Received April 13, 2021, accepted April 29, 2021, date of publication May 4, 2021, date of current version May 14, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3077518

Model-Based Ideal Testing of GUI Programs–Approach and Case Studies

ONUR KILINCCEKER[1,2], (Member, IEEE), ALPER SILISTRE[3], FEVZI BELLI[1,4], (Member, IEEE), AND MOHARRAM CHALLENGER[5], (Member, IEEE)

[1]Department of Computer Science, Electrical Engineering and Mathematics, Paderborn University, 33098 Paderborn, Germany
[2]Department of Computer Engineering, Mugla Sitki Kocman University, 48000 Menteşe, Turkey
[3]International Computer Institute, Ege University, 35040 İzmir, Turkey
[4]Department of Computer Engineering, Izmir Institute of Technology, 35430 İzmir, Turkey
[5]Department of Computer Science, University of Antwerp and Flanders Make, 2020 Antwerp, Belgium

Corresponding author: Onur Kilincceker (okilinc@mail.upb.de)

This work was supported in part by the University of Antwerp under Grant 43169.

ABSTRACT Traditionally, software testing is aimed at showing the presence of faults. This paper proposes a novel approach to testing graphical user interfaces (GUI) for showing both the presence and absence of faults in the sense of ideal testing. The approach uses a positive testing concept to show that the GUI under consideration (GUC) does what the user expects; to the contrary, the negative testing concept shows that the GUC does not do anything that the user does not expect, building a holistic view. The first step of the approach models the GUC by a finite state machine (FSM) that enables the model-based generation of test cases. This is always possible as the GUIs are considered as strictly sequential processes. The next step converts the FSM to an equivalent regular expression (RE) that will be analyzed first to construct test selection criteria for excluding redundant test cases and construct test coverage criteria for terminating the positive test process. Both criteria enable us to assess the adequacy and efficiency of the positive tests performed. The negative tests will be realized by systematically mutating the FSM to model faults, the absence of which are to be shown. Those mutant FSMs will be handled and assessed in the same way as in positive testing. Two case studies illustrate and validate the approach; the experiments' results will be analyzed to discuss the pros and cons of the techniques introduced.

INDEX TERMS GUI testing, holistic testing, ideal testing, model-based testing, mutation testing, test generation, regular expression.

I. INTRODUCTION

The main goal of program testing is to show the presence of faults, not to show their absence, which Dijkstra [1] expressed in 1970. This purpose becomes the main acceptance of the testing community. However, Goodenough and Gerhart [2] in 1975 proposed a theorem that claims adequately designed tests could show not only the presence but also the absence of faults. Based on their theorem, this test with the ability to show the absence of faults requires being both reliable and valid. Test results need to be consistent concerning reliability, while this test also needs to be skillfully designed to detect defects concerning validity. The test approach is called an ideal test if and only if it satisfies these conditions. Ideal testing refers to a methodology that satisfies reliability and validity requirements for testing both the presence and absence of faults.

Moreover, an exhaustive testing approach utilizing proper termination criteria can be the ideal test [2]. Chow [3] stated that a test approach holding reliability and validity conditions is not possible at the program level. He used the specification rather than program code to provide a viable solution for achieving the ideal test and provided formal proof to support his claim.

However, Chow [3] suggests that the steps required to obtain the ideal testing can only be achieved with a W-Method test generation algorithm. Also, he proposes proof that only satisfies the reliability criterion in his study. While it is emphasized in his study that it is not possible to obtain the ideal testing method for the code-based test, it is observed that reaching the ideal testing for the specification is insufficient only by satisfying the reliability criterion. The present study proposes a more systematic approach using the system's

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

specification and meeting both reliability and validity criteria.

Graphical User Interface (GUI) testing is an evaluation process for the correctness of the software's GUI. It is a very significant section of Software Engineering and a crucial part of the Software Development Life Cycle (SDLC). It is required to be a section of the development process from the beginning of application development. In general, software applications' correctness and usability are essential and may constitute a significant reason to choose one software over another. To attract the users, software developers must consider the User Experience (UX) and GUI of their applications and their correctness. Flaws or faults in GUIs will result in dissatisfaction of the customers. Because of this, catching flaws and hidden faults in an application GUI is critical before deploying the software.

GUI testing is a process of testing the visual elements and their design to limit the probable problems. Component type, size, color, font are just a few examples of those elements that we can test in an application. More importantly, the business logic of an application can be tested with GUI testing via automation. Automated GUI testing can detect faults in an application with the help of automation tools. Automated GUI testing is significant because manual GUI testing is a prolonged and costly process. With test automation, a significant reduction in test time and cost could be achieved.

As with other software testing methods, GUI testing approaches have been suggested to indicate the presence of a fault. However, GUI originated functional faults are mostly caused by GUI components. An example of the functional fault category is called the ''Action'' fault in the literature [4], which is frequently encountered in these components. It can be given when a GUI user presses a button, and there is no action or a faulty action. Instead of detecting the presence of such faults, showing their absence will make it easier for the tester and prevent the GUI user from experiencing such a fault. The present study suggests a method that allows showing both the presence and absence of the fault in this respect.

Along with introducing a toolchain for automated GUI testing, this paper introduces a methodology for GUI testing by addressing functional faults. It uses Holistic Testing (HT) [5], [6] and Mutation Testing (MT) [7], [8] to achieve ideal testing of the specification (model) of a GUI instead of its program code.

Conventionally, the HT offers an integrated perspective as a joint test of expected and unexpected functions. For example, for a banking application, it is a function that the user is expected to be able to log into the system using the correct personal information successfully. The fact that the same user can log into the system with wrong information is an unexpected function. The HT integrates this bi-directional perspective into its test methods. The present method applies the necessary steps for HT to a model-based testing approach. For this, it uses different and specific models that contain expected and unexpected functions. While the conventional HT uses models that use expected functions to test unexpected functions in certain studies, the current study uses a separate model for each unexpected function. In this way, it is possible to obtain test suites specific to each unexpected function. While advocating testing all certain unexpected functions with a single set of tests, the current study argues that different sets of tests are required for each different function.

MT, a technique for measuring test effectiveness [9], [10], is a fault-oriented testing technique. It utilizes mutants acquired by seeding faults into the specification (model) or directly to the program using proper mutation operators. To measure test inputs' effectiveness, the mutants are killed or survived related to the test execution of those test inputs on those mutants. Finally, the mutation score is calculated based on the results of those executions [11]. The MT was originally introduced to test code-based testing methods. Mutation operators are varying specific to systems written in different languages. For example, there are a total of 24 operators for object-oriented errors with the MuJava tool for the Java language. Adaptation of MT to specification-based test methods, unlike its use for code-based test methods, is among current research topics. This approach is commonly referred to as model-based mutation testing (MBMT) [6], [12]. Mutation operators are applied to the model in the MBMT. Therefore, mutation operators are diversifying for different models.

The current work uses the code-based mutation test (CBMT) for evaluation while using MBMT to obtain model-based mutants. Unlike conventional MBMT, mutant models are used for test generation in the current work. Moreover, traditional CBMT approaches randomly generate mutants using appropriate mutation operators. This results in a lot of redundant mutants that have nothing to do with real faults. Also, some mutants may be equivalent to the system under test. Eliminating equivalent mutants is one of the major challenges of mutation testing. While similar situations occur in MBMT approaches, randomly generated mutant models in MBMT may also cause non-determinism. Thus, identifying equivalent and non-deterministic mutant models in MBMT is a challenging process. The present study follows a more systematic and novel approach for the difficulties that arise in CBMT and MBMT using mutant and code-based mutants specific to the real faults that may occur in the system.

The proposed approach is called Model-based Ideal Testing (MBIT). This methodology is rather general and can be adapted to other application domains. We have other works on applying MBIT to the validation of hardware design. The current work adopted the HT because it is an integrated and complementary view for which it uses the negative testing (NT) aside from the positive testing (PT). The HT acquires the legal (expected) test inputs using the fault-free model, applied to the GUI under test for the PT. Moreover, it acquires the illegal (unexpected) inputs using the mutant model, which are also applied to the GUI under test for the NT.

The experimental and theoretical studies carried out within the scope of this study are designed to answer the research questions (RQs) given below:

1) Is it practically and theoretically possible to offer an ideal testing [2] approach for GUI testing?
   - What types of systems can be tested in this way?
   - What types of faults can be targeted with the proposed approach?
2) What is the cost of applying this approach to GUI testing?
3) How is scalability affected?

Considering the experimental and theoretical studies carried out in this work, the research questions mentioned above are examined in detail in Section VI-B.

Our previous conference paper [13] applied MBIT for validation of hardware design to target specific design fault. It is only evaluated on a demonstrating example with a little experimental setup without comparison. It also neglects two essential selection criteria in the MBIT. In the current work, we adapt and extend MBIT to GUI testing for targeting GUI-related functional faults and evaluate MBIT on mature case studies, including comparison with three different approaches. The current work uses two selection criteria for the algorithmic correctness of the models. To this end, the current work provides the following contributions:

1) Unlike conventional usages of the HT and MT, they are adapted to achieve the ideal test suites for GUI testing for the presence and absence of faults
   - The HT is adapted by offering different test suites for each different fault
   - The MT is customized by acquiring mutants for test generation
   - A methodology is provided to target functional faults for GUI testing, including an informal proof for being MBIT
2) An experimental evaluation for the current methodology is presented
   - Two mature GUI case studies are used to evaluate the current work
   - Three different test generation approaches and tools are utilized for comparison
3) A tool support is developed and provided
   - The MBIT is partially automated for GUI testing
   - The toolchain including examples and details are provided in a bundle[1]

By providing an experimental study with two case studies, a comparison was made for a total of four test generation approaches, including an industry scale tool called Graphwalker. Using the results, the proposed approach has been evaluated comparatively.

The rest of the paper is organized as follows: Section II summarizes the related work on the HT, code and model-based GUI testing, GUI testing, ideal testing, and PQ-Analysis. The proposed methodology is presented in Section III, including detailed stages and used notions. Section IV presents two case studies for experimental evaluation. Section VI presents a discussion on RQs and threats to their validity. Finally, Section VII concludes the paper and presents the possible further studies.

[1] MBIT, https://kilincceker.github.io/MBIT4SW/

II. ELEMENTS OF THE APPROACH AND RELATED WORK

This section presents related work and background information for the current work. Holistic testing (HT), code-based MT, MBMT, GUI testing, ideal testing, and PQ-Analysis are given in the following subsections.

A. HOLISTIC TESTING (HT)

The HT proposed by Belli [5] requires the testing of the system's desirable and undesirable features by using the PT and NT. In the PT, a system is checked against desired outcomes by using legal (expected) input variables. In the NT, a system is checked against undesired outcomes using illegal (unexpected) input variables. For example, consider testing a website for the user profile page; a tester enters a numeric variable for testing a box corresponding to the social security number in the PT. However, a tester enters the alphabetic variable for testing the same box in the NT. In this example, a numeric variable is a valid input, whereas an alphabetic variable is an invalid input.

Belli *et al.* [6] adapted the HT to model-based testing by generating legal (expected) and illegal (unexpected) test suites using models. They propose a graph-theoretic approach for modeling the system under test, and this model is called the Event Sequence Graph (ESG). They acquire test inputs from the ESG model in the PT and ESG's complements in the NT. In ESG's complements, all illegal (unexpected) features are included. Test inputs acquired from this complement graph contain illegal (unexpected) input variables representing these undesired functions. The HT is already used to model and test graphical user interfaces [14], web service composition [15], web application [16], interactive systems [17], hardware designs [18], and android applications [19].

We utilize holistic testing in the current work to show the presence and absence of specific faults concerning positive and negative testing.

B. CODE AND MODEL-BASED MUTATION TESTING

The code-based mutation is applicable in white box testing, where the source code of the software under test is available. Moreover, MBMT is appropriate for black-box testing in which the source code is not available. There is another approach called grey-box testing in which both source code and model are available.

