2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)

Random Test Generation from Regular Expressions for Graphical User Interface (GUI) Testing

Onur Kilincceker
University of Paderborn
Paderborn, Germany
Mugla Sitki Kocman University
Mugla, Turkey
okilinc@mail.upb.de

Alper Silistre
International Computer Institute,
Ege University,
Izmir, Turkey.
alpersilistre@gmail.com

Moharram Challenger
Electronics-ICT Dep.
University of Antwerp
Antwerp, Belgium.
moharram.challenger@uantwerpen.be

Fevzi Belli
University of Paderborn,
Paderborn, Germany.
Izmir Institute of Technology,
Izmir, Turkey.
belli@upb.de

***Abstract*— Generation of test sequences, that is, (user) inputs - expected (system) outputs, is an important task of testing of graphical user interfaces (GUI). This work proposes an approach to randomly generate test sequences that might be used for comparison with existing GUI testing techniques to evaluate their efficiency. The proposed approach first models GUI under test by a finite state machine (FSM) and then converts it to a regular expression (RE). A tool based on a special technique we developed analyzes the RE to fulfill missing context information such as the position of a symbol in the RE. The result is a *context table* representing the RE. The proposed approach traverses the context table to generate the test sequences. To do this, the approach repeatedly selects a symbol in the table, starting from the initial symbol, in a random manner until reaching a special, finalizing symbol for constructing a test sequence. Thus, the approach uses a *symbol coverage criterion* to assess the adequacy of the test generation. To evaluate the approach, mutation testing is used. The proposed technique is to a great extent implemented and is available as a tool called *PQ-Ran Test* (*PQ-analysis based Random Test Generation*). A case study demonstrates the proposed approach and analyzes its effectiveness by mutation testing.**

***Keywords*— Random Test Generation, GUI Testing, Regular Expression, Finite State Machine**

I. INTRODUCTION

Testing of software user interfaces (UI) or graphical user interfaces (GUI) is an important topic in software development process. Testing, in general, is a critical part of software development life cycle and should be considered during the development process. Customers generally tend to care the aesthetics and usability of the application she/he is interested in purchasing. There are many reasons for this, but one of the most important is that the Internet forms a rapid and very competitive environment, so users usually have alternatives of the same kind of products, which results in many challenges in testing GUIs to check and increase the attractiveness of offered products.

Application developers need to consider their GUI and user experience designs in order to attract more users. Poorly designed GUI will lead to unsatisfied customers who will get bored with the offered product.

It is very important to catch unnoticed bugs and flaws in GUI and in the project before their designs go into production. GUI testing is the process of testing the visual elements of an application and it's design to prevent the problems mentioned above before any user can perceive them. Those elements that influence the attractiveness can be color, font, size, etc., of the visual elements on the screen and business policy can be checked with the help of automated UI testing. Any undesirable event, such as a design flaw, that will occur during the automated GUI testing will reveal the problems at the core of the application. Manual GUI testing is a cumbersome process of checking the application before it goes to the market. This process tends to be sloppy because of the difficulties of the manual testing.

With the evolution of technology, new techniques and processes have emerged to improve testing aspects of software development life cycle. GUI testing is no longer considered as a job to be done by a tester manually. Nowadays, software projects become larger and larger, and thus the required labor for testing every part of their GUI by hand increases excessively, so that it almost becomes infeasible.

Random tests are in general not an effective way to validate products. However, random testing plays a crucial role in testing activity [1] due its simplicity that explains its popularity to be used as a yardstick to compare a novel, suggested method with existing testing techniques, and to evaluate its efficiency.

This work proposes an approach to random generation of test sequences for GUI testing to address sequencing and functional faults. In case of sequencing faults, GUI under test (GUT) is unable reach the final event, this might cause a system crash. In case of functional faults, the GUT is unable to provide the desired functionality even if it reaches the final event.

This paper suggests modeling the GUT by a finite state machine that will automatically be converted to a regular expression (RE) using a tool. The RE has the same expressional power as the corresponding FSM as both can be represented by a type-3 grammar, generating the same regular language. The tester can also model the GUT directly with a RE if he/she is familiar with working with RE.

