Event-Based GUI Testing and Reliability Assessment Techniques

An Experimental Insight and Preliminary Results

Fevzi Belli, Mutlu Beyazıt
Department of Electrical Engineering and Information Technology; University of Paderborn
Padeborn, Germany
belli@adt.upb.de, beyazit@adt.upb.de

Nevin Güler
Departments of Statistics
University of Muğla
Muğla, Turkey
nevinguler@hotmail.com

***Abstract*—It is widely accepted that graphical user interfaces (GUIs) highly affect—positive or negative—the quality and reliability of human-machine systems. In spite of this fact, quantitative assessment of the reliability of GUIs is a relatively young research field. Existing software reliability assessment techniques attempt to statistically describe the software testing process and to determine and thus predict the reliability of the system under consideration (SUC). These techniques model the reliability of the SUC based on particular assumptions and preconditions on probability distribution of cumulative number of failures, failure data observed, and form of the failure intensity function, etc. We expect that the methods used for modeling a GUI and related frameworks used for testing it also affect the factors mentioned above, especially failure data to be observed and prerequisites to be met. Thus, the quality of the reliability assessment process, and ultimately also the reliability of the GUI, depends on the methods used for modeling and testing the SUC. This paper attempts to gain some experimental insight into this problem. GUI testing frameworks based on event sequence graphs and event flow graphs were chosen as examples. A case study drawn from a large commercial web-based system is used to carry out the experiments and discuss the results.**

Keywords-GUI modeling and testing; reliability modeling/assessment/prediction; event flow graphs; event sequence graphs

I. INTRODUCTION AND RELATED WORK

Graphical user interfaces (GUIs) play a significant role in improving the usability of the software system, enabling easy interactions between user and system. Thus, a well-developed GUI is an important factor for software quality.

Graph-based abstractions are often used to model and test GUIs. The basic idea is to build a graph model of the system under consideration (SUC) and use some coverage criteria to generate test cases [1]. Achieving a proper level of coverage entails the generation of test cases and the selection of an optimal number of them. Thus, it ensures the cost-effective exercise of a given set of structural or functional features.

Several approaches exist in literature to automate testing of GUIs. Memon et al. [2] propose the use of event flow graph (EFG) models in the testing of GUIs. EFG models are later used in many works and become a key component of a commonly used GUI testing framework [3][4]. Based on directed graphs, Belli [5] uses event sequence graph (ESG) models to represent both the legal and the illegal behavior based on GUI events. This model is also used through several works and a corresponding testing framework is implemented [6][7].

Paiva et al. [8] present an approach to model and test GUIs based on hierarchical finite state machines (HFSMs), where vertices of the underlying finite state machines (FSMs) not only represent single states, but also groups of states. This method reduces the number of states in the equivalent flat FSM resulting from the exploration of the model. Furthermore, Shehady and Siewiorek [9] propose a method to automate testing of GUIs by making use of variable finite state machines (VFSMs). VFSM models cope with scaling problems of FSMs, which is an important issue in GUI testing.

From the point of view of the automata theory of formal languages, the above mentioned models are equivalent in the sense that they all model event-based regular systems generated by type-3 grammars. In FSMs, nodes represent states and arcs represent transitions, which are labeled by events in the system [10][11]. In EFGs and ESGs, nodes represent events and arcs represent sequences of these events. Thus, EFGs and ESGs focus on the events and do not process the states explicitly. In some sense, this makes EFGs and ESGs easier to learn and apply, even by users without any detailed knowledge of automata theory or graph theory.

One of the primary goals of testing is to improve reliability, since reliability is a user-centric measure. The present paper attempts to apply reliability analysis to GUI testing. Since the early seventies of the last century, several techniques have been developed to model and predict software reliability. They use statistical theory of stochastic processes and Bayesian approach; the most widely accepted of those models are collected and recommended in standards such as [12][13]. In this paper, we will refer to those models and techniques as "software reliability assessment techniques (SRATs)" to prevent confusion with the term "GUI models" used in this paper.

The problem with SRATs is that they are generally based on particular assumptions and have different prerequisites and parameters. Thus, there is no single one of them that is universally applied to all types of software. Therefore, the selection of proper SRAT is quite important to perform a correct reliability analysis of the SUC.

We expect that, in GUI testing, methods and frameworks used may affect the shape of fault data. Some SRATs may turn out to be totally infeasible. Therefore, SRATs which demonstrate a better performance may vary with respect to the selected GUI testing framework. This paper aims to gain some experimental insight in the use of SRATs in GUI testing by discussing the following issues.

- Which of the existing SRATs are suitable for GUI testing?
- Does the selected GUI model and its testing framework affect SRATs and the reliability analysis process?

To diversify the experiments and avoid biased results, tests are performed both in EFG-based and ESG-based frameworks. We selected EFGs and ESGs and exclude UML diagrams [13], since we are primarily interested in formal models that focus on events and have simpler syntax and semantics. Section II summarizes the test frameworks selected, whereas Section III outlines SRATs used to perform the case study.

Section IV reports the case study on a large GUI-based web application. To perform the case study, first two GUIs are selected in the system and their EFG and ESG models are constructed. For the testing process, real-life faults are seeded into the selected modules and their faulty versions (mutants) are created. Test generation is performed by using the EFG-based and ESG-based GUI testing frameworks and generated test cases are executed on the faulty versions. Based on the results of the test execution, fault reliability analysis is performed on the fault data obtained during test execution. Finally, results are interpreted and validity threats are outlined. Section V concludes the paper, summarizing the future work planned.

II. GUI TESTING FRAMEWORKS

In the following, GUI models and related test frameworks used for performing the experiments are outlined.

A. GUI Models

Definition 1: An *event flow graph* (*EFG*) for a GUI component C is a 4-tuple $<V, E, B, I>$ where

- V is a set of vertices representing all the *events* in the component. Each $v \in V$ represents an event in C.
- $E \subseteq N \times N$ is a set of *directed edges* between vertices. We say that event e_i follows e_j if and only if e_j may be performed *immediately* after e_i. An edge $(v_x, v_y) \in E$ if and only if the event represented by v_y *follows* the event represented by v_x.
- $B \subseteq V$ is a set of vertices representing those events of C that are available to the user when the component is first invoked.
- $I \subseteq V$ is the set of restricted-focus events of the component.