DeMillo *et al.* [8] proposed the MT in their seminal paper. The MT is a fault-oriented technique that uses a given software program's mutation. A mutation contains a simple fault caused by making small changes in the original software program. A generated test data is executed on each mutant, and the results are compared with the result of the original program's test execution results. If the result of the test data differs from the result of the original test data, then the corresponding mutant becomes dead; otherwise, it is still alive because the test data result does not make any difference. Therefore, the two cases could occur. The test data does not contain enough sensitivity to distinguish between the mutant and the original programs, so the mutant is equivalent; thus, there is no test data to detect the fault.

DeMillo *et al.* [8] emphasized the power of the coupling effect that states the test data that distinguish only simple faults could also be sensitive to cover more complex faults. The MT method is a powerful and elegant method that is applied to both software and hardware testing [13], [24]. The only consideration is the cost of this method, which increases very quickly related to the program's size that directly affects the number of the mutants. A comprehensive literature review in the form of a ''mini-handbook''-style road-map for the MT is given in [25].

King and Offutt [20] presented an MT framework with the 22 mutation operators for the Fortran 77 version of the Mothra system, a software testing environment. The Mothra achieves the highest mutation (adequacy) score for the set of test cases executed on mutant and original programs. The Mothra system generates 970 mutants for a 27-line program. These results are computationally and spatially expensive due to the excessive number of mutants. Therefore, the Mothra handles this problem by utilizing incremental compilation.

Wong and Mathur [26] offered an empirical study to reduce unacceptable computational expenses due to the number of mutants. One of the proposed solutions is randomly selected from a subset of all mutants (x). Earlier investigation shows that a random selection of 10 to 100 of all mutants makes dramatic reductions in requiring efforts while keeping the MT's effectiveness. They increase x by 5 up to 40 to examine the cost and power of the MT. Another offered solution is constrained mutation that requires selecting a few specific types of mutants and neglecting the others. They state that proper selection of a small set of mutant types significantly lessens the MT's complexity and still keeps nearly the same fault detection ability of the MT.

Ma *et al.* [21] introduced the MuJaVa tool for the MT, including the GUI of the Java programming language for both method and class-level mutation with related levels of mutation operators. The method-level mutation operators change the expressions by replacing, deleting, and inserting operators. The class level mutation operators are responsible for object-oriented attributes: inheritance, polymorphism, and dynamic binding. The MuJava contains the mutant generator, including an engine to detect equivalent mutants, the mutant executor, and the mutant viewer components. However, it is reported that the MuJava is still very slow for a large set of mutants.

Jia and Harman [27] presented a comprehensive analysis and survey for the MT. It is also mentioned that the new trend in the MT is going to be the semantic effects of mutants rather than syntactic effects.

Fabbri *et al.* [22] provided an MT technique to validate state chart-based specifications. The technique uses a set of mutation operators: the finite state machine, extended finite state machine, and state charts-feature-based operators [22]. The set contains 37 mutation operators. They also utilize an abstraction strategy, namely the Hierarchical Incremental Testing Strategy (HITS), to make the technique more feasible for conducting a modular and incremental testing activity. However, they also state that tool support becomes mandatory for testing large-size statecharts.

Belli and Beyazit [23] made a comparison of the event-based and state-based approaches for MBMT. The event-based approach uses the event sequence graphs (ESG), [23] whereas the state-based approach uses the finite state machine (FSM). The comparison criteria are mutation operators, coverage criterion, and test generation method. The mutation operators are sequence insertion, sequence omission, event insertion, and event omission for the ESG model, while transition insertion, transition omission, state insertion, and state omission are used for the FSM model. The coverage criterion is event pair coverage for ESG and transition coverage for FSM. However, the test generation method for specific coverage criteria roughly requires solving a well-known problem, namely the Chinese Postman Problem (CPP). The CPP requires visiting every edge of a graph to find a shortest path. They report that the FSM-based test sequences comprise more redundancy and cover 40 to 100 more failures. However, the cost becomes roughly 52 to 122 higher. The ESG covers 29 to 50 fewer failures while it costs roughly 30 to 55 less due to event sequences clustering. Experiments conclude that the FSM-based test results are more effective for covering more failures because of the redundancy.

Belli *et al.* [6] proposed an MBMT method providing proper mutation operators, namely omission and insertion operators, evaluated fault detection ability of test set acquired using the mutated model, and surveyed the literature on the MBMT. They validate the effectiveness of three examples that are industrial and commercial real-life systems. Experiments show that the insertion operator is more efficient than the omission operator because it reveals more faults.

Kilincceker *et al.* [13] proposed a hybrid MT approach that combines code-based mutation testing and MBMT to validate the hardware design. They use code-based mutation for test execution. They select the regular expression (RE) model for test generation due to its algebraic and declarative power. They also theoretically and experimentally proved that the proposed method satisfies the conditions of Goodenough and Gerhart's ideal testing [2].

To summarize the available studies in the scope of code and model-based mutation testing, we have provided a comparison table, Table 1, in which the mutation operators and effectiveness of each approach have been elaborated.

We adapt the code-based mutation testing approaches presented in [8], [20]–[22] to obtain code-based mutants from the original program using mutation operators. The authors of [6], [22], [23] offer model-based mutation testing that we

TABLE 1. Comparison of mutation testing methods.

	Method	Code and Model	Mutation Operator	Effectiveness
Code-Based Mutation	[8]	Fortran-like programs	Logical Expression replaces	Usage of simple faults to detect complex faults via coupling effect
	[10]	VHDL programs	The ten mutation operators	N/S
	[20]	Fortran Programs	The twenty-two mutation operators	Providing The Mothra software testing environment
	[21]	Java programs	twelve method level and twenty-eight class-level operators	N/S
Model-Based Mutation	[22]	Statecharts	Eight for FSM and eleven for Extended FSM	Providing Proteum/ST software testing environment
	[23]	Event Sequence Graph (ESG)	Insertion, Omission, Replace operators	Comprehensive comparison of event-based and state-based models
	[6]	Event Sequence Graph (ESG)	Insertion, Omission, Replace operators	Proposing a MBMT approach with three real life case studies.
Hybrid	[13]	Finite State machine (FSM)	Semantic mutants operators which require higher order mutation	Combine the model and code-based mutation testing

utilize in the current work to construct model-based mutants from the original (fault-free) model by using model mutation operators given in [18] for the FSM model. We use the similar idea proposed in [13] as being a hybrid approach by applying code-based and model-based mutation testing methods at the same time.

C. GUI TESTING

The process of testing the GUI-oriented of software applications, i.e., one that has a GUI front-end and there are available events(''enter a text'', ''click on a button'', ''select an item from a dropdown'') that can be applied on GUI widgets (e.g., ''text-field'', ''button'', ''dropdown'') to perform actions in the system, is called GUI testing. The GUI testing process can be carried out effectively through a well-selected model, i.e., Finite State Machine (FSM) [28], Event Flow Graph (EFG) [29], [30], Event Sequence Graph (ESG) [5]. The FSM, EFG, and ESG are graph-based models. Optimization and traversal algorithms need to be applied to them to produce test sequences.

Shehady and Siewiorek [28] implemented a formal way to describe a GUI, called a Variable Finite State Machine (VFSM). The VFSM is then transformed into an FSM to be used in test generation using a well-known W-Method, initially introduced by Chow [3]. The W-Method requires a completely defined FSM, so there could be many NULL transitions within the model. Although the VFSM requires fewer states than the FSM, the proposed algorithm runs on a large number of states of the FSM.

ESG is proposed by Belli [5] to be used in modeling the GUI. Additionally, a testing method is also proposed to be used in this novel model. The ESG describes the events at the vertexes and the relationship of the events at the edges of the given GUI. Testing methods in [5] merge the PT and NT to obtain a holistic viewpoint. In the NT, the system is tested against illegal inputs (incorrect behavior). In the PT, it is tested against legal inputs (correct behavior) in compliance with user expectations. The suggested approach is thus generic, describing all correct and incorrect behaviors.

Memon *et al.* [29] presented a test generation algorithm based on artificial intelligence-based planning using the EFG model. They also propose the generation of a hierarchical model from the given GUI structure. The EFG model is constructed, and then the planning algorithm is implemented, which involves the specification of a collection of operators, an initial state, and a target state. Then by considering GUI events and interactions, the algorithm produces test sequences between the initial and target states. They are also using GUI model decomposition to deal with the issue of scalability.

Memon [30] recast the existing idea of event-based GUI testing via model-based techniques called event-space exploration strategies (ESES). He decreases the cost and effort of event-flow techniques and automates the procedure to enable extensive experiments and simplify the model creation step.

Xie and Memon [32] presented a new concept called Minimal Effective Event Context (MEEC) and used this in an empirical way for fault detection. Because generally, GUIs are implemented as a collective of widgets with their event-handlers and response to event-handlers. Generating long test cases becomes expensive. The purpose of modeling MEEC is to create an abstract model of GUIs and then generate the shortest ''potentially'' problematic event sequences for test case generation.

Huang *et al.* [33] developed a method to repair GUI test suites, which suffer from in-feasibility because graphs are generally used to acquired test cases, and they are created from all possible sequences of events. There is a possibility that an event inside these kinds of test sequences may not be available for execution and terminate early. They used a genetic algorithm to fix these problematic test suites and increase test coverage.

Belli *et al.* [34] discussed and applied a case study about GUIs reliability and selecting a GUIs reliability model in human-machine systems. In the provided work, they indicate that selecting an appropriate modeling technique for GUI testing affects the quality of the assessment process, hence the software.

Banerjee *et al.* [35] researched GUI testing articles and studies and matched them with a systematic mapping technique. They defined selection criteria for studies from the pool of 230 articles written between 1991 and 2011 about GUI testing. They classified studies, provided an overview of existing approaches, and spotted areas that require more study and research.

Belli *et al.* [14] presented a study about reviewing and summarizing existing works on model-based GUI testing. They also provide the PT and NT with their examples taken from real projects. They gave examples from conventional and modern techniques for model-based GUI testing. They also covered test-case construction and optimization of the process.

Alegroth and Feldt [36] provide a case study including comprehensive and qualitative work for visual GUI testing (VGT) in industrial practice. The study is carried on

TABLE 2. **Comparison of GUI testing methods.**

Method	Model	Coverage Criteria	Pros	Cons
[28]	Variable Finite State Machine (VFSM)	N-switch set cover	Covers nearly all predefined faults	Results excessive number of redundant test cases
[31]	Finite State Machine (FSM)	Complete interaction sequences (CIS)	N/S	N/S
[5]	FSM and RE	Complete interaction sequences (CIS)	Testing GUI with the PT and NT	N/S
[29]	GUI Tasks	N/S	Intuitively and easily scalable for larger GUIs	Tasks are chosen by test designer that may yield inadequate coverage
[30]	Event Flow Graph (EFG)	N/S	Quickly generate test cases without scalability problem	N/S
[32]	Event Interaction Graph (EIG)	Genetic algorithm–CIT coverage	Increase the feasible coverage of these suites	N/S

well-known music streaming application company, Spotify. They attempt to answer three research questions about problems, challenges, and limitations for adaptation of automated VGT on the company using the Sikuli [37] test automation tool and Graphwalker [38] model-based testing tool at the industrial level. They also explain why the VGT was abandoned in the company due to organizational changes based on experiences. Finally, they present an automated GUI testing solution for Spotify company by focusing on adaptation of Graphwalker. However, they neglect effectiveness of Graphwalker than others. The current work not only adapts the Grapwalker to proposed methodology but also provides experimental evaluation over others.