There are several reasons why we prefer working with RE to working with FSM. First, RE form algebra (event algebra, see [2]) that allows algebraic operations that are considerably easier and more efficient to be handled than graph-based operations on FSM. Secondly, a RE model is mostly less spacious than a graph model. Last but not least, analyzing contextual relations can easier and more efficiently be carried out by RE than with graph models.

Therefore, a tool to make up the missing context information such as the position of a symbol that cannot directly be determined in the original model, that is, FSM, will analyze the RE. The result of this analysis is the *context table* representing the contextual relations of the symbols. As a next step, the context table will be traversed to generate the test sequences. To do this, a symbol in the table is repeatedly selected, starting from the initial symbol, in a random manner until reaching a special, finalizing symbol for constructing a test sequence. The selected symbol will be excluded from the further iterations and included in the list

978-1-7281-3925-8/19/$31.00 ©2019 IEEE
DOI 10.1109/QRS-C.2019.00044

of covered symbols. Thus, the approach uses a *symbol coverage criterion* to assess the adequacy of the test generation. Once the required symbol coverage ratio is achieved, the test generation terminates to exclude redundant test sequences. For example, setting symbol coverage to 100% requires covering all of the different symbols in the table. Once predefined coverage ratio achieved, test generation terminates and excludes redundant test sequences that are incomplete sequences. The proposed technique is implemented and is available as a tool called *PQ-Ran Test* (*PQ-analysis based Random Test Generation*). The case study in Section V demonstrates the utilization of this tool, among other aspects.

The approach comprises test preparation and testing steps. In test *preparation* step, the tester models the GUI by a FSM using the tool JFLAP [19] that also converts the FSM into a corresponding RE. A tool analyzes the RE to construct the context table that contains contextual information of the symbols contained in the RE. In *testing* step, the developed tool PQ-RanTest applies to the context table to generate test sequences.

For validation of the approach, mutation testing technique is used. This technique entails generation of mutants of the GUT as its faulty versions [28,29] to model functional and sequencing faults. Mutants are obtained by applying mutation operators to the source code of the GUT. These mutation operators form slight changes of the code, such as manipulation of an assignment. The test sequences generated from the RE and its context table will then be applied to these mutants. If they reveal a fault the mutant is said to be "killed", that is, the test sequence was successful. Otherwise we have a behavioral *equivalent* mutant of the GUT. For running the tests, a test automation tool will be used.

Section 2 summarizes related work on GUI testing and random test generation. Section 3 briefly summarizes the used notions and techniques as background information. Section 4 presents the proposed approach. A case study demonstrates and evaluates the proposed approach in Section 5. Section 6 provides the tool support. Section 7 summarizes the results of case study. Section 8 concludes the paper discussing the results and further perspectives.

II. Related Work

The term GUI testing refers to testing a GUI-based application, that is, one that has a graphical-user interface (GUI) front-end. GUI testing is based on executing sequences of operations as events, such as, "click on button", "enter text", "open menu" on GUI widgets, such as , "button", "text-field", "pull-down menu" [3].

The GUI testing can be effectively specified via a well-selected model, for example a finite state machine (FSM) [4], event sequence graph (ESG) [5,6], event flow graph (EFG) [7,8]. The FSM and ESG are graph-based models, thus requires usage of the graph traversal and optimization algorithms on the graph model to generate test sequences. Semi-formal models as EFG and UML diagrams are harder to be used for algorithmic operations, such as optimization of test suites, as their rich syntax and semantics do not always allow application of strict mathematical methods, for example graph theory.

Shehady and Siewiorek [4] introduced a formal way of representing a GUI, namely variable finite state machine (VFSM) that is constructed from the specification of the system. VFSM is then converted to a finite state machine (FSM) for test generation using the well-known W-Method, originally proposed by Chow [9]. The W-Method requires a fully specified FSM, thus, the model may contain a huge number of NULL transitions. Even though the VFSM involves less number of states than the FSM, test generation algorithm runs on the huge number of states of the FSM.

Belli [5] proposes event sequence graphs (ESG) for modeling GUI and test sequence generation techniques based on optimization algorithms. The ESG represents the given GUI's events at the vertexes and the relations of the events at the edges on this a graph like a model. Testing methods in [5] combines positive and negative testing to achieve a holistic view. The system is checked against legal inputs (correct behavior) in *positive* testing and illegal inputs (incorrect behavior) in *negative* testing in accordance with the expectations of the user. The proposed method is generic, i.e. defining both legal and illegal cases.

