Search-Based Automated Play Testing of Computer Games: A Model-Based Approach

Raihana Ferdous[1], Fitsum Kifetew[1(✉)], Davide Prandi[1], I. S. W. B. Prasetya[2], Samira Shirzadehhajimahmood[2], and Angelo Susi[1]

[1] Fondazione Bruno Kessler, Trento, Italy
{rferdous,kifetew,prandi,susi}@fbk.eu
[2] Utrecht University, Utrecht, The Netherlands
{s.w.b.prasetya,s.shirzadehhajimahmood}@uu.nl

Abstract. Computer game technology is increasingly more complex and applied in a wide variety of domains, beyond entertainment, such as training and educational scenarios. Testing games is a difficult task requiring a lot of manual effort since the interaction space in the game is very fine grained and requires a certain level of intelligence that cannot be easily automated. This makes testing a costly activity in the overall development of games.

This paper presents a model-based formulation of game play testing in such a way that search-based testing can be applied for test generation. An abstraction of the desired game behaviour is captured in an extended finite state machine (EFSM) and search-based algorithms are used to derive abstract tests from the model, which are then concretised into action sequences that are executed on the game under test.

The approach is implemented in a prototype tool `EvoMBT`. We carried out experiments on a 3D game to assess the suitability of the approach in general, and search-based test generation in particular. We applied 5 search algorithms for test generation on three different models of the game. Results show that search algorithms are able to achieve reasonable coverage on models: between 75% and 100% for the small and medium sized models, and between 29% and 56% for the bigger model. Mutation analysis shows that on the actual game application tests kill up to 99% of mutants. Tests have also revealed previously unknown faults.

Keywords: Game play testing · Search-based testing · Model-based testing

1 Introduction

A common approach to test a computer game is by *play testing* it, where human users are deployed to play the game in order to find flaws, usability issues,

This work is a result of iv4XR project, funded by the EU Horizon 2020 research and innovation programme under grant agreement No. 856716.

© Springer Nature Switzerland AG 2021
U.-M. O'Reilly and X. Devroey (Eds.): SSBSE 2021, LNCS 12914, pp. 56–71, 2021.
https://doi.org/10.1007/978-3-030-88106-1_5

and to give feedback on the game user experience. This process is expensive, so introducing automation could greatly reduce the cost. Unfortunately, so far there is not much automated testing technology available for computer games. A handful that exist are tailored for specific games (and not publicly available).

Computer games also come in a great variety of genres such as action, adventure, puzzle, strategy, building, etc. [1]. The difference between genres (or even within the same genre) is large, e.g. an action game is usually a fast moving event driven system but the story is linear, while an adventure game is much less event driven, but the story is often complex. While such variety is good to keep users entertained, it certainly does not help in developing an automated testing approach that would work for all, or at least most, game genres.

This paper presents model-based approach for automated play testing of computer games, relying on search-based testing for generating tests. Outside the Game domain, model-based testing (MBT) [12] has long been known as a versatile testing approach. Similarly, search-based testing (SBT) [9] has proven effective for generating tests, in particular when the search space is large and exact methods are not applicable. This paper aims to formulate game play testing in such a way that SBT can be applied for automated test generation.

We present an approach for modelling game behaviour using extended finite state machines (EFSMs) in such a way that the tester can model the desired aspect of the game behaviour. Once the model is defined, SBT is applied for test generation from the model, following a typical MBT cycle.

The approach is implemented in a prototype tool `EvoMBT` which allows the generation of abstract tests from EFSM models by applying search-based algorithms. Empirical evaluation is carried out by applying `EvoMBT` on a 3D game called `Lab Recruits`. The concretisation and execution of abstract tests on `Lab Recruits` is implemented by means of an agent-based API of `Lab Recruits`. Results show that the proposed application of SBT and MBT are effective in achieving reasonable levels of coverage on the model and exposing faults.

The main contributions of this work are:

1. an approach combining SBT and MBT for automated game play testing
2. a tool `EvoMBT` for generation of tests from an EFSM model, allowing experimentation with existing search algorithms
3. publicly available artifacts (tool, models, data) that enable reproducibility of results and facilitate further research.

The rest of the paper is organised as follows: Sect. 2 presents the running example used throughout the paper. Section 3 discusses issues in modelling games and presents definitions used in the rest of the paper. Section 4 introduces the testing problem and Sect. 5 presents our proposed search-based test generation approach. Experimental results are presented in Sect. 6 and related work is discussed in Sect. 7. Section 8 concludes, and outlines future work.

