applied sciences

MDPI

Article

Finite State GUI Testing with Test Case Prioritization Using Z-BES and GK-GRU

Sumit Kumar [1,*], Nitin [2] and Mitul Yadav [3,*]

[1] Department of Computer Science and Engineering, Veer Madho Singh Bhandari Uttarakhand Technical University, Dehradun 248007, India
[2] Department of Electrical Engineering and Computer Science, University of Cincinnati, Cincinnati, OH 45221, USA; delnitin@gmail.com
[3] Department of Computer Science and Engineering, Dev Bhoomi Institute of Technology, Dehradun 248007, India
* Correspondence: sumitnadar@gmail.com or sumit.kumar@abesit.in (S.K.); mitulyadav1905@gmail.com (M.Y.); Tel.: +91-9891502406 (S.K.)

Abstract: To deliver user-friendly experiences, modern software applications rely heavily on graphical user interfaces (GUIs). However, it is paramount to ensure the quality of these GUIs through effective testing. This paper proposes a novel "Finite state testing for GUI with test case prioritization using ZScore-Bald Eagle Search (Z-BES) and Gini Kernel-Gated recurrent unit (GK-GRU)" approach to enhance GUI testing accuracy and efficiency. First, historical project data is collected. Subsequently, by utilizing the Z-BES algorithm, test cases are prioritized, aiding in improving GUI testing. Attributes are then extracted from prioritized test cases, which contain crucial details. Additionally, a state transition diagram (STD) is generated to visualize system behavior. The state activity score (SAS) is then computed to quantify state importance using reinforcement learning (RL). Next, GUI components are identified, and their text values are extracted. Similarity scores between GUI text values and test case attributes are computed. Grounded on similarity scores and SAS, a fuzzy algorithm labels the test cases. Data representation is enhanced by word embedding using GS-BERT. Finally, the test case outcomes are predicted by the GK-GRU, validating the GUI performance. The proposed work attains 98% accuracy, precision, recall, f-measure, and sensitivity, and low FPR and FNR error rates of 14.2 and 7.5, demonstrating the reliability of the model. The proposed Z-BES requires only 5587 ms to prioritize the test cases, retaining less time complexity. Meanwhile, the GK-GRU technique requires 38945 ms to train the neurons, thus enhancing the computational efficiency of the system. In conclusion, experimental outcomes demonstrate that, compared with the prevailing approaches, the proposed technique attains superior performance.

Keywords: GUI testing; Software Testing Automation (STA); user requirements; State Transition Diagram (STD); ZScore-Bald Eagle Search (Z-BES); Gaussian sinusoid—bidirectional encoder representations from transformers; Test Cases (TC); word embedding; Gini Kernel-Gated Recurrent Unit (GK-GRU); State Activity Score (SAS)

Citation: Kumar, S.; Nitin; Yadav, M. Finite State GUI Testing with Test Case Prioritization Using Z-BES and GK-GRU. *Appl. Sci.* **2023**, *13*, 10569. https://doi.org/10.3390/app131910569

Academic Editor: Andrea Prati

Received: 21 August 2023
Revised: 15 September 2023
Accepted: 19 September 2023
Published: 22 September 2023

1. Introduction

Over the past decade, the graphical user interface (GUI) has proven to be the most promising component in the software development lifecycle due to the user-friendly interactions and experiences it provides. In addition, it is an essential component of most of today's software programs [1]. However, it is crucial to test graphical user interfaces (GUIs) to assure system dependability, which is essential for maintaining the operation of software products and the satisfaction of end users [2]. GUI testing entails performing a methodical analysis of the user interface's graphical elements, interactive components, and visual design to verify that it satisfies the criteria and functionalities outlined as intended [3]. Commonly, various techniques, such as manual-based, record-and-replay, and

model-based, are employed for GUI testing [4]. However, the traditional manual testing methodologies for GUIs exhibit certain limitations regarding efficiency, coverage, and scalability as the complexity of the software system increases [5]. To overcome these issues, significant efforts have been made to integrate machine learning (ML) techniques, including decision tree (DC), random forest (RF), support vector machine (SVM), etc., with GUI testing. This integration has the capacity to enhance testing methodologies, improve accuracy, and hasten the identification of defects, thus improving the GUI's performance [6].

The creation of automated testing processes, such as watir, jmeter, selenium, etc., that can learn from historical data, adapt to changing software environments, and provide valuable insights to developers and testers is enabled through ML algorithms [7]. One of the primary benefits of using ML in GUI testing is that it improves the testers' ability to effectively manage large-scale and complicated GUI designs [8]. However, conventional testing approaches cannot readily accommodate the extensive design space and diverse user interactions that modern applications demand [9,10]. The recurring patterns of user behavior and interactions might be automatically taught by the machine learning algorithms, which would facilitate more complete testing coverage without the need for operator involvement [11,12]. Additionally, the use of ML makes it possible for GUI testing to develop dynamically with the program, reacting to changes in the GUI as well as the associated functionality [13]. However, ML algorithms depend heavily on sufficient and representative training data; insufficient data can cause suboptimal testing outcomes that limit the system's consistency [14]. To resolve this issue, this study proposes a novel framework called "Finite state testing for GUI with test case prioritization using ZScore-Bald Eagle Search (Z-BES) and Gini Kernel-Gated recurrent unit (GK-GRU)" that effectively satisfies the user requirements.

1.1. Problem Statement

The prevailing limitations of this research include:

- Conventional methods often lack the adaptability to handle complex software, resulting in insufficient coverage of high-risk areas.
- Prevailing approaches uncover the hidden flaws and unintended behaviors in complex software systems.
- When dealing with intricate and extensive GUI designs, existing approaches exhibit limited scalability.

1.2. Objectives

- The Z-BES algorithm prioritizes the test cases, focusing on critical areas and efficiently allocating resources for better issue resolution.
- The STD visualization provides a comprehensive understanding of system behavior, helping identify gaps and ensuring alignment with expectations.
- The utilization of RL and fuzzy logic enables accurate labeling of test case outcomes, improving overall evaluation precision.