Memon et al. [7] present a planning algorithm based on artificial intelligence for automatic test generation from the EFG model. They also propose a hierarchical model generation from the given GUI structure. The EFG model is constructed and then the planning algorithm, which requires defining a set of operators, an initial state, and a goal state, is initiated. Finally, the algorithm generates test sequences between initial and goal states by considering GUI events and interactions. They also use decomposition of the GUI model to handle the scalability problem.

Besides mentioned models to represent GUI, Belli [5] offers RE model being equivalent to FSM in expressive power that is variety and quantity contained in the model. Moreover, RE offers compactness and different coverage metrics than FSM. Kilincceker and Belli [22] offer novel RE based coverage criteria and analysis their features regarding to test generation complexity. They also apply these coverage criteria on GUI testing.

Ravi et al. [23] introduce RE based testability analysis and optimization methods for register transfer level sequential circuits. Kilincceker et al. [24] present an RE-based technique for validation of hardware description language (HDL). They use abstract syntax tree (AST) representation of RE for which they suggest a traversal algorithm on this tree considering algebraic operators of RE model. RE becomes more popular and applies on different domains [25,26,27] of testing area due to its algebraic structure and compactness.

Random testing is a fundamental and important aspect of software testing due its simplicity [10]. It refers to selection of random test input from input domain of software under test. There are several approaches proposed for random testing such as Adaptive Random Testing (ART) [11] and Directed Automated Random Testing (DART) [12]. These approaches apply on source code of the system.

The present paper introduces a specific method for random test generation from RE model based on symbol coverage criteria. AutoTest [13] is a similar research tool developed by ETH Zürich. It generates test inputs from contract specifications. GraphWalker [14] is another model-based random test generation tool. It uses graph model of the

system and generates random test inputs from this model. It also enables to terminate test generation by means of predefined coverage criteria, which are edge coverage and vertex coverage.

To our best knowledge, this is the first work to propose a random test generation using a RE-based model. This work presents a test generation technique that consists of test preparation step and testing step, controlled by the coverage criterion specifically developed.

III. Preliminaries

The notions of finite state machine (FSM) and regular expression (RE) will be used in further sections extensively. The formal definition can be given as follows.

Definition 3.1. *Finite State Machine* (*FSM*) [15]: Following 5-tuple defines a FSM

<Q, Σ, δ, q_0, F> with

Q: a finite set of states

Σ: a finite set of input symbols (alphabet)

δ: a state transition function

q_0: an initial (starting) state belongs to Q

F: a finite set of final states belongs to Q.

Definition 3.2. *Regular Expression* (*RE*) [15]: A RE by means of rules is defined by an alphabet which is shown by a sequence of symbols a, b, c,... Symbols can occur zero or more times related to the following rules that define the RE.

- *Concatenation* – is represented by "." or ""(blank). For example, "xy" refers to 'a is followed by b'.
- *Selection* (Union), is represented by "+". For example, x+y refers to 'x (exclusive) or y'.
- *Iteration* (Kleene's Star), is represented by "*". For example, "x*" refers to 'x is iterated a desired times' (including zero times that means the *empty word* "λ"). Moreover, "x+" means at least one time iteration that is excluding λ.

Example 1: As an example, a RE below can be given.

[(xy(t+z)*)] (1)

The RE in (1) refers that the symbol "x" is followed by the symbol "y" which is followed by zero or more times iteration of symbol "t" or "z".

IV. Proposed Approach

The proposed approach comprises of two steps: *Test preparation* and *testing*. In test preparation step, we model the GUI behavior by means of a FSM using JFLAP tool [19] that also converts the FSM to RE and finally analyzes the RE to construct context table.

In testing step, the random test generator, called PQ-RanTest developed in our research work, is executed to generate test sequences. Fig. **1** depicts the concept of the proposed approach. Following subsections explain the details of the test preparation and testing steps.