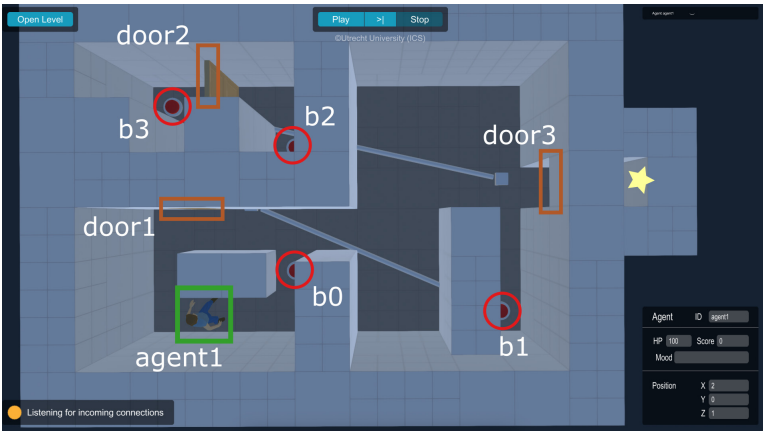

Fig. 1. Level `buttonDoors1` in `Lab Recruits`.

2 Running Example

This section introduces `Lab Recruits`[1], a 3D game developed for experimenting with intelligent agents. The application allows the definition of mazes, a set of rooms connected by doors. Each door is opened by one or more buttons, and each button activates one or more doors. The goal is to find the path to reach a certain room by opening doors in the right order. The game can be played by both humans and artificial agents [10]. `Lab Recruits` levels are defined as csv (comma-separated value) human-readable files allowing researchers to specify their tests of variable complexity.

As a running example, Fig. 1 shows a level of the `Lab Recruits` game named `buttonDoors1`. The level features three doors, `door1`, `door2`, and `door3`, and four buttons, `b0`, `b1`, `b2`, and `b3`. Door `door1` is activated by buttons `b1`, `b2`, and `b3`, while `door2` and `door3` are connected only to `b2`. Note that `b0` is not connected, therefore pressing it has not effect in the game. Agent `agent1` aims to reach the room marked with a star, and therefore to open `door3`. A possible path requires `agent1` to press `b1` to open `door1` and then `b2` to open `door3`. Since `b2` also acts on `door1`, at this point `agent1` cannot reach `door3`, but need to traverse `door2` and press `b3` to open `door1`. Now, `agent1` walks through `door1` and `door3`, finally reaching the star room. Even if the layout of the level is simple, the path to reach the final room is not trivial and it is not trivial for automated play testing.

3 Modelling Games

Computer games are stateful systems, and hence using state-based models to model them is natural. They are also very complex systems, so abstraction will

[1] https://github.com/iv4xr-project/labrecruits.

have to be applied, but not to the degree that we lose control and observability of the system. Plain finite state machines (FSM) or labelled transition systems are in most cases either cumbersome or insufficient, and we will need to use, for instance, EFSM that allow variables and assignments to be superimposed over a finite state model. The running example presented in Sect. 2 with buttons and doors whose states change dynamically can be modelled with a plain FSM, but its size would be quite large; whereas an EFSM model would be much more succinct and easy to understand.

Unlike other types of systems, modelling a computer game has an additional challenge due to the presence of the 'world' where the game is played on. E.g. the Lab Recruits game in Sect. 2 is played in a virtual lab building as its 'world'. A world imposes certain constraints. Triggering a state might require a certain interactable to be interacted with, but a test agent can only do that if the interactable is physically reachable from its current position. If there is a wall between them, this is obviously problematic. So in terms of modelling, such physical constraints need to be taken into account as well. That is, a transition in the model should be translatable to a concrete sequence of actions by the agent, that are also physically possible. The same goes with observation. When the model requires that a certain condition should be checked, e.g. as an invariant to check, or as the guarding condition of a transition, it implies that the agent should be able to observe the condition. In the game setup this is not always given. A wall might be blocking the agent's sight, and hence the agent might first need to move itself to a spot where it can observe the said condition. In terms of modelling, this means that introducing guards and invariants in the model implies that there should exist a feasible way for the agent to actually observe them.

In the next subsection, we present the EFSM notation we adopt in the rest of the paper, and introduce the modelling of `Lab Recruits` (see Sect. 2) which takes into consideration the issues mentioned above regarding the modelling of games for testing purposes.

3.1 EFSM Notation

An FSM models the behaviour of a system as a finite set of states connected by transitions, where a transition could be fired by an input and returns an output. EFSMs [4] introduce data information into FSM behavioral representation. An EFSM has an internal memory, a set of variables, to store data and extends FSM transitions with guards and update transformations. Guards specify whether a transition can be performed according to the values of the variables stored in the memory. Updates allow changing variable values as a result of a transition. In this paper we adopt the following definition of EFSM.