Besides automated methods based on test automation tools, some works utilize machine learning methods, especially deep reinforcement learning [39], [40] for automatically traversing the GUI. Eskonen *et al.* [39] present an image-based deep reinforcement learning method for the exploration of GUI structure. The introduced method mainly focuses on learning GUI behaviors by feeding screenshots of the GUI to the neural network and letting the learning method explore the GUI events. They also compare the exploration efficiency of the algorithm with Q-learning and random exploration methods. However, they do not provide an appropriate experimental evaluation of how the method effectively catches faults. Similar to [39], Adamo *et al.* [40] presents a reinforcement learning method for automated GUI testing based on exploration. They utilize a Q-learning algorithm that outperforms the random exploration approach based on experimental evaluation concerning only code coverage efficiency. They also do not provide any information regarding fault coverage.

Jan et. al [41] present a test generation approach to exploit security vulnerabilities of web application represented by GUI. They offer a security attack mechanism addressing SOAP communication of web application based on search-based techniques. They analyze four different algorithms and two fitness functions and provide a comprehensive experimental work based on a large set of case studies including two industrial examples. Based on experimental evaluation, their automated approach is effective to generate such security vulnerability of web application.

In addition to GUI testing of web applications, the automated test generation for mobile applications is having increasing interest in both academia and industry due to becoming indispensable part of our daily life of smart phones. Arnatovich and Wang provides [42] a systematic literature review for automated GUI testing of mobile applications. Jiang *et al.* [43] carry out a systematic study on factors affecting GUI testing of Android applications. Salihu *et al.* [44] offers an automated GUI testing approach (called AMOGA) for mobile application based on static-dynamic model generation. The AMOGA tool based on model-based testing utilizes static ana dynamic analysis to construct model (called Windows Transition Graph (WTG)) of mobile application. It uses a crawling algorithm to explore GUI states within a depth first search manner. It achieves 0.90 mutation score in the experimental part for which it uses MuDroid, a mutation testing tool, for mutant generation of 15 mobile applications.

Ardito et. al [45] introduce a testing framework which contains a test script language (1) for writing generic test scripts, a modeler (2) to define activities and widgets of the application, a classifier (3) to determine type of the application under test, an activity classifier (4) to define objective of the screens, and an adapter (5) for execution of implemented test scripts on the application. They utilize a deep neural network for the classification phase of the framework that are evaluated on 32 different mobile applications. However, a complete evaluation of the proposed framework with other approaches are missing and the used metrics for experiments are poor.

Table 2 presents a comparison among the studies related to GUI testing, focusing on each approach's advantages and disadvantages. It also gives the models used in each study and the coverage criteria for the study.

In the current work, we model GUI under test as given in [46] and partially in [28]. Then, we use the FSM model for a mutant generation. However, we convert them to the RE model for test generation that is different from [28], [46]. Our modeling approach is more similar to [5] in which the authors do not offer a test generation approach that we propose in the current work. In [30], [32], the authors use a very different modeling methodology than the current work using various node types for different GUI events. The advantages and disadvantages of these different models for GUI testing are elaborated in [47]. The main bottlenecks of the exploration-based solutions [39], [40] are unnecessary test inputs and automatic exploration of GUI events without knowing the correct input(s) to trigger a fault. However, the exploration-based methods are valuable for automatic extraction of the GUI model (called GUI ripping), accompanied by static analysis methods to eliminate low-quality and redundant test suites. Therefore, the current work focuses on the model-based testing approaches due to their robustness and determinism. In contrast to the current methodology, the approaches in [44], [45] provides a solution for GUI testing of mobile applications.

D. IDEAL TESTING

Goodenough and Gerhart [2] define the ideal test based on its principal conditions. The theorem states that in the case of test data satisfying these conditions, namely reliability and validity, this test data enables testing the absence of faults. They also provided proof of this fundamental theorem in [2]. Howden [48] introduced a testing method for analysis of paths, namely P-Testing, and evaluated the ideal test in terms of reliability condition. The reliability of P-Testing is checked against different types of common faults. However, Howden [48] stated that P-Testing is reliable or almost reliable for subset faults not covering all faults. Bouge [49] extended the ideal test's current conditions by offering additional features, namely bias and acceptability. Bouge [49] also provided a detailed relation of program testing and program proving for bias and acceptability conditions. Langmaack [50] presented sufficient and readable proof for compiler verification, considering the ideal test for verification and software testing. The main inspiration of the current work is based on the seminal work of Goodenough and Gerhart [2].

Before the informal definition of the ideal testing, some terms require to be clarified. These terms are Program (p), Test case (t), Selection criterion (c). Program (p) pairs domain (D) to range (R) as being a function. Test case (t) is an input (i) and expected output (o) tuple. Selection criterion (c) is a requirement to select some test cases for a specific reason (e.g., fault detection).

The following definition of the ideal test is summarized [51] based on the given terms above.

Definition 1: OK(d) implies the availability of the outcome of (t) as being a predicate based on p(t), execution of t on the program p. OK(d) = true if and only if (t) is an adequate output o. OK(d) = false if and only if (t) is not adequate output o.

The OK(d) examines the adequacy of a test case. In case entire test cases become adequate, T turns out successful in terms of the given definition below.

Definition 2: Successful(T) characterizes an achievement of a set of test cases (t_s) that belongs to T. T is a successful test if and only if (t_s) belongs to T and OK((t_s)). Successful (T) = true if and only if t belongs to T and OK(t) or Successful (T) = false test if and only if ts belongs to T and not OK((t_s)).

OK(d), Successful(T), Fail(T), Satisfy(t,c) as being predicates are supported to characterize reliability and validity conditions to attain an ideal test.

Definition 3: Reliable Criterion refers to consistency for the chosen test suite demonstrated by Reliable(T).

Definition 4: Valid Criterion refers to the capability of the chosen test suite for revealing the faults, demonstrated by Valid(C).

Therefore, we can define the ideal test by reliability and validity criteria.

Definition 5: A test (t) is called an ideal test if and only if for all ts belongs to T satisfying criterion ς that is both reliable and Valid.

E. PQ-ANALYSIS

PQ-Analysis proposed by Eggers and Belli [52], [53] indexes the provided RE to obtain missing state information during conversion from the FSM. Moreover, the context of the RE elements can cause ambiguity due to the same symbol appearing in different positions, which is copied after PQ-Analysis using indexing of the symbols. The primary purpose is to extract information regarding the analyzed system's fault tolerance capability using indexing and context tables (CTs). In the current work, it is utilized to increase the ability of test sequences acquired from CTs. The PQ-analysis contains seven steps, for which we provide the details, with an example, in the appendix in section VII. We adopt the PQ-Analysis as a base of our test generation approach using tables resulting from the PQ-Analysis. Test generation from these tables results in more efficient test suites than from the others based on different models, such as FSM, due to usage of redundancy provided by the PQ-Analysis.

III. PUTTING THE ELEMENTS OF THE APPROACH TO WORK

In the current section, we present the proposed approach, including necessary information. Test preparation and Testing steps are provided in Section III-B and III-C, being two main MBIT stages, offer necessary information supporting sub-steps. We use the FSM and the RE models as defined in [54] the current work.

In Fig. 1, the general flow of the current methodology is shown. The test preparation step follows the model and test generation sub-steps. The testing stage follows test execution and test selection sub-steps.

In Fig. 1, the straight lines perform the paths that are utilized by the current work. However, the dashed lines show other options that can be employed. For example, the FSM of the GUI program can be obtained from the specification by the designer and then given to the PQ-Analysis tool.

A. THE MBIT AND IT'S PROOF

The MBIT contains test preparation and test composition as being the main stages for which we present the following information.

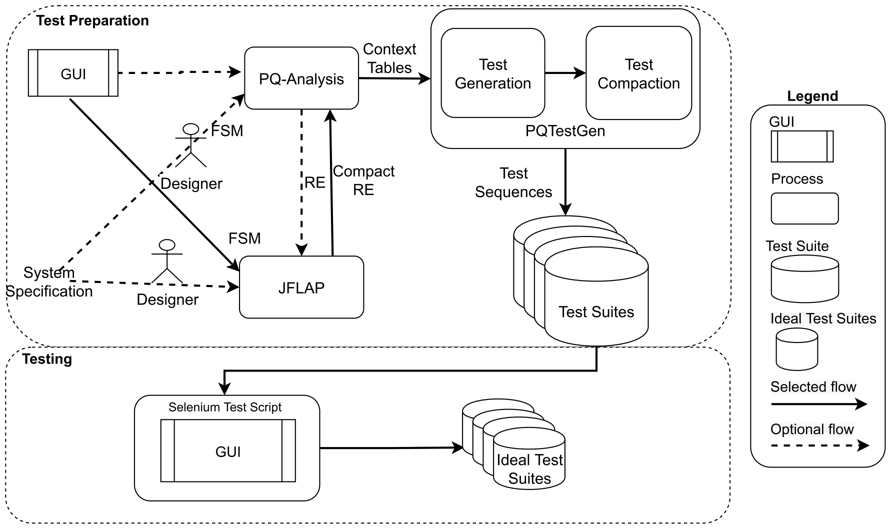


FIGURE 1. General flow of the current methodology.

1) TEST PREPARATION

We consider that an FSM as being a model provided or obtained from the GUI or its specification. The provided or obtained FSM model requires to be deterministic. The tester needs to be careful to avoid a non-deterministic model that causes invalid test sequences resulted from the test generation algorithm.

We utilize the MT to acquire mutant models for fault-specific models by mutation operators [18]. Then, we construct the CTs from transformed the RE. Then, we generate test suites based on these tables.

Finally, we obtain positive test suites (PTSs) and negative test suites (NTSs) using these suites.

2) TESTING

The PTSs are used to test the original (fault-free) GUI program on which these suites execute. For criterion 1, we select test sequences that are "passed". The set of these "passed" test sequences are called TS_1.

For criterion 2, the PTSs are used to test the faulty GUI program to acquire "failed" test sequences that are executed on this GUI program. The set of these "failed" test sequences are called TS_2.

In the NT, the NTSs are used to test the mutant (faulty) GUI program on which these suites execute. Then, we collect test sequences that are "failed" concerning criterion 3. The set of these "failed" test sequences are called TS_3.

For criterion 4, the NTSs are used to test the mutant (faulty) GUI program to acquire "passed" test sequences that are executed on this GUI program. The set of these "passed" test sequences are called TS_4.

Definition 6: Model-based Ideal Test (MBIT) suite: The resulting test suite T becomes MBIT suite if and only if any t(M) belonging to T satisfying a certain Criterion C_1 *or* C_2 *or* $C_3 or C_4$ that are reliable and valid. (See definitions 3 and 4)

TS_1, TS_2, TS_3, and TS_4 (Test Suites) are acquired with respect to C_1, C_2, C_3, *and* C_4, respectively.

The reliability and validity requirements will be examined to show that provided test suites are an ideal test by means of the following 2 lemmas including their proofs;

- *Lemma 1 (Reliability): TS_1, TS_2, TS_3, and TS_4* (Test Suites) for the PT and NT are either "pass" or "fail" for all test cases contained in the corresponding test suites.

  Proof:

  PT:

  Any t_i belonging to T acquired from criterion C_1 is Successful(t_i). Thus, C_1 is reliable from definition 3. Any t_j belonging to T acquired from criterion C_2 is Fail(t_j). Thus, C_2 is reliable from definition 3.

  NT:

  Any t_k belonging to T acquired from criterion C_3 is Fail(t_k). Thus, C_3 is reliable from definition 3. Any t_l belonging to T acquired from criterion C_4 is Successful(t_l). Thus, C_4 is reliable from definition 3.
- *Lemma 2 (Validity): TS_1, TS_2, TS_3, and TS_4* (Test Suites) are able to detect the faults or testify their absence.

  Proof:

  Any t_i belonging to T acquired from criterion C_1 is OK (t_i). Thus, C_1 is not valid from definition 4. Any t_j belonging to T acquired from criterion C_2 is not OK (t_j). Thus, C_2 is valid from definition 4. Any t_k belonging to T acquired from criterion C_3 is not OK (t_k). Thus, C_3 is valid from definition 4. Any t_l belonging to T acquired from criterion C_4 is OK (t_l). Thus, C_4 is not valid from definition 4.
- *Theorem: TS_2* and *TS_4* (Test Suites) selected using criteria $C_2 or C_3$ constitutes an MBIT suite based on the definition 5 given in Section 2.1.