A. Test Preparation

The GUI is the input of this step. A tester models the GUI manually to obtain an appropriate finite state machine (FSM) representation. The tester can also draw the FSM model from GUI using the JFLAP tool [19] that is then utilized to convert FSM into RE model. RE contains missing context information that is necessary for random test generation. Context refers to location of a symbol in RE and its relations with other symbols. For example, "[(xy (t+z) *)]" is a RE given in (1). Right side of symbol "x" is only "y" and left side is symbol "[". Thus, right side of the "x" symbol in the context table contains symbol "y" and left side is "[". We use terms *right context* and *left context* of a symbol. Context table contains for all symbols of the RE their *right* and *left context*. Moreover, PQ-Analysis uses indexing the RE to remove ambiguities in the RE that might cause missing context-based errors. Consider a same symbol in different locations in the RE, covering a symbol means addressing only one symbol in one location and missing others. Thus, PQ-Analysis uses indexing to overcome these problems.

B. Testing

The output of the PQ-Analysis tool [16,17] is the input of the *PQ-RanTest* tool which is developed and utilized within the scope of the proposed approach. *PQ-RanTest* traverses the context table starting from initial symbol to end symbol in a random manner.

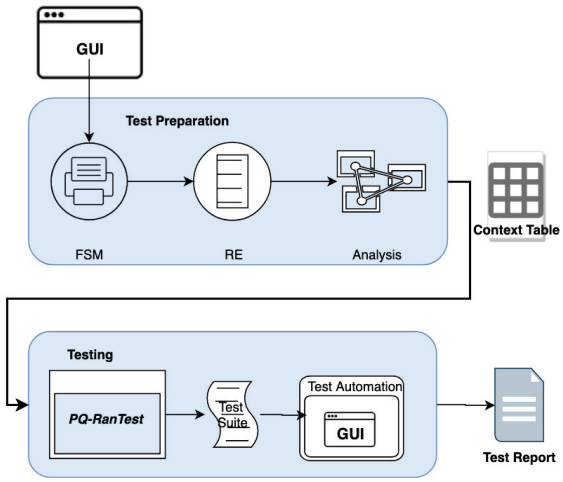

Fig. 1: Illustration of the Approach

To assess adequacy of the test generation, a symbol coverage criterion is used. The symbol coverage criterion is a predefined ratio to terminate test generation when it is achieved. For example, 80 coverage ratios refers to 80% of the all different symbols in the context table required to be contained in the already generated test sequences for test generation termination.

PQ-RanTest terminates the test generation process when the predefined symbol coverage ratio is achieved. Then, the generated test sequences are put together into a test suite. Finally, a test automation tool, such as Selenium, automatically executes this test suite on GUI under test (GUT). Test report from testing step finishes the procedure.

V. Case Study

In this paper, we select a case study that is a module of Web-based commercial tourist portal, the GUI of ISELTA (*Isik's System for Enterprise Level Web-Centric Tourist Applications* (*ISELTA*) [16]. ISELTA [16] 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 and the University of Paderborn. ISELTA implemented in PHP programming language. We apply mutation operators to the PHP source code of ISELTA to obtain mutants that model the addressed sequencing and functional faults.

"Special" module of ISELTA enables users to offer advertisements for special events, for example, a celebration event from high school. Thus, users provide special prices, rooms at specific date to customers. "Special" module contains "title" for event name, "number" for number of available rooms, "price" for event value and others shown in TABLE I.

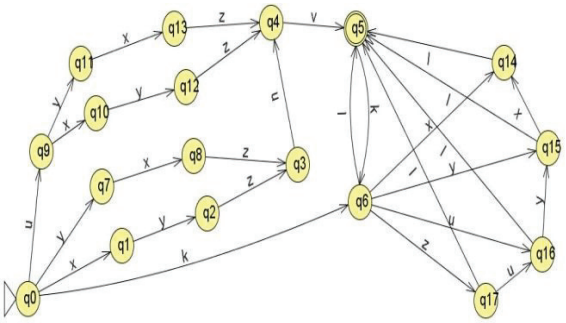

Fig. 2: ISELTA "Special" Module Original FSM

TABLE I: ABBREVIATIONS OF THE EVENTS ON FSM

Letter	Action
k	Click edit
l	Click save
v	Click Add
u	Set title text
x	Set number value
y	Set price value
z	Set description text
r	Remove all text
t	Remove title input
p	Remove price input
n	Remove number input

In the original FSM model of the "Special" module, see Fig. 2, we assign a symbol to each event as a transition label. The events represent actions such as filling an input, clicking a button, or removing some text from an input. These labels enable to implement Selenium test scripts. The event letters used in the current work are defined in TABLE I.