Definition 1 (EFSM). An EFSM E is a 7-tuple (S, I, O, D, F, U, T), where

- S is a set of states
- I is s set of input symbols and O is a set of output symbols
- $\bar{D} : D_1 \times \ldots \times D_n$ is an n-dimensional space.
- F is a set of enabling functions $f_i : \bar{D} \rightarrow \{0, 1\}$
- U is a set of update transformations $u_i : \bar{D} \rightarrow \bar{D}$
- $T : S \times F \times I \rightarrow S \times U \times O$ is a transition relation.

Symbol $\bar{x} = (x_1, \ldots, x_n)$ indicates an element of $D_1 \times \ldots \times D_n$. Given states s_1, s_2, input i, output o, $f \in F$, and $u \in U$, $(s_1, f, i) \rightarrow (s_2, u, o)$ denotes $T(s_1, f, i) = (s_2, u, o)$. Given a vector variables $\bar{x} \in \bar{D}$ at s_1, the notation specifies a transition from s_1 to s_2, triggered by the input i, and *provided* $f(\bar{x}) = 1$. The transition produces the output o, and updates $\bar{x}$ to $u(\bar{x})$.

A finite *path* P over an EFSM $E = (S, I, O, D, F, U, T)$ is a finite sequence of transitions $t_0 \ldots t_n \in T$. A configuration of E is a pair state $s \in S$ and vector variables $\bar{x} \in D$. A feasible path over E from a configuration $(s_0, \bar{x})$ is a path $t_0 \ldots t_n$ such that the enabling function of t_0 is 1 for $\bar{x}$ and for each $i \in [1, n]$, $f_i(u_{i-1}(\bar{x}_{i-1})) = 1$ with f_i enabling function of t_i, u_{i-1} update function on t_{i-1}, and x_{i-1} vector variables at $i - 1$.

EFSM Model for Lab Recruits. A model for `Lab Recruits` captures the essential features of the game while abstracting away from details that are not of interest to the tester. For instance, to check the consistency of the button-door connections in the game, a candidate model could consider only buttons and doors and the actions the player can perform: move from a door to a button or to another door, walk trough a door, and toggle a button. Such a model for `buttonDoors1` in Fig. 1 could be EFSM $LR1 = (S, I, O, D, F, U, T)$ in Fig. 2. The set of states S are buttons and doors. For each door `door_`, `d_p` and `d_m` model the two sides of `door_`. The n-dimensional space D records door status with $D = \{0, 1\} \times \{0, 1\} \times \{0, 1\}$, where x_i control `door_`i. The EFSM in Fig. 2 has three types of transitions: solid edges for free travel, when the agent can move from one entity to the other without traversing a door; this type of transition has empty enabling and update functions. Dotted transitions model guarded movements that happen when the agent walks through a door; the enabling function check the status of the corresponding variable, while update function is empty. Dashed self loop transitions are for toggle actions, i.e., the agent presses the button; the update function changes the status of the doors connected to the pressed button. Note that the concept of 'transitions' here also incorporates 'world travel'. That is, a transition in the model is guaranteed to be physically possible in the `Lab Recruits` world, and furthermore the guards guarding the transitions can be physically checked as well (through some concretisation that guides the agent to observe them, achieved via automated navigation in the underlying testing framework [11]). Input set $I = \{$`travel`, `toggle`$\}$ defines the actions an agent can perform, i.e., move (`travel`) or press a button (`toggle`). Output set O is empty.

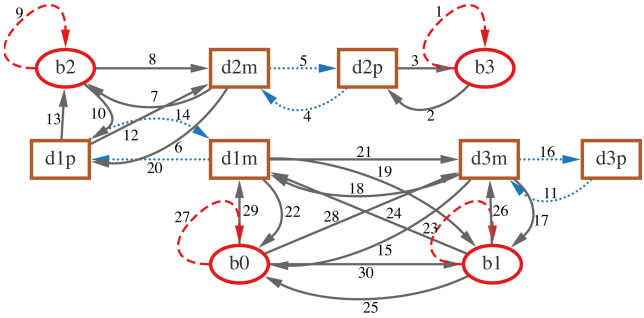

Fig. 2. EFSM model of `buttonDoors1` in Fig. 1

The game starts with the agent near `b0` and with all the doors closed, therefore $\bar{x} = (0, 0, 0)$. A feasible path to reach star room in Fig. 1 has to include transition 16 (t16, for short), and therefore opening `door3`. The `agent1` starts going to `b1` (t30) and pressing it (t23). Update transformation changes $\bar{x}$ to $(1, 0, 0)$, i.e., `door1` is open. Then, `agent1` goes to `d1m` (t24) and walks through `door1` reaching `d1p` (t20). Enabling function of t20 is 1, as $x_1 = 1$. After that, `agent1` goes to `b2` (t13) and presses it (t9), changing $\bar{x}$ to $(0, 1, 1)$, so `door1` is closed, while `door2` and `door3` are open. At this point, the agent goes to `door2` (t8), crosses it (t5), reaches `b3` (t3), and toggles it (t1). Button `b3` opens `door1` so that $\bar{x} = (1, 1, 1)$, i.e., all the doors are open. Now, `agent1` can reach star room following, for instance, t1, t4, t6, t14, t21, and t16. This gives an example of a feasible path from the initial position of the agent to the star room mimicking the steps an agent has to perform.