Refer to [2] for definitions of GUI component, restricted focus events, and a detailed discussion on EFGs.

Definition 2: An *event sequence graph* (*ESG*) is a tuple *(N, A, S, F)* where

- N is a finite set of nodes representing the *events*.
- $A \subseteq N \times N$ is a finite set of directed arcs representing *follows* relation between events; that is, for two events x and y in the graph, x *follows* y if and only if *(y, x)* is an arc in the graph.
- $S \subseteq N$ is a distinguished nonempty set of events representing *start* or *initial* events.
- $F \subseteq N$ is a distinguished nonempty set of events representing *finish* or *final* events.

More details on ESGs can be found in [5].

Both EFGs and ESGs are directed graphs where nodes are interpreted as events and arcs form a "follows relation". Even though they visualize different aspects, from a strictly formal point of view, EFGs are equivalent to FSMs having no final states and ESGs are equivalent to FSM having at least one final state.

By loosening the constraints on start and finish events, one can convert an EFG or an ESG to an FSM by interpreting these graphs as Moore-like machines [15] and FSMs as Mealy-like machines [16], and vice versa. However, the case of empty string and the absence of final states should be handled carefully, and one may need to use indexing [5] to assign a unique label to each event in the graphs. We conclude the theoretical view at this point and return to testing.

In testing practice, additional constraints are imposed on EFGs and/or ESGs, caused by their frameworks, such as:

- Each event in an EFG is reachable from at least one start event.
- Each event in an ESG is reachable from at least one start event, and at least one finish event is reachable from each event.

B. Example Models

A GUI example, given in Figure 1, is modeled in Figure 2 and Figure 3 by corresponding EFG and ESG, respectively.

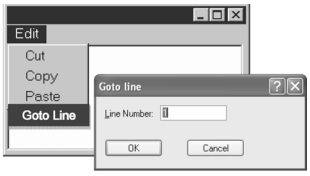


Figure 1. An example GUI (Taken from [17]).

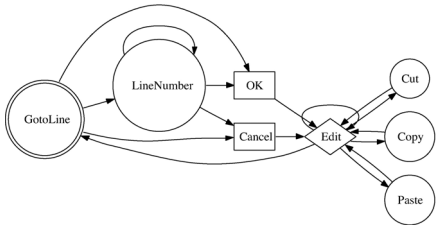


Figure 2. An example EFG of the GUI in Figure 1 (Taken from [17]).

In EFG (Figure 2), 'Edit' is a menu-open event (shown in diamond), 'Goto Line' is a restricted-focus event (shown in double circle), 'OK and 'Cancel' are termination events (shown in rectangles), and 'Cut', 'Copy' and 'Paste' events are system-interaction events (shown in circles). Furthermore, 'Edit' is the only start event.

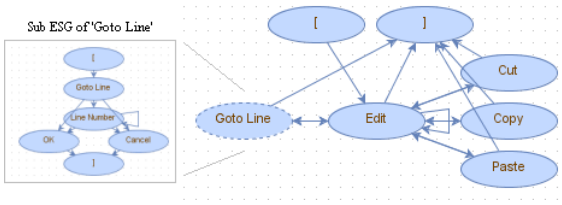


Figure 3. An example ESG of the GUI in Figure 1.

In ESG (Figure 3), all events shown in ellipses are simple events and the event shown in the dotted ellipse is a composite event having its own ESG. Note that start events follows '[', and ']' follows each finish event; that is, "[" and "]" are pseudo-events used to mark start and finish events, respectively.

C. GUI Testing Frameworks

Brief descriptions of EFG-based and ESG-based testing frameworks follow.

1) EFG-Based Framework

EFGs are actively used in GUITAR (**GUI** **T**esting Fr**a**mewo**r**k) [4]. The core of this framework is composed of the following elements:

- Tools to build EFG models
- Test case generation tool to cover all event sequences of specified length.

When GUITAR is used to generate test cases covering all the event sequences of length 2 from the EFG model given in Figure 2, test cases in Figure 4 are obtained.

In Figure 4, each test case ends with a different event sequence of length 2 which is intended to be covered.

Edit, Goto Line, Cancel, Edit
Edit, Copy, Edit
Edit, Cut, Edit
Edit, Copy
Edit, Cut
Edit, Edit
Edit, Goto Line
Edit, Paste
Edit, Goto Line, Cancel
Edit, Goto Line, Line Number
Edit, Goto Line, OK
Edit, Goto Line, Line Number, Cancel
Edit, Goto Line, Line Number, Line Number
Edit, Goto Line, Line Number, OK
Edit, Goto Line, OK, Edit
Edit, Paste, Edit

Figure 4. Test cases generated using GUTAR.

For our experiments, relevant tools of GUITAR are:

- GUIRipper rips the GUI and creates an XML file containing GUI structure (information of windows, widgets, their attributes, and user events).
- GUIStructure2EFGConvert generates an EFG from the XML file.
- TestCaseGenerator generates test cases from the EFG model. The most commonly used method for test generation is to cover all event sequences of specified length (SequenceLengthCoverage plugin).
- GUIReplayer automatically executes GUI test cases on the application.

The framework contains some other tools and also employs different graph-based models derived from EFGs, like *event interaction graphs (EIGs)* [17].

Features of EFG models used in this framework are:

- All the events in root window of GUI are considered as start events.
- Each event is reachable from at least one start event and no finish events are required.
- Edge types are used to identify hierarchies in the model.
- Events are divided into 5 types: Restricted-focus events, unrestricted-focus events, termination events, menu-open events, and system-interaction events [3].

2) ESG-Based Framework

ESGs are used in TSD (**T**est **S**uite **D**esigner) [7]. The core of ESG-based testing framework consists of the following elements:

- A tool to build ESG models
- A test case generation tool to cover all event sequences of specified length
- A tool for optimization of test suites.

As an example, when TSD is used to generate test cases covering all the event sequences of length 2 from the ESG model given in Figure 3, test cases in Figure 5 are obtained.