JFLAP tool [19] automatically converts the FSM in Fig. 2 into the RE model in (2).

Example 2: An example, a RE of the "Special" module is given in (2);

```
(kz(u(y(xl+l)+l)+l)+ku(y(xl+l)+l)+ky(xl+
l)+kxl+uyzv+uzyv+zyuv+kl+yzuv)(kz(u(y(xl
+l)+l)+l)+ku(y(xl+l)+l)+ky(xl+l)+kxl+kl)
*                                        (2)
```

After obtaining the RE model from the FSM, we input the RE into a software tool called *PQ-Analysis* [16,17]. This is specific software designed to analyze RE for context properties. Fig. 3 shows main screen of the PQ-Analysis tool.

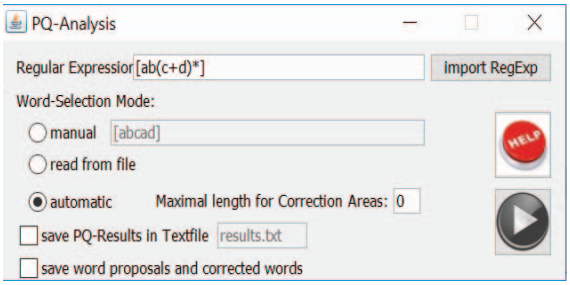

Fig. 3: PQ-Analysis tool

The regular expression field accepts the RE that we obtained from the JFLAP tool [19]. PQ-Analysis generates the context table shown in Fig. 4.

```
## CONTEXT TABLES ##
## RIGHT-LEFT-CONTEXT-TABLE OF T ##
              x'L                |Symbol|      x'R
---------------------------------+------+---------------
                                 | 1'[  |2'a + 3'b + 4']
---------------------------------+------+---------------
1'[ + 2'a + 7'a + 10'c + 11'a| 2'a  |2'a + 3'b + 4']
---------------------------------+------+---------------
1'[ + 2'a + 7'a + 10'c + 11'a| 3'b  |   5'a + 6'b
---------------------------------+------+---------------
1'[ + 2'a + 7'a + 10'c + 11'a| 4']  |
---------------------------------+------+---------------
          3'b + 6'b              | 5'a  |   7'a + 8'b
---------------------------------+------+---------------
          3'b + 6'b              | 6'b  |   5'a + 6'b
---------------------------------+------+---------------
             5'a                 | 7'a  |2'a + 3'b + 4']
---------------------------------+------+---------------
             5'a                 | 8'b  |  9'a + 10'c
---------------------------------+------+---------------
         8'b + 12'b              | 9'a  |  11'a + 12'b
---------------------------------+------+---------------
         8'b + 12'b              | 10'c |2'a + 3'b + 4']
---------------------------------+------+---------------
             9'a                 | 11'a |2'a + 3'b + 4']
---------------------------------+------+---------------
             9'a                 | 12'b |  9'a + 10'c
```

Fig. 4: PQ-Analysis result window

PQ-Analysis tool helps to save the results in a text file. We need a text file because in the next step our tool called PQ-RanTest takes the results.txt as an input and generates test sequences.

After generating test sequences, we utilize Selenium tool [20] to automate test execution that can run each mutant of ISELTA. Testing each case by hand would take a huge amount of time. With the help of the automated test process and Selenium, we can finish each mutant test suite within 2-3 minutes.

There are in total 12 mutants generated from original code of ISELTA by means of mutation operators. All mutants of the ISELTA system are available on [21].

TABLE 2: MUTANTS AND SEMANTICS

Mutant	Semantics
1	Add empty input boxes
2	Update empty input boxes
3	Add empty Number of Packages input box
4	Add empty price input box
5	Add empty title input box
6	Update empty title input box
7	Update empty price input box
8	Update empty number input box
9	Add click does not respond
10	Edit click does not respond
11	Both Add and Edit click does not respond
12	After edit save button move to initial state without saving

*Mutants are available online on [21].