  Proof: Lemma 1 and Lemma 2 show that TS_2 *and* TS_4 are ideal suites and constitute MBIT suites.

```
Open the SUT (GUI of a web page);
while (check all events of the SUT for all pages) {
    for (each elements (events) at the currentPage){
        if (event==radio button){
            click=radio button;
            Model=AddEvent(click);
        }else if (event== edit field ){
            click=edit field ;
            Model=AddEvent(click);
        }else if (event== text box){
            click=text box (addRandomText);
            Model=AddEvent(click);
        }else if (event== combo box){
            click=text box;
            Model=AddEvent(click);
        }else if (event== button){
            click=text box;
            Model=AddEvent(click);
        }
    }
    currentPage = nextPage;
}
```

Listing 1. Pseudocode of the model generation algorithm for a web page.

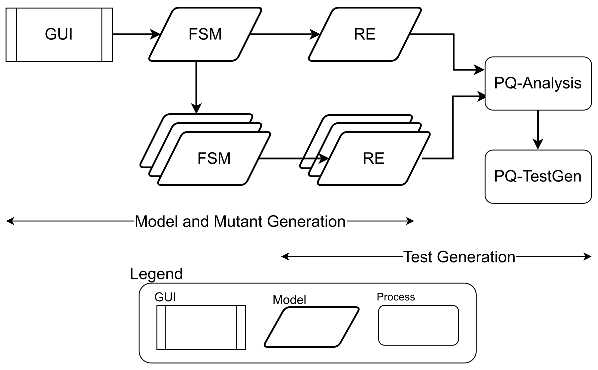


FIGURE 2. Test preparation step.

B. TEST PREPARATION

The GUI under test is represented by an FSM and then converted to a corresponding RE by the JFLAP tool shown in Fig. 2. To acquire mutants, artificial faults are seeded into the FSM. Each mutant can contain one or more faults.

An FSM model can be automatically generated from the specification of the GUT, or one of the GUI ripping methods [55], [56] automatically generates the proper model by using reverse engineering techniques. It is assumed that the specification is missing, and we generate the model of the GUI under test manually using the JFLAP.[2] We provide pseudo-code for generating a model from a GUI of a web page in listing 1. The algorithm given in listing 1 starts opening the system under test (SUT). Then, it proceeds by checking all elements of the current page of the SUT. These elements can be "radio button", "edit field", "text box", "combo box", or "button". Once selecting the current element (event), the corresponding entry is added to the model with its input and output response.

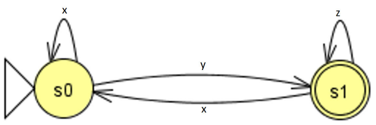


FIGURE 3. An FSM example.

After finishing all elements on the current page, the algorithm proceeds to the next page. This procedure continues until exploring all the elements for all pages of the SUT. While exploration carries on, the corresponding responses are added to the model. Once the exploration is finished, the model generation is also finished.

For example, let us suppose that SUT is Gmail login page.[3] Once the user clicks this page, he/she has got several options, such as a textbox for user email or telephone number. The user requires to enter his/her email account into this text box to proceed next step, or he/she can create a new account by clicking the "create account" button. Let us again suppose that the tester (the person responsible for the model generation) enters the correct email address to the textbox, then he/she needs to add this event to the model with Email entry and its corresponding response of the SUT to this action. This procedure must be applied for all elements of the login page by the tester and added to the model with corresponding responses. Finally, the tester constructs the Gmail login page model once he/she finishes all elements of this web page. The manual construction is straightforward and easy for this kind of login page that is the main bottleneck of automatic model generation approaches due to missing correct information of user accounts. Therefore, the automatic model generation approach requires user intervention to cope with this kind of problem. Another advantage of manual construction is to decide the capacity of the model by neglecting unnecessary features.

We provide a formal definition of a Finite State Machine (FSM) below to clarify further steps.

Definition 7: Finite State Machine (FSM) is defined by 5-tuples $\langle S, \sum, \delta, q_0, F\rangle$ in which; S is a finite set of states, $\sum$ is a finite set of symbols, δ is a state transition function q_0 is the initial state is an element of Q, and F is a finite set of final states is a subset of Q.

Example 1: An FSM example is defined by $(s0, s1, x, y, z, \delta, s0, s1)$, where $\delta = \{\delta(s0, x) = s0, \delta(s0, y) = s1, \delta(s1, z) = s1, \delta(s1, x) = s0\}$ is a transition function. Fig. 3 represents the FSM graphically.

We utilize insertion, omission, or replacement of the state(s) or transition(s) of the FSM to acquire mutants. The following definitions, including examples, are presented to elaborate on how to acquire mutant FSM models.

Definition 8: Insertion operator (IO) adds an extra transition(s) or state(s) into the FSM.

Example 2: IO(s0, z, s1) refers to adding an extra "z" transition from s0 to s1, or IO(s0, z, s2, x, s1) refers to

[2]JFLAP, Available online at http://www.jflap.org/

[3]Gmail Login Page, Available online at https://gmail.com/

adding an extra state s2 between s0 and s1 with "z" and "x" transitions.

Definition 9: Omission operator (OP) deletes a transition(s) or state(s) from the FSM.

Example 3: OP(s0, x, s0) refers to deleting the transition "x", or OP(s1) refers to deleting the state s1 with corresponding transitions.

Definition 10: Replace operator (RO) substitutes a transition(s) or state(s) from the FSM.

Example 4: RO(s0, x, s0, z) refers to replacing the transition "x" with "z", or RO(s1, s2) refers to replacing the state s1 with s2.

To model semantic faults, the mutants require higher-order mutation, in contrast to inserting a single fault in the model or code to create first-order mutants. In this higher-order, mutation applies the mutation operator more than once [57].

Once we acquire mutants, we transform the resulting FSM model into the RE models using the JFLAP tool that provides a more compact RE than the PQ-Analysis. The procedure of the FSM to RE conversion for the PQ-Analysis tool is presented in the Appendix section, including the pseudo-code for the conversion. The PQ-Analysis tool generates CTs that accommodate forward and backward information. This information is useful for generating more efficient test suites to increase the possibility of covering the faults because covering the only symbol without its right and left context does not guarantee the coverage of the modeled fault(s).

We use the PQ-TestGen [13] tool for test generation. The CT contains two different and independent tables, namely forward right and left CT. Therefore, we have two sets of test sequences from the forward right and left tables. We select the forward right table considering any overlapping between the two tables. PQ-TestGen [13] parses the table and then traverses, starting from the initial symbol in a depth-first search manner in the first phase. The traversing finishes once the final symbol is reached to construct complete test sequences. However, some sequences can be incomplete because of a different symbol in the final test suite by reaching coverage criteria to assess adequacy. For those partial sequences, the algorithm uses already complete sequences to complete them in the second phase. The algorithm utilizes a compaction procedure to eliminate redundant sequences in the final phase while keeping the coverage criteria in a predefined ratio.

PQ-TestGen [13] initiates from the opening symbol "[" and selects the next symbol from its forward right context. Test generation continues until assessing coverage criterion satisfied when all different symbols are in the resulting test suite. Kilincceker and Belli in [58] define the coverage criteria depending on the CT and extensively analyze their effectiveness for GUI testing.

C. TESTING

The test execution and then test composition are the sub-steps of the testing stage. In the test execution, the test suites are run automatically on the corresponding GUI programs, and then these sequences are collected into MBIT test suites in the test composition step.

We run the test suites on the fault-free (original) and faulty (mutant) GUI programs in this step (see Fig. 4). Hence, two different testing scenarios happen as follows;

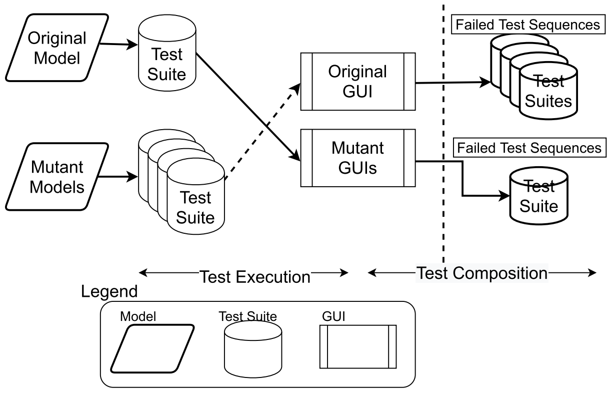


FIGURE 4. **Testing step.**

1. (Positive Testing (PT)) Executing test sequences obtained from original models on corresponding mutant GUIs under test. These test sequences contain legal inputs.

2. (Negative Testing (NT)) Executing test sequences obtained from mutant models on original GUI under test. These test sequences contain illegal inputs.

Test selection criteria are a filtering mechanism to select satisfying test cases. It is important to note that the coverage criteria and selection criteria are not to be mixed. Coverage criteria are termination criteria utilized for the test generation procedure. However, test selection criteria (also called test criteria) are a filtering mechanism to accomplish the ideal test conditions.

The current methodology does not intend to utilize a code level or function level coverage at the program level. The main intention of the used coverage criteria in the current work is to assess adequacy of GUI testing at the functional level with respect to events captured by test sequences. Therefore, we only use coverage of events based on black box testing from user perspective. On the other hand, the tester focuses on code coverage in white box testing from developers perspective to assess adequacy.

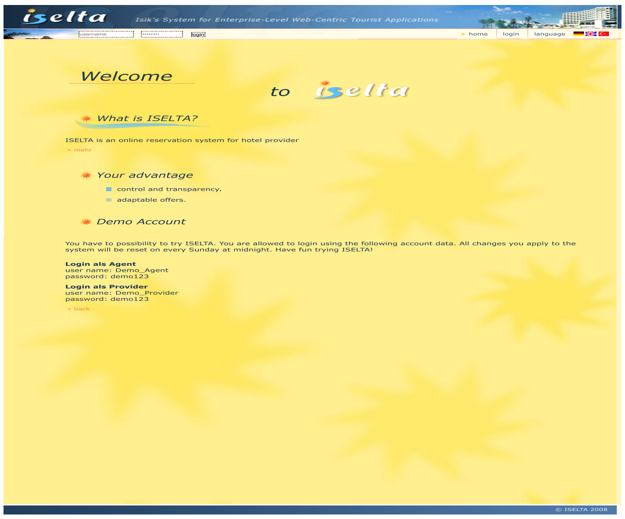
Based on the test criteria, "failed" test sequences are collected into ideal test suites concerning the PT and NT. To test the presence and absence of predefined faults, we utilize these test suites using the PT and NT, respectively.

Model Correctness: Test generation algorithm from the model under analysis is used to check model correctness. Therefore, the suite generated from the model is executed on the system modeled. The entire test suite requires to be "passed" on the system modeled to satisfy algorithmic correctness. The test generation algorithm must be deterministic in order to avoid different results in each generation.

We utilize criteria 1 and 4 to satisfy model correctness. Hence, the original model is checked by executing the test suite generated from the original model on the original system under test concerning criterion 1. Any mutant model is checked by executing the test suite generated from the corresponding mutant model on the mutant system under test. To this end, we can satisfy that all the models hold correctness.

IV. CASE STUDIES

This section provides the case studies, the GUI of ISELTA[4] website's "Special" and "Additional" modules (see Fig. 5). ISELTA is a commercial web portal for marketing tourist services and an online reservation system for hotel providers. It is a cooperative work between ISIK Touristic company and the University of Paderborn. The "Special" module provides agents' ability to promote special advertisements, such as the New Year event. The "Additional" module offers other advertisements rather than regular events. For each module, the GUI of ISELTA enables agents and providers to use different attributes for specific events to catch customers' interest. It is written in PHP programming language and contains 69323 lines of code inside five different modules for each provider under the "Hotels" section. Readers can log in to ISELTA using demo information given on the website as being a provider role (see Fig. 5 for details).


iselta
Isik's System for Enterprise-Level Web-Centric Tourist Applications
username
login
home
login
language
© ISELTA 2008

Welcome to iselta

What is ISELTA?

