GameRTS: A Regression Testing Framework for Video Games

Jiongchi Yu[1†], Yuechen Wu[2†], Xiaofei Xie[1*], Wei Le[3], Lei Ma[4,5], Yingfeng Chen[2*], Jingyu Hu[2] and Fan Zhang[6,7]
[1] Singapore Management University, Singapore
[2] NetEase Fuxi AI Lab, China [3] Iowa State University, USA
[4] University of Alberta, Canada [5] The University of Tokyo, Japan
[6] Zhejiang University, China [7] Zhengzhou Xinda Institute of Advanced Technology, China.

***Abstract*—Continuous game quality assurance is of great importance to satisfy the increasing demands of users. To respond to game issues reported by users timely, game companies often create and maintain a large number of releases, updates, and tweaks in a short time. Regression testing is an essential technique adopted to detect regression issues during the evolution of the game software. However, due to the special characteristics of game software (*e.g.*, frequent updates and long-running tests), traditional regression testing techniques are not directly applicable. To bridge this gap, in this paper, we perform an early exploratory study to investigate the challenges in regression testing of video games. We first performed empirical studies to better understand the game development process, bugs introduced during *game evolution*, and the *context sensitivity*. Based on the results of the study, we proposed the first regression test selection (RTS) technique for game software, which is a compromise between safety and practicality. In particular, we model the test suite of game software as a State Transition Graph (STG) and then perform the RTS on the STG. We establish the dependencies between the states/actions of STG and game files, including game art resources, game design files, and source code, and perform change impact analysis to identify the states/actions (in the STG) that potentially execute such changes. We implemented our framework in a tool, named *GameRTS*, and evaluated its usefulness on 10 tasks of a large-scale commercial game, including a total of 1,429 commits over three versions. The experimental results demonstrate the usefulness and effectiveness of *GameRTS* in game RTS. For most tasks, *GameRTS* only selected one trace from STG, which can significantly reduce the testing time. Furthermore, *GameRTS* detects all the regression bugs from the test evaluation suites. Compared with the file-level RTS, *GameRTS* selected fewer states/actions/traces (*i.e.*, 13.77%, 23.97%, 6.85%). In addition, *GameRTS* identified 2 new critical regression bugs in the game.**

I. Introduction

Video games have been an important part of Internet activities and daily life for many people worldwide. According to a recent report [1], more than 1 in 3 web users play video games every day, and around half of them (51%) play at least once a week. The COVID-19 lockdowns hurdle further push the heavy usage of Internet including game domains. With the increasing popularity of games, the quality of game software becomes essential. Game bugs not only can impact user experience, but have caused financial loss and even lead to security and privacy risks [2].

* Corresponding authors
† Contributed equally

To accommodate customers' sentiments and feedback, game software is often frequently updated, and creates and maintains a large number of versions. According to a previous study [3], the industrial game software can be updated with three internal versions per day. The rapid software evolution can pose big challenges for game quality. To identify the issues during version updates, *regression testing* is widely applied to traditional software. It re-executes the already executed test cases to ensure that existing functionalities do not break, and the new changes in the software do not introduce bugs. Regression testing can be rather costly, taking up to 80% of the testing budget [4]. For example, in Google, there are over 100 million test cases running each day [5]. To reduce the cost, the researchers have been actively developing regression test selection (RTS) [4]–[9], aiming to select tests that are affected by the software update changes. Similarly, consider the frequent update of video games that require much time to run all test cases, RTS is also necessary for game testing.

The general RTS usually includes two tasks [5]: 1) to identify which *software elements* can be executed by a given test case (*i.e.*, test dependency) and 2) to analyze the changed software elements. Then test cases that execute any changed software elements are selected. Existing RTS is difficult to be directly applied to video game software due to its unique characteristics. Specifically, a test case of a video game is mainly a sequence of actions that the player can perform, and it returns the state transitions from the game software. Often, many (sometimes even all) test cases can reach a same state (*e.g.*, a common game scenario). If this state is affected by the game changes, the conventional RTS techniques would select all the test cases. This is not affordable as running a game test can take tens of minutes to hours. Moreover, the game space is often very huge, and there are a large number of test cases to select from. To the best of our knowledge, there is still no effective RTS technique for game software, and the game companies mainly rely on ad-hoc strategies (*e.g.*, our collaborating company selects and runs the shortest test cases that can cover the basic tasks after the game update). Motivated by this, this paper aims to make an early attempt to investigate regression test selection for video game software.

To this end, we first perform a study to investigate the development and the evolution process in industrial games (Section II) to better understand the characteristics of the game development and the bugs introduced during game evolution.

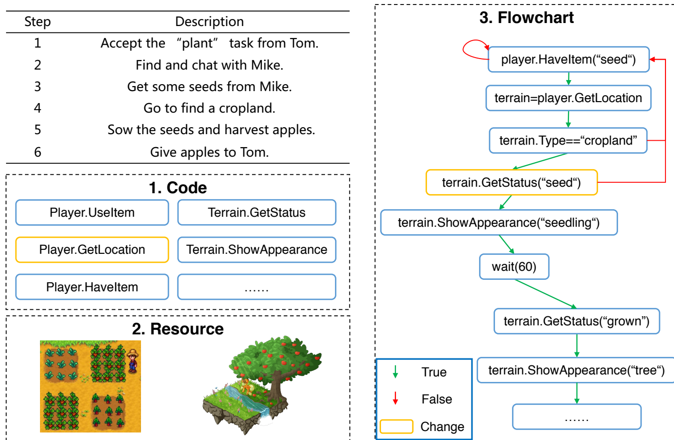

Fig. 1: A simplified example of a video game task

Specifically, we found that the game development often includes several stages: *game art resources* (*e.g.*, 2D and 3D art files), *game design* (*e.g.*, game sketches, storylines, game character features and interactions), and *code implementation* (*e.g.*, the basic function such as the use of a skill), which can cause regression issues. We manually investigated a total of 2,763 real-world bugs and analyzed the root cause of these bugs. We found that all of the above three stages in the game development can cause bugs during game evolution and most of the bugs (78.85%) are relevant to the game designing process, indicating that the changes in these three strategies should be considered in the RTS. In addition, we observe that a game can contain a large number of states, and the abstraction can be used to reduce the complexity and state space of testing.

To address the aforementioned challenge that all test cases could be selected using conventional RTS, we conduct another study to understand the *context sensitivity* of video game playing. We found that unlike traditional programs, the game playing manifest *Markov property* [10] in that its future actions usually depends on only the current state, but not the past history (*i.e.*, paths and previous states/actions that lead to the current state). For example, to reach a game state (*e.g.*, mission accomplished), players can use different strategies to play the game, which can finally reach the same game state. Based on this observation, in our work, we only select one test case that reaches a state, instead of covering all the test cases that come from different contexts before reaching the same state.

We propose a novel framework, named *GameRTS*, specially designed for regression testing of video games. In particular, given a new game software and a set of test cases, we first perform the abstraction and model them as a State Transition Graph (STG). We then develop static and dynamic analysis to establish the dependencies between test cases and various game software elements (*i.e.*, art resource files, design files and source code). Unlike the existing RTS techniques that calculate the dependencies between software elements and the whole test case, we propose to build the game test dependency between game files and abstract states/actions of the STG, which can further reduce the number of selected tests. When the game software is updated, a change impact analysis is performed on the game files between the two versions. Based on the test dependencies and the change information, we adopt a greedy-based selection strategy to select a small number of test cases (*i.e.*, traces in the STG) to cover the states and actions that are affected by the changes.