2. Related Literature Survey

A previously proposed automatic simulation-based testing approach [15] combined domain expert knowledge and autonomous systems' operating and environmental parameters. During the testing procedure, type-2 fuzzy logic was applied to facilitate the robust management of data uncertainty. The efficacy of this procedure was demonstrated experimentally. However, this strategy necessitated a significant time investment and resources and required specialized knowledge.

A testing technique for Android apps was also proposed that encompassed an app classifier, test scripting language, state graph modeler, activity classifier, and test adapter [16]. The framework automated the customization of test scripts for different apps and activities. The empirical evaluation provided evidence that the classifiers' performance was

significantly above average. However, the usefulness of the framework was constrained to applications with intricate user interfaces.

The YOLOv5 algorithm that enhanced the mobile app interface element recognition significantly improved the analysis accuracy and identification of minute components [17]. The experimental findings confirmed its superiority in GUI element identification, demonstrating promise for future development in robot testing automation for mobile applications. However, due to the automated nature of the system, a potential for bias existed.

A test prioritization system grounded in the RL method for user interface testing has also been developed [18]. The associated study sought to maximize the number of detected test defects while simultaneously lowering the necessary testing quantity. The findings demonstrated that the method successfully gained insightful knowledge of the test cases and the linkages between them. Despite this, the system had greater computing expenses, particularly when dealing with applications on a large scale.

A distributed state model inference approach for the GUI testing tool was proposed [19]. The methodology that allowed for the inference of a centralized model made use of a distributed architectural framework. The experiment demonstrated the feasibility of using a model with a distributed approach and the lower time requirement compared with models that are not distributed. However, the system required a large amount of memory and computing power, leading to potential scalability restrictions. The systems, namely, interactive event-flow graphs, GUI-level guidance, and enhanced crowd testers' coverage, have been previously described. The interactive event-flow graphs consolidated the interactions of the testers into a single directed graph, facilitating the visual evaluation of previously tested scenarios. In spite of this, the information that was provided through the interactive event-flow graphs was difficult to read and evaluate.

A deep learning (DL)-centric end-to-end trainable model for GUI similarity and isomorphic GUI identification was established [20]. Visually identifying GUI items while using a DL was required to complete the task. Relative entropy was utilized to allow the measurement of variations in GUI. In addition, the comparison results demonstrated the efficiency of the method. However, the model's performance was relatively sensitive to differences in the GUI design.

A distributed state model inference for scriptless GUI testing has been developed previosly. To conduct effective GUI tool testing, the procedure included empirical evaluation and an approach that did not require scripts. Therefore, the distributed system performed significantly better than the GUI validation test. The performance of the system was hindered by the exclusion of non-deterministic components in the model.

A previous study incorporated GUI modeling and test coverage analysis [21]. In this work, the unified modeling language (UML) was utilized to structure the GUI components. The test cases were then automatically generated from the UML models and underwent test coverage analysis, efficiently achieving GUI testing based on user perspectives. However, this model required extensive testing time.

An automated GUI functional test was presented based on Simplified Swarm Optimization (SSO) [22]. SSO established the relevant test cases to implement the GUI testing. Moreover, an event-interaction graph (EIG) was created to determine the optimal testing for the GUI tools. This system retained the effective performance of GUI testing but exhibited local optimization issues.

Thus, previous research reveals that GUI testing is generally performed via ML and DL models. Models tend to sustain better performance in GUI testing and fault identification to attract users. However, many traditional models face certain challenges, including a lack of data analytics, inefficiency due to variations, massive memory requirements, loss of power, high computational complexity, and limited resources. These types of drawbacks mainly affect the testing outcomes and cause poor performance. Hence, the proposed system focuses on test case prioritizing-based GUI testing, which can more efficiently address the common limitations of traditional models. The proposed work effectively satisfies user requirements by utilizing well-organized frameworks, such as GK-GRU and Z-BES.

3. Proposed Methodology

The proposed technique aims to identify potential mismatches between GUI designs and test cases, ultimately determining whether the GUI meets the requirements which is shown in Figure 1.