ISELTA is an online reservation system for hotel provider
» mehr

Your advantage

- control and transparency.
- adaptable offers.

Demo Account

You have to possibility to try ISELTA. You are allowed to login using the following account data. All changes you apply to the system will be reset on every Sunday at midnight. Have fun trying ISELTA!

Login als Agent
user name: Demo_Agent
password: demo123

Login als Provider
user name: Demo_Provider
password: demo123

« back

FIGURE 5. Iselta webpage.

A. TEST PREPARATION

This section presents the test preparation step for only the "Special" module case study. However, we provide a supplementary website for the "Additional" module,[5] which also introduces required information to reproduce the current work.

Firstly, an FSM is manually constructed from the GUI of the ISELTA's "Special" module. This module is called GUI Under Test. An omission, insertion, and replace mutation operators [18] are performed on these FSM(s) for a mutant generation. Then, further steps are carried on as provided in Section III-B and III-C.

There are many input areas and buttons in the main GUI of the "Special" module, and it is redundant and tedious to test each case of the module. Therefore, the current work restricts the scope to a relatively small module in the application for evaluation. In this step, the FSM model is acquired by the GUI under Test (GUT). To do this, the tester enters the ISELTA web page and proceeds to the "Special" module. He/She has listed all elements (events) of this module. These elements are given in Table 3. Then he/she applies the algorithm given in section III in the listing 1. First of all, he/she tries to set required input boxes such as "price", "title", "number" elements. While the tester provides this information to the SUT, he/she is also added this information, including SUT's responses to the FSM model. When the tester satisfies covering all elements and their combinations in the FSM model, he/she finishes the model construction procedure.

In the fault-free FSM of the "Special" module, a symbol is assigned for each event that becomes a transition label in the FSM. All symbols represent filling an input, clicking a button, or removing a text from an input. These action symbols enable the implementation of the Selenium test script. All event symbols are listed in Table 3.

TABLE 3. Event symbol list.

Symbol	Event	Symbol	Event
e	Back	k	Click Edit
l	Click Save	v	Click Add
u	Set Title	x	Set Number
y	Set Price	z	Set Description
r	Remove All	t	Remove Title
p	Remove Price	n	Remove Number

In the case study, we intend to catch functional faults that directly affect the system's desired operations based on user interaction with the GUI.

Definition 7: Functional fault is a higher level and event-based fault in which the system achieves the final event without providing expected output.

Example 7: Mutant 1 given in Table 4 is a functional fault in which the system ends up not adding a new offer due to required empty input boxes. However, it reaches the final event called the "add" event. Once the user clicks the "add" button, he/she receives a message. This type of fault is called functional fault.

GUI-related user interaction faults are high levels and different than low-level faults such as low-level security faults and performance faults. The reader may refer [4] for detailed classifications and qualifications of high-level user interaction faults.

In total, 12 different types of mutants are acquired at code and model levels. Table 4 gives the semantics of these mutants for the "Special" module, including mutation operators utilized to generate these mutants. GUIs under test enable only

[4]ISELTA, http://iselta.ivknet.de/
[5]MBIT4SW, https://kilincceker.github.io/MBIT4SW/

TABLE 4. Mutant semantics.

Mutant	Semantics	Mutation Operator(s)
1	Add with empty input boxes	OP,IO
2	Update with empty input boxes	OP,IO
3	Add with empty Number of Packages	OP,RO
4	Add with empty price	OP,RO
5	Add with empty title	OP,RO
6	Update with empty title	RO, OP
7	Update with empty price	RO, OP
8	Update with empty number	RO, OP
9	Add click does not respond	OP, RO
10	Edit click does not respond	OP, IP
11	Add and Edit click does not respond	OP, IP
12	Save button move to initial state	OP, IO, RO

these number of semantic faults at a model level, based on experimentation for a mutant generation. Note that the number of possible mutants at the code level can be more than the number of modeled mutants related to the utilized mutation operators. In our case study, we only focus on semantic mutants that can be acquired by the FSM. Hence, the mutant and test generation carried out on the FSM model, but the test execution and selection steps were carried out on the code level in this study.

In this step, we use the JFLAP tool to transform original and faulty FSMs into REs. Then, we apply the PQ-Analysis [53], [59] tool to these RE models to obtain CTs.

The original (fault-free) RE is provided below. The RE in (1) is converted from the FSM model and contained symbols provided in Table 3. The RE in (1) is shortened to fit the page.

$$R = [(ve) * (knl(el) * exl + \ldots + ky(xl + l) + kxl + kl)*] \tag{1}$$

The PQ-TestGen [13] tool acquires test suites using each CTs of both original (fault-free) and mutant (faulty) RE models. Finally, the PQ-TestGen tool utilizes test compaction to obtain the test suites' final form by removing redundant sequences. Those sequences are the ones for the test execution part of the study, which operates on Selenium[6] test automation.

$$t1 = \text{``}klktleulkl\text{''}, \quad t2 = \text{``}klknlexlkl\text{''} \tag{2}$$

$$t3 = \text{``}yxzveuvkl\text{''}, \quad t4 = \text{``}yuzvexvkl\text{''} \tag{3}$$

For instance, test sequences are given above in (2) and (3) are acquired from RE given in (1) utilizing the PQ-TestGen tool.

B. TESTING

We utilize a Selenium test script for test execution. The Selenium script runs all test sequences on each original and faulty GUTs that are "Special" and "Additional" modules of the ISELTA. Then test scripts result in either "Pass" or "Fail" for each test sequence. We will then group those to analyze them and decide the required set of test cases for constructing the MBIT suites. These suites agree with the conditions of the ideal testing.

[6]Selenium Test Automation, https://www.selenium.dev/

In the current work, 12 different mutants for the ISELTA website's "Special" and "Additional" modules are obtained, and 12 different test suites from these mutants are acquired for the NT.

We utilize Selenium for test automation, which enables us to automatically execute test sequences on the web-based software and collect the test execution results. It contains two parts, which are the Web-Driver and the IDE. The Web-Driver provides functionality with Java and Python programming languages for automation of the test execution. The IDE contains an easy-to-use interface, including plugins for specific internet browsers and simple record-and-playback interactions with the browser. There are too many commercial or non-commercial web or mobile test automation tools, especially for test execution. We select Selenium due to its robustness, simplicity, and popularity among the scientific community. Besides Selenium, Sikuli[7] is also an open-source and robust solution. Sikuli uses image recognition powered by OpenCV to capture GUI events. Both Selenium and Sikuli can run the various internet browsers from different vendors. Selenium is more community-driven and supported among other testers. We could easily adapt Selenium to our methodology for test execution.

V. EVALUATION

The proposed approach was evaluated through the ISELTA "Special" and "Additional" modules with respect to experimental studies. For this evaluation, 24 mutants of the ISELTA "Special" and "Additional" modules (see Table 5) at code and model levels were obtained. The list of these mutants is provided in section IV with details. To test the presence and absence of the faults, the evaluation was carried out. Considering the presented general methodology, the test generation is only one of the stages. However, in the evaluation phase, the test generation has become a priority.

TABLE 5. Mutant profiles.

Fault Type	Quantities		Total
	Special Module	Additional Module	
Functional Fault	12	12	24

In order for the FSM models used within the scope of experimental studies to comply with the definition of algorithmic correctness proposed in Section III, the test suites obtained from the original model within the scope of criterion 1 were also run on the original GUI system. It was observed that all test sequences passed the test successfully. Similarly, each test suite generated from mutant FSM models is executed on the corresponding mutant GUI system based on criterion 4 to avoid wrong model utilization.

Test generation is optional for the current methodology. Thus, the general methodology can be realized by changing the test generation stage. However, an approach that provides coverage of all modeled faults has been proposed

[7]Sikuli Test Automation, http://sikulix.com/

TABLE 6. Results of the PT and NT for Special module.

	PQTestGen		PQRTestGen100		PQRTestGen60		Grapwalker	
Symbol Coverage	100		100		60		90	
Mutation Score	1*	0,92	0,75	0,75	0,58	0,75	0,92	0,92
Fault Coverage (%)	100*	92	75	75	58	75	92	92
Test Suite Size (Symbols)	412	486	228	209*	193	154	767*	594*
Test Generation Time (ms)	249	178	263	203	216	157	3870*	3897*
Test Execution Time (s)	207	180	84	72	120	71	283*	262*

PS: Positive Testing (left), NT: Negative Testing (right), ms: milliseconds, s: seconds

TABLE 7. Results of the PT and NT for Additional module.

	PQTestGen		PQRTestGen100		PQRTestGen60		Graphwalker	
Symbol Coverage	100		100		60		90	
Mutation Score	1*	1*	0,58	0,75	0,5	0,58	0,92	0,83
Fault Coverage (%)	100*	100*	58	75	50	58	92	83
Test Suite Size (Symbols)	486	412	215	215	180	160	554*	508*
Test Generation Time (ms)	254	181	275	218	217	158	3856*	3676*
Test Execution Time (s)	104	88	47	47	40	40	133*	132*

PS: Positive Testing (left), NT: Negative Testing (right), ms: milliseconds, s: seconds

in this study. An approach that produces random test generation has been developed and used for this evaluation. Also, an industrial-level model-based testing tool called Graphwalker is adapted to the methodology and utilized for evaluation. Thus, the extent to which the overall approach is effective and appropriate to test for the presence and absence of faults has also been evaluated.

In the literature, various metrics are utilized to evaluate the approaches proposed for software testing. The most important and preferred of these is fault coverage. Other important metrics are the time for test generation and test execution and test suites size. All of these metrics were considered to evaluate the current methodology.

Using Selenium, an open-source test automation tool, we automate the test execution process.

For example, in mutant number 3 (Add with empty Number of Packages input box) for Special module, in the original system, the user cannot add a new form to the system if he/she does not fill the "Number of Packages" input box. However, to create a mutant and a test sequence for this mutant, the adding "Number of Packages" input state from the FSM is removed in mutant three. Because the "Number of Resource" input state is removed from the FSM, the mutant test file does not contain the test sequence for adding the "Number of Packages" action. When the test suite is executed on the original GUI under test, the system fails in some test sequences for adding the new special form action. This is because the original system expects the "Number of Packages" input box to be filled to save the form to the system database. These validation points in the application lead to several failing test sequences in each mutant because those parts from the mutant are deliberately removed. Later, these failing sequences are used to assert the required parts of the GUI under test.

Together with the mentioned metrics for evaluation, we provide Table 6 and Table 7 with a comparison of the other three techniques. The symbol coverage criterion is set to 100 and 60 for the random test generation tool, PQRTestGen100 [60], and PQRTestGen60 [60]. We created a new FSM model to adapt the Graphwalker [38] using its visual editor. Then, we run Graphwalker to generate test sequences by setting symbol coverage to 100. After running Graphwalker for about one hour, we stopped the process due to excessive memory usage and resulted in an enormous output file. Experimentally, we decreased the coverage value and decided that 90 coverage was the optimum value. The proposed approach is called PQTestGen in the corresponding tables. We also utilized the RETestGen [61] tool for test generation. However, RETestGen resulted in excessive size test suites, which were unable to execute on the GUI under test at the acceptable times. Therefore, we neglect RETestGen from the experimentation. To eliminate randomness on evaluated metrics, we run each tool using a random test generation algorithm ten times and selected average test suite size among others with their corresponding metrics.