We implemented *GameRTS* and evaluated the tool on 10 tasks of a large-scale industrial game (including 1,429 commits between three versions). The experimental results demonstrate that 1) *GameRTS* can achieve better balance between practicality and safety. In most cases (except one), only 1 trace is selected, which can significantly reduce the testing time. Moreover, =*GameRTS* detects all the regression bugs. 2) Our fine-grained test dependencies can select much fewer state/action/traces than the file-level dependency (*i.e.*, 13.77%, 23.97%, 6.85%) and 3) *GameRTS* identified 2 critical bugs in the latest version of the game. In addition, *GameRTS* has been used in more than 15 games.

The main contributions of this paper include:

- We performed an empirical study to characterize the game evolution (on 2,736 real bugs) and its context sensitivity.
- We proposed a novel RTS framework that is more lightweight and practical. To the best of our knowledge, this is the first technique that performs RTS on a state transition graph. Particularly, the dependency analysis and selection are performed on the states and actions of the STG instead of the whole test, which can reduce the complexity.
- We implemented an instance of the framework *GameRTS*[1] for video game software. Without loss of generality, *GameRTS* can be extended to other software with GUI. A fined-grained change impact analysis is developed on different kinds of files, *i.e.*, resource files, game design files, and source code.
- We evaluated the effectiveness and efficiency of *GameRTS* on 10 tasks on a large-scale industrial video game across 3 versions. Moreover, *GameRTS* found 2 new regression bugs in the latest version.

II. An Empirical Study on Game RTS

A. Three Sources of Game Software

A typical video game development [12], [13] usually includes preparing art-effect resources(*e.g.*, the video and 3D model), designing gameplay rules (*e.g.*, storylines and characters), and achieving basic utility code (*e.g.*, maintaining the status of a player and facilitating network communications).

Definition 1: A *video game* can generally be represented as a 3-tuple (P, R, D), where P is a set of source code files that implement the basic functions, R is a set of art resource files, and D is a set of design files. A game software implements a set of game tasks, where each game task uses a flowchart in a design file to define the detailed process of the task.

[1]The detailed configuration and experimental data is available on our website [11].

TABLE I: Regression bugs in a video game

Priority	Art	Flowchart	Code
High	36	1,192	39
Normal	79	1,375	29
Low	1	12	0

Example: Figure 1 shows a simplified game task, as shown in the upper left corner: (1) the player first accepts the task from a *non-player character (NPC)* Tom; (2) the player finds another NPC Mike and starts a conversation with Mike; (3) the player receives some seeds by chatting with Mike; (4) the player finds some cropland to sow these seeds; (5) apples will ripen in a short period, and (6) the player harvests the apples and give them to Tom.

To implement the task, *art resources*, *design flowcharts* and *source code* are required as shown in Figure 1. Specifically, the code implements `Player` and `Terrain` classes. For example, `Player.UseItem` supports players to use their items. `Player.GetLocation` gets the location of a player on the map. The resource includes the game art provided by artists, *e.g.*, the apple tree and the map. The flowchart records the workflow of the task. First, the designer uses `Player.UseItem` to check whether the player has "seeds". Then, it calls *Player.GetLocation* to check whether the player finds some cropland; if so, the UI will show the seed sowing process via `terrain.ShowAppearance`. After 60 seconds, it shows the apple tree growing on the cropland. At each step of the flowchart, the corresponding code will be invoked and the art resources will be fetched. The flowchart combined with code and art resources is automatically compiled into the game executable.

The evolution of the game potentially involves the changes of all three sources. For example, in the *Flowchart* of Figure 1, the yellow box indicates that we add a new node `terrain.GetStatus("seed")` in the new version. This change fixes a game logic bug. In the game, after the players obtain the seeds, they can send the seeds to others (this step is defined in another game task and not shown in Figure 1). If it does not check the existence of seeds before growing them using `terrain.GetStatus("seed")`, the seeds will incorrectly grow in the terrain.

In Table I, we present the results of our empirical studies on regression bugs. We selected a large-scale industrial game as a subject. We collected *2,763* bugs, starting from August 2020, back to more than two years until June 2018. We worked with developers in the game company and manually studied which game file changes caused such bugs as well as the importance of the bugs, marked as *High*, *Normal* and *Low* priorities in the figure. The results show that the changes to three sources all can lead to bugs. To test the new game version for detecting such bugs, the company usually re-runs all tests or runs only some in an ad-hoc fashion for efficiency. To address this issue and improve the effectiveness, RTS techniques that can systematically select necessary test cases are needed.

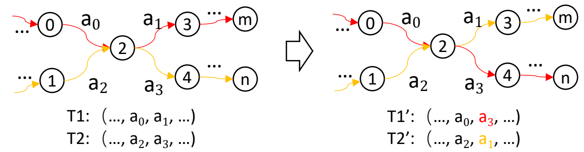

Fig. 2: Evaluation of Context Sensitivity

B. Game Execution Behaviors Manifesting Markov Property

Another challenge we found is that some game changes often affect most or even all test cases. For example, if we change the resource file that is used in a common state of all test cases, all test cases could be affected, which makes existing RTS techniques (that calculate dependencies between changes and the test case) not work well. Thus, we conducted an empirical study towards understanding the effect of the game context in game playing.

Informally, the Markov property of game execution means when different test cases reach a same state with different contexts, the different contexts in the same state have little effect on subsequent executions. For example, players may adopt different strategies to complete the same game quest and reach a same state, which usually has no impact on the following actions and states. Note that the context sensitivity will determine the granularity of regression test selection analysis. If the context is sensitive, we need a fine-grained dependency analysis and test selection such that test cases with different contexts can be selected. If insensitive, then we can consider only selecting minimal test cases that cover all changed states for efficiency.

The design of the empirical study is presented in Figure 2. There are two test cases (T_1 and T_2) that can reach the same state s_2 (called intersection state) from different playing histories (*i.e.*, different context). Consider T_1 that covers the state sequence $(\ldots, s_0, s_2, s_3, \ldots, s_m)$, we try to execute a sequence of actions (T_1') that include the actions of T_1 before the common state s_2 (context from T_1) and the following actions of T_2 after s_2. If T_1' can be replayed successfully (*i.e.*, T_1' can lead to the state transitions $0 \rightarrow 2 \rightarrow 4 \rightarrow n$), then we could determine the context of T_1 has little effect on the future execution (*i.e.*, insensitive). Similarly, we try to use the context of T_2 and replay the following actions of T_1 (*i.e.*, T_2'). Note that different strategies reaching the same state will have similar effects on subsequent executions, while the different strategies reaching the same status would be considered context-insensitive due to the similar final status. We measure the context sensitivity based on the success rate of the cross execution. The higher the success rate, the less sensitive the context.

