


Article

GUI Component Detection-Based Automated Software Crash Diagnosis

Seong-Guk Nam and Yeong-Seok Seo * 

Department of Computer Engineering, Yeungnam University, Gyeongsan 38541, Republic of Korea; sd05031@yu.ac.kr

* Correspondence: ysseo@yu.ac.kr; Tel.: +82-53-810-3534

Abstract: This study presents an automated software crash-diagnosis technique using a state transition graph (STG) based on GUI-component detection. An STG is a graph representation of the state changes in an application that are caused by actions that are executed in the GUI, which avoids redundant test cases and generates bug-reproduction scenarios. The proposed technique configures the software application STG using computer vision and artificial intelligence technologies and performs automated GUI testing without human intervention. Four experiments were conducted to evaluate the performance of the proposed technique: a detection-performance analysis of the GUI-component detection model, code-coverage measurement, crash-detection-performance analysis, and crash-detection-performance analysis in a self-configured multi-crash environment. The GUI-component detection model obtained a macro F1-score of 0.843, even with a small training dataset for the deep-learning model in the detection-performance analysis. Furthermore, the proposed technique achieved better performance results than the baseline Monkey in terms of code coverage, crash detection, and multi-crash detection.

Keywords: software crash; object detection; deep learning; Android application; GUI; automation



Citation: Nam, S.-G.; Seo, Y.-S. GUI Component Detection-Based Automated Software Crash Diagnosis. *Electronics* **2023**, *12*, 2382. <https://doi.org/10.3390/electronics12112382>

Academic Editor: Anna Fasolino

Received: 27 March 2023

Revised: 17 May 2023

Accepted: 22 May 2023

Published: 24 May 2023



Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Advanced software tools support devices and platforms to facilitate their efficient use in various fields. Furthermore, software can improve work efficiency by replacing human labor in tasks that are difficult or impossible for humans to accomplish. Driven by recent progress in the industrial market and technology, new devices, platforms, and user requirements continue to emerge [1], which are accompanied by related technologies such as new programming languages, frameworks, development tools, and development methodologies, thereby increasing the diversification of software-development methods [2,3].

Current software is usually equipped with a graphical user interface (GUI) that provides a visual environment, such as buttons, text boxes, and combo boxes, to enhance user convenience by enabling users to use the available functions with ease. As GUI components have similar shapes and can be used across devices or platforms, they are intuitive for users to operate without the need to learn. Although software applications provide easy-to-use GUIs, users experience bugs in many different forms [4,5]; for example, by operating the program in an illogical manner owing to a lack of background knowledge of the program or by causing bugs while using GUI-driven software. Bug resolution is one of the most important tasks in maintaining software, and developers are interested in analyzing these bugs to improve software quality [6]. That is, software bugs occur when the software is not operated as intended by the developer. Software bugs are often hidden under unintended circumstances and are particularly difficult to detect [7]. Developers usually work under time pressure, which may lead to errors. Bugs are detrimental to software quality and user satisfaction and may lead to astronomical financial losses or even human casualties. Therefore, it is necessary to run tests to detect hidden bugs prior to software delivery and

distribution, which is as cost-intensive as the software development itself [8]. However, despite the importance of preventing bugs by testing software, companies often have difficulty in hiring bug-testing professionals.

A range of automated test tools has been developed to detect hidden bugs efficiently [9–16]. However, it is impossible to predict unexpected actions that are likely to be taken by users who do not accurately understand the intention of the developer and to detect all such unexpected actions with rule-based bug detectors. In particular, users of video game software can break the balance of a game by operating certain functions using manipulation techniques that developers cannot anticipate [17]. Therefore, running tests with actual users of software products, which is known as beta testing [18], is an ideal testing technique.

Beta testing [19] is associated with difficulties such as recruiting user groups for testing and allocating resources prior to every software release [20]. Thus, alpha testing is performed using in-house personnel who were not involved in the software development to overcome the drawbacks of beta testing. However, some companies are not in a position to conduct beta or alpha testing, and users of software products that are distributed without undergoing beta or alpha testing may encounter bugs, thereby resulting in an increasing number of bug reports. This leads to user inconvenience and software quality degradation and may even result in huge economic damage and human casualties in software application fields such as healthcare, finances, and national defense, which highlights the importance of ensuring software quality [21].

In general, GUI testing is conducted for bug detection using a development tool following GUI-based software development. Numerous programming languages and development tools have emerged in the software-development market in recent years. Outdated software-development tools also exist, for which technical support has been discontinued. The testing of the entire spectrum of software products is associated with difficulty in learning how to use the tools for developing individual software products, which makes software testing an increasingly difficult task. However, the GUI components shown to users have similar characteristics. In GUI applications, GUI components show similar shapes, although it is difficult to find the same shapes. We use the features of GUI components and object detection, a vision-based AI technique, to detect GUI components. If GUI components can be detected in any environment, it will be possible to perform general-purpose crash diagnosis in applications that provide GUI. Our goal is to automatically diagnose software crashes in GUI applications using GUI-component detection even though the detailed GUI information (the detailed GUI-related documents) is not available.

Therefore, many researchers have conducted studies on the automation of bug detection and the importance of bug-detection technology continues to increase [22–26]. In particular, the automation of manipulating GUI widgets can significantly increase testing efficiency [27–30].

This study presents a method of automated software crash diagnosis based on GUI-component detection. This technique, based on advanced technologies such as computer vision and artificial intelligence (AI), is used to analyze the visual GUI characteristics [31,32] and reduce the testing costs by addressing the aforementioned difficulties.

The recent progress in computer vision and AI technologies has significantly improved the performance of techniques for detecting objects in static images. The proposed method recognizes GUI components, such as buttons, text boxes, and combo boxes, on the screen of a running program using an object-detection technique and identifies their types, locations, and sizes. The derived component data are used to configure a state transition graph (STG) for testing purposes. Subsequently, the STG is used to avoid redundant or meaningless test cases during software testing. Using the STG and test automation, a dataset is generated, which is used to train the policies or rules to prioritize locations that are likely to be affected by bugs [33].

The major contributions of this study are summarized as follows:

- An automated crash diagnosis method that extracts GUI components using an object-detection technique and is applicable to all GUI-based software is proposed.
- During crash diagnosis, an STG is generated to avoid meaningless test cases by structuring the changes in the software state based on information regarding the class, location, and size of the screen and GUI components.
- Cross-platform black-box testing can be provided.
- Quantitative and empirical crash diagnoses were conducted using open-source Android application datasets.
- An application was developed and tested to establish an environment to detect multiple crashes.
- The software STG that was generated during the crash diagnosis was used as a training dataset to prepare the groundwork for research on reinforcement-learning-based GUI software-crash-diagnosis techniques.

The remainder of this paper is organized as follows: Section 2 describes related work; Section 3 outlines the proposed approach; Section 4 presents a performance evaluation of the proposed technique; and Section 5 summarizes the conclusions and provides suggestions for future research.

2. Related Work

GUI-based application screen-capture testing, object detection, and AI-based software-testing techniques are described in this section.

As the number of mobile applications rapidly increases, the importance of testing automation for GUI-based applications is also increasing. Before releasing the application, verification is performed through as many types of testing as possible. This section describes GUI-based application testing. Screen-capture testing performs tests such as input from devices and screen touches through an automation tool that captures the execution screen of an application and extracts data. Screen-capture testing is based on image-processing technology and is a method of finding and identifying the components of applications. Ranorex and Sikuli are currently available as screen-capture testing tools that can be applied to both web and mobile environments and are constantly being updated [34,35]. Ranorex is a non-free GUI automation testing tool that provides tests such as black boxes, cross-browsers, data, keywords, regression analysis, and functionality [34]. It has the advantage of being able to quickly define a test case and being intuitive in its use. The testing procedure for Ranorex is as follows: First, test cases are recorded from a capture screen for distribution or executable files. Second, it performs all the tasks recorded through simulation in the recorded test case. Finally, a report on the test results is generated and presented to the user. Sikuli was launched in 2009 as an open-source research project by MIT's User Interface Design Group as an automation tool to define users' tasks from the GUI components of their applications [35]. Sikuli can automate tasks for anything that appears on the screen on Windows and Mac. It performs OpenCV-based image processing to identify and control GUI components. SikuliX can be useful for automating tasks in common applications, web pages, and mobile applications that require repetitive tasks.