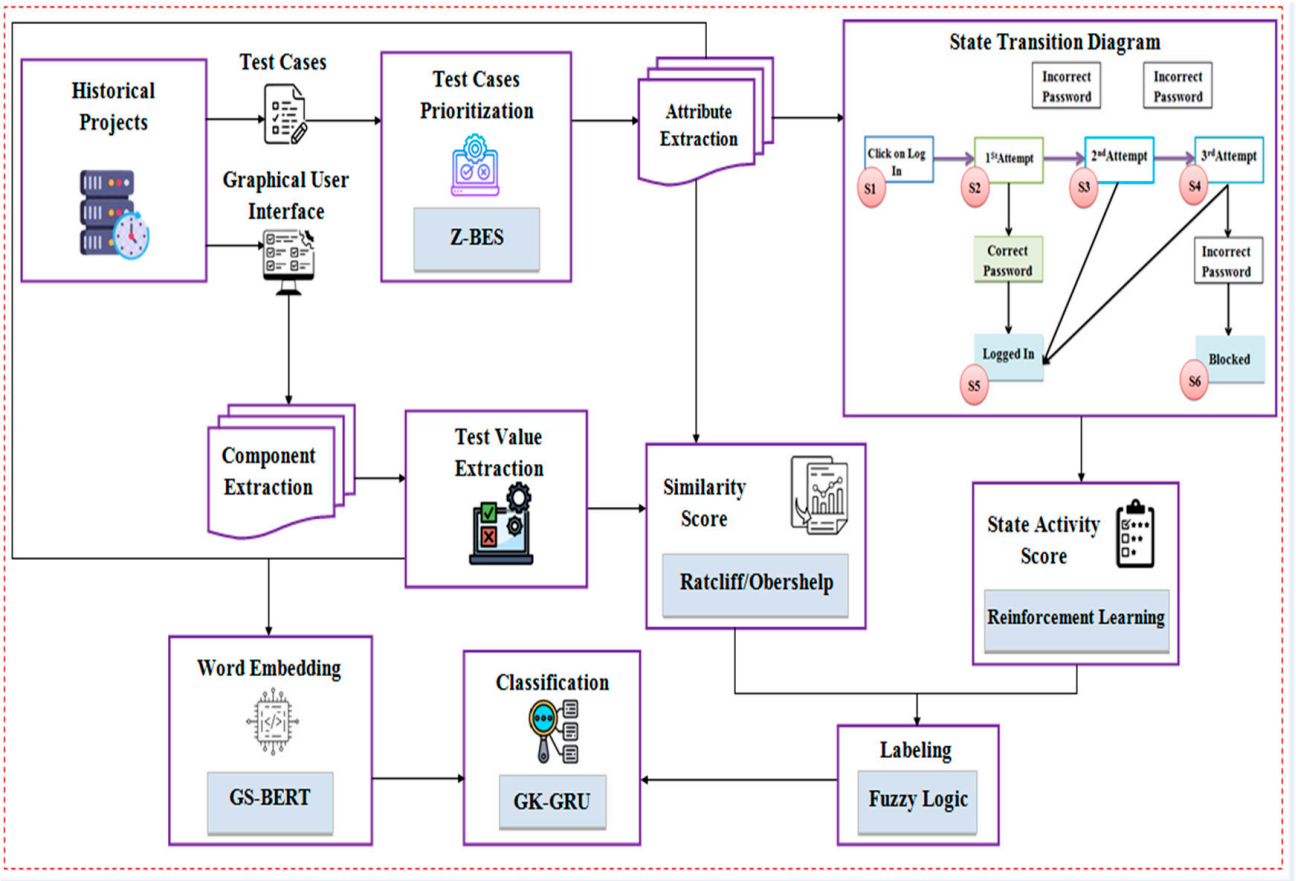

Figure 1. Proposed architecture.

3.1. Historical Projects

The historical project dataset was gathered, comprising the GUI files, test cases, and logs of past testing activities.

3.1.1. Test Cases

Test cases (*Tc*), which outline the steps and expected results when testing the software, are predefined scenarios. Test cases (*Tc*) are represented as follows:

$$Tc = Rq + S + St + E_{out} \quad (1)$$

where *Rq* specifies the requirements, *S* signifies scenarios, *St* exemplifies outline steps, and E_{out} implies the expected outcomes.

3.1.2. Test Case Prioritization

The Tc are prioritized based on their importance and potential impact on the project. By prioritizing Tc, testing efforts are concentrated on areas most likely to impact the project's success. The Tc is prioritized by Z-BES. The BES algorithm achieves optimum solutions with a low number of repetitions. However, due to the influence of mean calculations, the BES algorithm cannot readily determine the optimal areas. The utilization of the mean may not be appropriate for distributions that exhibit significant skewness, which might result in delayed convergence. To mitigate this, the ZScore technique is introduced.

Selecting Stage

The Bald Eagle test population (test cases (Tc)) is represented as:

$$Tc_z = \{Tc_1, Tc_2, Tc_3, \ldots\ldots, Tc_{z\max}\} \tag{2}$$

The fitness function (high prioritized $(Tc)[H(Tc)]$) is then defined as follows:

$$Ft = H(Tc) \tag{3}$$

During hunting, the (Tc) selects the optimal spot within a chosen search area, which is expressed as follows:

$$\beta = \beta_{Best} + \ell * Rd(\beta_{Z-score} - \beta_i) \tag{4}$$

$$Z - score = \frac{(Tc - \mu)}{\sigma} \tag{5}$$

where Rd specifies a random number from 0 to 1, ℓ is the parameter that controls the position changes, β implies the new position, β_{Best} symbolizes the best location, $\beta_{Z-score}$ represents the $Z - score$ position of all (Tc), β_i delineates the current position of the (Tc), μ is the mean of the (Tc), and σ is the standard deviation of the (Tc).

Searching Stage

Tc assesses prey within the chosen search area during this phase. The optimal position $(\beta_{i,New})$ is defined as follows:

$$\beta_{i,New} = \beta_i + P(i) * (\beta_i - \beta_{i+1}) + Q(i) * (\beta_i - \beta_{Z-Score}) \tag{6}$$

where β_{i+1} implies the next position of the (Tc), and $P(i)$ and $Q(i)$ signify scaling factors.

Swooping Stage

The (Tc) moves toward its target prey from the optimal position in the search space:

$$\beta_{i,New} = Rand * \beta_{Best} + P_1(i) * (\beta_i - c_1 * \beta_{Z-Score}) + Q_1(i) * (\beta_i - c_2 * \beta_{Best}) \tag{7}$$

here, c_1 and c_2 are the controlling parameters. Therefore, the $H(Tc)$ is represented as:

$$H(Tc)_{\varsigma} = \left\{H(Tc)_1, H(Tc)_2, H(Tc)_3, \ldots\ldots, H(Tc)_{\varsigma\max}\right\} \quad \zeta = 1, 2, \ldots, \zeta\max \tag{8}$$

The pseudo-code for Z-BEO is presented in Algorithm 1 as follows:

Algorithms 1 Pseudo-code for Z-BEO

Input: Test cases (Tc)
Output: High Prioritized Test Case $H(Tc)$

Begin
Initialize the optimization parameters β_i, Max_I
Calculate the fitness function Fs
For $i = 1$ to Max_i **do**
Select the search space using
$\beta = \beta_{Best} + \ell * Rd(\beta_{Z-score} - \beta_i)$
Search the prey in the search space using
$\beta_{i,New} = \beta_i + P(i) * (\beta_i - \beta_{i+1}) + Q(i) * (\beta_i - \beta_{Z-Score})$
Swoop the prey with
$\beta_{i,New} = Rand * \beta_{Best} + P_1(i) * (\beta_i - c_1 * \beta_{Z-Score}) + Q_1(i) * (\beta_i - c_2 * \beta_{Best})$
If $Fs == Satisfied$
Return $H(Tc)$
Else
$i = i + 1$
End If
End For
End

3.1.3. Attribute Extraction

The important attributes, namely test case ID, test date, version, prerequisites, form name, test data, test scenario, testcase description, step details, expected result, actual result, and customer assigned priority, are extracted from the $H(Tc)$. The extracted attributes (A_b) provide context and information about each test case and are defined as:

$$A_b = \{A_1, A_2, A_3, \ldots\ldots\ldots, A_B\} \tag{9}$$

where B denotes the maximum number of attributes.