In Figure 5, although '[' and ']' are pseudo start and finish events in an ESG, the arcs connected to these events are also covered, and each test case is a path from a start event to a finish event.

[, Edit, Goto Line, OK, Edit, Goto Line, Cancel, Edit,]
[, Edit, Edit, Cut,]
[, Edit, Cut, Edit, Copy,]
[, Edit, Copy, Edit, Paste,]
[, Edit, Paste, Edit, Goto Line, Line Number, Line Number, OK,]
[, Edit, Goto Line, Line Number, Cancel,]

Figure 5. Test cases generated using TSD.

TSD is a single tool and it provides following facilities:

- A GUI to build hierarchical ESG models. Thus, each ESG node can be another ESG.
- Minimized test case generation covering all event sequences of specified length by finding a solution to Chinese Postman Problem [18] over the given ESG model.
- Hierarchy-based test case generation (optionally using some heuristics).
- Ability to associate events with code pieces and generate ready-to-run test scripts.

For ESG models used in this framework:

- There should be exactly one start event and one finish event. Thus, pseudo-nodes "[" and "]" are used to mark (multiple) start and finish events, respectively.
- Hierarchies are embedded into (composite) nodes as sub ESGs. Thus, there are simple events and composite events.
- There is a tool for test suite optimization.

III. SOFTWARE RELIABILITY ASSESSMENT TECHNIQUES

Since the early seventies of the last century, more than 80 different SRATs have been developed to attempt to estimate and predict software reliability; these techniques roughly correspond to software development lifecycle phases [19]: early prediction models, software reliability growth models, input-domain-based models, architecture-based models, hybrid black-box models, and hybrid white-box models. Particularly reliability growth models have been widely accepted and used in the practice and continue to be recommended by several industrial standards [12][13]. This paper also uses software reliability growth models in accordance with our following assumptions.

- Fault data and its trend are used to predict reliability.
- Faults are immediately corrected when they are detected during the course of software testing.
- The process of correcting a fault does not introduce a new fault (perfect debugging). Therefore, software reliability tends to increase.

SRATs can be classified in different ways:

- According to the shape of the expected value function of cumulative faults, i.e., concave and S-shaped.
- According to the cumulative number of faults observed in infinite time, i.e., finite and infinite.
- According to the distribution of the cumulative number of detected faults, i.e., Poisson and binomial.

As we did not have any prior knowledge as to the exact nature of the fault data, we proceeded by following these steps:

- Sort out the SRATs that (i) yield no initial values or (ii) do not really possess good-fit. These techniques are not feasible for our applications.
- The remaining set of SRATs still has to be carefully checked to determine whether or not they have slightly different properties than the fault data observed, which would then hinder good-fit.

Therefore, a very wide range of different techniques are considered and the SRATs that yield some initial values in our case study (and in at least one case) are taken. Table I summarizes basic features of the SRATs selected in this paper to be applied to our fault data (For more information, reader should refer to [12][13][20][21][22]). Accordingly, many other SRATs, such as Gompertz, Yamada Raleigh, Yamada Exponential, and Hossain-Dahiya G-O, have been discarded because of the reasons mentioned above.

Before selecting the proper SRATs, a time parameter should be determined. The amount of testing time needs to be measured using this parameter. There are several different ways to do this, such as using calendar time, execution time, number of test runs, number of test cases, or number of events executed, etc. In this paper, the number of events executed is used as the time parameter.

In order to analyze the fitness of an SRAT and select the proper SRATs, the steps below are followed.

First, fault data set is partitioned into two mutually exclusive subsets called the training set and testing set, in different proportions according to the Holdout method [23] to measure performance of the SRAT. A training set is used to adjust the parameters of the SRAT, and a test set is used to evaluate generalization and predictive capability of the SRAT.

TABLE I. SOFTWARE RELIABILITY ASSESSMENT TECHNIQUES

SRAT	**Type**	**Mean Value Equation** ($\mu(t)$)
Basic Musa (M)	-Concave -Poisson-Finite	$N(1-e^{-\lambda_0/N})$
Goel-Okumoto (G-O)	-Concave -Poisson-Finite	$a(1-e^{-bt})$
Musa-Okumoto (M-O)	-Concave -Poisson-Infinite	$\frac{1}{\theta}\ln(\lambda_0\theta t+1)$
Duane (D)	-Concave or S-Shaped (according parameter values) -Poisson-Infinite	at^b
Schneidewind (S)	-Concave -Poisson-Finite	$\frac{\alpha}{\beta}\left(1-e^{-\beta i}\right)$
Jelinski-Moranda (J-M)	-Concave -Binomial-Finite	$N(1-e^{-\phi t}))$
Littlewood-Verrall (L-V)	-Concave	$(1/b_1)\sqrt{b_0^2+2b_1 t\alpha}$ (for linear form)
Pareto (P)	-Concave -Poisson-Infinite	$a(1-(1+t/\beta))^{1-\beta}$
Weibull (W)	-Concave -Poisson-Finite	$a(1-e^{-bt^c})$
Logistic Growth (LG)	-S-Shaped -Poisson-Finite	$a/\left(1+ce^{-bt}\right)$
Delayed S-Shaped (DSS)	-S-Shaped -Poisson-Finite	$a(1-(1+bt)e^{-bt})$
Inflection S-Shaped (ISS)	-S-Shaped -Poisson-Finite	$N\left(\frac{1-e^{-\phi t}}{1+re^{-\phi t}}\right)$
Log Power (LP)	-Concave or S-Shaped (according to parameter values) -Poisson-Infinite	$a\ln^b(1+t)$

Second, initial parameter values are selected for the SRAT using the training data set following a similar initial value selection methodology as in [24]. Based on these values, maximum likelihood estimation (MLE) is used to estimate parameters by maximizing the likelihood that the observed data coming from a distribution with these parameters.