We select all the test cases used in our evaluation (see Section VI) to identify all intersection states among them. Each intersection state has M predecessors and N successors where $M > 1$, denoted as (M, N) intersection. Finally, we collected an average of 119.8 (M, N) intersections (for each task), including 56.9 $(M, 1)$ intersections. We construct 500

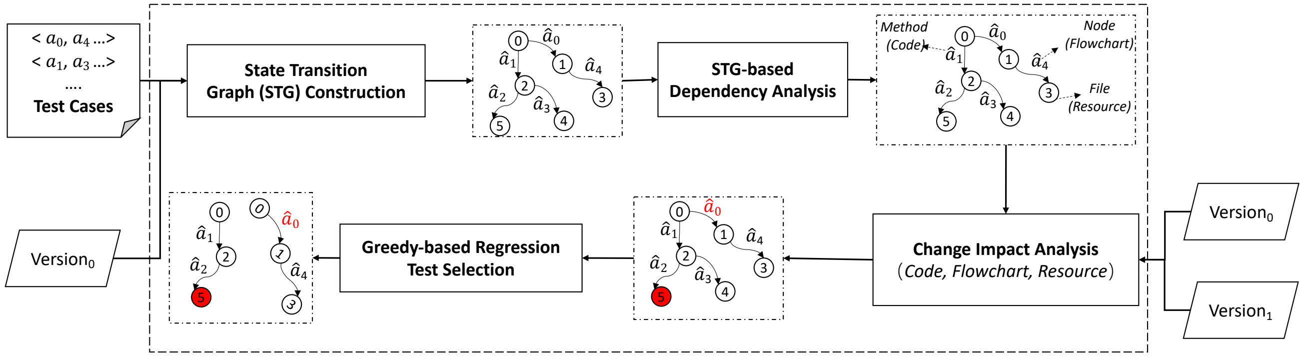

Fig. 3: The Overview of *GameRTS*

pairs of traces for the cross execution, where each pair has at least one intersection ($N > 1$). The final results show that the success rate of the cross execution is 100%, which indicates the context insensitivity of game execution. More detailed results can be found on our Website [11].

III. Problem Formulation and Overview

A. Problem Definition

We formulate the definition of regression test selection problem as follows:

Definition 2 (Game Regression Test Selection): Given a game software S, the modified version S', a given test suite T where each $t \in T$ is a sequence of actions and an abstraction function α that can be used to build a state transition graph g, the game regression test selection (RTS) is to find a test suite T' from the graph g, with which to test S'.

The algorithm of finding the test suite T' depends on the test goal. For example, in non-game software, RTS techniques usually consider two test goals [5]: *precision* to avoid selecting non-affected tests by game changes and *safety* to avoid missing any affected tests that can detect regression bugs. Based on our empirical results, we found that game changes are likely to affect all tests, *i.e.*, we have to select all tests for safety. Hence, game RTS is a trade-off between practicality and safety, especially for large-scale video games where running a test takes a very long time. In this paper, our goal is to design an effective and practical RTS framework for detecting regression bugs effectively and efficiently. Thanks to the Markov property of game execution (see Section II-B), we design a practical tool *GameRTS*, which is theoretically not safe but does not miss any known regression bugs.

B. Overview

Figure 3 shows the overview of *GameRTS* for video games. The inputs include a given set of test cases and two or more versions of the game. Each test case is a sequence of actions a player will take to navigate through the games in GUI. Our first step is to construct a *State Transition Graph (STG)* to represent the game. Typically, the number of actions and states can be very large when running a game, so the STG is built on the abstract states and transitions.

In the second step, we perform the *test dependency analysis* based on STG and link the states and transitions in STG to the corresponding game software files (*i.e.*, the art resource, the design flowchart, and the source code). The abstraction in STG leads to the small size of states and transitions, so we only need to calculate a small number of dependencies. *GameRTS* computes test dependency at different granularity for different types of files. For the art resources, we perform file-level dependency analysis, while for design files(*i.e.*, flowcharts), we map each state and the transition of the STG to the nodes and edges in the flowchart; and for source code, we perform the method-level analysis.

After a new version of game software ($Version_1$) is constructed, we analyze changes between the older version $Version_0$ and the new version $Version_1$ for the art resource files, the design flowchart and source code. Then, we leverage change impact analysis to identify those states and transitions in the STG that are affected by the changes (see the red nodes/transitions in Figure 3).

Based on the STG, we propose a greedy-based RTS technique with the guidance of the affected states and transitions to select as few test cases as possible. We re-run the selected test cases on the new version (*i.e.*, $Version_1$) to detect the regression bugs. For the test cases that are no longer feasible to the new version due to the changes, we applied the random exploration strategy from the last reached state. Finally, we update the STG of the old version to the new version and collect the new dependencies after running regression tests. The updated STG will be used in future versions (*e.g.*, $Version_2$).

IV. State Transition Graphs for Games

For regression test selection, a set of test cases are given for the first version. A test case for a game is a valid sequence of actions $(a_0, \ldots, a_n)$ that can be run in the game, leading to a sequence of state transitions $(s_0, \ldots, s_n, s_{n+1})$, where states are a representation of the game (*e.g.*, whatever information it can be seen on the screen) and actions are something the game player can perform to update game states (*e.g.*, moving and attacking). We use A_s to represent a set of actions that can be executed under a state s.

A state can be generally represented as a vector $< v_1, v_2, \ldots, v_n >$, where v_i is an attribute in the game, *e.g.*, items of the player and location. Essentially, a game can have a large number of states or even infinite states if the variable v_i is continuous (*e.g.*, health points). The state exploration problem increases the complexity of the regression analysis, *e.g.*, building the dependencies between states and changed game files. To reduce the complexity, we conduct the state and action abstraction to build a state transition graph. Note that the abstraction (*i.e.*, α in Definition 2) requires some domain knowledge of the target software, which could vary in different kinds of software. We emphasize that developing different abstraction techniques is orthogonal to our proposed RTS framework. In this paper, for the selected video games, we work together with the corresponding developers to design the abstraction strategies as follows.

State Abstraction. Generally, the states of video games usually can be classified as the environment state and the player state. For example, the player state mainly contains the player name, the player ID, the location, the health point, the magic point, the profession, the items and some buff. The environment state mainly contains some monsters, the NPC information similar with the player state (*e.g.*, name, ID and the location), the dropped items, the trigger in the map and *etc.*. The major abstraction includes three steps: 1) removing the unimportant attributes such as name and profession; 2) ignoring the environment information that is far from the current player (*e.g.*, NPCs who are 100 meters away are considered irrelevant information); 3) adopting the interval abstraction that discretizes the continuous values (*e.g.*, location information) into buckets, following the previous research [14]. For example, for the coordinate of the location, we divide the range to equal buckets (*e.g.*, 100 meters per bucket); For the health points, we abstract them as a binary value (*i.e.*, alive if greater than 0, otherwise dead). The states fall into the same bucket will be merged as an abstract state.

Action Abstraction. For the actions between two abstract states, we perform an abstraction that merges a sequence of concrete actions $(a_0, a_1, \ldots)$ as one abstract action $\hat{a}_0$. Consider the example in Figure 1, to find and start the conversation with Mike, the player needs to perform a sequence of actions such as moving on the map, looking at a person and confirming that it is Mike. We use an abstract action *goto* to represent this moving process. *goto* is an internal API[2] in the games that can automatically play the game. The abstract actions make the state/action space unnecessarily large and complex.

Example. Take Figure 1 as the example, the concrete state of the player includes many variables such as the detailed information of all NPCs, items, portal and current player in the map, which are too fine-grained. After the abstraction, one abstract state (in step 2) can be

```
{'env':{'npc':[{'id':1001, 'hp':alive},
    'loc':{'x': 125,'y': 961, 'z':0}]},
```

[2]Video games provide the inner functions that can help players to automatically accomplish some missions, *e.g.*, to find a specific NPC or fight against some monsters automatically.