Mutant 1-8 model functional faults and mutant 9-12 model sequencing faults. For example, mutant one has "add empty input boxes" semantics meaning that someone enables to add a special feature without filling required information in the boxes. This mutant models functional fault due to undesired action by passing required fields without system warning. To model this fault, the deletion and insertion mutation operators change the code of the system. Fig. 5 shows that condition of "if" statement, red under lined in (a) and (b), deleted and "true" is inserted.

```
################################
########## add Special ##########
################################

} elseif(isset($_POST['btn_addSpecial'])) {
    $obj_product = $_SESSION['productController']->

    if($allValid && specialsAreValid($obj_product))
        if(!$obj_product->addSpecial($_POST['frm_a
            error("COULD_NOT_SAVE");
            debug("FEHLER addSpecials: Zuweisung i
        } else {
            HTMLElement::flushElements();
        }
```

(a) Original Code

```
################################
########## add Special ##########
################################

} elseif(isset($_POST['btn_addSpecial']))
    $obj_product = $_SESSION['productCont

    if(true){
        if(!$obj_product->addSpecial($_POS
            error("COULD_NOT_SAVE");
            debug("FEHLER addSpecials: Zu
        } else {
            HTMLElement::flushElements();
        }
```

(b) Mutated Code

Fig. 5: Original (a) and mutated (b) code segments

VI. Tool Support

A set of tools are available for the proposed approach. Within the scope of the current work, we developed some of the tools, and we obtained others from different sources. We integrated all tools into a tool chain that helps us to get rid of tedious and error-prone manual work. Fig. 6 shows the tools used in the current work. Among these tools, JFLAP [19], PQ-Analysis [16,17], and PQ-RanTest are domain independent tools, whereas Selenium [20] is domain dependent designated for test automation for software testing.

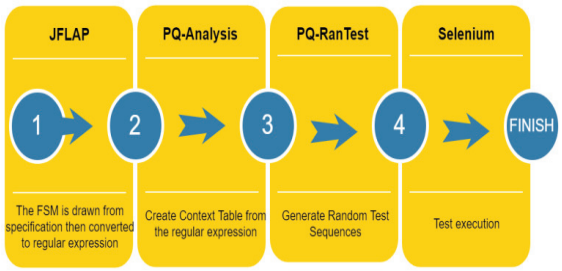

Fig. 6: The tools used in the proposed approach

A. JFLAP

Java Formal Languages and Automata Package (JFLAP) assists to design and experiment with techniques known from formal languages and automata theory [19]. JFLAP supports several formal models such as finite-state automation, Mealy and Moore machine, and regular expression. It also enables conversion between these models. It is extensively used in both teaching formal languages and experimenting in academia.

We use JFLAP for modeling a given GUI of software by means of a finite state machine (FSM) and converting it to a regular expression (RE). There are several algorithms for conversion from a FSM to a RE. The algorithm used in JFLAP results in more compact RE than others as we found from on our experiments. Therefore, JFLAP is selected for both modeling and conversion.

B. PQ-ANALYSIS

PQ-Analysis [16,17] tool is a software for modeling the GUT, which can be a software system or a hardware system coded in a hardware description language (HDL).

PQ-Analysis results in a table that contains contextual information. This context table contains valuable information for test generation that is normally missing in a conventional regular expression. To be more precise, we use the context table resulted from PQ-Analysis for test generation.

PQ-Analysis tool enables to convert FSM into RE model. However, we choose JFLAP for conversion due to size of the resulted RE model. JFLAP results in compact RE based on the difference between the algorithms used for conversion.

PQ-Analysis results in a context table divided into two parts that are forward and backward context table and each part containing right and left context information. Kilincceker and Belli [22] propose coverage criteria based on forward and backward parts of the context table.

C. PQ-RANTEST

PQ-RanTest is a tool for random test sequence generation from a context table resulted from PQ-Analysis tool. It utilizes a symbol coverage criterion for the context table that contains symbols. It traverses the context table to generate test sequences starting from the opening symbol "[" and ending at the closing symbol "]" in a random manner. PQ-RanTest finishes test generation when the required symbol coverage ratio achieved, and resulting test sequences are collected into a test suite.