Third, goodness of fit (GOF) measures for both the training set and the test set are calculated. These measures are used to describe how well the SRAT fits a set of observations. GOF measures used in this study are mean square error (MSE) and mean absolute percentage error (MAPE) [25]. These measures are calculated as shown in Equation (1) and Equation (2), respectively.

$$MSE = \sum_{i=1}^{n}(y_i-\hat{y}_i)^2 \Big/ n \tag{1}$$

$$MAPE = \left(\sum_{i=1}^{n}\frac{|y_i-\hat{y}_i|}{y_i}\Big/ n\right)x100 \tag{2}$$

In Equation 1 and Equation 2, y_i is the cumulative number of faults obtained during test execution process, $\hat{y}_i$ is the predicted value obtained from the SRAT and n is the number of observations.

Finally, after calculating goodness of fit measures for each SRAT, one can decide which ones are better fitting or not. General rule of thumb is to select the SRATs that have relatively smaller goodness of fit (MSE and MAPE) values for both the training set and the test set.

IV. CASE STUDY

A non-trivial case study drawn from a large commercial web-based system is used to perform our experiments.

A. System under Consideration

SUC is a commercial web portal, ISELTA (Isik's System for Enterprise-Level Web Centric Tourist Applications [26]) for marketing tourist services implemented in PHP. ISELTA enables travel and tourist enterprises, for example hotel owners, to create their own individual search & service offering masks. These masks can be embedded in the existing homepage of the hotels as an interface between customers and system. Potential customers can then use those masks to select and book hotel rooms and benefit from other services.

The system also provides a search mask to book arrangements. This component of the system is used within this case study. The arrangements component of the system consists of two modules: additional services ("`additionals`") and special offers ("`specials`"). Figure 6 and Figure 7 show GUIs to set up additional and special services, respectively.

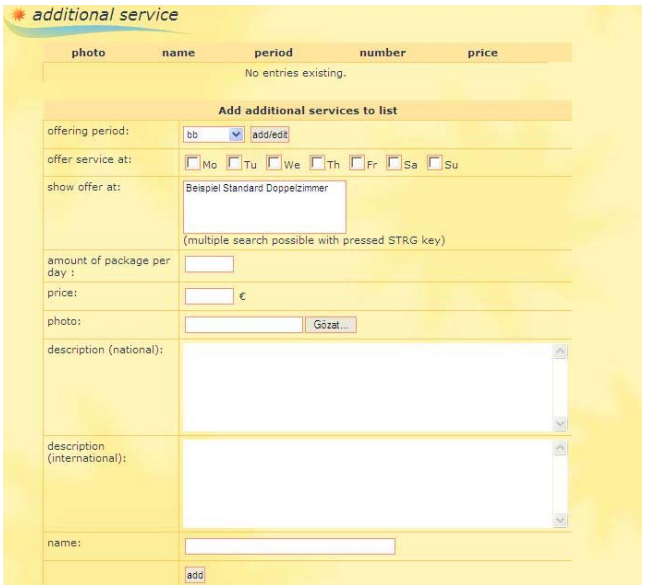


Figure 6. Website to set up `additionals`.

`Additionals` provide offering additional prices for facilities, such as extra beds or extra rooms in specified periods and service days. To add an additional service, at least the period, service days, room type, amount per day, price, and a unique name should be provided. Optionally, descriptions and photos can be included, and existing additional services can also be edited or deleted using this module.

Using `specials`, a hotel owner or a travel agent is able to add special prices to the facilities of the hotel marketed. To add a special, at least the room type, number of rooms of this type, basic price, and time period information should be provided, together with a unique name for the special. One can also upload a photo or write additional descriptions. Using this module, the existing specials can also be edited or deleted.

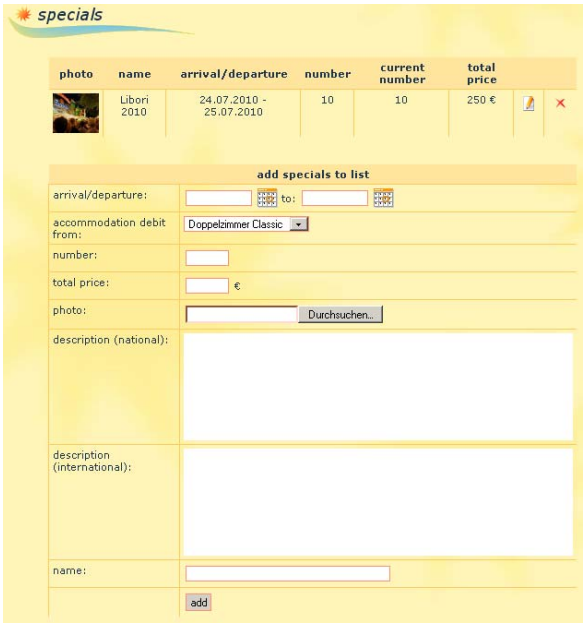


Figure 7. Website to set up `specials`.

B. GUI Models

The TSD tool is used to create ESG models. However, we experienced that creating EFG models was harder. Although, GUITAR provides easy mechanisms to create an EFG model from a running GUI instance, the following elements were not available to us [4].

- A tool to rip GUIs implemented in PHP and construct an EFG model (and a tool to execute generated test cases on PHP-based GUIs)
- A GUI-based tool to build EFG models by hand.

Hence, due to the fact that created ESG models are quite large (the fully resolved `additionals` ESG model contains 641 nodes and 1888 arcs, and the fully resolved `specials` ESG model contains 450 nodes and 1509 arcs [27]), ESG models are used to create EFG models by implementing a 'Save As EFG' feature in TSD tool.

1. "WidgetId" and "Action" tags of "Event" tag in EFG XML file are not used.
2. The "Type" tag of an "Event" tag is also not specified in EFG XML file.
3. To ensure successful test generation for each arc (e_i, e_j) in an ESG, the corresponding adjacency matrix entry in the EFG XML file is marked as "2."

The absence of data in Point 1 seems not critical for us, because they are related to the GUI structure XML file. In addition, Point 2 does not seem to prevent the generation of test cases because the test case generator tool of GUITAR works on the created EFG files. As for Point 3, although matrix entry "2" implies a hierarchy between events, it does not introduce any incorrect sequences. Also, by definition, in each event *e* in an ESG, *e* is marked as an initial event in the corresponding EFG if and only if its incoming degree is 0.