Generally, application-testing automation without human intervention is not easy because each application has a different configuration of GUI components. However, unlike the existing state-of-the-art techniques, the proposed method uses the state transition graph (STG), which plays an important role in automating the generation of test scenarios for diagnosing software crashes. Because STG can record state changes caused by interactions between GUI components of applications as data, the proposed method prevents duplicate-action scenarios and enables automation of test-scenario generation, which are difficulties encountered in the existing techniques. Thus, the proposed method provides an improvement in automation quality and has great advantages in both time and cost of testing.

Object detection analyzes input image data to determine the class, position, and size of the desired object. Convolutional neural network (CNN)-based AI technology has rapidly

expanded with the recent developments in high-performance computing [36–41]. These object-detection techniques have been validated in world-famous competitions such as the VOC PASCAL Challenge, COCO, ImageNet Object Detection Challenge, and Google Open Images Challenge [42–45]. Image data, which are easily recognizable by the human eye, have characteristics that are difficult for computers to analyze [46]. Image data are complex because they are sensitive to small changes (in color, brightness, and noise) and are represented by red, green, and blue (RGB) values. The visual features of objects were previously analyzed using traditional machine-learning algorithms such as support vector machines [47].

Among the deep-learning algorithms, CNNs have been used extensively to construct object-detection models and have achieved breakthroughs in object detection. Deep-learning-based object detection techniques perform two processes [48]: segmentation of the regions that contain objects from the image and classification of the objects that are included in the segmented regions. You Only Look Once (YOLO) is a popular real-time one-stage object-detection algorithm (current version: YOLO-v7) that performs object-region detection and object classification simultaneously [49]. R-CNN, which adopts a two-stage approach, is not sufficiently fast to be used as a real-time object detector because it performs object-region segmentation and object classification separately. As R-CNN performs better in object detection than one-stage models despite its slow performance, many researchers have contributed to improving its detection speed to the level of real-time detection. Faster R-CNN is currently the representative two-stage detector [50]. Whereas YOLO and Faster R-CNN detect objects inside a rectangular boundary box, Mask R-CNN stores the shape of an object as segmented masks that match its original shape to the maximum possible degree to improve the image representation accuracy [51]. Failure to detect GUI components during the testing process results in degraded testing quality and reliability. Therefore, Faster R-CNN was selected and trained as the GUI-component detection model in this study.

These deep-learning techniques exhibit a high analysis performance in detecting objects that are similar in terms of their overall external morphological characteristics but differ extensively in terms of their detailed characteristics, which are revealed through further analysis. For example, human bodies share similar characteristics in their overall appearance but there may be an infinite number of different cases depending on the age, gender, race, and clothing style, and the likelihood of people looking exactly alike is extremely low. Nevertheless, these techniques exhibit high performance, even for the detection of people. Users can easily use GUI functions without specific instructions by inferring them from similar features. Thus, deep-learning-based object-detection techniques are expected to achieve high performance in GUI component analysis owing to their image-analysis performance.

Many researchers have attempted to perform software testing using various deep-learning techniques. Sharif et al. [52] developed DeepOrder, which is a deep-learning-based model that assigns priority orders to test cases by learning failed test cases. Qiao et al. [53] proposed deep-learning neural-network-based defect prediction, which predicts the number of defects that are contained in a given source code. Kim et al. [54] proposed a model that generates test cases based on reinforcement learning. Liu et al. [55] presented DeepSQLi, which is a deep-learning model based on natural-language processing that automatically generates queries to test the techniques that are frequently used for SQL injection attacks on web application security. Mirabella et al. [56] generated automated test cases for testing RESTful web APIs and proposed a deep-learning-based approach to predict the validity of an API request before responding to it. Oz et al. [57] proposed an LSTM-based test-script generation technique that automates the generation of test scripts for web applications and reduces the number of possible sequences.

Amalfitano et al. studied test-automation techniques using the state of GUI applications [58,59]. MobiGUITAR created the GUI state of the application, allowing for more precise modeling of the state-sensitive behavior of the mobile application. They showed that the combination of model learning and model-based testing is a promising approach to

improving fault-detection performance in Android app testing. In addition, through follow-up studies, they defined Gate GUI that only functions when specific conditions are satisfied and proposed juGULAR to automatically detect Gate GUI by classifying the attribute values of GUI components using a machine-learning approach. They presented improved exploration capabilities in terms of Covered Activities and Covered Lines of Code.

In this study, a software user interface was developed using a deep-learning-based object-detection technique. Whereas existing techniques depend on source code and development tools, the proposed technique uses only the corresponding software version without software development- and configuration-related information (source code), and performs alpha testing in a state that is unrestricted by platforms and operating systems (OSs).

3. Overall Approach

This section presents a detailed explanation of GUI software crash diagnosis using the proposed technique. The schematics in Figure 1 provide an overview of the proposed technique.

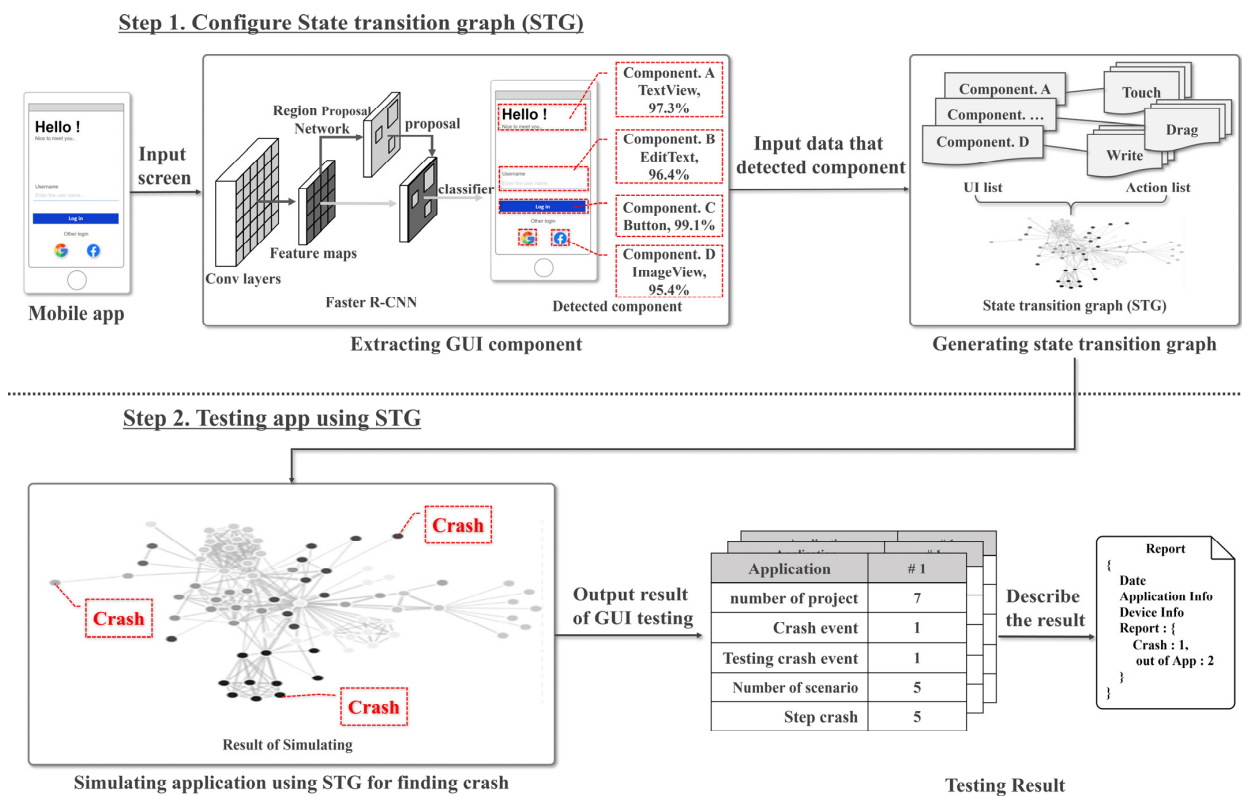


Figure 1. Overall Approach.

First, all states in the application during the test are captured. This process is important for extracting GUI component information from the screen and should be performed whenever the screen is switched or actions through GUI components (e.g., panels, buttons, and list boxes) are operated. The captured screen is used to analyze the current state of the application and to detect information on the GUI component shown on the screen. Subsequently, the captured application screen is transferred as an input value to the GUI-component detection model that is pretrained with Faster R-CNN. The GUI-component detection model provides information for extracting executable actions in each application state (screen) by detecting the components that comprise the application screen. The state information and information on all executable actions are converted into an application STG using the test scenario graph-generation algorithm. Finally, the STG that is generated from the software under testing is employed to run the test cases that are generated using a depth-first search (DFS) and provides a report containing information on the detected bugs.