3.1.4. State Transition Diagram

The STD is generated using (A_b), which provides a clear overview of how the system behaves and transitions between different states. By visualizing all possible state transitions, the diagram can reveal missing or unintended transitions that could result in errors. The STD has been presented in Figure 2.

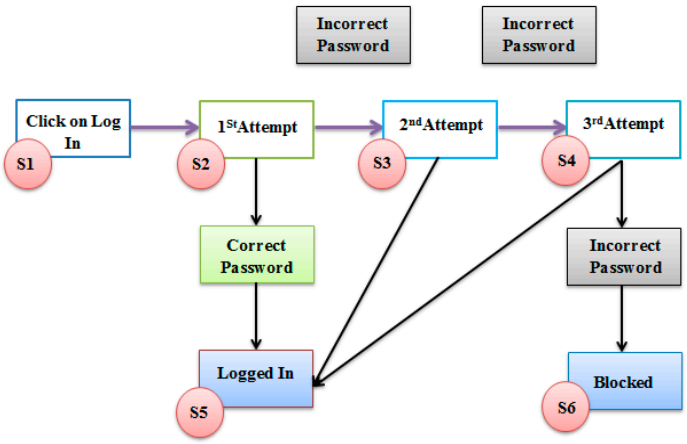

Figure 2. STD architecture.

3.1.5. State Activity Score

Based on the STD parameters, the SAS is computed to quantify the relative importance of each state within the system's behavior. For the score assignment, the RL is utilized. The RL offers advantages in assigning SAS owing to its adaptability to dynamic systems, enabling the algorithm to learn and optimize score assignments centered on interactions and outcomes.

Learning Environment: The RL agent (Ag) interacts with an environment via different states (St) and takes action (Ac) to maximize cumulative rewards (Rw).

Z-value Learning: The (Ag) uses expected cumulative rewards (Z-values) to make decisions. Primarily, Z-values are often initialized randomly.

Reward Feedback: The (Ag) receives (Rw) from the environment for each (Ac) taken. These (Rw) guide the (Ag) toward desirable outcomes.

Updating Z-Values: After each (Ac), the Z-values are updated using Equation (10):

$$Z(St, Ac) = (1 - \kappa) * Z(St, Ac) + \kappa * (Rw + \varphi * Max(Z(\overline{S}t, \overline{A}c))) \quad (10)$$

where the learning rate is implied as κ, the discount factor is notated as φ, and $\overline{S}t$ and $\overline{A}c$ specify the next state and action, respectively.

SAS: The SAS is derived from the Z-values of each state. Higher Z-values specify more active or valuable states. The SAS is notated as (δ).

3.2. GUI

The GUI is based on historical projects. GUIs make software user-friendly by providing a visual and intuitive way for users to interact with software.

3.2.1. GUI Components

The different *GUI* components (G_C), namely buttons, labels, checkboxes, and radio buttons, are identified. This information guides further analysis and testing strategies. The different *GUI* components (G_C) are mathematically termed as:

$$G_C = \{G_1, G_2, G_2, \ldots\ldots, G_{MaxC}\} \quad (11)$$

3.2.2. Text Value Extraction

The next step involves extracting the text content associated with each component. This comprises extracting labels, instructions, options, messages, and any other textual information presented to users. The text values of the *GUI* $E(Tx)$ are represented as follows:

$$E(Tx) = \sum (Tx(GUI)_j) \quad For\, j = 1\, to\, n \quad (12)$$

where $\sum (Tx(GUI)_j)$ signifies the sum of text content from all *GUI* elements and n implies the maximum text values.

3.2.3. Similarity Score

Here, by comparing $E(Tx)$ with the (A_b), a similarity score (α_s) is determined. This comparison helps assess how closely the textual content in the *GUI* aligns with the expected behavior as specified in the test cases. The α_s is computed by the Ratcliff/Obershelp similarity technique. The α_s is estimated by:

$$\alpha_s = \frac{2 * L(lcs(E(Tx), A_b)}{L(E(Tx)) + L(A_b)} \quad (13)$$

where $L(lcs(E(tx), A_b)$ specifies the length of the longest common subsequence $(E(Tx), A_b)$, $L(E(tx))$ signifies the length of $(E(tx))$, and $L(A_b)$ implies the length of (A_b).

If α_s is high, the text content aligns closely, suggesting that the requirement has not changed and the GUI design is likely accurate. Contrarily, if α_s is low, a significant difference

exists in the text values, indicating a potential mismatch between the *GUI* design and the requirement.

3.3. Labelling

Subsequently, α_s and (δ) are inputted into a fuzzy algorithm to determine a label (G). If the similarity score and SAS are higher, the test case result is designated as "pass." However, if either or both scores are low, the test case result is designated as "fail." The proposed work uses the fuzzy algorithm to establish an interpretation of pass or fail based on the combination of these two significant scores. This is presented as:

$$IF\,(\alpha_s = High) AND (\delta = High) \qquad THEN\, Tc_{\text{Result}} = PASS \tag{14}$$

$$IF\,(\alpha_s = High) AND (\delta = Low) \qquad THEN\, Tc_{\text{Result}} = FAIL \tag{15}$$

$$IF\,(\alpha_s = Low) AND (\delta = High) \qquad THEN\, Tc_{\text{Result}} = FAIL \tag{16}$$

$$IF\,(\alpha_s = Low) AND (\delta = Low) \qquad THEN\, Tc_{\text{Result}} = FAIL \tag{17}$$

3.4. Word Embedding

Here, word embedding is performed by the text values of the GUI components $E(Tx)$ and the test case attributes (A_b). Word embedding converts the textual information into numeric vectors, enabling the GK-GRU to process and analyze the data more effectively. The process of word embedding is executed using the GS-BERT algorithm. Although BERT demonstrates proficiency in understanding natural language, it may encounter challenges in appropriately recognizing word order and the impact of word positions. To address this concern, the Gaussian sinusoid encoding method is incorporated into BERT. This entails the incorporation of sinusoidal functions into the positional embeddings (PE) utilized by BERT, enhancing the model's capacity to accurately determine the relative position of words in a given sequence. Consequently, the utilization of GS-BERT results in enhanced numerical embeddings of textual data.