Due to the reasons discussed above, the EFG and the ESG models use the same set of events and "follows" relation. Hence, their graphical forms are quite similar. In order to save space, a simplified ESG model of `additionals` and a simplified EFG model of `specials` are given in Figure 8 and Figure 9, respectively.

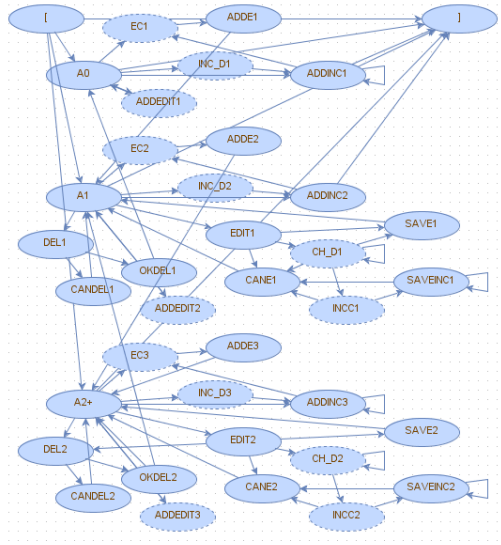


Figure 8. A simplified ESG model of `additionals`.

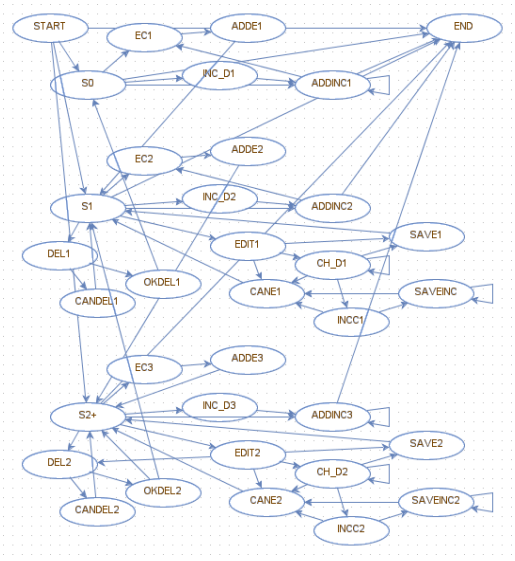


Figure 9. A simplified EFG model of `specials`.

See [27] for the complete sets of the ESG models (in graphical form) and the EFG models (in XML format).

C. *Seeding Faults*

In order to simulate a testing process and obtain fault data over time, a version of the SUC containing faults similar to real faults is created by seeding faults. As shown by Andrews et al. [28], faults generated by using mutation operators are quite similar to real-life faults. Furthermore, the mutation operators used are selected from the set of mutation operators that are shown to be in the set of "sufficient mutation operators for measuring test effectiveness" in the literature [28][29].

TABLE II. MUTATION OPERATORS AND NUMBER OF FAULTS

Mutation operator	**Additionals**		**Specials**	
	Maximum Number of Faults	**Number of Faults Generated**	**Maximum Number of Faults**	**Number of Faults Generated**
Delete Statement	441	44	232	23
Constant Replacement	151	15	80	8
Arithmetic Operator Replacement	9	1	17	2
Bitwise Operator Replacement	1	0	1	0
Logical Operator Replacement	14	1	11	1
Relational Operator Replacement	57	6	55	6
Increment/Decrement Mutation	10	1	7	1
Bitwise Negation	1	0	1	0
Logical Negation	14	2	11	1
Logical Context Negation	120	12	53	5
Sum	**818**	**82**	**468**	**47**

To add numerous faults for each module, the selected mutation operators are simultaneously applied to the PHP source code, resulting in a single high order mutant that is used as the faulty version of the SUC. To limit the number of injected faults, 1/10 sampling is performed; that is, for each mutation operator, the maximum number of faults is calculated and 10% of these faults are seeded into the system. For example, consider the "Constant Replacement" mutation operator for the `specials` module. It is possible to create a maximum of 80 mutants using this operator. Therefore, using 1/10 sampling, this operator is applied 8 times to inject different faults into `specials`.

Table II shows mutation operators used, maximum number of faults that can be generated using each mutation operator, and the number of faults that are actually generated using each mutation operator for the `additionals` and `specials` modules.

D. Test Generation and Execution

The frameworks outlined in Section II.C are used in the test generation process. For each EFG and ESG model, the corresponding framework is used to generate test cases.

In addition, the same structural coverage criterion is used during test generation. More specifically, for the given EFG or ESG model, test cases covering all the arcs (that is, event sequences of length 2 or event pairs) are generated. The respective test cases have different properties in the following ways.

- The test set generated using EFG-based framework covers each event pair in a different test case by computing a path from a start node to the event pair. Thus, the number of test cases is equal to the number of event pairs.
- The test set generated using ESG-based framework is minimized and each test case is a path from a start node to a finish node. Due to optimization, some test cases tend to be (much) longer than others. However, the total length of generated test cases is minimized.

In our experiments, the test execution is performed as follows. Each test case in the given test set is executed on the SUC until a failure is observed or until completion. Upon observing a failure, a single corresponding fault is found and corrected in the code, and the test case revealing this fault is executed again from the beginning. If the test case reveals no fault and runs until completion, it is never executed again. This process is repeated until no more faults are detected using the test set.

Note that the above test execution style satisfies the following two arguments, which are generally assumed by SRATs.

- The fault is repaired immediately when it is discovered.
- The fault repair introduces no new faults.

Table III summarizes the results of the test execution process by outlining the following data for models and modules used in this case study: the number of test cases generated (NTC), the number of events executed (NEE), the number of faults detected (NF), and the fault detection ratio (FDR) that represents the number of faults detected per executed event.

TABLE III. RESULTS OF TEST EXECUTION PROCESS

	EFG		ESGs	
	Additionals	**Specials**	**Additionals**	**Specials**
NTC	1889	1509	50	90
NF	76 (of 82)	41 (of 47)	80 (of 82)	43 (of 47)
NEE	10550	8099	27394	6750
FDR	0.0072	0.0051	0.0029	0.0064

One time unit is taken to be the time it takes to execute a single event and it is assumed that execution of each event takes the same amount of time. Since this leads to too many data points, data on the cumulative number of faults are grouped and plotted per 50 time units. Figure 10 and Figure 11 show the EFG-based and ESG-based fault data for `additionals` and `specials`, respectively.