3.1. Step 1: Construction of the STG

This subsection provides a detailed description of the process of generating an application STG. An STG is a graph representation of the state changes according to the actions that are executed during application testing. It is used to avoid redundant test cases during testing and to record data. The extracted GUI components and action lists are combined to construct an STG.

3.1.1. Extraction of GUI Components

Figure 2 depicts the process of GUI-component extraction from a screen, which is performed in three steps: screen capture, screen analysis, and GUI-component analysis.

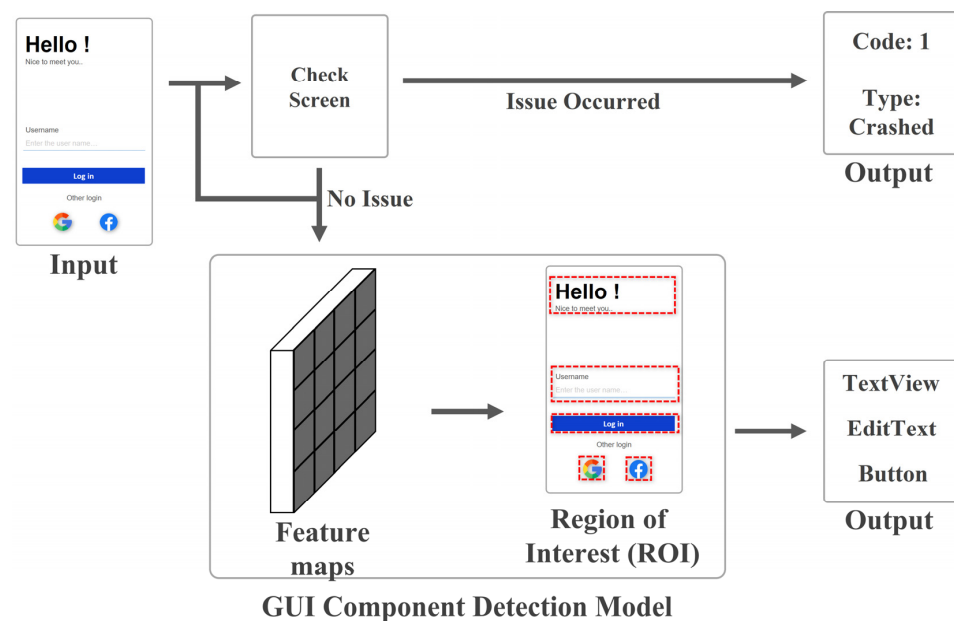


Figure 2. Block diagram illustrating GUI-component detection process.

First, all screens of the actions that are executed during testing are saved as image data, which are subsequently used to analyze the application state and GUI component information that are included in each screen.

Second, the state of the application is analyzed by visual (image) inspection to detect out-of-application (OOA) and crash events prior to the GUI component analysis. OOA and crash events may occur while certain actions are executed during testing, making it impossible to continue testing. If an OOA or crash event occurs, testing is discontinued, the application status is plotted on the graph, and the output is returned. Once all of these steps have been taken, the application under testing is initialized for restarting and all cache data are cleared.

Third, if the screen testing reveals no problems, the class, location, size, and reliability of the GUI components are analyzed using a GUI-component detection model that is pretrained with Faster R-CNN deep learning. The Faster R-CNN model extracts a feature map from an input image and selects the region of interest (ROI). The ROIs indicate regions that contain GUI components, from which the location and size information can be derived. A feature map is used to classify the image class within an ROI and to derive its reliability. Only the GUI-component detection model outputs with a reliability that is higher than the threshold value are valid. Reliability refers to the accuracy of an output, and low reliability is considered to be prone to false positives.

3.1.2. Generation of STG

An STG plots the state (screen) changes when a specific action is executed in a specific state (screen) of an application. The nodes in an STG represent the application-screen (state)

information, whereas the edges connect the changes between screens that are caused by actions. The proposed technique performs testing using actions based on screen touch. It is necessary to create an action list and plot a graph to generate an STG.

First, the information on the GUI components that are analyzed in the step outlined in Section 3.1.1 and the actions that can be executed by each GUI component are combined to generate an action list. The GUI is composed of different actions that can be executed depending on the class. For example, a button can be clicked, and a scroll (list) view can be dragged down. These contents are used to predetermine the sets of actions that are executable by the GUI components of different classes. That is, an action list is generated by combining all GUI components that are extracted from a screen with all sets of actions that are executable by the components on that screen.

Second, the generated action list is used to configure the graph. An action in an action list is a means of moving from the current to the next state. The listed actions comprise the edges of the current state. The generated edges are not connected to the nodes because the state change from the current state is not known. Therefore, the initial edge should be identified from the generated edges, the action command should be executed, and the next node should be generated and connected.

The graph search is terminated if any of the following events occur:

- No change on the screen;
- Crash;
- OOA;
- Maximum steps exceeded;
- No executable action.

Algorithm 1 presents the STG-generation algorithm that was used for testing. The algorithm inputs are the device, GUI-component detection model, and application under testing. The output is not specified.

Algorithm 1: Configure Initial STG

Input: device, model, application

```

1  device.install(application)
2  nodeList = new List()
3  lastNode = null
4  step = 0
5  while step < MAX_STEP // Iterate until the MAX step
6      image = device.getScreen()
7      screenStatus = screenAnalysis(image)
8      if screenStatus != null // Check for crash, OOA, and state change
9          lastNode.getAction(index=0).setStatus(screenStatus)
10         break
11     node = makeNode(image) // Generate nodes for the graph
12     nodeList.add(node)
13     if lastNode != null // Add no action to the initial node
14         lastNode.getAction(index=0).setNode(Node)
15     UI_List = GUI_Component_detect(model, image) // Detect GUI components
16     actionList = makeActions(UI_List)
17     Node.addAction(actionList)
18     // Create commands to execute actions matching device type
19     command = node.getAction(index=0)
20     .getCommand(device_type=device.getType())
21     device.execute(command)
22     lastNode = node
23     step = step + 1

```

Lines 1–4 of Algorithm 1 indicate the initial state configuration of the algorithm: install the application for testing on the device and initialize the node list, last node, and step of the graph. The last state node connects the new state node to the edge following a state change. A step is used to count the number of iterations of an action. MAX_STEP is set to prevent an infinite loop.

Lines 6–10 of Algorithm 1 capture the screen of the device, load the image, and analyze the state. The status of the application under testing is determined as normal, crash, OOA, or no change. Any state that is not normal is marked on the graph and the testing is terminated. No state change means that the executed action does not cause any change to the screen. The GUI-component detection technique prevents the generation of an incorrect STG by erroneously generating meaningless actions or executing undefined actions by analyzing the action before responding to it.

Lines 11–17 of Algorithm 1 describe the STG node and edge generation. If no problem is found in the application state, the node generation, GUI component analysis, action list generation, and edge connection in the graph are executed sequentially based on the current screen.

Lines 18–21 of Algorithm 1 describe the executed actions. The test is executed by creating a command at the 0th edge of the current node, saving the current node to the last state node variable, and increasing the step value by one.

3.2. Step 2: Testing with STG

This subsection describes the execution and reporting of the application testing using the proposed technique. The constructed STG indicates a state in which all state transitions of the application are not completely connected. Therefore, a detailed explanation is provided, with a focus on the process of expanding the STG by searching for and discovering unconnected nodes. Moreover, a report that summarizes the test results is written based on the complete STG, and the contents of the report are explained.

3.2.1. STG-Based Testing Process

As the initial state STG that is generated in Step 1 is not the result of searching all states of the application, edges exist that are unconnected to the nodes, where the search (test) state is null. The paths of the unsearched edges are generated using test scenarios based on the STG and DFS techniques. To this end, the process of expanding the STG is iterated until all edges are sequentially connected to their respective nodes during the application testing according to a given test scenario. Figure 3 is an example of an initial STG. In Figure 3, solid line nodes are already visited screens, dotted line nodes are not-visited screens, and edges are GUI components. In this case, the list of test scenarios contains [[execute B], [execute A, execute D], [execute A, execute E]]. Whenever nodes that are not visited are found during the test, they are continuously added to the list of test scenarios.