D. SELENIUM

Selenium is a test automation tool to execute test sequences and produce a test report. It consists of the components *Web-Driver* and *IDE*. The Web-Driver provides functions for integrating the programming languages Java and Python for automation of the test execution. IDE has an easy-to-use interface including plug-ins for specific internet browser and simple record-and-playback of interactions with the browser.

The proposed approach uses Web-Driver part for integration and automation. Web-Driver containing generic java test scripts performs test execution for test sequences generated from PQ-RanTest.

VII. RESULTS AND DISCUSSION

This paper uses "Special" module of ISELTA to validate the proposed approach. To evaluate effectiveness, we address fault coverage, length of generated test sequences, time for test generation and time for test execution metrics. TABLE 3 shows results based on these metrics on "Special" module of ISELTA.

As a first step, we construct the FSM model of "Special" module of ISELTA that will automatically converted into a corresponding RE. Our specific technique, PQ-Analysis acquires the context table by analyzing the RE. We generate test sequences from the context table using PQ-RanTest. We give details in Section 5.

We carry out an experiment on the mutants of the of "Special" module of ISELTA, by executing test sequences generated by PQ-RanTest. The mutants model functional and sequencing faults. We measure effectiveness of PQ-RanTest for revealing modeled functional and sequencing faults.

TABLE 3: TEST RESULTS FOR THE RANDOM TEST GENERATION AND TEST EXECUTION

	Symbol Coverage	
	100%	**60%**
Fault Coverage (%)	100	87.5
Length of Test Sequences (Symbols)	3056	608
Time for Test Generation (secs)	2.8	0.7
Time for Test Execution (secs)	857	235

Results in TABLE 3 show that PQ-RanTest achieves 100% fault coverage in 2.8 seconds for test generation, 857 seconds for test execution and total length of test suite are 3056 symbols. While setting the symbols coverage to 60% results in 87.5% fault coverage in 0.7 seconds for test generation, 235 seconds for test execution and total length of test suite are 608 symbols.

The above summarized results of the case study show that the proposed approach is effective to reveal functional and sequencing the faults. We have modeled eight functional and four sequencing faults by utilizing mutation operators that are applied on source code of GUT. We observe that a decrease in the symbol coverage reduces the fault coverage ratio. Thus, the symbol coverage and the fault detection ratio correlate.

VIII. CONCLUSION

This paper proposes a context-driven random test generation approach for testing GUI using regular-expression-based techniques. We demonstrated the proposed approach using a case study. Results show that the proposed approach is effective to reveal functional and sequencing faults. Coverage-oriented random test generation enables to control the test process by setting desired coverage ratio. The context-driven test generation from RE model offers a new perspective to model-based test generation area.

As a future work and for evaluation and improvement of the proposed approach we plan further experiments with other model-based random test generation approaches such as AutoTest [13] and GraphWalker [14]. Moreover, we intent to conduct more experiments using different fault models to evaluate effectiveness of PQ-RanTest and thus improve our techniques and tools.

REFERENCES

1. Chen, T. Y., Kuo, F. C., Merkel, R. G., & Tse, T. H. (2010). Adaptive random testing: The art of test case diversity. *Journal of Systems and Software*, *83*(1), 60-66.
2. Salomaa, A., Two Complete Axiom Systems for the Algebra of Regular Events, Journal of the ACM 13(1):158-169, DOI: 10.1145/321312.321326 (1966).
3. Banerjee, Ishan, Bao Nguyen, Vahid Garousi, and Atif Memon. "Graphical user interface (GUI) testing: Systematic mapping and repository." Information and Software Technology 55, no. 10 (2013): 1679-1694.
4. R.K. Shehady, D.P. Siewiorek, A method to automate user interface testing using variable finite state machines, in: Proceedings of the 27th Annual International Symposium on Fault-Tolerant Computing (FTCS 1997), IEEE Computer Society, Washington, DC, 24–27 June, 1997, pp. 80–88.
5. Belli, F., Finite state testing and analysis of graphical user interfaces. Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on. IEEE, (2001).
6. Belli, F., Budnik, C. J., Hollmann, A., Tuglular, T.,Wong, W. E., Model-based mutation testing approach and case studies. Science of Computer Programming, *120*, 25-48, (2016).
7. A.M. Memon, M.E. Pollack, M.L. Soffa, Hierarchical GUI test case generation using automated planning, IEEE Trans. Software Eng. 27 (2) (2001) 144–155.
8. Memon, Atif M. "An event-flow model of GUI-based applications for testing." Software testing, verification and reliability 17.3 (2007): 137-157.