We also implemented a random level generator for `Lab Recruits` for experimental purposes. The generator builds on the observation that the EFSM of a `Lab Recruits` level has a specific structure. First, a room is represented by the set of buttons it contains. Given the total number of buttons *n_buttons* in a level and the mean *mean_buttons* number of buttons in a room, the algorithm extracts random integers from a Poisson distribution with mean *mean_buttons*, until all buttons are used. For instance, given *n_buttons* = 10 and *mean_buttons* = 2, the generated sequence $2, 3, 2, 1, 2$ corresponds to the number of buttons in a level with 5 rooms. Then, given the number of doors *n_doors*, we randomly connect two rooms until all doors are used. The algorithm guarantees that there are not unconnected rooms. Given a room, the corresponding EFSM model has a state for each button and door side, and all the states are connected by a free travel Each button has a self loop with update function that represents button-door connections. Finally, the models corresponding to linked rooms are connected by guarded travel transition, where the enabling function checks the status of the door. The generated EFSM can be transformed into the corresponding csv file level and opened on `Lab Recruits`.

4 Problem Definition

The play testing problem involves finding a sequence of actions that achieve a desired goal in the game. Given the model of the system under test (SUT), the play testing problem could be represented as a coverage problem on the model. Testing a specific play action in the game (corresponding to a transition T in the model) is equivalent to finding a *prefix play* that would reach the source state of T, and subsequently a suffix play to verify the effect of T. Hence, generating a test suite from the model that would cover all transitions corresponds to exercising the corresponding game play actions in the SUT. Stronger coverage criteria, such as k-transition coverage and path coverage, represent more rigorous interactions of game play actions.

5 Test Generation

Our proposed approach follows the generic model-based test generation approach where the *SUT* is abstracted into a *model* which is then used to generate *abstract tests*. The abstract tests are then concretised into *concrete tests* that can be executed on the SUT.

The goal of the work presented in this paper is to investigate the feasibility of applying the model-based approach by incorporating search-based test generation for the *test generation* phase.

The *abstraction* phase, where the model of the SUT is built, typically involves human involvement as it requires a good understanding of the behaviour of the SUT. In our experiments, we have used models which were crafted manually as well as randomly generated ones. However, the test generation approach presented here is independent of how the model is generated, as long as it is as described in Sect. 3.1. The *concretisation* and *execution* phases are specific to the SUT and could be implemented in different ways, depending on the nature of the SUT. For our experiments we have built automated transformers from abstract tests to concrete tests, and adopted an agent-based API provided by the SUT for executing the tests. For a different SUT, different concretisation (and execution) mechanisms are needed, however the generation of abstract tests remains the same, as long as the model of the SUT is provided.

In the remainder of this section, we present the search-based test generation approach for deriving abstract tests from the model of the SUT.

5.1 Search-Based Test Generation

Test generation from models could be driven by different goals. In this paper we outline a search-based approach that can be applied to find test suits that satisfy a desired model coverage criterion, e.g., transition coverage. We present the various ingredients needed for applying a search algorithm for test generation, including individual representation, search operators, and fitness function.

Individual Representation. Given an EFSM, we represent an individual as a path (sequence of transitions), starting from the initial state of the model (see Sect. 3.1). Individuals can be of different length, up to a pre-defined maximum. For our running example (see Fig. 2), $I_1 = \langle t_{27}, t_{28}, t_{18}, t_{21} \rangle$ represents an example of an individual. Note that paths in the model may or may not be feasible, hence an individual, as generated initially, is not guaranteed to be feasible.

Search Operators. With individuals represented as paths in the model, different operators could be implemented. Here we describe crossover and mutation operators that we used in our experiments. Clearly, other operators could be implemented and experimented with.

Crossover: one possible way of implementing crossover is to adopt a straightforward application of *single point relative crossover*. Given two individuals, a common state is chosen at random and the tails of the two individuals are swapped. For our running example, if $I_1 = \langle t_{30}, t_{26}, t_{16}, t_{11} \rangle$ and $I_2 = \langle t_{29}, t_{21}, t_{17}, t_{25} \rangle$, crossover at state $d3m$ results in offspring $O_1 = \langle t_{30}, t_{26}, t_{17}, t_{25} \rangle$ and $O_2 = \langle t_{29}, t_{21}, t_{16}, t_{11} \rangle$.

Mutation: we propose three mutation operators, applied with equal probability:

1) insert self transition: insert a self transition on a randomly chosen state of the model, if such a transition is allowed
2) delete self transition: remove a self transition at random
3) delete a transition: remove a transition at random.