Primarily, the input text (In), which is the combination of $E(Tx)$ and (A_b), is broken into sub-words. Each token is then represented as a word embedding vector (WE) presented as follows:

$$WE(Tk) = \chi_f(Tk) \tag{18}$$

where Tk delineates tokens and χ_f is the word-to-vector function.

PEs are added to the word embeddings to convey sequence order. Here, by using the GS function, the PE is performed and represented as follows:

$$GS(Ps, v) = \text{Sin}(Ps/10000^{(2*(v/Dim))}) * Exp(-((Ps - Max_{Ps}/2)/(0.1 * Max_{Ps}))^2) \tag{19}$$

where Ps implies the position, Dim specifies the dimension at index v, and Max_{Ps} symbolizes the maximum position.

Thereafter, the multi-head self-attention (X) computes weights, which indicate the importance of each word's relation to others. This output is then linearly transformed to produce the final attention representation:

$$X(y_1, y_2, y_3) = \zeta(y_1 {y_2}^{Tp} / \sqrt{Dim_{y_2}}) * y_3 \tag{20}$$

where y_1, y_2, and y_3 are query, key, and value matrices, respectively, and $\sqrt{Dim_{y_2}}$ is the dimension of keys.

A stack of transformer encoder layers (E) captures contextual relationships and produces the numeric vectors (Nu):

$$E(WE(In)) = X(In) + Rc(\gamma(In)) \tag{21}$$

where $X(In)$ computes attention-based representations of the (In), $\gamma(In)$ is a neural network for enhancing the attention output, and $Rc()$ converts In to the transformed output (Nu).

3.5. Classification

Finally, (Nu) and their corresponding labels (G), collectively designated (Y), are fed into the GK-GRU, which accurately predicts whether the test cases pass or fail.

GRU, which is faster than LSTM, utilizes less memory; however, GRU models may encounter challenges, such as slow learning efficiency and extended training times. Hence, the GK function is introduced to address these concerns. This function is designed to enhance the learning process within the GRU architecture, aiming to mitigate issues associated with prolonged training durations and optimize model performance. The GK-GRU architecture is presented in Figure 3.

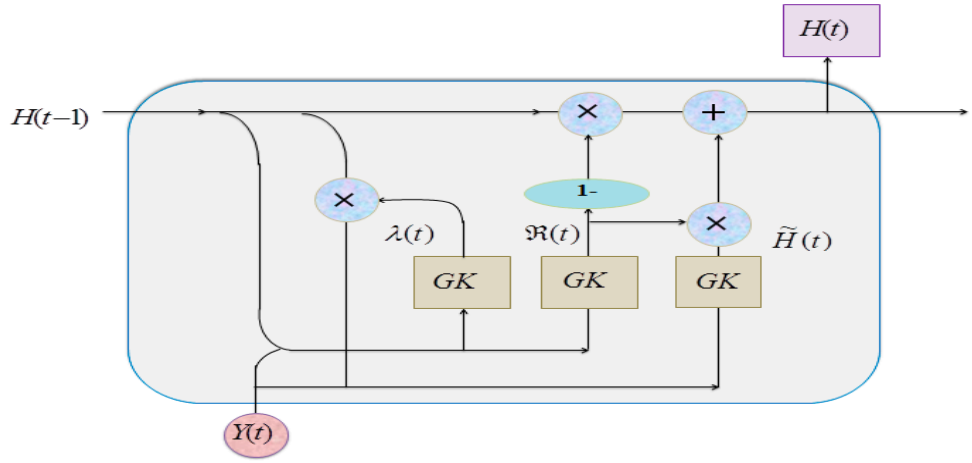

Figure 3. GK-GRU architecture.

The Y is inputted to the GK-GRU, represented by Equation (22):

$$Y_{itr} = \{Y_1, Y_2, Y_3, \ldots\ldots, Y_{itr_{Max}}\} \tag{22}$$

Update gate $(\lambda(t))$: The $\lambda(t)$ controls how much of the previous memory to retain and how much of the new information to incorporate:

$$\lambda(t) = GK(wt_\lambda * [H(t-1), Y(t)]) \tag{23}$$

Here, GK is the Gini kernel activation function, represented as:

$$GK(\lambda) = \frac{1}{2}\left(1 + Er\left(\frac{\lambda}{\sqrt{2}}\right)\right) \tag{24}$$

where λ is the error function, which maps (λ) to the range between 0 and 1.

Reset Gate $\Re(t)$: The $\Re(t)$ is computed to determine how much of the previous hidden state (H) to forget:

$$\Re(t) = GK(wt_\Re * [H(t-1), Y(t)]) \tag{25}$$

The candidate activation $\widetilde{H}(t)$ is computed as per Equation (26), representing the new information to be added to the memory cell:

$$\widetilde{H}(t) = GK(wt_{\widetilde{H}} * [\Re(t) * H(t-1), Y(t)]) \quad (26)$$

$(H(t))$ and memory cell $(M(t))$ are updated using the $(\lambda(t))$ and $\widetilde{H}(t)$.

$$H(t) = (1 - \lambda(t)) * H(t-1) + \lambda(t) * \widetilde{H}(t) \quad (27)$$

$$M(t) = H(t) \quad (28)$$

where t signifies the time step, while wt_{λ}, $wt_{\Re}$, and wt_{H} are weight matrices, and $(H(t))$ determines whether the testcase is designated as a pass or fail.

4. Results and Discussion

Here, the experiments conducted on the working platform of PYTHON are presented.

Performance Analysis

This phase validates the proposed technique's performance. The performance of the proposed GK-GRU and prevailing GRU, long short-term memory (LSTM), recurrent neural network (RNN), and deep neural network (DNN) is elucidated in Figure 4. The proposed GK-GRU achieved remarkable results with a precision of 98.85%, recall of 98.64%, F-measure of 98.95%, accuracy of 98.15%, sensitivity of 98.65%, and specificity of 98.46%, while the other remaining classifiers obtained approximate rates of precision, recall, F-measure, accuracy, sensitivity, and specificity of 93%, 95%, 94%, 93%, 95%, and 91%, respectively. Figure 4 suggests that the GK-GRU model exhibits superior performance to existing models due to its capacity to optimize the learning process, leading to enhanced overall performance.

