problems. In: Proceedings of the Thirtieth Hawaii International Conference on System Sciences, Vol. 5. IEEE, pp. 89–98.
Taylor, R.N., 1983. General-purpose algorithm for analyzing concurrent programs. Commun. ACM;(United States) 5.
Thong, W.J., Ameedeen, M., 2015. A survey of Petri net tools. In: Advanced Computer and Communication Engineering Technology. Springer, pp. 537–551.
Togelius, J., Schmidhuber, J., 2008. An experiment in automatic game design. In: 2008 IEEE Symposium on Computational Intelligence and Games. IEEE, pp. 111–118.
Varvaressos, S., Lavoie, K., Massé, A.B., Gaboury, S., Hallé, S., 2014. Automated bug finding in video games: A case study for runtime monitoring. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation. pp. 143–152.
Yessad, A., Mounier, I., Labat, J.-M., Kordon, F., Carron, T., 2014. Have you found the error? A formal framework for learning game verification. In: European Conference on Technology Enhanced Learning. Springer, pp. 476–481.
Zheng, Y., Xie, X., Su, T., Ma, L., Hao, J., Meng, Z., Liu, Y., Shen, R., Chen, Y., Fan, C., 2019. Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 772–784.
Ziftci, C., Cavalcanti, D., 2020. De-flake your tests : Automatically locating root causes of flaky tests in code at google. In: 2020 IEEE International Conference on Software Maintenance and Evolution. ICSME, pp. 736–745.