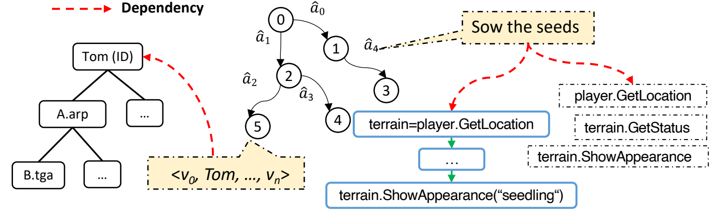

Fig. 4: Example of test dependencies

```
'player':{'hp':alive, 'bag':[3006, 3009],
   'loc':{'x': 0-100,'y': 700-800, 'z':0-100}}}
```

The concrete state can be found on the website [11]. In the abstract state, we ignore and simplify the environment information (*e.g.*, some NPCs and the attributes are ignored). The health points (hp) are abstracted as a binary (*i.e.*, die or alive). The location (x, y, z) of the player is abstracted into buckets, *e.g.*, 0-100. The location of the NPC is concrete value because this NPC does not move. The abstract actions can be:

```
[('talk', 1001), ('goto', 1001), ('useitem', 3006),...]
```

where *talk*, *goto* and *useitem* are inner APIs, which can largely remove the actions and states. For example, the player can find the NPC 1001 within only one abstract action by using *goto*. Without *goto*, we need a sequence of concrete actions (*i.e.*, moving up, down, left or right) taken from the player.

Definition 3 (STG): A State Transition Graph (STG) is a 5-tuple $(S, A, T, \hat{s}_0, F)$, where S is a set of abstract states, A is a set of abstract actions, $T : S \times A \to S$ is the transition function, $\hat{s}_0 \in S$ is an initial state (*e.g.*, starting the game playing) and F is a set of accepting states (*e.g.*, accomplishing the game or entering a stuck state).

V. STG-based Regression Test Selection

A. Test Dependency Analysis on STG

The goal of test dependency analysis is to determine for a given test case, which files or methods will be exercised by the test case. Here, we determine states and actions in the STG that can be affected by modified game files. In the STG, the states, which represent the status of the game at some points, are more related to the resource files. The actions, which represent the operations of players, are more related to the flowcharts and the code. The flowcharts and the code define the rules and the steps taken during the game playing. We develop a static-based method to calculate the state dependencies and a dynamic-based method to calculate the action dependencies. Figure 4 shows an example of test dependency constructed for Figure 1. The STG nodes and edges have been linked to the art resource files (on the left) and the design flowchart (on the right). We use this figure to explain more details of state and action dependencies in the following.

a) State Dependencies: We implement a static analysis tool to build the relationship of the art resource files based on their *def-use* relations. The relationship is hierarchical and can be represented using a tree. Each node of the tree is a file. The parent node uses the files from its children nodes.

For example, in Figure 4, the left tree shows that the model *Tom* is constructed using the resource files such as the model file (*e.g.*, *A.arp* is the model file of *Tom*) that is created using other files (*e.g.*, the image *B.tga*).

Each art file has a unique ID (*e.g.*, 1001 for Tom) that is used by the game software. We use this ID to establish the dependencies with the states in the STG. For example, in Figure 4, suppose that the state 5, shown in the middle, is $< v_0, Tom, \dots, v_n >$. This state has a dependency on the model *Tom*. Any changes in the art files under the tree of *Tom* will mark this state as affected by the changes.

b) Action Dependencies: For actions, we execute the existing test cases and dynamically monitor the code and the flowchart to collect the set of functions in code and the set of nodes in the flowchart, which are accessed during the execution of an action. To collect the affected functions, we implemented the *flame graph* [15] for the code instrumentation, *e.g.*, at the start of the constructor, the start of the static method and the start of the class method, which can record the methods called. Note that the dependency analysis conducted with different granularities (*e.g.*, method-level or file-level) could affect the precision (*i.e.*, the number of tests selected).

To monitor the activation of nodes in the flowchart, we instrumented the code, which is automatically generated based on the flowchart. When an action is executed, the instrumentation will return which nodes are activated in the flowchart. Note that, an action can activate multiple nodes of a flowchart. Consider the example in Figure 4, when we execute the action "sow the seeds", we observe that the nodes *terrain = player.GetLocation*, *terrain.Type == "cropland"*, *terrain.GetStatus("seed")* and *terrain.ShowApperance("seeding")* are activated. The functions *Player.GetLocation*, *Terrain.GetStatus*, *Terrain.ShowAppearance* are invoked.

Definition 4 (STG-based Test Dependency): Given a game (P, R, D) and the corresponding STG $(S, A, T, \hat{s}_0, F)$, we define its test dependencies as three functions $\{f_R, f_P, f_D\}$, where $f_R : S \rightarrow R$ represents the dependencies between the states and the art resource files, $f_P : A \rightarrow P_m$ and $f_D : A \rightarrow D_n$ represent the dependencies between the actions and the methods of game software (*i.e.*, P_m) and the flowchart nodes (*i.e.*, D_n), respectively.

B. Change Impact Analysis for Games

Given two versions of game software, we identify their changes in terms of the three types of game files. We perform change analysis for different types of files. Specifically, for art files, we first identify which files are added, deleted, and changed based on the commit information. The dependencies of these files are marked in the tree we constructed. For the code changes, we apply the Lua Parser [16] to obtain the abstract syntax tree (AST) of the Lua program. From the AST, we can obtain the methods as well as line numbers of each method. A flowchart is saved into a JSON file. For a changed file, we parse the nodes of each flowchart, and compare the corresponding nodes (*i.e.*, the old version and the new version) to identify the changed ones.

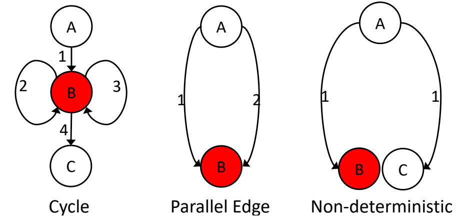

Fig. 5: Challenges of test case selection

Finally, we identify all the changes, annotated as $\delta = \{\Delta_R, \Delta_P, \Delta_D\}$, where Δ_R, Δ_P and Δ_D represent the changed art resource files, methods and flowchart nodes. With the change information (Δ) and the test dependencies (see Definition 4), we calculate the affected states C_S and transitions C_A as follows:

$$C_S = \{s | \forall s \in S \wedge f_R(s) \cap \Delta_R \neq \emptyset\}$$

$$C_A = \{a | \forall a \in A \wedge (f_P(a) \cap \Delta_P \neq \emptyset \vee f_D(a) \cap \Delta_D \neq \emptyset)\}$$

Intuitively, the states and actions are marked as affected if their test dependencies (*i.e.*, the corresponding software files) are changed. For example, for each state s, $f_R(s)$ represents the corresponding resource files that are used in the state. If some of these files are changed in the new version (*i.e.*, $f_R(s) \cap \Delta_R$ is not empty), then s is the affected state.

C. Regression Test Selection and Execution

Considering that 1) the game software can be very frequently updated, and 2) the game playing is time-consuming (a test case often takes several dozens of minutes), *GameRTS* is a trade-off between practicality and safety (see more discussions in Section III). We select test cases to cover a changed state or an action at least once based on the context-insensitivity (see Section II-B). The experimental result shows all the known regression bugs could be found by *GameRTS*, which indicates a good balance between practicality and safety.

1) Algorithms for test selection: In Figure 5, we show the challenges of selecting test cases from the STG, where we assume state B is affected by the change. First, there exist cycles in the STG, which can lead to infinity exploration that we want to avoid in the RTS. In regression testing, we require to traverse the cycles once all the affected states and actions are covered. For the *parallel edges* (one state can transit to another state via two different actions) shown in the middle of Figure 5, we only require to select one of the parallel edges. Finally, the game has non-determinism in that the same action may transit to two different states (*e.g.*, a *fighting* action could defeat the monster successfully or unsuccessfully). In this case, we will repeat the action several times and try to reach the target state (*e.g.*, state B).