9. Chow, T. S. (1978). Testing software design modeled by finite-state machines. IEEE transactions on software engineering, (3), 178-187.
10. Anand, S., Burke, E. K., Chen, T. Y., Clark, J., Cohen, M. B., Grieskamp, W., ... & Li, J. J. (2013). An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software, 86(8), 1978-2001.
11. Chen, T. Y., Kuo, F. C., Merkel, R. G., & Tse, T. H. (2010). Adaptive random testing: The art of test case diversity. Journal of Systems and Software, 83(1), 60-66.
12. Godefroid, P., Klarlund, N., & Sen, K. (2005, June). DART: directed automated random testing. In ACM Sigplan Notices (Vol. 40, No. 6, pp. 213-223). ACM.
13. Ciupa, I., Leitner, A.: Automatic testing based on design by contract. In: Proceedings of Net. ObjectDays 2005, pp. 545–557 (2005).
14. Olsson, N., and K. Karl. "Graphwalker: The open source model-based testing tool." (2015).
15. Hopcroft, John E., Rajeev Motwani, and Jeffrey D. Ullman. "Automata theory, languages, and computation." International Edition 24.2.2 (2006).
16. F. Belli, Extending Regular Languages for Self-Detection and Self-Correction of Syntactical Faults (PhD Thesis in German; Technical Univ. Berlin), Bericht 119 der Gesellschaft für Mathematik und Datenverarbeitung, , Oldenburg Verlag, 1978.
17. PQ-Analysis Tool, Available online: http://download.ivknet.de/, (last access: May 2019).
18. ISELTA website, Available online: http://iselta.ivknet.de/, (last access: May 2019).
19. JFLAP Tool, Available online: http://www.jflap.org/, (last access: May 2019).
20. Selenium Test Automation Tool, Available online: https://www.seleniumhq.org/, (last access: May 2019).
21. ISELTA Mutants, Available online: http://iseltamutants.ivknet.de/, (last access: May 2019).
22. Kilincceker, Onur, and Fevzi Belli. "Regular Expression based Coverage Criteria for Graphical User Interface", CEUR Workshop Proceedings, (in Turkish), 2017.
23. Ravi, Srivaths, Ganesh Lakshminarayana, and Niraj K. Jha. "TAO: Regular expression-based register-transfer level testability analysis and optimization." IEEE Transactions on Very Large Scale Integration (VLSI) Systems 9.6 (2001): 824-832.
24. Kilincceker, O., Turk, E., Challenger, M., & Belli, F. (2018, July). Regular Expression Based Test Sequence Generation for HDL Program Validation. In 2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C) (pp. 585-592). IEEE.
25. Kilincceker, O., Turk, E., Challenger, M., & Belli, F. (2018, April). Applying the Ideal Testing Framework to HDL Programs. In ARCS Workshop 2018; 31th International Conference on Architecture of Computing Systems (pp. 1-6). VDE.
26. Arcaini, Paolo, Angelo Gargantini, and Elvinia Riccobene. "Fault-based test generation for regular expressions by mutation." Software Testing, Verification and Reliability 29.1-2 (2019): e1664.
27. Mercan, G., Akgündüz, E., Kılınççeker, O., Challenger, M., & Belli, F.. "Preliminary work on ideal testing of android application", (2018), (in Turkish), CEUR Workshop Proceedings.
28. DeMillo, Richard A., Richard J. Lipton, and Frederick G. Sayward. "Hints on test data selection: Help for the practicing programmer." Computer 11.4 (1978): 34-41.
29. Belli, Fevzi, Christof J. Budnik, and W. Eric Wong. "Basic operations for generating behavioral mutants." *Second Workshop on Mutation Analysis (Mutation 2006-ISSRE Workshops 2006)*. IEEE, 2006.