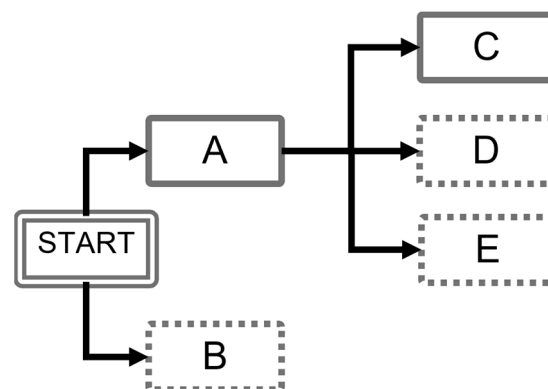


Figure 3. Example of an initial STG (START, A, B, C, D, and E are screens of a mobile application (activities of an Android application)).

Algorithm 2 represents the algorithm for the STG-based testing process. The algorithm input consists of a device, a GUI-component detection model, an STG node list, the application under testing, and a list of test scenarios; the output is not specified. The algorithm selects the 0th node of the graph, which represents the initial state of the application under testing, and conducts a state check for the presence of executable actions during testing.

Algorithm 2: STG-based testing

Input: device, model, nodeList, application, testScenarioList

```

1  startNode = nodeList.get(index=0)
2  // Iterate until nothing remains to be done at the initial node
   while startNode.isClosed() != True
3      device.restart(application)
4      step, nowNode, history = doTest(testScenarioList)
5      // Execute when the step is smaller than MAX_STEP
       while step < MAX_STEP
6          image = device.getScreen()
7          screenStatus = screenAnalysis(image)
8          if screenStatus != null
9              nowNode.setStatus(screenStatus)
10             break
11         // Replace the existing node with a new one and connect it
           nowNode = getAddConnectNode(nowNode, image)
12         actions = nowNode.getActions() // Non-close action
13         if action == null
14             nowNode.setClose()
15             break
16         action, others = select(actions)
17         add_scenario(history, others)
18         history.add(action)
19         device.execute(action.command())
20         step = step + 1

```

In lines 3 and 4 of Algorithm 2, the application is initialized and one of the actions in the action list of the test scenario is executed. If the test scenario has been completely executed, the number of actions executed, current node, and action record are returned. If no test scenario has been executed, a list with an empty history (step = 0, nowNode = initial node) is returned. If the number of executed actions is lower than the maximum number set in the test scenario, the testing is restarted.

In lines 6–10 of Algorithm 2, the screen of the device is captured and analyzed; if an abnormality is identified in the state, the abnormal state is recorded in the node and the corresponding test run is terminated.

If no abnormality is identified in the application state that is analyzed in lines 11 to 15 of Algorithm 2, the nodes and action list are generated. If the action list is empty, the state of the related node is set to “closed” and the test run is terminated. No test scenarios are generated during testing for closed nodes.

In lines 16–20 of Algorithm 2, one non-closed node is selected from the action list, while the remaining nodes are fed into the history and added to the test scenario list. The selected action is added to the history and the step value is increased by one once the action has been executed.

3.2.2. Reporting

If a new test case is not created in the STG-based testing stage, it is converted into the “testing complete” state and the proposed technique generates a report based on all information that is collected during testing. The test report is a summary of the application-testing results and provides useful information for users to understand the contents easily.

The report includes metadata on the test environment (device status and application information) and issues (crash and OOA events), as illustrated in Figure 4.

```

1  {
2    "Date": "2022-12-08 13:46:09",
3    "Application Info": {
4      "Package Name": "de.vier_bier.habpanelviewer",
5      "Version": "0.9.23"
6    },
7    "Device Info": {
8      "Device Name": "Pixel 4 API 26",
9      "OS": "Android 8.0 Google Play | x86",
10     "Device Size": {
11       "Width": 1080,
12       "Height": 2280
13     },
14     "Orientation Mode": "Portrait",
15     "Battery Info(%)": 100,
16     "Network Mode": "Wifi"
17   },
18   "Report": {
19     "Number of Crash": 1,
20     "History of Crash": [
21       [
22         "tap 915 2038",
23         "tap 915 2038",
24         "tap 915 2038",
25         "tap 915 2038",
26         "tap 915 2038"
27       ]
28     ],
29     "Number of OutOfApp": 0,
30     "History of OutOfApp": []
31   }
32 }

```

Figure 4. Example of results reporting.

Lines 2 to 17 of Figure 4 contain the metadata relating to the testing environment. The metadata include the testing date and time, application information, and device-status information. The application information includes the package name and version. The device-status information includes the device name, OS, display size, orientation mode, battery-charge state, and network mode.

Lines 18 to 31 of Figure 4 contain information regarding the issues that are detected during testing; that is, crash and OOA events that are collected through the STG search, which provide information such as the occurrence of issues, number of issues incurred, and reproduction scenarios. If issues occur during the STG search, the number of events is increased and reproduction scenarios are generated using the commands that are executed along the search path from the initial to the current node in the graph. Lines 21 to 27 of Figure 4 provide an example of reproduction scenarios in the form of a list. In the case of no events, an empty list is returned, as indicated in line 30 of Figure 4.

4. Experiment

4.1. Configuration of Experimental Environment

Three experiments were conducted to evaluate the performance of the proposed software-crash diagnosis method using an STG based on GUI-component detection. The following aspects were investigated: (i) the performance of the GUI-component detection model, (ii) code-coverage analysis and crash detection using open-source application datasets, and (iii) the reliability of crash detection in the multi-crash environment of an application. In this experimental evaluation, only actions related to screen touch (click,

double click, drag, zoom-in, zoom-out, etc.) were considered, and experiments on other actions (text input, hardware button input, etc.) were not considered in this experiment. In addition, we defined two or more crashes present in the application as “multi-crash” and conducted experiments to quantitatively evaluate the performance of the proposed technique compared to the baseline. The applications provided by the open-source dataset contain only one bug per application. However, we constructed a multi-crash environment to test whether multiple crashes can be detected in one application using the proposed technique. The multi-crash environment was established to enable a novel testing method and an application for bug simulation was developed for this experiment.

4.1.1. Design for Performance Testing of the GUI-Component Detection Model

The first experiment was conducted to analyze the performance of the pretrained GUI-component detection model in analyzing the GUI-component information from the application screen. A total of 100 open-source Android projects were collected from GitHub, and the Faster R-CNN model with modified parameters was used for pretraining to implement and train the model. In general, many applications on the market are for profit purposes, making it difficult to create datasets. Therefore, in order to perform objective verification of the proposed technique using GUI applications with as many types of GUI components as possible in this paper, we first collected many applications, including GUI, from Android repositories opened on GitHub. We collected 295 XML files from 100 open Android repositories. To convert XML files into images, we modified some source code for the Layout build of Android Studios and used it in the preprocessing process. In addition, we extracted and recorded the coordinates of GUI components corresponding to images in the process of converting XML files into images. The coordinates consist of the X starting point, Y starting point, height, and width of the GUI component. The data set for model training consists of image files converted from XML files, class numbers, and coordinates of GUI components, which are labels of data. The total number of extracted GUI components (label) is 726 and was divided based on the label number into 8:1:1 for training, testing, and verification. In addition, data augmentation was not performed. Next, we filtered out some bad data that could degrade the quality of the collected dataset. Filtering criteria include the absence of any GUI components in the layout, the configuration of GUI components through an external library, and errors in the format of the file. The processed dataset is described in Table 1.

Table 1. Dataset for evaluating GUI-component detection performance.

Item	Description
Number of Android Repositories	100
Number of collected images from XML	295
Number of labeled datasets	726
Data processing	<ol style="list-style-type: none"> 1. Extract XML files from Android repositories 2. XML files to PNG files (images) 3. XML files to class type and coordinate data (labels) of GUI components
Data filtering	<ul style="list-style-type: none"> - layout without GUI components - GUI components using external libraries - XML format error
Ratio of training dataset	80%
Ratio of test dataset	10%
Ratio of validation dataset	10%
The link to the dataset	https://github.com/sd05031/Dataset_for_GUI_components (accessed on 21 May 2023)

The performance of the pretrained GUI-component detection model was assessed in terms of precision, recall, F1-score, and accuracy. Among the data that were predicted as true by the model, those whose actual class was true were true positives (TPs), and those whose actual class was false were false positives (FPs). Among the data that were predicted as false by the model, those whose actual class was true were false negatives (FNs), and those whose actual class was false were true negatives (TNs). All misclassified cases were defined as false in this study. The accuracy, which is obtained by dividing the number of correctly classified data by the total number of data, was used as an indicator of the performance reliability of the classification model. The accuracy is calculated using Equation (1).