We present the details of our RTS approach in Algorithm 1. The inputs of the algorithm include the set of affected states C_S, the set of affected actions C_A, and the STG of the

Algorithm 1: Regression Test Selection for Games

```
input   : C_S, C_A: the affected states and actions
          STG(S, A, T, s_0, F): the STG of the game
output  : T: a set of selected test cases
1  t = (), t.selected = false, T = ∅;
2  RTS(s_0, C_S, C_A, t, ∅, T);
3  return T;
4  Procedure RTS(s, C_S, C_A, t, visited, T)
5      if s ∈ C_S then
6          C_S := C_S \ {s};
7          t.selected := true;
8      if s is an accepted state then
9          if t.selected then
10             T := T ∪ {t};
11         if C_S ≠ ∅ ∨ C_A ≠ ∅ then
12             t' = (), t'.selected = false;
13             RTS(s_0, C_S, C_A, t', ∅, T);
14         else
15             return T;
16     for (s, a, s') ∈ s.next() do
17         if (s, a, s') ∉ visited then
18             visited := visited ∪ {(s, a, s')};
19             if a ∈ C_A then
20                 C_A := C_A \ {a};
21                 t.selected := true;
22             t.append(a);
23             RTS(s', C_S, C_A, t, visited, T);
```

game. The output is a set of test cases that cover the affected states and actions at least once. The method RTS performs a recursive traversal over the STG.

Specifically, during the traversal, if the current state s is an affected state (Line 5), we remove it from the target set C_s and mark that the current test case should be selected (Line 6-7). If the current state s is an accepted state, and the test case t covers some affected states or actions, we add it to the selected test set T (Line 10). If there are still some states or actions that are not covered, we continue to identify other test cases (Line 11-2). If all states or actions are covered, we have selected a set of test cases T (Line 15).

If the current state is not an accepted state, we select an action under the current state s. $s.next()$ returns an unordered set of transitions starting from s (Line 16). At each iteration, we select one transition (s, a, s') if it is not visited before (Line 17). If a is an affected action, we remove it from the target set C_A and mark t as a "selected" test case (Line 19-21). We add the transition into the current test and continue the traversal (Lines 22-23).

2) Test Execution and STG Update: After the RTS is completed, we execute the selected test cases and update the STG. Since the RTS is performed on the STG constructed for the old version, some of the selected transitions are no longer feasible in the new version, which will be removed. On the other hand, after exercising a changed state, the new version of the game may lead to a completely new state, which will be added to the STG. If some affected states/actions are unreachable with the selected test cases, we rerun the RTS algorithm on the updated STG to select new test cases.

VI. Evaluation

To evaluate the effectiveness and efficiency of *GameRTS*, we investigate the following research questions[3]:

- **RQ1:** Can *GameRTS* effectively detect the regression bugs?
- **RQ2:** Can *GameRTS* save computational resources?
- **RQ3:** What is the coverage of changed states and actions?
- **RQ4:** How do different dependency granularities (*e.g.*, method-level vs file-level dependency) impact the results? How do changes in three software files contribute to *GameRTS*?
- **RQ5:** How useful is *GameRTS* in detecting new regression bugs?

A. Experimental Setup

1) Implementation: We have implemented *GameRTS* including about 17,400 lines of Python code and 1,100 lines of Lua code. The Lua side performs the STG-based dependencies analysis (*e.g.*, the code instrumentation) while the Python code performs other analyses including the STG construction, change analysis, test selection, test case replay, *etc.*.

2) Subject: We selected three recent versions of a large-scale commercial game in our evaluation (denoted as V_0, V_1 and V_2), collected on January 19, 2022, January 26, 2022 and March 08, 2022, respectively. We also used a newly developed version on March 15, 2022, where we did not know any regression bugs to evaluate whether our techniques can find new real-world regression bugs. This game is a very popular online role-play game, which has more than 30 million registered users, more than 1.7 thousand tasks, more than 160 thousand art files and millions of lines of code. There are 313 commits from V_0 to V_1, and 1,116 commits from V_1 to V_2. It usually takes days even months for a player to accomplish all the tasks in a video game. Thus, game testing is usually performed task by task rather than the end-to-end testing. We selected 10 tasks from the game for evaluation. The game and the tasks are selected based on that: 1) this game is quite popular and large one (the top 3 of more than 100 commercial games in the partner company); 2) we can get the authorization and support from the game company, *e.g.*, accessing the development/testing environment, defining the abstraction strategies, developing game specific codes (instrumentation and change analysis) with the developers, and confirming bugs; and 3) the selected tasks belong to the main quests and cover comprehensive action patterns of players in the game.

For the regression testing of each task, the game provider maintains a number of test cases, which are from recorded manual tests, existing game playing scripts and automatically generated tests. As each test case takes about 10-20 minutes

[3]Due to the space limit, more detailed experimental results (*e.g.*, setting, STG size, STG visualization and videos) can be found in [11].

to run, we cannot select all test cases for the evaluation (that requires more than 20,000 CPU-hours to run). Hence, for each task, we select 50 test cases as the initial test suite of *GameRTS* based on the importance of these test cases. Specifically, all recorded manual tests and scripts were prepared by professional testers, which are of high quality, and thus selected. Then a certain number of automatically generated tests (*e.g.*, by reinforcement learning), considered low quality, are selected such that the total number is 50.

3) Metrics and baselines: To answer RQ1 and RQ2, we use the metrics of bugs found and the computation time. We select three baselines: 1) running all the test cases (denoted as All); 2) running randomly selected test cases (denoted as Random). To keep impartiality, we select the same number of test cases to compare random strategy and *GameRTS*, and 3) manual testing that is performed by 5 game testers in the company (denoted as Human). We select manual testing as the baseline because it is one of the major methods for regression testing used in the company. Similar to random strategy, each game tester plays the game a number of times which is the same as the number of test cases selected by *GameRTS*. Note that all the professional testers are employed by the collaborating company with zero prior knowledge about the study, and are well-trained to test games with different playing strategies. For RQ4, we will compare the number of test cases selected under different dependency granularities or with different changed files.

B. RQ1 and RQ2: Bugs Detected and the Cost

In this experiment, we evaluated how many regression bugs can be detected by the test cases selected by different methods. We replay such tests in the real game environment to monitor whether bugs are triggered and recorded the time cost. Note that the game environment provides the oracle for detecting these bugs including some logical bugs, crashes and stuck bugs.

Table II shows the results of the two revisions. Column *Time* shows the end-to-end time used by different methods (minutes or hours). Note that, the time of *GameRTS* includes the change analysis, the test selection from STG, the execution and the dependency collection. Column *#Bug* shows the total number of known regression bugs in the corresponding version. Column *All* shows the results of selecting all initial test cases. Column *Random* and Column *Human* show the results of the random strategy and manual testing, respectively.

Note that, *GameRTS* selected two test cases for Task 1 ($V_1 \rightarrow V_2$). For other tasks in the update, *GameRTS* selected only *one* test case that can cover all changed files. For the fair comparison, *Random* selects the same number of test cases with *GameRTS* from the initial test set. To reduce the randomness, we repeat 10 times for *Random* strategy to calculate the average results. For *Human*, each tester plays each task 3 times and we calculate the average results of all repetitions of all testers.

The number of bugs (Column *#Bug*) under *All* can be considered as the total number of known bugs introduced during the game update. Overall, by comparing the results under *All-#Bug* and *GameRTS-#Bug*, we can observe that *GameRTS* does not miss any bug by only selecting one test case for each task in each revision, indicating the effectiveness of *GameRTS*. As for the time cost, *GameRTS* only needs an average of 10.12 minutes per task while executing all test causes needs an average of 10.25 hours per task, indicating its practicality in industrial video games. Most of the time spent in *GameRTS* is the *execution* of the selected test case while other analysis takes less than 10 seconds.