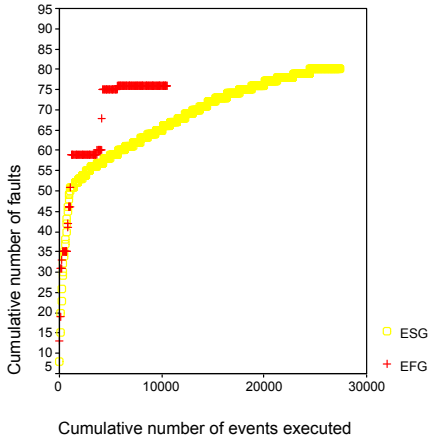


Figure 10. EFG-based and ESG-based fault data for `additionals`.

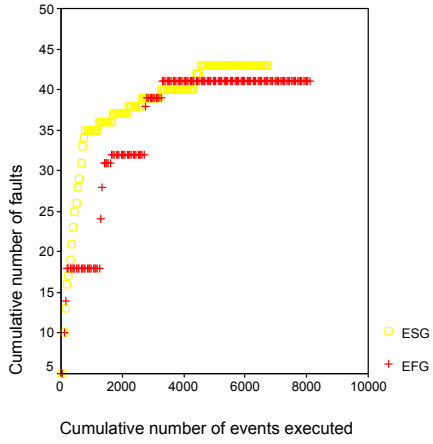
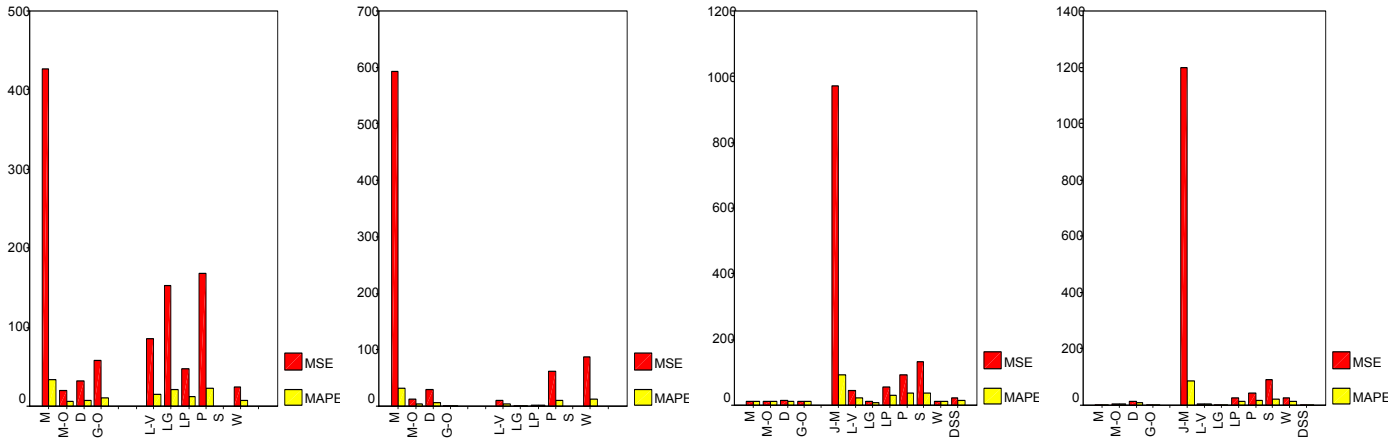
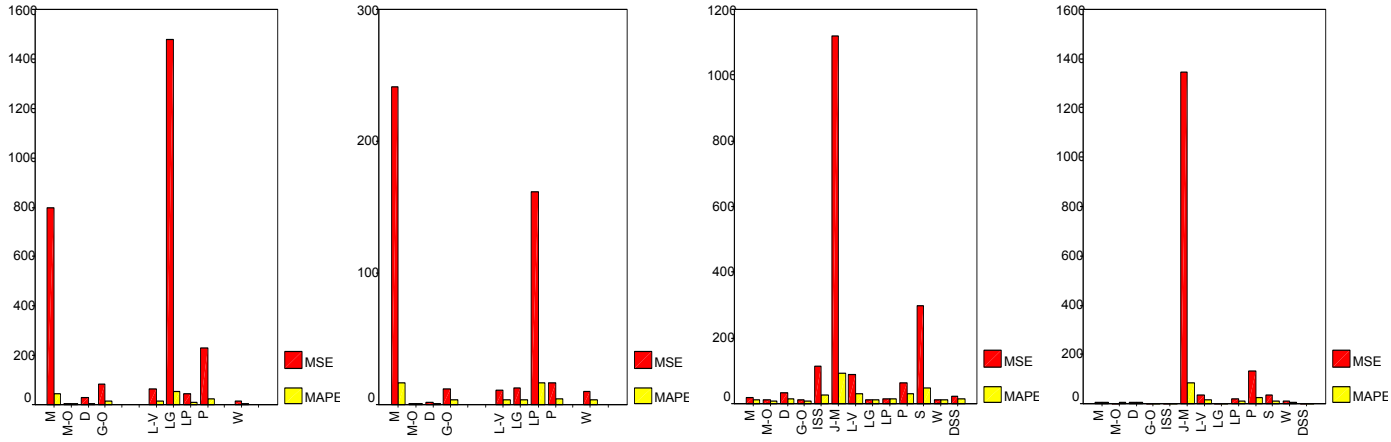


Figure 11. EFG-based and ESG-based fault data for `specials`.



(a) Additionals Training Set (b) Additionals Test Set (c) Specials Training Set (d) Specials Test Set

Figure 12. GOF Measures of SRATs for EFGs



(a) Additionals Training Set (b) Additionals Test Set (c) Specials Training Set (d) Specials Test Set

Figure 13. GOF Measures of SRATs for ESGs

As seen in Figure 10 and Figure 11, most of the faults are detected at the beginning of test execution process. In later stages, the fault detection rate decreases and the number of faults approximates a finite value. Therefore, data curves are concave.