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \quad (1)$$

However, as accuracy is an unreliable indicator when the dataset is skewed, the precision, recall, and F1-score were also used to evaluate the performance. Precision refers to the ratio of data that are classified as true by the model to the data that are labeled as true, as expressed by Equation (2). Recall refers to the ratio of data that are labeled as true to the data that are classified as true by the model, as expressed by Equation (3).

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad (2)$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad (3)$$

Given that precision and recall have an inversely proportional relationship (tradeoff), it is unlikely that both will achieve high performance if a classification model cannot perform classification accurately. Therefore, the F1-score, which is the harmonic mean of the precision and recall scores, was used to determine whether precision and recall simultaneously achieved high performances, so as to measure the model-classification performance regardless of data skewness. The F1-score can be obtained by dividing the product of precision and recall by their sum. A high F1-score can only be obtained when both precision and recall exhibit high performance. When both precision and recall are set to 1.0, the maximum value that is yielded by the calculation formula is 0.5. Therefore, it is multiplied by 2 for correction, as expressed by Equation (4).

$$\text{F1-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad (4)$$

Table 2 presents an overview of the experimental environment and setup for the performance evaluation of the GUI-component detection model. We used the PyTorch framework to use deep-learning techniques and the MMDetection framework to develop and train object-detection models. In addition, it does not collect information on GUI components provided by external libraries that are not provided by Android Studio. Performing a test based on a GUI-component detection model takes time to detect GUI, but becomes non-dependent on the framework for the test. Although all GUI states included in the application were captured and analyzed, it takes only 0.28 s on average to analyze components from images on the captured screen. People need a little time to recognize and select GUI components.

Table 2. Experimental environment for evaluating GUI-component detection performance.

Item	Description
Implementation framework	Deep learning: PyTorch
GUI-component detection model	Object detection: MMDetection Faster R-CNN
Component types to be detected	13 types (Button, Image, TextView, ToggleButton, RadioButton, EditText, ProgressBar, SeekBar, RatingBar, ScrollView, Switch, Spinner, CheckBox)
Average time for component detection	0.28 s

4.1.2. Design for Testing Open-Source Applications

Application testing was simulated using the Android Emulator as the experimental device and AndroR2 [60] as the dataset, as outlined in Tables 3 and 4, respectively. The Android Emulator simulates an Android device on a PC, thereby creating a virtual environment that is similar to a real device. This is convenient for testing because the environment can be configured and controlled easily.

Table 3. Experimental environment configuration.

Item	Description
Device name	Android Emulator
Device type	Virtual device
OS	Android API 26 Android 8.0 OS (Oreo)
Resolution	Width: 1080 px Length: 2280 px
RAM capacity	1536 MB (1.5 GB)
GPU hardware	Yes

Table 4. The dataset for crash detection testing.

Item	Description
Name of the dataset	AndroR2
Number of total bug reports being published	90 items
Bug types	Crash, Output, GUI
Data feed	Files containing bugs (APK), bug reports, reproduction scripts (Java), metadata (JSON)
Used bug reports for crash detection testing	Crash type 3 bug reports: #7, #11, and #50

Bug report ID	Bug type	Application name	Android OS version reported	GUI actions in bug scenarios
7	Crash	HAB Panel Viewer	8.1	5
11	Crash	Noad Player	6.1	2
50	Crash	Berkeley Mobile	9.0	1

AndroR2 is a dataset of Android applications that contains real bugs. It provides 90 bug reports that are associated with bug reproduction scripts created using the GitHub Issues tracker for applications that are available on the Google Play Store and GitHub. This dataset provides files for installing applications and scripts containing how to reproduce bugs corresponding to reports. The application can be installed through an APK extension file and is provided in the same version as written in the bug report. The script for reproduction is written in the source code of the Java language, and when executed, it performs GUI actions that reproduce bugs in the report. In addition, JSON data are provided as metadata, including the GitHub address of the open source, the OS version of Android, the type of

bug, and the number of GUI actions for bug reproduction. Three types of bugs exist: Crash (an application termination that is caused by a crash event), Output (errors in the output results), and GUI (errors in GUI properties). The experiments were conducted using four applications that operate normally in the Android Emulator environment and bug reports.

The second experiment measured the code coverage to determine how many codes could be executed by the proposed technique. Monkey [61], which has been used as a comparison tool in many studies, was selected as the baseline, and the Android Code Coverage Tool (ACVTool) [62] was used as the coverage measurement tool. ACVTool measures the code coverage by analyzing bytecode without the project source code. Table 5 describes the experimental environment for the code coverage evaluation. The application of bug report #62 (PDF Converter) was used as the data to avoid crashes during the code coverage analysis, which would make normal measurement impossible.

Table 5. Experimental environment configuration for code coverage measurement.

Item	Description
Measurement tool	Android Code Coverage Tool (ACV Tool) [62]
Data applied	AndroR2 [60]
Baseline	Bug report #62 (Output type)
Length of the test scenario	Monkey [61]
Number of events set for Monkey test	5 and 10
Seed value of Monkey	100, 500, and 1000
Measurement object	1 and 777
	Total code coverage

This application is classified as the “output” bug type and operates normally in the Android Emulator. The test scenario lengths were set to 5 and 10 to compare the difference according to the number of test executions for each scenario of the proposed technique. The length of the test scenario refers to the number of actions performed during the test through the GUI component from the start of the application. The number of events was set to 100, 500, and 1000 and the number of test executions was not specified for the Monkey test. The seed value was set to 1 and 777 to ensure that the Monkey test, which is based on the execution of random actions, can always be performed under the same conditions. Seed is a value used to control random events that occur during Monkey tests. The same values lead to the same results. We use them to obtain fixed test results.

In the second experiment, we also evaluated the bug detection performance of the proposed technique. Among the available bugs in the dataset, those that are classified as “Crash” in bug reports #7, #11, and #50, which can be detected using computer vision (image analysis) without source code, were used for testing with the application running normally on the Android Emulator. The maximum length of the test scenario was set to 10. Monkey was selected as the baseline to compare the testing performance from the black-box perspective. The number of random events was set to 1000, with the seed values ranging from 1 to 5, in the Monkey test.

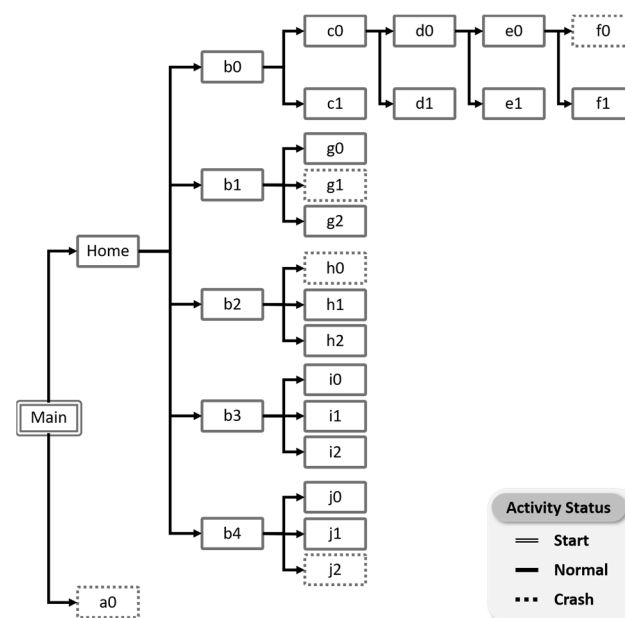
4.1.3. Crash Detection Testing Design

For the third experiment, an application that directly generates crashes was developed and a testing environment that could induce a crash event was constructed to evaluate the performance of the proposed technique. Five user-defined exception-handling crashes were specified (Crash 1 through Crash 5) in the application to verify the occurrence of crash events and analyze the crash location from the device log data. The configuration details are presented in Table 6. The application was developed using the Kotlin language for programming on the Android platform, and it was configured for use on devices with Android 8.0 (Oreo) or higher. There were no permission requests to use the application and the Android Emulator was used in the experiments.

Table 6. Configuration of experimental environment for code coverage measurement.