Fitness Function. Given an EFSM model and a given coverage criterion, the fitness function should guide the search towards covering all coverage targets. However, since the individual may not be feasible, the fitness function should also guide the search towards turning the individual into a feasible one. As a result, the fitness function has two components: 1) related to path feasibility, and 2) related to the search target. A high level algorithm of the fitness function we adopted is shown in Algorithm 1. To calculate the fitness of an individual with respect to a coverage target, first the individual is executed on the model and the execution trace as well as the outcome of the execution are returned (line 6 in Algorithm 1). If the individual is a feasible path, then the algorithm checks to see if the current target is present in the individual. If present (line 9) then target is covered, otherwise, the fitness value should estimate the distance from satisfying the target. In this case, we opt for a simple heuristic, i.e., penalising the individual by a predefined constant value (*PENALTY1*). Other heuristics could be applied here as well. If however the individual happens to be infeasible (line 14 in Algorithm 1, this means that a transition guard in the path represented by the individual has failed. In this case, we compute the approach level and branch distance for the path (lines 15 and 16). Approach level is computed as the number of transitions in the path yet to be traversed. Branch distance is

Algorithm 1. Fitness function

```
1: Input
2:     I    individual
3:     T    target
4: Output
5:     f    fitness value
6: trace, feasible ⟵ executeOnModel(I)
7: if feasible then                          ▷ individual I represents a feasible path in the model
8:     al_feasiblity, bd_feasiblity ⟵ 0
9:     if T ∈ I then
10:        al_target, bd_target ⟵ 0
11:    else
12:        al_target, bd_target ⟵ PENALTY1    ▷ even if I eventually turns feasible, T remains
       uncovered, but at least I is feasible
13:    end if
14: else if not feasible then                ▷ individual I represents an infeasible path
15:    al_feasiblity ⟵ length(I) − passdTransitions
16:    bd_feasiblity ⟵ computeBranchDistance(trace)
17:    if T ∈ I then
18:        al_target, bd_target ⟵ 0          ▷ fitness takes the value of feasiblity_fitness
19:    else
20:        al_target, bd_target ⟵ PENALTY2   ▷ even if I turns feasible, T remains uncovered
21:    end if
22: end if
23: f_feasiblity = al_feasibility + normalise(bd_feasibility)
24: f_target = al_target + normalize(bd_target)
25: f = f_feasibility + f_target
```

computed based on the guard expression of the failing transition, as typically done in code-based testing [9]. We then check whether the individual contains the current target (line 17). If yes, no penalty is applied, otherwise, a penalty is applied ($PENALTY2 \gg PENALTY1$). Finally, the algorithm computes the *feasibility fitness* and *target fitness* values, and sums them up to find the fitness value of the individual (lines 23–25).

Search Algorithms. Once the individual encoding, search operators, and fitness function are defined, existing search algorithms could be applied to generate tests. In practice though, the corresponding machinery for implementing the test generation is needed. We have implemented a prototype tool `EvoMBT` that uses EvoSuite [5] as a library. `EvoMBT` implements all the model related parts, including the operators discussed above. It implements EvoSuite's interfaces in such a way that search algorithms already implemented in EvoSuite can be used out-of-the-box. Details are discussed in Sect. 6.

6 Evaluation

In this section we present the experiment we carried out in order to get insight into the feasibility of the proposed test generation approach combining search based algorithms with model based testing.

6.1 Prototype: `EvoMBT`

`EvoMBT` provides an implementation of the EFSM used in this paper and the necessary machinery for generating/executing tests from/on the model, compute fitness values, and collect coverage. `EvoMBT` uses search algorithms implemented in EvoSuite [5] (i.e., EvoSuite is used as a library). `EvoMBT` currently implements state and transition coverage criteria, relative point crossover, and a number of mutation operators. At the moment, several search algorithm found in EvoSuite can be used with `EvoMBT` without any modification. For the purpose of experimentation, `EvoMBT` also implements mutation operations on the `Lab Recruits` application, and enables concretisation and execution of generated tests on `Lab Recruits` (both mutated and original). It generates different reports that enable analysis of results as well as debugging of eventual faults. `EvoMBT` is publicly available in Github: https://github.com/iv4xr-project/iv4xr-mbt. We also provide an executable jar with all the necessary resources, and additional plots that could not fit in the paper, here: https://doi.org/10.5281/zenodo.4768470.

6.2 Models of the System Under Test

We use three different models of `Lab Recruits`: i) `buttonDoors1`: the running example (see Sect. 2) with 10 states and 30 transitions, ii) `randomMedium`: randomly generated model as described in Sect. 3.1 with $n\_doors = 8$ and $n\_buttons = 10$ having 26 states and 116 transitions, and iii) `randomLarge`: randomly generated with $n\_doors = 15$ and $n\_buttons = 20$ having 50 states and 194 transitions.