E. Reliability Analysis

For reliability analysis, as mentioned in Section III, fault data sets are partitioned into two mutually exclusive subsets using the Holdout method, i.e., 2/3 of the data are designated as the training set and the remaining 1/3 are designated as the test set [23]. Later, the training set is used to determine the initial parameter values of the SRATs by the MLE method. Then, both the training set and the test set are used in calculating the GOF measures: MSE and MAPE. The results are given in Figures 12 and 13. See [30] for complete sets of MSE and MAPE goodness of fit measures and their numerical values.

Figure 12 and Figure 13 can be used to determine better-fitting SRATs. Note that SRATs having smaller GOF measures are relatively better fits (Thus, the higher the bar of an SRAT gets, the worse fit it becomes). Furthermore, no GOF measures were calculated for some SRATs because proper initial parameter values could not be determined even though predicted values along with a range of sensible values were tried.

F. Interpretation of Results

This section discusses the results outlined in Section IV.D and Section IV.E.

1) Test Execution Results

As can be seen from Table III, Figure 10, and Figure 11, using minimized test cases does not always increase the testing performance, and all the fault data obtained using EFG-based and ESG-based testing frameworks have a concave shape.

For both `additionals` and `specials`, both EFG-based and ESG-based test cases managed to detect a majority of the faults. However, the number of faults detected by ESG-based minimized test cases is greater in both cases.

For `additionals`, using ESG-based test cases, 1.60 times more events were executed to detect 4 more faults (2

faults could not be detected), leading to a relatively poor FDR of 2.9%. Whereas, EFG-based test cases yielded an FDR of 7.2% (while leaving 6 faults undetected).

For `specials`, ESG-based test cases yielded an FDR of 6.4% (4 faults went undetected). Using EFG-based test cases, 0.20 times more events were executed, 2 fewer faults were detected (6 faults were undetected), and an FDR of 5.1% is achieved.

2) Reliability Analysis Results

Results outlined in Figure 12 and Figure 13 suggest that better, good, and not-good fitting SRATs (from Table I) vary according to the frameworks used, and they are also dependent on the module being tested. For example, in the testing of `specials`, L-V SRAT is a good fit for data obtained using EFG-based framework, whereas it is a poor fit for data obtained using ESG-based framework. On the other hand, the same SRAT performs equally well for both frameworks in the testing of `additionals`.

Furthermore, although fault data is concave, S-shaped SRATs may provide a good fit, as DSS does while using EFG-based framework to test `specials`. Furthermore, some concave SRATS may turn out to be really poor for all fault data, as J-M does.

Our results suggest that SRATs that provide the best and acceptably good overall performance are in general concave and Poisson type. M-O and G-O SRATs are overall best fits for all obtained testing data, and W, LG, LP, L-V, and D SRATs show in general acceptably good performance. On the other hand, P, S and JM SRATs show the worst overall performance.

One can use GOF measures (Figure 12 and Figure 13) to rank the SRATs in two categories.

a) Predicting parameters (w.r.t. training sets)

For the results obtained using the EFG-based framework, LG, W, **G-O**, **M-O,** D, and LP provided relatively better performance; that is, they were observed to be better fitting SRATs. M and L-V turned out to be good fitting SRATs, and DSS, **P**, **S**, **JM,** and ISS are SRATs that are not really good fits.

W, **M-O,** LP, **G-O,** and D were determined to be better fitting SRATs in the results obtained using the ESG-based framework. Whereas, LG and L-V are good fitting SRATs, and M, DSS, **P**, **S**, **JM** and ISS are poor fits.

b) Predicting faults in later stages (w.r.t. test sets).

G-O, LG, L-V, **M-O,** and LP are better fits in the results obtained using the EFG-based framework. D, M, and DSS are good fitting SRATs, and W, **P, S, JM,** and ISS are poor performing SRATs.

G-O, **M-O**, LG, L-V, D, and W were determined to be better fitting SRATs in the results obtained using the ESG-based framework. Whereas, M, DSS, and ISS can be counted as good fits. SRATs that are not really good fits are LP, **P, S,** and **JM**.

When the reliability values for the best fitting M-O and G-O SRATs are calculated for the end stages of testing (Table IV), it can be seen that M-O yields relatively smaller reliability values because M-O is an infinite SRAT and assumes that cumulative number faults always increases, having no finite value. On the other hand, G-O is a finite SRAT; that is, it assumes that cumulative number of faults is bounded. Therefore, reliability values (for the end stages of testing) are high. This is consistent with the data given in Table III and indicates that almost all faults were revealed using both testing frameworks.

TABLE IV. RELIABILITY VALUES OF M-O AND G-O SRATS

RAT	Additionals		Specials	
	EFG	ESG	EFG	ESG
M-O	0.936	0.978	0.946	0.934
G-O	0.999	0.999	0.998	0.999

G. Threats to Validity

Only two modules of a web-based application were selected for the case study. In practice, using several different types of GUI-based applications, testing frameworks, and performing larger number of experiments would achieve a greater confidence in the obtained results.

Also, EFG models are derived from ESG models. Thus, preparing EFG models independently may change the results.

Finally, all the data obtained in this paper and all the results derived are valid with respect to the SUC outlined in Sections IV.A and IV.B, the testing frameworks described in Section II.C, the discussion in Section IV.C, and the test execution strategy outlined in Section IV.D. Changing them may yield different outcomes.

V. CONCLUSION AND FUTURE WORK

This paper analyzes the software reliability models, or software reliability assessment techniques (SRAT) as preferred here, in event-based GUI testing. The analysis is carried out by using different concave and S-shaped SRATs. Two event-based GUI testing frameworks, employing EFG and ESG models and different test generation algorithms, are considered in the analysis.

It is worth noting that our main intention is not to compare event flow graph (EFG) and event sequence graph (ESG) models (or related testing frameworks) against each other, but rather to examine the use of software reliability assessment techniques in event-based GUI testing using different testing frameworks.

To do this, a case study of two large modules of the commercial web portal ISELTA was carried out. To stimulate the testing process, several similar-to-real faults were seeded and faulty versions of the modules were obtained. Later, GUI testing frameworks were used to generate test cases to be executed on these faulty versions in order to observe fault data. When the observed data was analyzed, the following results were derived.