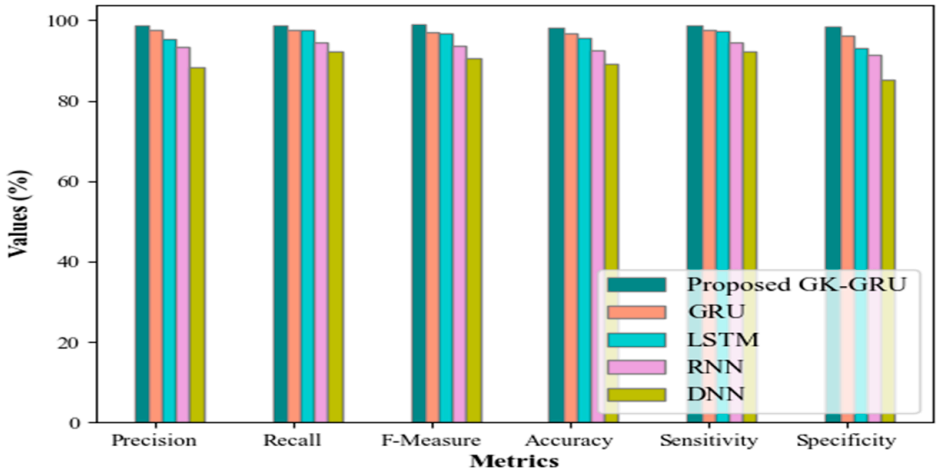

Figure 4. Performance comparison.

Figure 5 compares the performance metrics, including the true negative rate (TNR), false negative rate (FNR), true positive rate (TPR), false positive rate (FPR), and positive predictive value (PPV) for the proposed GK-GRU and the existing models. The GK-GRU mitigates the challenges related to slow learning efficiency and extended training times. Thus, the GK-GRU model exhibits higher TPR (92.25%) and TNR (85.12%) along with lower FPR (14.25) and FNR (7.54) compared to the other models.

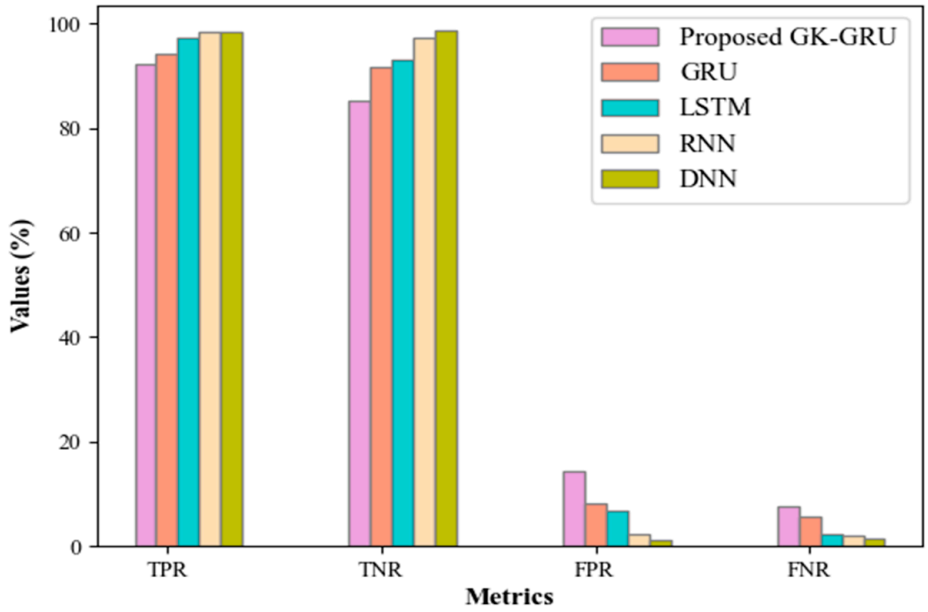

Figure 5. Comparative analysis of the proposed GK-GRU.

Fitness values for the proposed Z-BES and existing Bald Eagle Search (BES), Galactic Swarm Optimization (GSO), Cockroach Swarm Optimization (CSO), and Bacterial Foraging Optimization (BFO) with various iterations (10, 20, 30, 40, and 50) are presented in Figure 6. The proposed Z-BES algorithm attains increased fitness over iterations (5236–9451) as it enhances convergence by providing a more suitable measure for area selection during optimization.

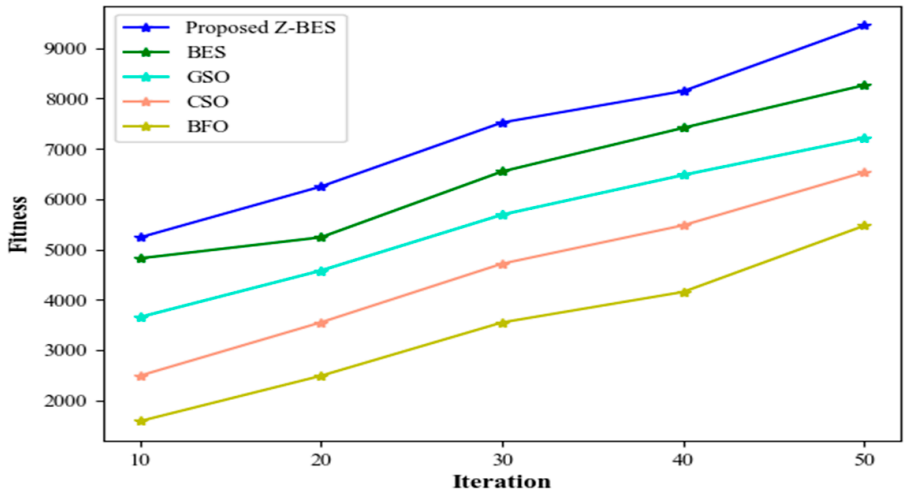

Figure 6. Fitness versus iteration.

The prioritization time for the proposed Z-BES and the existing techniques is presented in Table 1. While the existing technique achieves a prioritization time of 8279 ms, the proposed Z-BES attains the shortest prioritization time of 5587 ms. BES can make more informed decisions regarding the direction and magnitude of changes in search areas using the ZScore, leading to improved convergence with limited time.