6.3 Experiment Setup

Experiments are aimed at assessing: *feasibility of search-based algorithms for generating abstract test sequences from the model*, *practicality of abstract tests for execution on the actual system under test*, and *fault finding potential of the generated tests*. Consequently we formulate the following research questions to guide our experimental evaluation:

RQ1 - Suitability of SBT how suitable are search based algorithms for test generation from the models?

RQ1.1 - Model coverage how much of the models are covered by the test generation algorithms?

RQ2 - Test execution are the model-based tests feasible in terms of execution on the actual application?

RQ3 - Fault finding what is the fault finding potential of the tests generated from the models?

For **RQ1**, we report on the search algorithms we used for test generation and the coverage they achieved on the models (**RQ1.1**). For this purpose we use

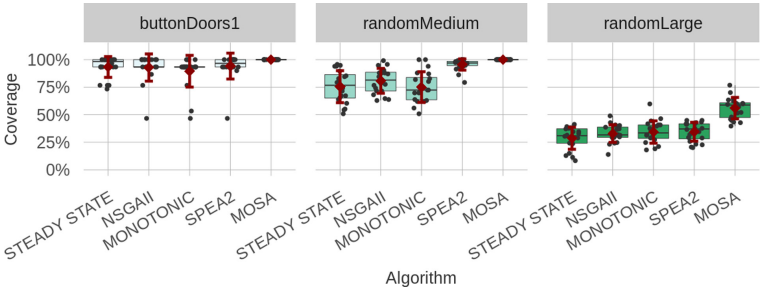

Fig. 3. Coverage achieved by the search algorithms for the three models

transition coverage criterion, computed as the ratio of covered transitions to the total number of transitions in the model. For **RQ2**, we measure the test execution time on the `Lab Recruits` application. For **RQ3**, we measure the mutation score of the tests on mutants injected into the `Lab Recruits` application. Mutation score is computed as the ratio of killed mutants to the total number of mutants generated.

Experimental Settings. For the search algorithms, we kept the default values in EvoSuite. For search budget, we use 300 s, collecting statistical data every 10 s. Experiments were run on computers with Intel Core i7 processors with 8 cores @2.80 GHz and 8 GB memory, running Ubuntu Linux.

Experiment Procedure. For **RQ1.1** we run each algorithm on a given model 20 times, to account for the random nature of the algorithms. Hence, we performed 5 algorithms $\times$ 3 (models) $\times$ 20 (repetitions) = 300 runs for a total of 300 $\times$ 300 (seconds) = 90000 s (25 h). For **RQ2**, we report test execution times on the `Lab Recruits` application for all test suites generated. For **RQ3**, we report the mutation scores for all algorithms generated on `buttonDoors1`, `randomMedium`, and `randomLarge`.

6.4 Results

Suitability of SBT (RQ1). The first set of results are related to the suitability of search based testing for the generation of tests from models. We have applied 5 different search algorithms: *MONOTONIC GA, MOSA, NSGAII, SPEA2, STEADY STATE GA* for the generation of tests from the models. Figure 3 shows the coverage achieved by each algorithm on the three models of `Lab Recruits`.

As can be seen from Fig. 3, the algorithms achieve different levels of coverage on the three models. On the running example (`buttonDoors1`), which is the least complex of the three with 30 transitions in total, all algorithms achieve high levels of coverage with MOSA achieving 100%. On the `randomMedium` model which has 116 transitions, MOSA still achieves full coverage, while the other algorithms achieve less than for the smaller model. On the largest of the three,

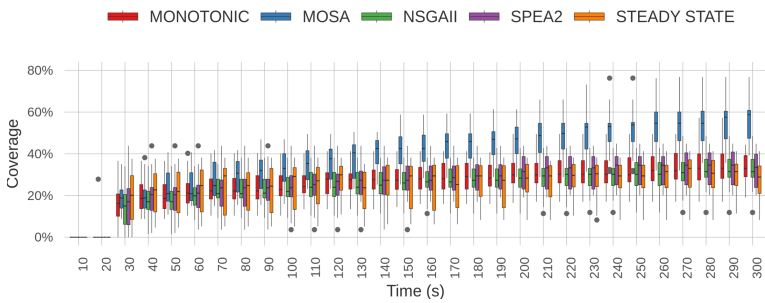

Fig. 4. Coverage achieved on `randomLarge` by the search algorithms over time

Table 1. Test execution time on `Lab Recruits` (in minutes), the tests are organized into 20 test suites for each algorithm