Consider the other two baselines *Random* and *Human* that execute the same number of test cases with *GameRTS*, they can also capture some regression bugs but much less than *GameRTS* (40.0% for *Random* and 14.3% for *Human*). It demonstrates the effectiveness of *GameRTS*. In terms of computation time, although all these three methods select the same number of test cases, the average time costs are very different. Specifically, *GameRTS* is much faster than *Random*, *e.g.*, the average time (from two revisions) of *Random* and *GameRTS* are 16.94 minutes and 10.17 minutes, respectively. Our in-depth analysis shows that *GameRTS* selects shorter trace than *Random* because: 1) *GameRTS* only executes when there is a cycle in STG (see Figure 5) and 2) *GameRTS* terminates the execution once all affected states/actions are covered (see Line 15 in Algorithm 1).

Compared to others, *Human* triggers the smallest number of bugs. We observe that the length of traces from *Human* is shorter than *GameRTS* and *Random* since game testers are unknown of the changes (*e.g.*, source code) and adopt a smarter strategy to complete tasks. Therefore, it reduces the exploration diversity. Note that, although relatively shorter, testing by *Human* takes more time than *GameRTS* because human clicks have some delays and game testers may watch some animations that are skipped by automatic game replay.

> **Answer to RQ1 & RQ2**: *GameRTS* achieves better balance between practicality and safety. Specifically, it detects *all* known regression bugs and significantly saves computational resources compared to selecting all test cases.

C. RQ3: State and Action Coverage

Table III shows the state and action coverage of *GameRTS*. Column *TC* shows the total number of changed states/actions. Column *Cov.* shows the number of changed states/actions that can be covered by the test cases selected from *GameRTS*. Column *New* shows the number of new states introduced by the new version.

We can observe that some changed states and actions cannot be covered because the update makes them infeasible. For example, from V_0 to V_1, 92.3% of changed states and 97.6% of changed actions are covered. On the other hand, some new states can be added due to the update. During the test case execution, we also add the newly explored states in STG. For example, we added 100 states in total from V_0 to V_1.

TABLE II: Comparative results on the number of regression bugs triggered and the computation time

ID	$V_0 \rightarrow V_1$								$V_1 \rightarrow V_2$							
	GameRTS		All		Random		Human		*GameRTS*		All		Random		Human	
	Time(m)	#Bug	Time(h)	#Bug	Time(m)	#Bug	Time(m)	#Bug	Time(m)	#Bug	Time(h)	#Bug	Time(m)	#Bug	Time(m)	#Bug
1	14.8	2	15.1	2	14.4	0.8	17.2	2	15.6	1	17.2	1	14.6	1.0	18.4	0
2	10.1	3	18.5	3	10.7	1.0	11.8	0	10.0	3	19.5	3	13.3	1.3	10.0	0.8
3	10.0	3	15.6	3	15.5	1.8	8.6	0	9.6	1	17.9	1	15.3	0.0	11.9	0
4	10.0	1	6.4	1	15.4	1.0	9.7	0	10.0	1	9.5	1	17.1	1.0	8.8	0
5	10.0	2	8.0	2	19.0	0.0	15.6	0.6	10.1	2	12.3	2	21.0	0.0	14.8	0
6	9.4	1	5.7	1	15.8	1.0	11.3	0	10.0	1	6.5	1	20.4	0.0	10.6	0
7	10.9	1	7.5	1	14.9	0.0	8.4	0	9.7	2	8.5	2	15.5	0.5	10.1	0
8	5.8	1	12.8	1	15.2	0.0	4.9	0	6.8	0	9.7	0	18.4	0.3	5.9	0
9	10.0	1	5.6	1	20.4	0.5	15.1	0	10.0	1	6.3	1	20.3	1.0	14.8	0.6
10	10.2	1	7.5	1	20.8	0.0	15.0	0	10.3	0	12.9	0	20.8	0.0	15.2	0
Total	101.2	16	102.5	16	162.1	6.1	117.6	2.6	102.1	12	120.3	12	176.7	5.1	120.4	1.4

TABLE III: Coverage results of *GameRTS*

ID	V0->V1					V1->V2				
	State			Action		State			Action	
	TC	Cov.	New	TC	Cov.	TC	Cov.	New	TC	Cov.
1	34	31	11	50	50	30	30	1	109	102
2	32	32	22	22	21	31	31	18	54	51
3	12	12	1	10	10	37	36	18	99	99
4	30	29	8	29	29	29	29	6	106	105
5	28	26	1	42	42	27	24	5	59	55
6	22	19	2	28	28	21	21	21	62	62
7	32	32	26	23	23	30	30	16	57	57
8	7	6	2	19	14	6	5	2	70	67
9	19	14	18	21	21	17	15	4	75	75
10	4	2	31	7	7	4	4	9	68	68
Total	220	203	122	251	245	232	225	100	759	741

Answer to RQ3: An average of 94.7% states and 97.6% actions are covered by our selected test cases. Due to the update, some changed states and actions become infeasible and therefore cannot be covered. Meanwhile, the test cases triggered some new states in the new game version.

D. RQ4: Impact of Dependency Granularity and Sources

1) The Choice of Dependency Granularity: In *GameRTS*, we build the test selection based on the art files, the methods of the code and the nodes of the flowcharts. Following the existing work [4], [5], we select the file-level test dependencies as the baseline. Specifically, in the file-level baseline, we compute test dependency using more coarse information, *i.e.*, art file, code file and flowchart file. In Table IV, we report the number of states and actions selected from V_0 to V_1. Results from V_1 to V_2 can be found on our website [11].

Under columns *AllFile*, we observe that *all* states (1,082) and actions (968) in the STG are selected for each game. However, under Column *GameRTS*, we only need to select 149 states (13.77%) and 232 actions (23.97%). Regarding the number of test cases selected by Algorithm 1, in Table IV, the file-level selection selects a total of 146 test cases to cover all states and actions while our method only selects 10 test cases (6.85%). Thus, the fine-grained dependencies can help largely reduce the overhead of the testing.

Answer to RQ4-1: The file-level dependency is too coarse for regression testing of the game software. The fine-grained test dependency in *GameRTS*, on the other hand, selected much fewer states, actions and traces.

2) Effect of different files: We evaluate how different types of game files (*i.e.*, the art files, the source code, and the flow chart) impact the results of the RTS. Specifically, we used the art files, the methods in source code and the nodes of flowcharts to perform the RTS separately. The columns *Art, Meth. and Node* in Table IV show the respective results.

The results show that using only one type of file tends to select fewer states, actions and test cases than using all of them. The number of test cases selected by using only one type of file (*i.e.*, *Art, Meth. and Node*) is 1, which is omitted from the Table IV. In addition, as art resource files only affect the states while methods and nodes only affect the actions, all states in *GameRTS* are selected based on the art files, *i.e.*, the numbers of states under Column *GameRTS* and Column *Art* are the same. We show the number of regression bugs captured by the selected tests (shown in Column *#Bug*). It is not surprising that using only one type of file (*i.e.*, resource file, design file, and source code) can miss some bugs since changes on other files are missing.

Answer to RQ4-2: Art files only affect the states while code files and flowcharts affect the actions. Using only one type of file misses some bugs.