Item	Content
Platform	Android Native
Programming language	Kotlin
Minimum OS requirements	Android API 26 Android 8.0 OS (Oreo)
Permission requests	None
Number of activities	28
Number of action events	32
Number of crash events	5
Crash 1 event activity	A0
Crash 2 event activity	F0
Crash 3 event activity	G1
Crash 4 event activity	H0
Crash 5 event activity	J2

Figure 5 outlines the activities, moves between activities, and crash events in the application under testing. The solid line bordered nodes indicate normal-state activities, the dotted line bordered nodes indicate activities that triggered crashes within the nodes, and the double line bordered nodes indicate the starting activity of the application. An attempt was made to test whether the proposed technique could detect crashes and move between activities without being constrained by specific circumstances (depth and width) on the application STG by generating crash events on activities a0, f0, g1, h0, and j2. An additional experiment was conducted to compare the performance of the proposed technique with that of the baseline method. The performance of Monkey in detecting bugs was tested in 100 iterations, with the number of events set to 100, 500, and 1000.

**Figure 5.** Testing application configuration.

4.2. Experimental Results

4.2.1. Results of Performance Evaluation of GUI-Component Detection Model

The images that were collected from the first experiment were used to evaluate the performance of the GUI-component detection model in detecting 13 GUI components. Table 7 presents the results of the performance evaluation experiments. The model exhibited stable performance ranging from 0.83 to 0.86 on all evaluation items. However, scores

of 0.97 or higher are required to achieve detection accuracy, which suggests that a larger dataset should be used.

Table 7. Results of performance evaluation of GUI-component detection model.

TEST ITEM	Value
Macro precision	0.8529
Macro recall	0.8349
Macro F1-score	0.8430
Accuracy	0.8659

4.2.2. Results of Open-Source Application Testing Performance

Table 8 displays the results of the second experiment; that is, the code-coverage measurement of an open-source application. The code coverage was compared according to the maximum depth, with the number of action executions set to 5 and 10 for each test scenario of the proposed technique. The results demonstrate that the proposed technique did not execute meaningless actions. In the Monkey test, the application activities were randomly executed or changed, and the actions were randomly generated and executed. Monkey offers the advantage of freely executing any activity compared with the proposed technique. However, its excessively random search increased the rate of redundant test cases and its overall code-coverage results were lower than those of the proposed technique.

Table 8. Code-coverage measurement results.

Technique	Proposed Technique			Baseline (Monkey)		
MAX_STEP (depth)	5	10	-	-	-	-
Number of events	-	-	100	500	1000	1000
Seed value	-	-	1	1	1	777
Statement coverage	6.543%	7.629%	5.331%	6.382%	6.438%	6.628%
Function coverage	8.354%	9.317%	7.360%	8.497%	8.579%	8.277%
Class coverage	12.056%	13.517%	11.346%	12.279%	12.279%	11.812%
Activity coverage	17.460%	68.254%	17.160%	17.460%	17.460%	17.460%
Code coverage	6.141%	7.089%	5.331%	6.382%	6.483%	6.216%

The numbers in bold indicate the highest.

Table 9 presents the results of the crash-detection performance experiment; that is, the testing of the three open-source applications. Actions that matched the test scenarios or triggered other crash events were detected using all three applications. OOA events were also detected. Crash and OOA events could be reproduced with a high success rate by executing the test actions that were presented via the proposed technique.

Table 9. Detected crash measurement results.

Technique	Detection Performance	1 (# 7)	2 (# 11)	3 (# 50)
Proposed technique	Number of crash detections	1	1	9
	Number of GUI action executions	5	2	1, 3
	Number of OOA detections	0	2	0
Baseline (Monkey)	Number of crash detections (seed 1)	0	0	3
	Number of crash detections (seed 2)	0	0	2
	Number of crash detections (seed 3)	0	0	2
	Number of crash detections (seed 4)	1	0	2
	Number of crash detections (seed 5)	2	0	3
	Detection exception type	Illegal State Exception	None	Null Pointer Exception

In the comparison test, the baseline (Monkey) could detect crashes in only two of the three applications: 2/5, 0/5, and 5/5 in the first, second, and third applications, respectively, with all detected crashes showing the same exception.

4.2.3. Crash-Detection Testing Results

Table 10 presents the results of the testing in the final experiment, for which a bug-simulation application was developed in-house. The proposed technique performed a search (test), detected all crashes in the application and generated bug reports. However, the baseline technique failed to detect certain crashes, detecting 2/5, 4/5, and 5/5 crashes when testing was performed with 100, 500, and 1000 events per loop, respectively. More than 77,000 events had to be generated for Monkey to detect all five crashes at 1000 events per loop.

Table 10. Analysis of application testing results of the proposed technique.

Test Execution	Crash 1	Crash 2	Crash 3	Crash 4	Crash 5
Success in detection	O	O	O	O	O
Number of GUI actions	2	4	4	4	7
Number of events per loop: 100					
Number of events detected	-	-	57/100	76/100	-
Number of loops detected	-	-	63/100	35/100	-
Number of searches	<10,000	<10,000	5763	3576	<10,000
Success in detection	X	X	O	O	X
Number of events per loop: 500					
Number of events detected	258/500	340/500	481/500	160/500	-
Number of loops detected	28/100	27/100	6/100	10/100	-
Number of searches	14,285	13,840	3481	5160	<50,000
Success in detection	O	O	O	O	X
Number of events per loop: 1000					
Number of events detected	716/1000	318/1000	133/1000	236/1000	823/1000
Number of loops detected	40/100	4/100	21/100	42/100	77/100
Number of searches	40,716	4318	21,133	42,236	77,823
Success in detection	O	O	O	O	O

Figure 6 depicts the application-testing process of the proposed technique using a preconfigured application and the test results of each process step. In Step 1, an action list was generated by extracting the UI list from the input application activities. The initial STG, which was composed of nodes and edges, was generated based on an action list and activity information. Subsequently, three GUI components were extracted from the screen of the preconfigured application, which was followed by the generation of an action list with four items by combining the components with the predefined action set. However, no action list items were generated for the text view owing to a lack of matching actions in the action set. Finally, two items were generated for each button in the action set.

In Step 2, an initial-state STG was constructed using the node and edge information that was generated in the previous step. Nodes with values that are marked with “none” in the initial-state STG were unsearched and provided no information on their action-related states.

In Step 3, a DFS-based STG search was performed to complete the initial-state STG, and test cases were generated for the none-state nodes that were detected during the search. The test runs continued until no none-state nodes were identified in the STG and the test was terminated when the STG was completed.

Finally, in Step 4, the metadata on the test environment and information on crashes and OOA events that were detected during testing were recorded in a report and returned. The metadata included the testing date and time, application name and version, name

of the tested device, OS, and screen size. The crash and OOA information included the number of events and GUI actions that were used to reproduce the issue.

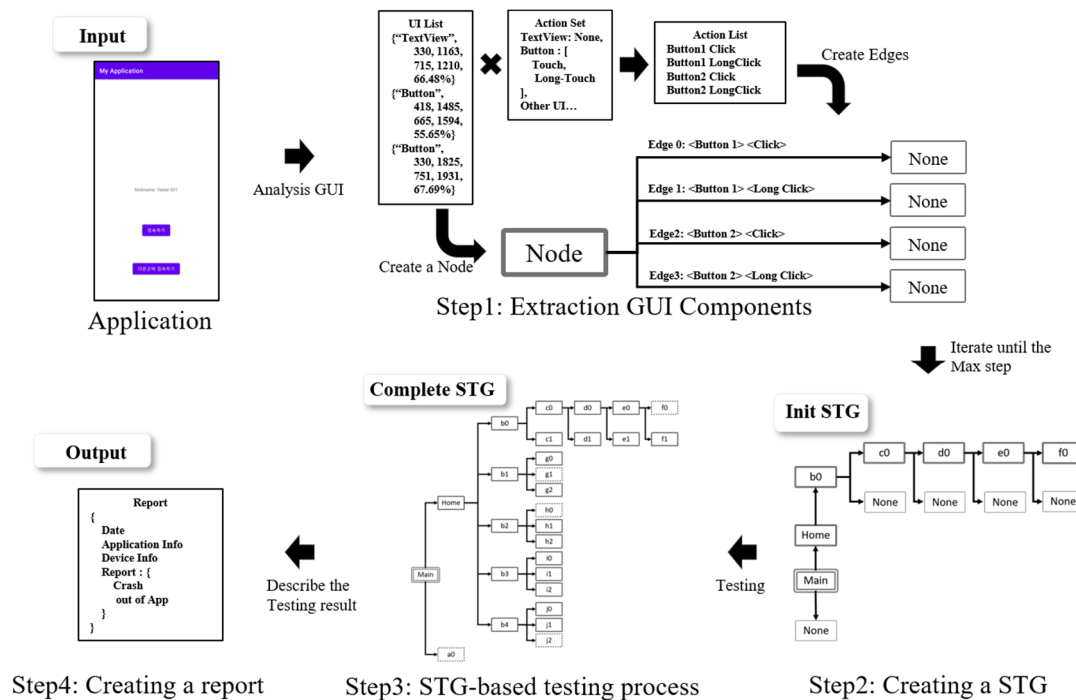


Figure 6. Application testing process of the proposed technique.