As provided in Table 6 and Table 7, PQTestGen attained the highest coverage of faults. Moreover, for the PT, PQTestGen has the highest mutation score that is 1. Therefore, for the entire set of mutants, we acquired MBIT test suites for Special and Additional GUI under test using PQTestGen. For the PQRTestGen method, while the fault coverage and mutation score values decrease, the symbol coverage value also decreases. However, the PQRTestGen method resulted in the same coverage for the "Special" module in the NT.

Graphwalker resulted in a larger test suite than other techniques, and its test generation time is about 15 times higher than others, in absolute figures, that is about 4 CPU seconds for the case studies. On the other hand, its test execution time is about 300 seconds for Special and 130 seconds for Additional modules provided in Table 6 and 7. The reason for the larger test suite is the random test generation

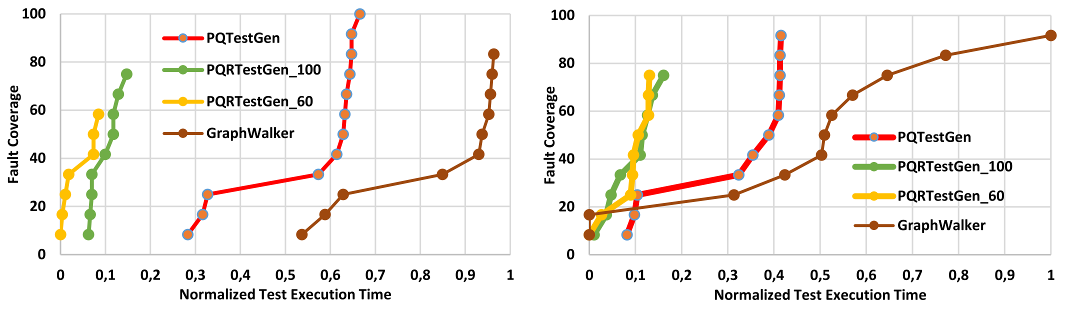


FIGURE 6. Fault coverage and normalized test execution time curve for Special module (left diagram is for the PT, and right diagram is for the NT).

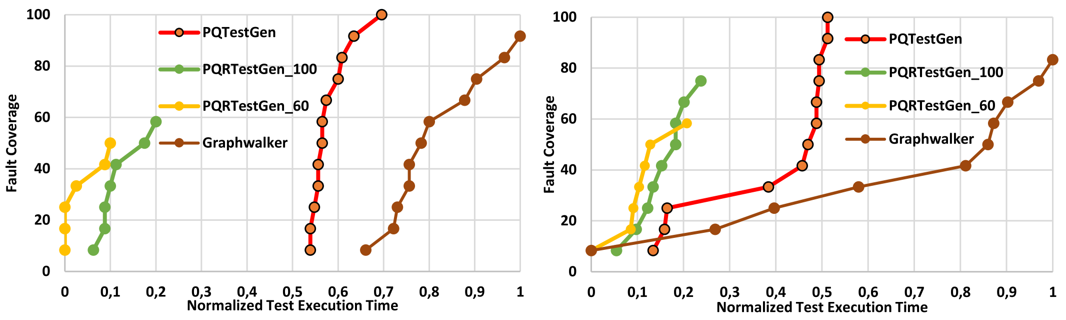


FIGURE 7. Fault coverage and normalized test execution time curve for Additional module (left diagram is for positive and right diagram is for negative testing).

algorithm utilized by Graphwalker. We run Graphwalker on the command-line interface (CLI) and calculate test generation time using its jar file. Execution of jar files may explain excessive test generation time of Graphwalker, among others. Except for Graphwalker, PQTestGen yielded the largest test suite results among others. This is because the test generation algorithm used by PQTestGen includes even each repeating symbol in the coverage value as if it were a different symbol. Thus, while some redundant symbols are included in the test suite, this situation directly affects the fault detection capability.

The cumulative distribution curves for fault coverage and normalized test execution time are given in Fig. 6 and Fig. 7 for the "Special" and the "Additional" modules for each tool. All the techniques have at least 58 fault coverage in around 225 seconds for both GUIs under tests. In about 207 seconds (PT) and 180 seconds (NT) for the "Special" module, PQTestGen achieves the maximum fault coverage. It is around 104 seconds and 88 seconds in the PT and NT, respectively, for the "Additional" module. To calculate normalized test execution times, the minimum (min) and maximum (max) values of all test execution times for the respective GUT were found. Then, the normalized form of the x value as (x-min) / (max-min) was calculated. The normalization process resulting from the proportional difference between PQTestGen and PQRTestGen test execution times is needed. Thus, Fig. 6 and Fig. 7 were obtained.

Using automatized processes, we collected these results. We neglected the manual effort. Usually, the manual effort requires more time than the automatized process. The highest manual effort is the mutant generation. To this end, we plan to automatize these manual efforts.

The proposed approach is efficient to test the presence and absence of the faults in the model's scope based on the collected results. Finally, using a Selenium test script, we automatically collected MBIT test suites for the "Special" and "Additional" modules.

The Web-Driver part is integrated into the current framework. We use Selenium for the test execution and test selection steps that are performed using the Web-Driver containing generic Java test scripts to automatically execute test sequences from PQ-TestGen and PQ-RanTest. Then, it collects the results that are "pass" or "fail".

VI. DISCUSSION

This section discusses the current work concerning testing techniques and selection of mutant generation based on our experimental and theoretical works. We provide answers to research questions, including the internal and external validity of the current work.

A. TEST AND MUTANT GENERATION TECHNIQUES

To acquire mutants and to obtain test suites, we utilize the FSM and RE, respectively. However, alternatively, it is possible to use others, such as the ESG or EFG, for these processes. We adapted the FSM and RE to the current work.

We use the PQTestGen tool in this study to obtain test suites due to its effectiveness in detecting faults. The PQTestGen is an efficient tool that uses the tables that eliminate the uncertainty using the right and left context of each symbol in the RE. This enhances the capability of fault detection.

B. CHECKING THE RESEARCH QUESTIONS

The research questions provided in Section I are answered concerning theoretical and experimental evaluation within the current work scope.

1) Is it practically and theoretically possible to offer an ideal testing [2] approach for GUI testing? The experiments' result is that the conditions of being an ideal test [2] are satisfied with the GUI programs based on the "Special" and "Additional" modules of the ISELTA. Also, it is applicable based on theoretical evidence. Therefore, the answer to the RQ1 is yes.
   - What types of systems can be tested in this way? In the current work, we address the GUI system of a computer application, representing an application by a combination of icons, menus, buttons, bars, boxes, and windows. However, the current work applies to any mobile GUI program. Other types of computer programs such as the GUI of game programs are neglected, which requires the utilization of different abilities and methods.
   - What types of faults can be targeted with the proposed approach? The fault type addressed in the current work is a functional fault [62] in the GUI under test that is unable to deliver desired and expected behavior/function in case this fault occurs.
2) What is the cost of applying this approach to GUI testing? The proposed approach's computational complexity is O(M*N3 + M*K*N) for N states, M mutants, and K is the total number of test sequences from each mutant. We provided Table 6 and Table 7 for the cost of the "Special" and "Additional" modules of the ISELTA case studies, including the test suite's size.
   - How is scalability affected? The CT-based test generation is the main bottleneck of the current work due to the transformation of the models for analysis. Nonetheless, we automate the main steps by using the tool-chain.

C. THREATS TO THE VALIDITY

We provide potential internal and external threats of the current work in this section.

1) INTERNAL VALIDITY

The current work presents the MBIT methodology for testing the GUI program using the HT and the MT. Theoretical evidence to confirm this assertion is also presented in the same section to show the requirements to be accepted as the ideal test. Additionally, the experiments are provided with two case studies in section IV to evaluate theoretical evidence. In the experimentation, the selected faults are seeded into both code and model to address functional faults in the GUI under test. The experimental and theoretical evidence verify that the claim is correct, indicating no threat to internal validity. We show the MBIT effectiveness results in section V with tables and graphs, which let us see the improvements and benefits of this approach.

However, there is an essential point that we need to address here and which will lead us to ask this question about the model utilized in the proposal: How to make sure that the model obtained from the GUI under test is correct? For more than three decades, this question has been asked by people studying in the model-based testing domain because the model's correctness has vital importance in the study and results. A wrong model will certainly lead us to wrong results no matter which techniques are used in testing. This critical question has satisfactorily been answered: use every possibility, every method to validate the model. For instance, use model checkers [63], [64] or, more importantly, get feedback from end-users as early as possible to assess the model long before starting with test generation and testing itself. However, we use test generation algorithm to ensure the model complies with the system modeled. We utilize test selection criteria 1 and 4 to get rid of the model correctness problem provided in Section III. Belli and Gueldali [65] also proposed a test generation approach based on model checking that can be also a solution to model correctness.

2) EXTERNAL VALIDITY

In the test composition part of the proposed approach, there is the step of running the test suites. Although these suites run automatically thanks to the Selenium tool, each test run can take a few minutes, as shown in the tables of section IV. This time varies in proportion to the number of states in the model. Moreover, considering that the PT and NT are applied for each mutant, the total number of Selenium test operations will be the mutant number times 2. This can be a few hours, even in the small GUI form used in our case study. This approach will take much longer when applied in larger models with more mutants. For this reason, the most important external validity can be considered as this complexity problem.

To cope with the complexity issue that may lead to a state explosion problem, we may utilize one of the GUI ripping methods [55], [56] to obtain the model automatically. On the other hand, there is another solution [66], [67] to cope with huge models utilizing layered modeling methods that are well-suited for the hierarchical structure of GUI systems. These layers can be manually created by the tester or automatically extracted from a non-layered model by utilizing a community detection algorithm introduced in [46] to mitigate the complexity thread.

The current work addresses the detection of functional faults rather than other types of faults related to visual attributes as mostly utilized in the GUI of games. This may lead to external validity. Models functionally represent systems under test in the current work. Testing visual attributes on the screen is not suitable for the current work due to the constructed model's use. Testing such visual attributes requires the utilization of white box code-based testing methods.

3) CONSTRUCT VALIDITY

The current methodology requires manual efforts for the model and mutant generation. Test expert carries out the model generation using the JFLAP tool. However, the test expert may construct the wrong model as an original model, as it is also addressed in the internal validity. In this way, the test expert manually constructs the wrong mutant models from the wrong original model. This is also considered a potential threat to construct validity. To overcome this threat, we execute generated test suites from an original model on the original SUT and from mutant models on the corresponding mutant SUTs. Furthermore, we check that all test cases pass on these executions to ensure that the model and the SUT are equivalent concerning the generated test suites. We also plan to automatize these manual efforts, which may lead us to the wrong models.

VII. CONCLUSION AND FUTURE WORK

In the current work, the MBIT approach is introduced for the GUI programs to test the presence and absence of functional faults. The proposed methodology is composed of two main steps, which are test preparation and testing. In the test preparation step, we use a GUI program model described by an FSM that is converted to a RE. The same procedure is also applied to the mutant models acquired by seeding faults into the original GUI model. We generate test suites from the fault-free and the mutant models concerning the positive testing (PT) and negative testing (NT). We utilize code-based mutation operators to obtain mutant GUI programs and model-based mutation operators for mutant models. The mutated models are used for test generation, and the mutated GUI programs are used for test execution. In the testing step, we run generated test suites on the fault-free and mutant GUI programs. We collect the ideal test suites from these executed test suites considering selection criteria. These test suites enable us to test both presence and absence of functional faults within the scope of the GUI models.

We evaluate the proposed methodology using two case studies, namely "Special" and "Additional" modules of the ISELTA webpage. The results show that the proposed methodology achieves ideal testing utilizing the PT and NT and provides ideal test suites to test the presence and absence of functional faults. We compare the proposed test generation method, called PQTestGen, with PQRTestGen100 and PQRTestGen60 methods. PQTestGen achieves a higher fault coverage than the other methods. Besides, PQTestGen has a higher mutation score than the other PT methods, meaning that it can kill all mutants for "Special" and "Additional" modules of the case study. However, PQTestGen results in a more extensive test suite than the other methods due to the Breadth-First Search (BFS) algorithm. This large test suite size in PQTestgen also increases test execution time. However, test sets' size is within acceptable limits considering time for test generation and execution steps that finish in milliseconds.