E. RQ5: Detecting Real-world Bugs

To evaluate the usefulness of *GameRTS*, we applied *GameRTS* in a newly developed version (March 16, 2022) by performing RTS between V_2 and the selected version. We totally found 2 critical issues, which were confirmed by developers. We summarized the 2 bugs [11] as follows:

1) *Bug 1 (Design Bug)*. The previous version defines the task that the players have to kill 10 monsters. In the new version, to improve the playability, the designer modified the types of monsters, where one of them can become invisible. This invisible monster can only be killed by Area of Effect (AOE) skills. If players do not use AOE skills, the task can never be completed. The developers fix this bug by removing the invisible monster.
2) *Bug 2 (Source Code Bug)*. There is one step in the task that requires the QTE checking (Quick Time Event)[4]. The previous version only checks whether a key is pressed within the specified time. To increase the difficulty, the designer changes the flow, *i.e.*, a key must be pressed 9 times. The buggy checking code is $press_num == n$,

[4]https://en.wikipedia.org/wiki/Quick_time_event

TABLE IV: Results of RTS with different test dependencies ($V_0 \rightarrow V_1$)

Task ID	#State					#Action					#Test Cases		#Bug		
	GameRTS	AllFile	Art	Meth.	Node	*GameRTS*	AllFile	Art	Meth.	Node	*GameRTS*	AllFile	Art	Meth.	Node
1	39	104	39	0	0	45	88	0	42	3	1	17	1	2	1
2	6	111	6	0	0	34	121	0	33	2	1	16	1	2	0
3	9	69	9	0	0	30	137	0	23	7	1	9	1	3	1
4	6	65	6	0	0	3	148	0	3	0	1	10	1	0	0
5	3	145	3	0	0	10	111	0	10	4	1	16	0	2	1
6	5	66	5	0	0	11	73	0	11	0	1	18	0	1	0
7	30	214	30	0	0	26	107	0	23	3	1	16	1	0	0
8	17	69	17	0	0	13	53	0	13	0	1	11	0	1	0
9	28	185	28	0	0	56	78	0	50	6	1	17	1	1	0
10	6	54	6	0	0	4	52	0	4	0	1	16	1	1	1
Total	149	1082	149	0	0	232	968	0	212	25	10	146	7	13	4

where n is equal to 9 in this version. However, when a key is frequently pressed, $press_num$ can be greater than n, which makes the $press_num == n$ not satisfied. Developers have changed the checking as $press_num >= n$. This bug is difficult to be triggered by players or game testers since humans usually do not click a key so quickly.

For the comparison, we also run the random strategy that selects the same number of tests with *GameRTS* (for 10 times) and the no-selection strategy (using all tests). The average number of bugs identified by random strategy is 0.2 while using all tests can also identify the two bugs but take 100+ hours. The results demonstrate the usefulness of *GameRTS*.

VII. Threats to Validity

One threat to validity is the subjects selected in our experiments. To mitigate it, in this paper, we selected multiple tasks in a commercial game, a total of 7,245 commits. Randomness is also a threat to the validity, so we repeat multiple times to mitigate the threat. Another threat to validity is the construction of the STG, which depends on the initial test set. To mitigate this problem, we selected different types of test cases as the initial test set, *e.g.*, the recorded manual tests, the scripts and other automatically generated tests.

A further threat lies in the context-insensitivity of the game. If a new application is context-sensitive, then we need to select more test cases from STG to cover the changed states and actions that may lead to different execution behaviors of the games. This is a tradeoff we can explore between safety and practicality of the regression testing.

VIII. Related Work

A. Regression Testing

Since re-running the entire test suite is time-consuming, various approaches have been studied to reduce the cost of regression testing [17] such as regression test selection [4], [5], [8], [18]–[20], test suite minimization [21]–[26] and test case prioritization [27]–[30]. In addition, there are some techniques [31]–[36] that focus on test case generation.

1) Traditional RTS.: Xu *et al.* [20] proposed regression testing of Aspect-oriented software. They proposed a control-flow representation of Java programs and a two-phase graph traversal algorithm to identify dangerous edges that may lead to semantic differences. Then a set of tests that can capture such differences are selected. Zhang *et al.* [18] detected the software changes at the method-level and select test cases that execute such changed methods. Gligoric *et al.* [4] proposed EKSTAZI, a lightweight RTS technique based on the file dependencies. Compared with the basic-block level and method-level RTS techniques, file-level RTS is more efficient in the dependency calculation but may select more test cases. Zhang [5] proposed a hybrid test selection technique that combines the strengths of RTS at different granularity, *i.e.*, the combination of the method-level and file-level.

2) RTS on GUI-based Applications: Although there are few RTS works on game software, there have been some techniques proposed for GUI-based applications, *e.g.*, Android applications and Web applications. DetReduce [37] proposes a test minimization algorithm (works in a single version) by identifying and removing some common forms of redundancies introduced in the GUI testing tools. To enable the RTS on Android applications, ReTestDroid [38] proposes a precise interprocedural control flow graph (ICFG) that can handle the asynchronous tasks, the life cycle of fragments and the native code. The ICFG allows fine-grained dependency analysis, making more precise RTS. Qadroid [39] proposes the event-aware regression selection tool for Android apps TestSage [40] proposes a function-level dynamic RTS technique for Web applications. It uses two-pass execution (a same test is run in non-instrumented system and instrumented system) to improve the efficiency. Nakagawa *et al.* [41] proposed the method-level RTS for Web applications and WebRTS [42] proposes a file-level RTS technique that tracks file-level dependencies in parallel and support distributed Web applications.

The existing RTS techniques in traditional software and GUI-based applications are hard to be used in video games. The major challenges are: 1) running a test case in game software usually takes longer time (*e.g.*, tens of minutes to hours) and 2) it is often the case that most or all of the tests could be affected by the changes from different files. The existing RTS techniques could select many tests, making them impractical. The biggest difference in our work is that we build the STG to represent test cases of game software, and calculate the hybrid dependencies between the states/actions (instead of whole tests) and software files, which can significantly reduce the complexity.

B. Game Testing

Due to the complex interactions between games and users, existing automatic testing techniques are ineffective in testing games. Some works are conducted to analyze the characteristics of game software [43]–[45]. In addition, there have been some techniques proposed for automated game testing. Iftikhar *et al.* [46] proposed a model-based testing approach for automated testing of platform games. Zheng *et al.* [14] proposed Wuji, an automated testing technique for testing game software. Specifically, evolutionary deep reinforcement learning is adopted to train policies that can explore more wide and deep states of the game. Different from our work, these techniques mainly focus on general-purpose game testing on a single version instead of regression testing.

However, to the best of our knowledge, there is little work for RTS on the evaluation of game software. Wu *et al.* [3] recently propose to generate test cases targeting the different behaviors of two versions of a game. The DRL is used to train the difference-aware policy by designing the divergence rewards. The difference is that it works on the test case generation while our work mainly focuses on the regression test selection.

IX. Conclusions

Automated game testing is still a challenging task, especially on large-scale video games. This paper first performed a study on game development, game evolution and the root causes of the regression bugs. Based on the understanding, we proposed the first regression testing framework including how to represent test suites of games and how to select regression tests. The experimental results demonstrated that *GameRTS* is more effective than the strategies of random selection, selecting all tests and the file-based regression test selection. In the future, we plan to extend *GameRTS* to other software with GUI (*e.g.*, Android and Web applications), which requires interactions between users and the software. Without loss of generality, for the new software, we need to design the abstraction function to construct the STG and develop the test dependency analysis based on the corresponding software elements (*e.g.*, code, resource).

Acknowledgment

This work was partially supported by the Ministry of Education, Singapore under its Academic Research Fund Tier 1 (21-SIS-SMU-033), National Natural Science Foundation of China (62227805, 62072398), SUTD-ZJU IDEA Grant for visiting professors (SUTD-ZJUVP201901), National Key R&D Program of China (2020AAA0107700), Alibaba-Zhejiang University Joint Institute of Frontier Technologies, National Key Laboratory of Science and Technology on Information System Security (6142111210301), State Key Laboratory of Mathematical Engineering and Advanced Computing, and by Key Laboratory of Cyberspace Situation Awareness of Henan Province (HNTS2022001). We thank our industrial research partner NetEase, Inc., especially the Fuxi AI Lab and Game Testing Department of Leihuo Business Groups for their support with the experiments and constructive discussion.