5. Conclusions

An automated software crash diagnosis technique using an STG based on GUI-component detection has been proposed. The executable actions on the application screen and STG are generated by combining the GUI components and a predefined action set. The STG is used to generate test scenarios, prevent redundant test cases, and report crash and OOA events.

The proposed technique differs from existing test tools in that it can perform testing without the source code of the application. Upon completion of the testing, it provides a bug report that enables STG-based bug reproduction. This also allows for cross-platform testing using computer vision. The strength of the proposed technique is that, by automating the test of the GUI application in a human-like manner, it has higher coverage than Monkey while maintaining the reliability and stability of the test. It can also be tested through the visual recognition of GUI components in an environment where the source code of the software is not provided. The proposed technique can perform valuable tests, unlike Monkey, which performs tests indiscriminately without probability, by performing tests through the recognition of GUI components. We conducted a comparative experiment with Monkey on the proposed technique, and as a result, we were able to verify the excellence of the proposed technique as follows.

Performance evaluations of the GUI-component detection model, code coverage measurement, and crash detection tests were conducted. A total of 100 open-source Android applications were collected and the GUI-component detection model was pretrained to analyze its performance. Subsequently, the accuracy of the model was assessed, along with its macro precision, macro recall, and macro F1-score. Although the deep learning model was trained with a small dataset, it exhibited stable performance ranging between 0.83 and 0.86. Given that the GUI-component detection performance is directly associated with the testing performance, the performance must be continuously improved.

Thereafter, code-coverage analysis and crash-detection performance evaluation experiments were conducted using the Android Emulator and AndroR2 bug dataset, and the

results were compared with those of the baseline technique (Monkey). A comparison of the code coverage measurement results revealed that, despite the imperfect performance level of the GUI-component detection model, the proposed technique outperformed Monkey, which executed 1000 random actions. We also performed a crash-detection experiment that could be analyzed using computer-vision technology to evaluate the testing performance using the three applications. The proposed technique detected all crashes in the three applications and generated result reports with the scenarios, including GUI actions that could reproduce each crash and OOA. The baseline technique detected crashes in only two of the three applications.

For the final experiment, we constructed a multi-crash testing environment by configuring an application containing five exception-handling crashes to analyze the testing performance of the proposed technique from various aspects. All five crashes were detected using the proposed technique and each crash could be reproduced based on the test report. In contrast, more than 77,000 events had to be generated for the baseline technique to detect all five crashes.

In future research, we will perform a study to automatically test actions that are not screen-touch-based. The dataset we collected did not take into account detection for hierarchical views. We will also perform the detection of hierarchical views in the next study. In addition, we plan to apply the MLOps [63] technique for the performance maintenance and improvement of the GUI-component detection model through continuous GUI-design changes. Furthermore, we will attempt to reduce the testing execution time by distributing test tasks using virtual instruments and cloud-computing techniques. Finally, we will investigate a reinforcement-learning-based GUI testing tool by training bug-prone tasks to enhance testing efficiency.

Author Contributions: Conceptualization, S.-G.N. and Y.-S.S.; Data curation, S.-G.N.; Funding acquisition, Y.-S.S.; Investigation, S.-G.N. and Y.-S.S.; Methodology, S.-G.N. and Y.-S.S.; Project administration, Y.-S.S.; Resources, Y.-S.S.; Software, S.-G.N.; Supervision, Y.-S.S.; Validation, S.-G.N. and Y.-S.S.; Visualization, S.-G.N.; Writing—Original draft, S.-G.N.; Writing—Review & editing, S.-G.N. and Y.-S.S. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the 2023 Yeungnam University Research Grant.

Data Availability Statement: The data presented in this study are openly available in GitHub at https://github.com/sd05031/Dataset_for_GUI_components (accessed on 26 March 2023).

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1. Basili, V.R. Software development: A paradigm for the future. In Proceedings of the Thirteenth Annual International Computer Software & Applications Conference, Orlando, FL, USA, 20–22 September 1989; pp. 471–485.
2. Yu, J. Research process on software development model. *IOP Conf. Ser. Mater. Sci. Eng.* **2018**, *394*, 032045. [CrossRef]
3. Dingsøyr, T.; Moe, N.B. Exploring software development at the very large-scale: A revelatory case study and research agenda for agile method adaptation. *Empir. Softw. Eng.* **2018**, *23*, 490–520. [CrossRef]
4. Hu, C.; Neamtiu, I. Automating GUI testing for Android applications. In Proceedings of the 6th International Workshop on Automation of Software Test, Honolulu, HI, USA, 23–24 May 2011; Association for Computing Machinery: New York, NY, USA; pp. 77–83.
5. Sui, Y.; Zhang, Y. Event trace reduction for effective bug replay of Android apps via differential GUI state analysis. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, 26–30 August 2019; Association for Computing Machinery: New York, NY, USA, 2019; pp. 1095–1099.
6. Moran, K. Enhancing android application bug reporting. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, Bergamo, Italy, 30 August–4 September 2015; Association for Computing Machinery: New York, NY, USA; pp. 1045–1047.

7. Ko, Y.; Zhu, B. Fuzzing with automatically controlled interleavings to detect concurrency bugs. *J. Syst. Softw.* **2022**, *191*, 111379. [\[CrossRef\]](#)
8. Jovic, M.; Adamoli, A. Catch me if you can: Performance bug detection in the wild. In Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, Portland, OR, USA, 22–27 October 2011; Association for Computing Machinery: New York, NY, USA, 2011; pp. 155–170.
9. Choi, W.; Necula, G. Guided gui testing of android apps with minimal restart and approximate learning. *ACM Sigplan Not.* **2013**, *48*, 623–640. [\[CrossRef\]](#)
10. Dong, Z.; Böhme, M. Time-travel testing of android apps. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, New York, NY, USA, 27 June–19 July 2020; pp. 481–492.
11. Gu, T.; Sun, C. Practical GUI testing of Android applications via model abstraction and refinement. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; pp. 269–280.
12. Wang, J.; Jiang, Y. ComboDroid: Generating high-quality test inputs for Android apps via use case combinations. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Seoul, Republic of Korea, 5–11 October 2020; pp. 469–480.
13. Machiry, A.; Tahiliani, R. Dynodroid: An input generation system for android apps. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, Saint Petersburg, Russia, 18–26 August 2013; Association for Computing Machinery: New York, NY, USA, 2013; pp. 224–234.
14. Mao, K.; Harman, M. Sapienz: Multi-objective automated testing for android applications. In Proceedings of the 25th International Symposium on Software Testing and Analysis, Saarbrücken, Germany, 18–20 July 2016; Association for Computing Machinery: New York, NY, USA, 2016; pp. 94–105.
15. Pan, M.; Huang, A. Reinforcement learning based curiosity-driven testing of Android applications. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, 18–22 July 2020; Association for Computing Machinery: New York, NY, USA, 2020; pp. 153–164.
16. Su, T.; Meng, G. Guided, stochastic model-based GUI testing of Android apps. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, Paderborn, Germany, 4–8 September 2017; Association for Computing Machinery: New York, NY, USA, 2017; pp. 245–256.
17. Zheng, Y.; Xie, X. Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA, 10–15 November 2019; pp. 772–784.
18. Dolan, R.J.; Matthews, J.M. Maximizing the utility of customer product testing: Beta test design and management. *J. Prod. Innov. Manag.* **1993**, *10*, 318–330. [\[CrossRef\]](#)
19. Pelivani, E.; Cico, B. A comparative study of automation testing tools for web applications. In Proceedings of the 2021 10th Mediterranean Conference on Embedded Computing (MECO), Budva, Montenegro, 7–11 June 2021; pp. 1–6.
20. Jiang, Z.; Scheibe, K.P. The Economics of Public Beta Testing. *Decis. Sci.* **2017**, *48*, 150–175. [\[CrossRef\]](#)
21. Lamkanfi, A.; Demeyer, S. Predicting the severity of a reported bug. In Proceedings of the 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), Cape Town, South Africa, 2–3 May 2010; pp. 1–10.
22. Sharma, M.; Kumari, M. Multiattribute based machine learning models for severity prediction in cross project context. In Proceedings of the Computational Science and Its Applications–ICCSA 2014: 14th International Conference, Guimarães, Portugal, 30 June–3 July 2014; pp. 227–241.
23. Chaturvedi, K.K.; Singh, V.B. Determining bug severity using machine learning techniques. In Proceedings of the 2012 CSI Sixth International Conference on Software Engineering (CONSEG), Indore, India, 5–7 September 2012; pp. 1–6.
24. Lamkanfi, A.; Demeyer, S. Comparing mining algorithms for predicting the severity of a reported bug. In Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering, Oldenburg, Germany, 1–4 March 2011; pp. 249–258.
25. Menzies, T.; Marcus, A. Automated severity assessment of software defect reports. In Proceedings of the 2008 IEEE International Conference on Software Maintenance, Beijing, China, 28 September–4 October 2008; pp. 346–355.
26. Tian, Y.; Lo, D. Information retrieval based nearest neighbor classification for fine-grained bug severity prediction. In Proceedings of the 2012 19th Working Conference on Reverse Engineering, Kingston, ON, Canada, 15–18 October 2012; pp. 215–224.
27. Yatskiv, S.; Voytyuk, I. Improved method of software automation testing based on the robotic process automation technology. In Proceedings of the 2019 9th International Conference on Advanced Computer Information Technologies (ACIT), Ceske Budejovice, Czech Republic, 5–7 June 2019; pp. 293–296.
28. Ma, Y.W.; Lin, D.P. System design and development for robotic process automation. In Proceedings of the 2019 IEEE International Conference on Smart Cloud (SmartCloud), Tokyo, Japan, 10–12 December 2019; pp. 187–189.
29. Maalla, A. Development prospect and application feasibility analysis of robotic process automation. In Proceedings of the 2019 IEEE 4th Advanced Information Technology, Electronic and Automation Control Conference (IAEAC), Chengdu, China, 20–22 December 2019; pp. 2714–2717.
30. Yatskiv, N.; Yatskiv, S. Method of robotic process automation in software testing using artificial intelligence. In Proceedings of the 2020 10th International Conference on Advanced Computer Information Technologies (ACIT), Deggendorf, Germany, 13–15 May 2020; pp. 501–504.