We automate all procedures for the proposed methodology except for the model and mutant generation steps that the test expert carries out. These steps require knowledge about the system under test. However, we plan to automate the model generation step using one of the proper GUI ripping methods to extract the model from the GUI program automatically. We also plan to automate the mutant generation step using omission, insertion, and replace mutation operators on the FSM models. Finally, we plan to obtain an end-to-end solution for test automation of GUI programs for the MBIT approach.

APPENDIX
PQ-ANALYSIS

A. THE MAIN IDEA

Eggers and Belli [52], [53] introduced a method based on regular expression (RE) for detection, localization, and finally, correction of faults. They used the method in compiler construction. This method is also used in the general theory of fault tolerance and its applications to sequential systems, such as system-user interactions [5] and hardware [59] by Belli. The ability of self-detection and self-correction is analyzed for a system under consideration (SUC). In case SUC does not have these abilities, the method offers functional redundancy [53] by expanding the SUC. The method is provided with minimum redundancy.

The method is applied to strictly sequential SUCs to utilize the RE model by the syntax-based technique. The RE is a composition of the symbols with concatenation, union, and star operators. The symbols can be defined differently, such as events being inputs and outputs. The method uses a set of hypotheses to handle context-based faults. The hypothesis is utilized by insertion (I), replacing (R), and deletion (D) of symbol(s). For example, "insertion of a symbol between other symbols". The method applies the pairwise and mutually exclusive hypothesis for an explicit self-correction of a fault. For instance, hypotheses P and Q out of I,R,D are applied to a fault. In the numerous master's and Ph.D. projects on this topic, the students coined the method as "PQ-Analysis".

The current paper uses the PQ-Analysis as a base of the test generation using the context tables obtained from the PQ-Analysis.

B. CONCEPT AND STEPS

We can generate sequences (strings) from a given regular expression T using a regular (type 3) language L(T) that accepts T. The main aim of PQ-Analysis is to obtain the context relations of the symbols from T called context and compatibility tables and then is to utilize these tables for checking whether a string corresponds to the language L(T) or not. Therefore, we avoid expensive backtracking. The steps of the method are provided in Fig. 8.

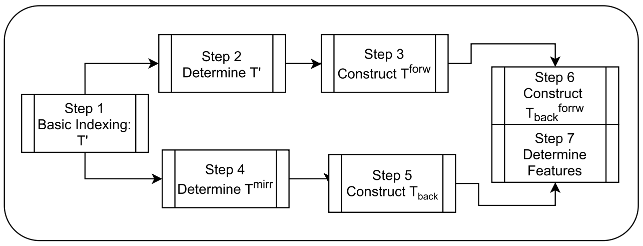
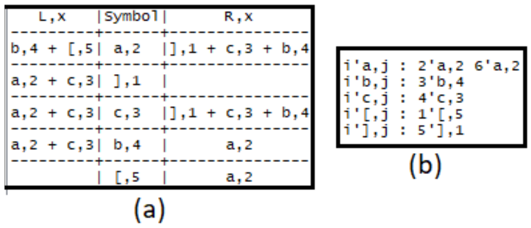


FIGURE 8. Steps of the PQ-Analysis.



```
    L,x    |Symbol|       R,x
----------+------+-----------------
b,4 + [,5| a,2  |],1 + c,3 + b,4
----------+------+-----------------
a,2 + c,3| ],1  |
----------+------+-----------------
a,2 + c,3| c,3  |],1 + c,3 + b,4
----------+------+-----------------
a,2 + c,3| b,4  |       a,2
----------+------+-----------------
         | [,5  |       a,2
```
(a)

```
i'a,j : 2'a,2 6'a,2
i'b,j : 3'b,4
i'c,j : 4'c,3
i'[,j : 1'[,5
i'],j : 5'],1
```
(b)

FIGURE 9. (a) Context table and (b) Compatibility table.

We present the method's steps, including an example for the corresponding steps to make the concept clear as follows.

Example: Given the following regular expression:

$$T = [(a(ba + c*)*)] \tag{4}$$

the related PQ-Analysis tables can be provided by following the Step 1-7.

Step 1: Indexing the given regular expression T that leads to the indexed T′.

$$T' = [^1(a^1(b^1a^2 + (c^1)*)*)]^1 \tag{5}$$

Step 2: Convert T′ to an equivalent FSM E^{forw} [68], [69].

We determine the states of the FSM as follows;

(a) i =: j (input a transfers the state i into state j) initial state = 0, ([)0 = ([^{1})0 =: 1 ≡ [^{1}, (a) 1 = $(a^1)1$ =: 2 ≡ aa^1, (])1 = (]^1)1 =: 3 ≡]^1, (b) 2 = $(b^2)1$ =: 4 ≡ b^1, (a) 2 = $(a^2)2$ =: 2 ≡ a^2, (c) 1 = $(c^1)1$ =: 5 ≡ b^3, (])1 = (]^1)2 =: 3 ≡]^1, Final State = 3.

Step 3: Scan T through E^{forw} to obtain T^{forw}.

$$T^{forw} = [^1(a^2(b^3a^6 + (c^4)*)*)]^5 \tag{6}$$

Step 4: Reverse (mirror) T' to have T^{mirr}.

$$T^{mirr} =]^1((c^1)* + (a^2b^1))*a^1[^1 \tag{7}$$

Step 5: Scan T^{mirr} through E^{back} to obtain $T^{mirr+forw}$.

$$T^{mirr+forw} =]^1((c^3)* + (a^2b^4))*a^2[^5 \tag{8}$$

Step 6: Reverse (mirror) T^{mirr_forw} to construct $T^{mirr+forw+mirr}$.

$$T^{mirr+forw+mirr} = [^5(a^2(b^4a^2 + (c^3)*)*)]^1 \tag{9}$$

Step 7: Determine the characteristic relations of T resulting from the context and compatibility tables given in Fig. 9.

We use the algorithm given listing 2 to convert the FSM model into the RE model. The procedure starts from reading

```
Input: State Table of the FSM; D[x,x]
Output: Regular expression; R

R = update (r);
function update (x){
    arden(x);
        for each i <= x-1
                d[x-1,i] = d[x-1,i] + d[x,i].d[x,x+1]
    if x >= 1
            update (x);
    else return(d[1,1);
}
arden (x){
    for each i <= x-1
            d[x,i] = d[]x,x]*.d[x,i]
}
```

Listing 2. Pseudocode of the model conversion from FSM to RE.

the state table representation of an FSM. Then, it uses an update function to iterate over the entries of the state table and uses the Arden function if needed.

REFERENCES

[1] W. E. Dijkstra, *Notes on Structured Programming*. New York, NY, USA: Academic, 1972, ch. 1, pp. 1–82.

[2] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," *IEEE Trans. Softw. Eng.*, vol. SE-1, no. 2, pp. 156–173, Jun. 1975.

[3] T. S. Chow, "Testing software design modeled by finite-state machines," *IEEE Trans. Softw. Eng.*, vol. SE-4, no. 3, pp. 178–187, May 1978.

[4] V. Lelli, A. Blouin, and B. Baudry, "Classifying and qualifying GUI defects," in *Proc. IEEE 8th Int. Conf. Softw. Test., Verification Validation (ICST)*, Apr. 2015, pp. 1–10.

[5] F. Belli, "Finite state testing and analysis of graphical user interfaces," in *Proc. 12th Int. Symp. Softw. Rel. Eng.*, Nov. 2001, pp. 34–43.

[6] F. Belli, C. J. Budnik, A. Hollmann, T. Tuglular, and W. E. Wong, "Model-based mutation testing—Approach and case studies," *Sci. Comput. Program.*, vol. 120, pp. 25–48, May 2016.

[7] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Trans. Softw. Eng.*, vol. SE-3, no. 4, pp. 279–290, Jul. 1977.

[8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, Apr. 1978.

[9] J. Offutt, "A mutation carol: Past, present and future," *Inf. Softw. Technol.*, vol. 53, no. 10, pp. 1098–1107, Oct. 2011.

[10] C. Robach and M. Scholive, "Simulation-based fault injection and testing unsing the mutation technique," in *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. Boston, MA, USA: Springer, 2003, pp. 195–215.

[11] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "MuJava: An automated class mutation system," *Softw. Test., Verification Rel.*, vol. 15, no. 2, pp. 97–133, 2005.

[12] B. K. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, R. Schlick, and S. Tiran, "Killing strategies for model-based mutation testing," *Softw. Test., Verification Rel.*, vol. 25, no. 8, pp. 716–748, Dec. 2015.

[13] O. Kilincceker, E. Turk, M. Challenger, and F. Belli, "Applying the ideal testing framework to HDL programs," in *Proc. ARCS Workshop 31th Int. Conf. Archit. Comput. Syst.*, 2018, pp. 1–6.

[14] F. Belli, M. Beyazıt, C. J. Budnik, and T. Tuglular, "Advances in model-based testing of graphical user interfaces," *Adv. Comput.*, vol. 107, pp. 219–280, 2017, doi: 10.1016/bs.adcom.2017.06.004.

[15] F. Belli, A. T. Endo, M. Linschulte, and A. Simao, "A holistic approach to model-based testing of Web service compositions," *Softw., Pract. Exper.*, vol. 44, no. 2, pp. 201–234, Feb. 2014.

[16] F. Belli and M. Linschulte, "On negative tests of Web applications," *Ann. Math., Comput. Teleinformatics*, vol. 1, no. 5, pp. 44–56, 2008.

[17] F. Belli, C. J. Budnik, and A. Hollmann, "Holistic testing of interactive systems using statecharts," in *Sicherheit 2006, Sicherheit-Schutz und Zuverlässigkeit*. Bonn, Germany: Gesellschaft für Informatik, 2006, pp. 345–356.

[18] O. Kilincceker and F. Belli, "Towards uniform modeling and holistic testing of hardware and software," in *Proc. 1st Int. Informat. Softw. Eng. Conf. (UBMYK)*, Nov. 2019, pp. 1–6.

[19] G. Mercan, E. Akgündüz, O. Kılınççeker, M. Challenger, and F. Belli, "Android uygulaması testi için ideal test ön çalışması," presented at the 12th Turkish Nat. Softw. Eng. Symp. (UYMS), A. Tarhan and E. Murat, Eds., Istanbul, Turkey, Sep. 2018, pp. 1–12.

[20] K. N. King and A. J. Offutt, "A fortran language system for mutation-based software testing," *Softw., Pract. Exper.*, vol. 21, no. 7, pp. 685–718, Jul. 1991.

[21] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "MuJava: A mutation system for Java," in *Proc. 28th Int. Conf. Softw. Eng.*, 2006, pp. 827–830.

[22] S. C. P. F. Fabbri, J. C. Maldonado, T. Sugeta, and P. C. Masiero, "Mutation testing applied to validate specifications based on statecharts," in *Proc. 10th Int. Symp. Softw. Rel. Eng.*, Nov. 1999, pp. 210–219.

[23] F. Belli and M. Beyazit, "Event-based mutation testing vs. state-based mutation testing—An experimental comparison," in *Proc. IEEE 35th Annu. Comput. Softw. Appl. Conf.*, Jul. 2011, pp. 650–655.

[24] T. Nguyen and C. Robach, "Mutation testing applied to hardware: The mutants generation," in *Proc. 11th IFIP Int. Conf. Very Large Scale Integr.*, 2001, pp. 118–123.

[25] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Mutation testing advances: An analysis and survey," *Adv. Comput.*, vol. 112, pp. 275–378, 2019, doi: 10.1016/bs.adcom.2018.03.015.

[26] W. E. Wong and A. P. Mathur, "Reducing the cost of mutation testing: An empirical study," *J. Syst. Softw.*, vol. 31, no. 3, pp. 185–196, Dec. 1995.