References

[1] A. L. MAG, "Video games now part of daily life," Feb. 2019. [Online]. Available: https://www.mediametrie.fr/en/video-games-now-part-daily-life
[2] J. Culp, *Online Gaming Safety and Privacy*. The Rosen Publishing Group, Inc, 2013.
[3] Y. Wu, Y. Chen, X. Xie, B. Yu, C. Fan, and L. Ma, "Regression testing of massively multiplayer online role-playing games," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 692–696.
[4] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 211–222. [Online]. Available: https://doi.org/10.1145/2771783.2771784
[5] L. Zhang, "Hybrid regression test selection," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, 2018, p. 199–209.
[6] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Softw. Test. Verif. Reliab.*, vol. 22, no. 2, p. 67–120, 2012.
[7] L. C. Briand, Y. Labiche, and G. Soccar, "Automating impact analysis and regression test selection based on uml designs," in *International Conference on Software Maintenance, 2002. Proceedings.*, 2002, pp. 252–261.
[8] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi, "Regression test selection for java software," in *The 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and App.*, ser. OOPSLA '01, 2001, p. 312–326.
[9] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, no. 2, p. 173–210, Apr. 1997. [Online]. Available: https://doi.org/10.1145/248233.248262
[10] I. Borovikov, J. Harder, M. Sadovsky, and A. Beirami, "Towards interactive training of non-player characters in video games," *arXiv preprint arXiv:1906.00535*, 2019.
[11] GameRTS. [Online]. Available: https://sites.google.com/view/gamerts
[12] D. Liming and D. Vilorio, "Work for play: Careers in video game development." *Occupational Outlook Quarterly*, vol. 55, no. 3, pp. 2–11, 2011.
[13] A. S. Falim and J. Prestiliano, "The use of board games as learning media of project time management," *Journal of Nonformal Education*, vol. 4, no. 1, pp. 69–78, 2018.
[14] Y. Zheng, X. Xie, T. Su, L. Ma, J. Hao, Z. Meng, Y. Liu, R. Shen, Y. Chen, and C. Fan, "Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 772–784.
[15] C.-P. Bezemer, J. Pouwelse, and B. Gregg, "Understanding software performance regressions using differential flame graphs," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 535–539.
[16] O. Scholdström, "Lua parser," May 2020. [Online]. Available: https://github.com/fstirlitz/luaparse
[17] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software testing, verification and reliability*, vol. 22, no. 2, pp. 67–120, 2012.
[18] L. Zhang, M. Kim, and S. Khurshid, "Localizing failure-inducing program edits based on spectrum information," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2011, pp. 23–32.
[19] H. Hemmati and L. Briand, "An industrial investigation of similarity measures for model-based test case selection," in *2010 IEEE 21st International Symposium on Software Reliability Engineering*. IEEE, 2010, pp. 141–150.
[20] G. Xu and A. Rountev, "Regression test selection for aspectj software," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 65–74.

[21] J. Chen, Y. Bai, D. Hao, L. Zhang, L. Zhang, and B. Xie, "How do assertions impact coverage-based test-suite reduction?" in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2017, pp. 418–423.

[22] D. Di Nardo, N. Alshahwan, L. Briand, and Y. Labiche, "Coverage-based regression test case selection, minimization and prioritization: A case study on an industrial system," *Software Testing, Verification and Reliability*, vol. 25, no. 4, pp. 371–396, 2015.

[23] D. Hao, L. Zhang, X. Wu, H. Mei, and G. Rothermel, "On-demand test suite reduction," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 738–748.

[24] J. A. Jones and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," *IEEE Transactions on software Engineering*, vol. 29, no. 3, pp. 195–209, 2003.

[25] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov, "Balancing trade-offs in test-suite reduction," in *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 2014, pp. 246–256.

[26] A. Shi, T. Yung, A. Gyori, and D. Marinov, "Comparing and combining test-suite reduction and regression test selection," in *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, 2015, pp. 237–247.

[27] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei, "A unified test case prioritization approach," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, pp. 1–31, 2014.

[28] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry, "An information retrieval approach for regression test prioritization based on program changes," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 268–279.

[29] K. Zhai, B. Jiang, and W. Chan, "Prioritizing test cases for regression testing of location-based services: Metrics, techniques, and case study," *IEEE Transactions on Services Computing*, vol. 7, no. 1, pp. 54–67, 2012.

[30] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei, "Bridging the gap between the total and additional test-case prioritization strategies," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 192–201.

[31] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury, "Regression tests to expose change interaction errors," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 334–344. [Online]. Available: https://doi.org/10.1145/2491411.2491430

[32] H. Palikareva, T. Kuchta, and C. Cadar, "Shadow of a doubt: Testing for divergences between software versions," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 1181–1192.

[33] G. Yang, S. Person, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 1, Oct. 2014. [Online]. Available: https://doi.org/10.1145/2629536

[34] A. Orso and T. Xie, "Bert: Behavioral regression testing," in *Proceedings of the 2008 International Workshop on Dynamic Analysis: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, ser. WODA '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 36–42. [Online]. Available: https://doi.org/10.1145/1401827.1401835

[35] W. Jin, A. Orso, and T. Xie, "Automated behavioral regression testing," in *2010 Third International Conference on Software Testing, Verification and Validation*, 2010, pp. 137–146.

[36] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "Nezha: Efficient domain-independent differential testing," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 615–632.

[37] W. Choi, K. Sen, G. Necul, and W. Wang, "Detreduce: minimizing android gui test suites for regression testing," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 445–455.

[38] B. Jiang, Y. Wu, Y. Zhang, Z. Zhang, and W.-K. Chan, "Retestdroid: towards safer regression test selection for android application," in *2018 IEEE 42nd annual computer software and applications conference (COMPSAC)*, vol. 1. IEEE, 2018, pp. 235–244.

[39] A. Sharma and R. Nasre, "Qadroid: regression event selection for android applications," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 66–77.

[40] H. Zhong, L. Zhang, and S. Khurshid, "Testsage: Regression test selection for large-scale web service testing," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2019, pp. 430–440.

[41] T. Nakagawa, K. Munakata, and K. Yamamoto, "Applying modified code entity-based regression test selection for manual end-to-end testing of commercial web applications," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2019, pp. 1–6.

[42] Z. Long, Z. Ao, G. Wu, W. Chen, and J. Wei, "Webrts: A dynamic regression test selection tool for java web applications," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 822–825.

[43] H. Khalid, M. Nagappan, E. Shihab, and A. E. Hassan, "Prioritizing the devices to test your app on: A case study of android game apps," in *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 610–620.

[44] S. Aleem, L. F. Capretz, and F. Ahmed, "Critical success factors to improve the game development process from a developer's perspective," *J. Comput. Sci. Technol.*, vol. 31, no. 5, pp. 925–950, 2016.

[45] G. Lovreto, A. T. Endo, P. Nardi, and V. H. S. Durelli, "Automated tests for mobile games: An experience report," in *17th Brazilian Symposium on Computer Games and Digital Entertainment, SBGames 2018, Foz do Iguaçu, Brazil, October 29 - November 1, 2018*, 2018, pp. 48–56.

[46] S. Iftikhar, M. Z. Iqbal, M. U. Khan, and W. Mahmood, "An automated model based testing approach for platform games," in *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 2015, pp. 426–435.