31. Battina, D.S. Artificial intelligence in software test automation: A systematic literature review. *Int. J. Emerg. Technol. Innov. Res.* **2019**, *6*, 2349–5162.
32. Bajammal, M.; Stocco, A. A survey on the use of computer vision to improve software engineering tasks. *IEEE Trans. Softw. Eng.* **2020**, *48*, 1722–1742. [[CrossRef](#)]
33. Jia, L.; Dong, W. Bug Finder Evaluation Guided Program Analysis Improvement. In Proceedings of the 2019 IEEE 7th International Conference on Computer Science and Network Technology (ICCSNT), Dalian, China, 19–20 October 2019; pp. 122–125.
34. Ranorex. Available online: <https://www.ranorex.com> (accessed on 27 April 2023).
35. Sikulix. Available online: <http://sikulix.com/> (accessed on 27 April 2023).
36. Krizhevsky, A.; Sutskever, I. Imagenet classification with deep convolutional neural networks. *Commun. ACM* **2017**, *60*, 84–90. [[CrossRef](#)]
37. Szegedy, C.; Liu, W. Going deeper with convolutions. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Boston, MA, USA, 7–12 June 2015; pp. 1–9.
38. LeCun, Y.; Bottou, L. Gradient-based learning applied to document recognition. *Proc. IEEE* **1998**, *86*, 2278–2324. [[CrossRef](#)]
39. He, K.; Zhang, X. Deep residual learning for image recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 26 June–1 July 2016; pp. 770–778.
40. Hinton, G.E.; Osindero, S. A fast learning algorithm for deep belief nets. *Neural Comput.* **2006**, *18*, 1527–1554. [[CrossRef](#)] [[PubMed](#)]
41. Hinton, G.E.; Salakhutdinov, R.R. Reducing the dimensionality of data with neural networks. *Science* **2006**, *313*, 504–507. [[CrossRef](#)] [[PubMed](#)]
42. Everingham, M.; Eslami, S.A. The pascal visual object classes challenge: A retrospective. *Int. J. Comput. Vis.* **2015**, *111*, 98–136. [[CrossRef](#)]
43. Coco Detection Challenge (Bounding Box). Available online: <https://competitions.codalab.org/competitions/20794> (accessed on 14 March 2023).
44. ImageNet. Imagenet Object Localization Challenge. Available online: <https://www.kaggle.com/c/imagenet-object-localization-challenge> (accessed on 14 March 2023).
45. G. Research. Open Images 2019—Object Detection Challenge. Available online: <https://www.kaggle.com/c/open-images-2019-object-detection> (accessed on 14 March 2023).
46. Bouma-Sims, E.; Reaves, B. A First Look at Scams on YouTube. *arXiv* **2021**, arXiv:2104.06515.
47. Cortes, C.; Vapnik, V. Support vector machine. *Mach. Learn.* **1995**, *20*, 273–297. [[CrossRef](#)]
48. Zhao, Z.Q.; Zheng, P. Object detection with deep learning: A review. *IEEE Trans. Neural Netw. Learn. Syst.* **2019**, *30*, 3212–3232. [[CrossRef](#)]
49. Wang, C.Y.; Bochkovskiy, A. YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors. *arXiv* **2022**, arXiv:2207.02696.
50. Ren, S.; He, K. Faster r-cnn: Towards real-time object detection with region proposal networks. *Adv. Neural Inf. Process. Syst.* **2015**, *28*, 1–14. [[CrossRef](#)]
51. He, K.; Gkioxari, G. Mask r-cnn. In Proceedings of the IEEE International Conference on Computer Vision, Venice, Italy, 22–29 October 2017; pp. 2961–2969.
52. Sharif, A.; Marijan, D. DeepOrder: Deep learning for test case prioritization in continuous integration testing. In Proceedings of the 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), Luxembourg, 27 September–1 October 2021; pp. 525–534.
53. Qiao, L.; Li, X. Deep learning based software defect prediction. *Neurocomputing* **2020**, *385*, 100–110. [[CrossRef](#)]
54. Kim, J.; Kwon, M. Generating test input with deep reinforcement learning. In Proceedings of the 11th International Workshop on Search-Based Software Testing, Gothenburg, Sweden, 28–29 May 2018; Association for Computing Machinery: New York, NY, USA, 2018; pp. 51–58.
55. Liu, M.; Li, K. DeepSQLi: Deep semantic learning for testing SQL injection. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, 18–22 July 2020; Association for Computing Machinery: New York, NY, USA, 2020; pp. 286–297.
56. Mirabella, A.G.; Martin-Lopez, A. Deep learning-based prediction of test input validity for restful apis. In Proceedings of the 2021 IEEE/ACM Third International Workshop on Deep Learning for Testing and Testing for Deep Learning, Madrid, Spain, 1 June 2021; pp. 9–16.
57. Oz, M.; Kaya, C. On the use of generative deep learning approaches for generating hidden test scripts. *Int. J. Softw. Eng. Knowl. Eng.* **2021**, *31*, 1447–1468. [[CrossRef](#)]
58. Amalfitano, D.; Fasolino, A.R. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Softw.* **2015**, *32*, 53–59. [[CrossRef](#)]
59. Amalfitano, D.; Riccio, V. Combining automated GUI exploration of android apps with capture and replay through machine learning. *Inf. Softw. Technol.* **2019**, *105*, 95–116. [[CrossRef](#)]
60. Wendland, T.; Sun, J. Andor2: A dataset of manually-reproduced bug reports for android apps. In Proceedings of the 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), Madrid, Spain, 17–19 May 2021; pp. 600–604.

61. UI/Application Exerciser Monkey. Available online: <https://developer.android.com/studio/test/monkey> (accessed on 14 March 2023).
62. Pilgun, A.; Gadyatskaya, O. Fine-grained code coverage measurement in automated black-box android testing. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **2020**, *29*, 1–35. [CrossRef]
63. MLOps. Available online: <https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning> (accessed on 14 March 2023).

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.