Table 1. Prioritization time evaluation.

Techniques	Prioritization Time (ms)
Proposed Z-BES	5587
BEA	6847
GSO	7659
CSO	8765
BFO	9845

The receiver operating characteristic (ROC) curve for the proposed GK-GRU and the existing techniques is depicted in Figure 7. A higher area under the ROC curve indicates that the GK-GRU model has a better ability to correctly classify positive cases while minimizing false positives, reflecting its strong discriminatory power and efficiency in evaluating test cases.

Figure 8 compares the efficiency of the proposed GK-GRU and the existing techniques. The Z-BES prioritizes methodology and facilitates the concentration of testing efforts on crucial areas, effectively distributing resources and promptly addressing significant concerns. Additionally, the GK-GRU model improves the learning capabilities inside the system, addressing issues such as reduced efficiency and prolonged training durations. This adaptation contributes to enhanced overall efficiency. The proposed model has an efficiency rate of 98%, whereas that of all other associated techniques is 90%. Thus, the proposed system retains better performance than the other state-of-the-art techniques.

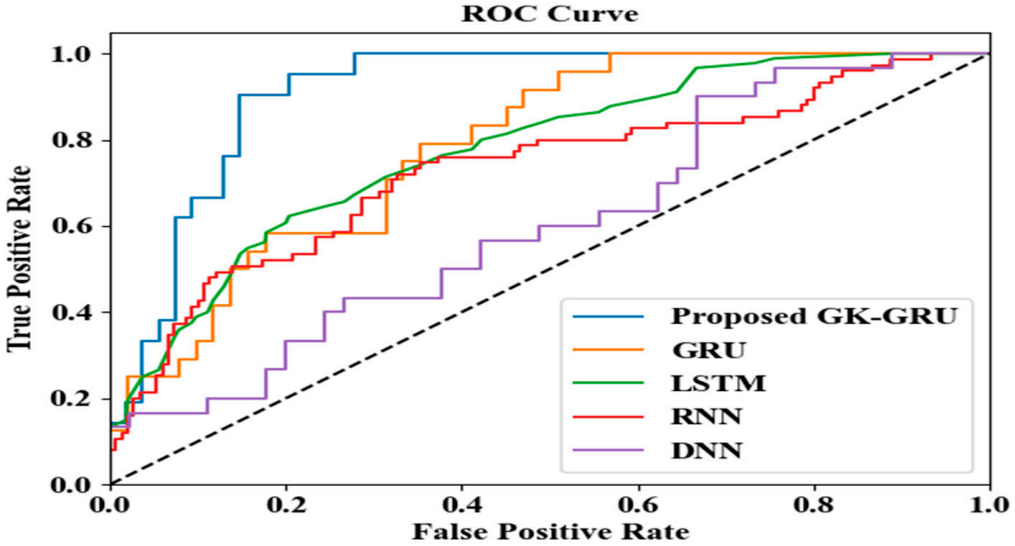

Figure 7. ROC curve.

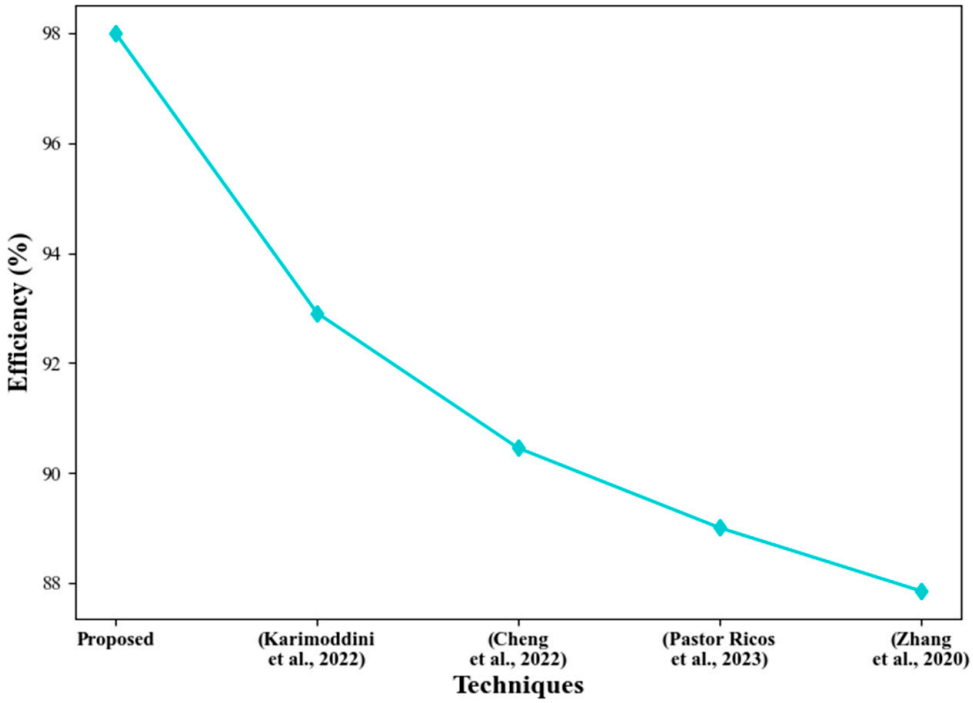

Figure 8. Efficiency comparison [15,17,19,20].

5. Conclusions

By combining Z-BES prioritization and the GK-GRU model, the proposed "finite state testing for GUI with test case prioritization using Z-BES and GK-GRU" framework tackles GUI testing challenges. The proposed technique's performance has been validated by experimentation analyses. The developed Z-BES gains a minimum prioritization time of 5587 at the 10th iteration, which improves the GUI testing process. Likewise, the proposed GK-GRU demonstrates impressive performance metrics, including 98.85% precision, 98.64% recall, 98.95% F-measure, 98.15% accuracy, 98.65% sensitivity, and 98.46% specificity. Moreover, the proposed GK-GRU requires an average of 38,945 ms for the training process, which reduces the time requirements. Furthermore, the proposed technique exhibits low error values and a 98% efficiency rate. Overall, the proposed technique outperforms the prevailing systems and is more reliable and robust. In this work, GUI testing was performed based on the similarity between GUI component text values and test case attribute values, along with state transition. Although this framework performs well for GUI testing, it has small error rates due to the missing GUI appearance and activity attributes that are not well-structured or follow unconventional design patterns. In the future, GUI segmentation might be applied to distinguish the GUI components (e.g., shapes, colors, visual layouts, and activity diagrams) to improve the performance of GUI testing.