Model	Algorithm	Tests	Passed	Failed	Time (avg per test)
`buttonDoors1`	STEADY STATE	237	220	17	36 (0.15)
	NSGAII	222	195	27	34.2 (0.15)
	MONOTONIC	239	205	34	29.5 (0.12)
	SPEA2	227	204	23	28.4 (0.13)
	MOSA	400	380	20	36.1 (0.09)
`randomMedium`	STEADY STATE	436	391	45	661.3 (1.52)
	NSGAII	614	502	112	1176 (1.92)
	MONOTONIC	564	495	69	1324.5 (2.35)
	SPEA2	594	527	67	1104.8 (1.86)
	MOSA	919	833	86	1362.2 (1.48)
`randomLarge`	STEADY STATE	249	31	218	243.2 (0.98)
	NSGAII	401	22	379	412.8 (1.03)
	MONOTONIC	466	25	441	613.8 (1.32)
	SPEA2	307	31	276	267.4 (0.87)
	MOSA	546	28	518	430.1 (0.79)

which has 194 transitions, the coverage achieved by the algorithms decreases with MOSA achieving 59% median coverage while the others achieve below 37%. Given the size of the model, the results could potentially improve if search budget is increased. As can be seen in Fig. 4, the trend shows that the coverage is likely to increase with increased search budget.

*We answer **RQ1** positively: different search algorithms could be applied for test generation from models, achieving reasonable levels of transition coverage.*

Test Execution (RQ2). We measured the time it takes to execute the generated tests on the actual `Lab Recruits` application. The abstract tests are

Table 2. Mutation analysis results

Model	Mutants	Algorithm	Suits	Tests	Mutantion score (avg. per suite)
`buttonDoors1`	5	STEADY STATE	20	220	0.46
		NSGAII	20	196	0.41
		MONOTONIC	20	204	0.34
		SPEA2	20	204	0.37
		MOSA	20	377	0.59
`randomMedium`	8	STEADY STATE	20	386	0.83
		NSGAII	20	506	0.80
		MONOTONIC	20	487	0.79
		SPEA2	20	508	0.92
		MOSA	20	850	0.98
`randomLarge`	24	STEADY STATE	20	32	0.08
		NSGAII	20	20	0.03
		MONOTONIC	20	25	0.11
		SPEA2	20	31	0.08
		MOSA	20	29	0.12

concretised for automated execution on `Lab Recruits` via its agent based API. The execution of tests on `Lab Recruits` involves the player (driven by the test agent) actually interacting with the game environment (e.g., going from one room to another, pressing buttons, etc.). Hence, the execution of tests is time taking.

As can be seen from Table 1, test execution on `Lab Recruits` is rather time taking. Hence, if the test generation were to be done directly on `Lab Recruits`, it would have taken an extremely long period of time. To give an idea, on `buttonDoors1`, MOSA performed, on average, 54848 fitness evaluations (i.e., executed abstract tests on the model). Executing that many tests directly on `Lab Recruits` would take several days.

*For **RQ2**, overall the model based approach for test generation combined with search algorithms gives an efficient means for generating tests for such systems as `Lab Recruits` where test execution is slow.*

Fault Finding (RQ3). With **RQ3**, we assess the fault finding potential of the generated tests. We created mutants in the `Lab Recruits` application in which the association between buttons and doors is changed, i.e., a link between a button and a door is removed. We created a number of mutants and executed the generated test suites on each mutant. Given that the tests may not all run successfully on the original application (see Table 1, 'Failed' column), we executed only the passing tests on the mutants, and calculated the mutation

score. The results, presented in Table 2, show that the generated tests are effective in detecting the injected faults. It is worth noting that the mutation scores for `buttonDoors1` and `randomLarge` are low because several tests failed when executed on the original `Lab Recruits` application (see Sect. 6.4), reducing the coverage of the tests. These test failures are due to bugs in `Lab Recruits` and the agent based API. In particular, in `Lab Recruits`, under certain circumstances, pressing a button fails to open a door controlled by it. On the agent API, we found instances where the agent gets stuck while navigating the game world, which is not supposed to happen. All faults have been reported to the developers of `Lab Recruits`.

*Concerning **RQ3**, experimental results show that generated tests are effective in detecting injected faults and revealing actual bugs in the application under test.*

7 Related Work

Although SBT has been successful for various types of software, its application in computer games has not been much studied. Directly using a search algorithm to search for test sequences without any model has not been attempted, as far we as know. The search space is far too large for such an approach to work. Instead, existing works tend to employ search algorithms for optimizing learning-based automated agents. For example Holmgård et al. [6] uses Monte Carlo Search Tree (MCTS) based agents to replace human play testers. A genetic algorithm is used to evolve MCTS' selection policy towards a desired play style. The case study is small; a game played in a 10×20 grid. It is unclear if the approach would scale to bigger games.