[27] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Sep. 2011.

[28] R. K. Shehady and D. P. Siewiorek, "A method to automate user interface testing using variable finite state machines," in *Proc. IEEE 27th Int. Symp. Fault Tolerant Comput.*, Jun. 1997, pp. 80–88.

[29] A. M. Memon, M. E. Pollack, and M. L. Soffa, "Hierarchical GUI test case generation using automated planning," *IEEE Trans. Softw. Eng.*, vol. 27, no. 2, pp. 144–155, Feb. 2001.

[30] A. M. Memon, "An event-flow model of GUI-based applications for testing," *Softw. Test., Verification Rel.*, vol. 17, no. 3, pp. 137–157, 2007.

[31] L. White and H. Almezen, "Generating test cases for GUI responsibilities using complete interaction sequences," in *Proc. 11th Int. Symp. Softw. Rel. Eng. (ISSRE)*, Oct. 2000, pp. 110–121.

[32] Q. Xie and A. M. Memon, "Using a pilot study to derive a GUI model for automated testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 2, pp. 1–35, Nov. 2008.

[33] S. Huang, M. B. Cohen, and A. M. Memon, "Repairing GUI test suites using a genetic algorithm," in *Proc. 3rd Int. Conf. Softw. Test., Verification Validation*, 2010, pp. 245–254.

[34] F. Belli, M. Beyazit, and N. Güler, "Event-oriented, model-based gui testing and reliability assessment—Approach and case study," *Adv. Comput.*, vol. 85, pp. 277–326, 2012, doi: 10.1016/B978-0-12-396526-4.00006-0.

[35] I. Banerjee, B. Nguyen, V. Garousi, and A. Memon, "Graphical user interface (GUI) testing: Systematic mapping and repository," *Inf. Softw. Technol.*, vol. 55, no. 10, pp. 1679–1694, Oct. 2013.

[36] E. Alégroth and R. Feldt, "On the long-term use of visual gui testing in industrial practice: A case study," *Empirical Softw. Eng.*, vol. 22, no. 6, pp. 2937–2971, Dec. 2017.

[37] T. Yeh, T.-H. Chang, and R. C. Miller, "Sikuli: Using GUI screenshots for search and automation," in *Proc. 22nd Annu. ACM Symp. User Interface Softw. Technol. (UIST)*, 2009, pp. 183–192.

[38] N. Olsson and K. Karl. (2015). *Graphwalker: The Open Source Model-Based Testing Tool*. [Online]. Available: http://graphwalker.org/index

[39] J. Eskonen, J. Kahles, and J. Reijonen, "Automating GUI testing with image-based deep reinforcement learning," in *Proc. IEEE Int. Conf. Autonomic Comput. Self-Organizing Syst. (ACSOS)*, Aug. 2020, pp. 160–167.

[40] D. Adamo, M. K. Khan, S. Koppula, and R. Bryce, "Reinforcement learning for Android GUI testing," in *Proc. 9th ACM SIGSOFT Int. Workshop Automating TEST Case Design, Selection, Eval.*, Nov. 2018, pp. 2–8.

[41] S. Jan, A. Panichella, A. Arcuri, and L. Briand, "Automatic generation of tests to exploit XML injection vulnerabilities in Web applications," *IEEE Trans. Softw. Eng.*, vol. 45, no. 4, pp. 335–362, Apr. 2019.

[42] Y. L. Arnatovich and L. Wang, "A systematic literature review of automated techniques for functional GUI testing of mobile applications," 2018, *arXiv:1812.11470*. [Online]. Available: http://arxiv.org/abs/1812.11470

[43] B. Jiang, Y. Zhang, W. K. Chan, and Z. Zhang, "A systematic study on factors impacting GUI traversal-based test case generation techniques for Android applications," *IEEE Trans. Rel.*, vol. 68, no. 3, pp. 913–926, Sep. 2019.

[44] I.-A. Salihu, R. Ibrahim, B. S. Ahmed, K. Z. Zamli, and A. Usman, "AMOGA: A static-dynamic model generation strategy for mobile apps testing," *IEEE Access*, vol. 7, pp. 17158–17173, 2019.

[45] L. Ardito, R. Coppola, S. Leonardi, M. Morisio, and U. Buy, "Automated test selection for Android apps based on APK and activity classification," *IEEE Access*, vol. 8, pp. 187648–187670, 2020.

[46] A. Silistre, O. Kilincceker, F. Belli, M. Challenger, and G. Kardas, "Community detection in model-based testing to address scalability: Study design," in *Proc. Federated Conf. Comput. Sci. Inf. Syst.*, Sep. 2020, pp. 657–660.

[47] A. Silistre, O. Kilincceker, F. Belli, M. Challenger, and G. Kardas, "Models in graphical user interface testing: Study design," in *Proc. Turkish Nat. Softw. Eng. Symp. (UYMS)*, Oct. 2020, pp. 1–6.

[48] W. E. Howden, "Reliability of the path analysis testing strategy," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 3, pp. 208–215, Sep. 1976.

[49] L. Bougé, "A contribution to the theory of program testing," *Theor. Comput. Sci.*, vol. 37, pp. 151–181, 1985, doi: 10.1016/0304-3975(85)90090-8.

[50] H. Langmaack, "Contribution to Goodenough's and Gerhart's theory of software testing and verification: Relation between strong compiler test and compiler implementation verification," in *Foundations of Computer Science*. Berlin, Germany: Springer, 1997, pp. 321–335.

[51] K. Naik and P. Tripathy, *Software Testing and Quality Assurance: Theory and Practice*. Hoboken, NJ, USA: Wiley, 2011.

[52] B. Eggers and F. Belli, "Eine theorie der analyse und konstruktion fehlertolerierender systeme," in *Fehlertolerierende Rechensysteme*. Berlin, Germany: Springer, 1984, pp. 139–149.

[53] F. Belli, "Extending regular languages for self-detection and self-correction of syntactical faults," (in German), Ph.D. dissertation, Tech. Univ. Berlin, Berlin, Germany, vol. 119.

[54] J. E. Hopcroft, R. Motwani, and J. D. Ullman, "Introduction to automata theory, languages, and computation," *ACM SIGACT News*, vol. 32, no. 1, pp. 60–65, 2001.

[55] A. Memon, I. Banerjee, and A. Nagarajan, "GUI ripping: Reverse engineering of graphical user interfaces for testing," in *Proc. 10th Work. Conf. Reverse Eng.*, 2003, pp. 260–269.

[56] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. D. Carmine, and A. M. Memon, "Using GUI ripping for automated testing of Android applications," in *Proc. 27th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2012, pp. 258–261.

[57] Y. Jia and M. Harman, "Higher order mutation testing," *Inf. Softw. Technol.*, vol. 51, no. 10, pp. 1379–1393, Oct. 2009.

[58] O. Kılınççeker and F. Belli, "Grafiksel kullanıcı arayüzleri için düzenli ifade bazlı test kapsama kriterleri," presented at the 11th Turkish Nat. Softw. Eng. Symp. (UYMS), Alanya, Turkey, Oct. 2017, pp. 332–343.

[59] F. Belli, "Regular expressions for fault handling in sequential circuits," in *Proc. ARCS 28th Int. Conf. Archit. Comput. Syst.*, 2015, pp. 1–5.

[60] O. Kilincceker, A. Silistre, M. Challenger, and F. Belli, "Random test generation from regular expressions for graphical user interface (GUI) testing," in *Proc. IEEE 19th Int. Conf. Softw. Qual., Rel. Secur. Companion (QRS-C)*, Jul. 2019, pp. 170–176.

[61] O. Kilinccceker, E. Turk, M. Challenger, and F. Belli, "Regular expression based test sequence generation for HDL program validation," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur. Companion (QRS-C)*, Jul. 2018, pp. 585–592.

[62] F. Belli, N. Nissanke, C. J. Budnik, and A. Mathur, "Test generation using event sequence graphs," Tech. Rep., 2005.

[63] E. M. Clarke, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model Checking*. Cambridge, MA, USA: MIT Press, 2018.

[64] P. E. Ammann, P. E. Black, and W. Majurski, "Using model checking to generate tests from specifications," in *Proc. 2nd Int. Conf. Formal Eng. Methods*, Jun. 1998, pp. 46–54.

[65] F. Belli and B. Güldali, "Software testing via model checking," in *Proc. Int. Symp. Comput. Inf. Sci.* Berlin, Germany: Springer, 2004, pp. 907–916.

[66] A. C. Paiva, N. Tillmann, J. A. C. Faria, and R. F. Vidal, "Modeling and testing hierarchical GUIs," in *Proc. 12th Int. Workshop Abstract State Mach.*, 2005, pp. 1–17.
[67] F. Belli, N. Güler, and M. Linschulte, "Layer-centric testing," *FERS-Mitteilungen*, vol. 30, no. 1, pp. 55–62, Apr. 2012.
[68] V. M. Glushkov, "The abstract theory of automata," *Russian Math. Surv.*, vol. 16, no. 5, p. 1, 1961.
[69] R. McNaughton and H. Yamada, "Regular expressions and state graphs for automata," *IRE Trans. Electron. Comput.*, vol. 9, no. 1, pp. 39–47, 1960.

ONUR KILINCCEKER (Member, IEEE) received the bachelor's degree in applied mathematics and the master's degree in informatics from Ege University, Turkey, in 2005 and 2011, respectively. He is currently pursuing the Ph.D. degree in electrical engineering and information technologies with the University of Paderborn, Germany. In 2009, he joined the Department of Computer Engineering, Mugla Sitki Kocman University, as a Research Associate. He is also involving in scientific and technical research in the fields of model-based testing, mutation testing, and holistic testing of software and hardware systems. He is also a member of ACM.

ALPER SILISTRE received the B.Sc. degree in software engineering from the Izmir University of Economics, in 2015. He is currently pursuing the M.Sc. degree in information technology with the International Computer Institute, Ege University, İzmir, Turkey. He is also a Software Engineer. His research interests include software testing and software engineering.

FEVZI BELLI (Member, IEEE) received the B.S., M.S., Ph.D., and Habilitation (a German postdoctoral) degrees in information technology and computer science from Technical University Berlin. He is currently a Professor Emeritus of Software Engineering with the University of Paderborn and the Izmir Institute of Technology. He has more than 35 years' experience in research, development and teaching software engineering, validation and verification, fault tolerance, and quality assurance. He started as a programmer in the aircraft industry and wrote programs to create simulation environments and to validate safety critical features. In 1983, he was awarded a Professorship with the University of Applied Sciences in Bremerhaven; in 1989, he moved to the University of Paderborn. He was also, for many years, a Faculty Member of the University of Maryland, College Park, MD, USA, and the European Division. He was also the Founding Chair of the Computer Science Department, University of Economics in Izmir, Turkey. He has an interest and experience in software reliability/fault tolerance, model-based testing, and test automation.

MOHARRAM CHALLENGER (Member, IEEE) received the Ph.D. degree in information technology from the International Computer Institute, Ege University, in February 2016. From 2005 to 2009, he had been a Faculty Member and a Senior Lecturer with the Computer Engineering Department, IAU-Shabestar University. From 2010 to 2013, he was a Researcher and a Team Leader of a bilateral project between Slovenia and Turkey (TUBITAK). From 2012 to 2016, he was the Research and Development Director of UNIT IT Ltd. leading a national project funded by TUBITAK and two international software-intensive projects in Europe called ITEA ModelWriter and ITEA Assume. From 2017 to 2018, he had been a member of the faculty as an Assistant Professor with Ege University. From January 2019 to July 2020, he was a Postdoctoral Researcher with the University of Antwerp working in a Flanders Make projects called PACo and DTDesign. He is currently a tenure-track Assistant Professor with the Department of Computer Science, University of Antwerp. His research interests include domain-specific modeling languages, multi-agent systems, cyber-physical systems, and the Internet of Things.