Author Contributions: Validation, M.Y.; Writing—original draft, S.K.; Writing—review & editing, N. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data are contained in the article.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Kilincceker, O.; Silistre, A.; Belli, F.; Challenger, M. Model-Based Ideal Testing of GUI Programs-Approach and Case Studies. *IEEE Access* **2021**, *9*, 68966–68984. [CrossRef]
2. Eskonen, J.; Kahles, J.; Reijonen, J. Automating GUI testing with image-based deep reinforcement learning. In Proceedings of the 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2020, Online, 17–21 August 2020; pp. 160–167. [CrossRef]
3. Jeong, J.W.; Kim, N.H.; In, H.P. GUI information-based interaction logging and visualization for asynchronous usability testing. *Expert Syst. Appl.* **2020**, *151*, 113289. [CrossRef]
4. Bons, A.; Marín, B.; Aho, P.; Vos, T.E. Scripted and scriptless GUI testing for web applications: An industrial case. *Inf. Softw. Technol.* **2023**, *158*, 107172. [CrossRef]
5. Jung, S.K. AniLength: GUI-based automatic worm length measurement software using image processing and deep neural network. *SoftwareX* **2021**, *15*, 100795. [CrossRef]
6. Prazina, I.; Becirovic, S.; Cogo, E.; Okanovic, V. Methods for Automatic Web Page Layout Testing and Analysis: A Review. *IEEE Access* **2023**, *11*, 13948–13964. [CrossRef]
7. Yan, J.; Zhou, H.; Deng, X.; Wang, P.; Yan, R.; Yan, J.; Zhang, J. Efficient testing of GUI applications by event sequence reduction. *Sci. Comput. Program.* **2021**, *201*, 102522. [CrossRef]
8. Xie, M.; Feng, S.; Xing, Z.; Chen, J.; Chen, C. UIED: A hybrid tool for GUI element detection. In Proceedings of the 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual, 8–13 November 2020; pp. 1655–1659. [CrossRef]
9. Broer Bahaweres, R.; Oktaviani, E.; Kesuma Wardhani, L.; Hermadi, I.; Suroso, A.I.; PermanaSolihin, I.; Arkeman, Y. Behavior-driven development (BDD) Cucumber Katalon for Automation GUI testing case CURA and Swag Labs. In Proceedings of the 2nd International Conference on Informatics, Multimedia, Cyber, and Information System, ICIMCIS 2020, Jakarta, Indonesia, 11–19 November 2020; pp. 87–92. [CrossRef]
10. Samad, A.; Nafis, T.; Rahmani, S.; Sohail, S.S. A Cognitive Approach in Software Automation Testing. *SSRN Electron. J.* **2021**, 1–6. [CrossRef]
11. Jaganeshwari, K.; Djodilatchoumy, S. An Automated Testing Tool Based on Graphical User Interface with Exploratory Behavioural Analysis. *J. Theor. Appl. Inf. Technol.* **2022**, *100*, 6657–6666.
12. Zhu, P.; Li, Y.; Li, T.; Yang, W.; Xu, Y. GUI Widget Detection and Intent Generation via Image Understanding. *IEEE Access* **2021**, *9*, 160697–160707. [CrossRef]
13. Vos, T.E.J.; Aho, P.; Pastor Ricos, F.; Rodriguez-Valdes, O.; Mulders, A. Testar—Scriptless Testing Through Graphical User Interface. *Softw. Test. Verif. Reliab.* **2021**, *31*, e1771. [CrossRef]
14. Ionescu, T.B.; Frohlich, J.; Lachenmayr, M. Improving Safeguards and Functionality in Industrial Collaborative Robot HMIs through GUI Automation. In Proceedings of the IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2020, Vienna, Austria, 8–11 September 2020; pp. 557–564. [CrossRef]
15. Karimoddini, A.; Khan, M.A.; Gebreyohannes, S.; Heiges, M.; Trewhitt, E.; Homaifar, A. Automatic Test and Evaluation of Autonomous Systems. *IEEE Access* **2022**, *10*, 72227–72238. [CrossRef]
16. Ardito, L.; Coppola, R.; Leonardi, S.; Morisio, M.; Buy, U. Automated Test Selection for Android Apps Based on APK and Activity Classification. *IEEE Access* **2020**, *8*, 187648–187670. [CrossRef]
17. Cheng, J.; Tan, D.; Zhang, T.; Wei, A.; Chen, J. YOLOv5-MGC: GUI Element Identification for Mobile Applications Based on Improved YOLOv5. *Mob. Inf. Syst.* **2022**, *2022*, 8900734. [CrossRef]
18. Nguyen, V.; Le, B. RLTCP: A reinforcement learning approach to prioritizing automated user interface tests. *Inf. Softw. Technol.* **2021**, *136*, 106574. [CrossRef]
19. Pastor Ricos, F.; Slomp, A.; Marin, B.; Aho, P.; Vos, T.E.J. Distributed state model inference for scriptless GUI testing. *J. Syst. Softw.* **2023**, *200*, 111645. [CrossRef]
20. Zhang, T.; Liu, Y.; Gao, J.; Gao, L.P.; Cheng, J. Deep Learning-Based Mobile Application Isomorphic GUI Identification for Automated Robotic Testing. *IEEE Softw.* **2020**, *37*, 67–74. [CrossRef]
21. Paiva, A.C.; Faria, J.C.; Vidal, R.F. Towards the integration of visual and formal models for GUI testing. *Electron. Notes Theor. Comput. Sci.* **2007**, *190*, 99–111. [CrossRef]
22. Ahmed, B.S.; Sahib, M.A.; Potrus, M.Y. Generating combinatorial test cases using Simplified Swarm Optimization (SSO) algorithm for automated GUI functional testing. *Eng. Sci. Technol. Int. J.* **2014**, *17*, 218–226. [CrossRef]