There were very few studies, as far as we could find, on the use of MBT for testing computer games, e.g. [2,7,13]. Ariyurek et al. [2] use a scenario graph for generating abstract test sequences. Such a graph is essentially an FSM whose states are decorated with a set of predicates that abstractly describe the state of a game under test. MCTS is used to search for the concrete sequence of actions that implement the steps in an abstract test sequence. The case studies are however small scale games, played in a grid not larger than 10×11; it is not clear if MCTS would scale to a larger search space. To deal with larger game worlds Prasetya et al. investigate the use of navigation mesh as a model [11] of the game world under test, subjected to their agent-based automated testing framework iv4XR/aplib [10]. The idea is taken from pathplanning, where the walkable areas of a physical or virtual world is divided into a finite set of connected shapes (e.g. triangles). This reduces the initially infinite search space into a finite graph which is then used as a model to guide agents' navigation, e.g. using A*.

Iftikhar et al. [7] use UML state machine, which is an EFSM, to model an open source variant of the Super Mario game. The study does not however fully explore the implication of using *extended* FSMs. For generating tests, an N+ strategy is used [3], that is aimed at covering all transitions and round trips. This strategy is not strong enough to handle an EFSM with complex constellations of conditions. It essentially unrolls the FSM into a tree where along any full path in the tree no node is repeating, except if it is the last node.

Outside the game domain, search-based algorithms are used for generating tests from EFSMs. Many of the works are focused on finding valid paths from the models and eventually covering predefined goals (e.g., [8]). Our work is aimed at exploring the feasibility of SBT for automated play testing via modelling. In this regard, existing works on SBT from EFSMs are complementary with ours, and could eventually be experimented with in `EvoMBT` so as to increase the effectiveness and efficiency of test generation. For instance, the fitness function of Kalaji et al. [8] could be implemented in `EvoMBT` in order to assess its feasibility for the play testing use case.

8 Conclusion and Future Work

We have presented an approach for automated game play testing by employing the combined application of search-based and model-based testing. The main objective of the work presented is exploratory in nature where we tried to assess the suitability of search-based testing for automated game play testing. Game play behavior is abstracted into an EFSM model and search-based algorithms are used to generate abstract tests, which are then converted into concrete tests that can be executed on the game. Experimental results are promising where a number of search algorithms were experimented with and achieved reasonable model coverage, mutation score, and exposed real bugs in the game under test.

The current work makes a number of choices with respect to heuristics used for test generation, such as fitness function and search operators. There are different alternatives that could be implemented and experimented with as part of future work.

References

1. Apperley, T.H.: Genre and game studies: toward critical approach to video game genres. Simul. Gaming **37**(1), 6–23 (2006)
2. Ariyurek, S., Betin-Can, A., Surer, E.: Automated video game testing using synthetic and humanlike agents. IEEE Trans. Games **13**(1), 50–67 (2021). https://doi.org/10.1109/TG.2019.2947597
3. Binder, R.: Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Professional (2000)
4. Cheng, K.T., Krishnakumar, A.S.: Automatic functional test generation using the extended finite state machine model. In: 30th ACM/IEEE Design Automation Conference, pp. 86–91. IEEE (1993)
5. Fraser, G., Arcuri, A.: EvoSuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, pp. 416–419 (2011)
6. Holmgård, C., Green, M.C., Liapis, A., Togelius, J.: Automated playtesting with procedural personas through MCTS with evolved heuristics. IEEE Trans. Games **11**(4), 352–362 (2018)

7. Iftikhar, S., Iqbal, M.Z., Khan, M.U., Mahmood, W.: An automated model based testing approach for platform games. In: 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 426–435. IEEE (2015)
8. Kalaji, A.S., Hierons, R.M., Swift, S.: An integrated search-based approach for automatic testing from extended finite state machine (EFSM) models. Inf. Softw. Technol. **53**(12), 1297–1318 (2011). https://doi.org/10.1016/j.infsof.2011.06.004
9. McMinn, P.: Search-based software test data generation: a survey. Softw. Test. Verif. Reliab. **14**(2), 105–156 (2004). https://doi.org/10.1002/stvr.294
10. Prasetya, I.S.W.B., Dastani, M., Prada, R., Vos, T.E.J., Dignum, F., Kifetew, F.: Aplib: tactical agents for testing computer games. In: Baroglio, C., Hubner, J.F., Winikoff, M. (eds.) EMAS 2020. LNCS (LNAI), vol. 12589, pp. 21–41. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-66534-0_2
11. Prasetya, I., et al.: Navigation and exploration in 3D-game automated play testing. In: Proceedings of the 11th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (ATEST), pp. 3–9 (2020)
12. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Softw. Test. Verif. Reliab. **22**(5), 297–312 (2012). https://doi.org/10.1002/stvr.456
13. Zhao, H., Sun, J., Hu, G.: Study of methodology of testing mobile games based on TTCN-3. In: 2009 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, pp. 579–584. IEEE (2009)