- The majority of faults were detected using both EFG-based and ESG-based testing frameworks. It was observed that minimized test cases (as produced in the ESG framework) do not always increase the overall

testing performance, even when relatively more faults were detected.

- The ranking of SRATs depends on the GUI testing framework used. Of course, the module being tested is also an influencing factor.
- All the fault data have concave shape, meaning that the fault detection rate is greater in relatively earlier stages of testing. However, it does not mean that concave SRATs always provide better performance than the S-shaped SRATs.
- Nevertheless, overall better fitting SRATS seem to be in general concave and Poisson type.

To sum up from the reliability analysis point of view, EFG-based and ESG-based testing frameworks differ slightly and in following aspects:

- The better fitting SRATs for the data obtained using EFG-based framework are Logistic Growth, Goel-Okumoto, Musa-Okumoto and Log Power reliability assessment techniques.
- The better fitting SRATs for the data obtained using ESG-based framework are Weibull, Musa-Okumoto, Goel-Okumoto and Duane reliability assessment techniques.
- For both GUI testing frameworks, Goel-Okumoto and Musa-Okumoto turn out to be the best overall fitting SRATs.

As already mentioned in Section IV.G, our results have some limitations; that is, they are not universally valid for GUI testing practice, but they provide an experimental insight. Therefore, to obtain more general results in the future, we plan to include other (including non-event-based) GUI testing frameworks, use different test generation algorithms, and increase the number of experiments by also considering different types of GUI-based applications.

ACKNOWLEDGMENT

The authors would like to thank Atif M. Memon and Bao N. Nguyen for providing us with GUITAR.dated.11.22.2010 and answering our questions about EFGs.

REFERENCES

[1] B. Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, 1990.

[2] A. M. Memon, M. L. Soffa, M. E. Pollack, "Coverage Criteria for GUI Testing," Proc. of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering ESEC/FSE-9, ACM, 2001, pp. 256-267.

[3] A.M. Memon, I. Banerjee, A. Nagarajan, "GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing," Proceedings of 10th Working Conference on Reverse Engineering (WCRE 2003), IEEE, Nov. 2003, pp. 260-269.

[4] GUITAR – a GUI Testing frAmewoRk (GUITAR.dated.11.22.2010), http://guitar.sourceforge.net [Dec. 22, 2010].

[5] F. Belli, "Finite-State Testing and Analysis of Graphical User Interfaces," Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE 2001), IEEE CS Press, Nov. 2001, pp. 34-43.

[6] F. Belli, C.J. Budnik, and L. White, "Event-based modeling, analysis and testing of user interactions: Approach and case study," *Journal of Software Testing, Verification and Reliability (STVR),* vol. 16(1), John Wiley & Sons, Ltd., 2006, pp. 3-32.

[7] TSD – Test Suite Designer, Angewandte Datentechnik, University of Paderborn.

[8] A.C.R. Paiva, N. Tillmann, J.C.P. Faria, and R.F.A.M. Vidal, "Modeling and Testing Hierarchical GUIs," Proceedings of 12th International Workshop on Abstract State Machines (ASM 2005), 8-11 Mar., 2005.

[9] R. K. Shehady and D.P. Siewiorek, "A Method to Automate User Interface Testing Using Variable Finite State Machines," Proc. 27th Int. Symp. Fault-Tolerant Com-puting FTCS, 1997, pp. 80-88.

[10] A. Salomaa, I. N. Sneddon, *Theory of Automata*, Pergamon Press, 1969.

[11] J. E. Hopcroft, R. Motwani, J. D. Ullman, *Introduction to Automata Theory, Languages and Computation* (2nd Edition), Addison-Wesley, 2000.

[12] "IEEE Recommended Practice on Software Reliability," IEEE STD 1633-2008, June 27, 2008, pp.c1-72.

[13] "Guidance on Software Aspects of Dependability (IEC 56/1349A/CD:2009)," DIN IEC 62628, March, 2010.

[14] Object Management Group, "Unified Modeling Language (UML)," http://www.omg.org/spec/UML/.

[15] E.F. Moore, "Gedanken Experiments on Sequential Machines," *Automata Studies, Ann. of Math. Studies No. 34*, Princeton U. Press, 1956, pp. 129-153.

[16] G.H. Mealy, "A Method for Synthesizing Sequential Circuits," *Bell System,* Tech. J 34, Sep. 1955, pp.1045-1079.

[17] Q. Xie, A. M. Memon, "Using a Pilot Study to Derive a GUI Model for Automated Testing," *ACM Trans. Softw. Eng. Methodol*., vol. 18, no. 2, Nov. 2008, pp. 1-35.

[18] J. Edmonds, E.L. Johnson, "Matching, Euler Tours, and the Chinese Postman," *Math. Programming*, vol. 5, 1973, pp. 88-124.

[19] S. H. Chang, S. S. Choi, J. K. Park, H.G. Kim, "Development of an advanced human-machine interface for next generation nuclear power plants," *Reliability Engineering and System Safety*, 64, 1999, pp. 109-126.

[20] M. R. Lyu, *Handbook of Software Reliability Engineering*, McGraw-Hill, 1996.

[21] A. Wood, "Software Reliability Growth Models," *Tandem Tech. Report 96-1*, Tandem Computers Inc., Corporate Information Center, Cupertino, Calif., 1996.

[22] M. Zhao, M. Xie, "On the log-power model and its applications," Proceedings of the International Symposium on Software Reliability Engineering, IEEE Computer Society, 1992, pp.14-22.

[23] R. Kohavi, "A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection," Proc. of the Int. Joint Conference on Artificial Intelligence, 1995, pp. 1137-1145.

[24] F. Belli, N. Güler, and M. Linschulte, "Are longer test sequences always better?- A Reliability Theoretical Analysis," Proceedings of the 4th International Conference on Secure Software Integration and Reliability Improvement (SSIRI 2010), IEEE, 2010, pp. 78-85.

[25] U. Raja, D. Hale, J. Hale, "Modeling software evolution defects: A time series approach," *Journal of software maintenance and evolution,* vol. 21, no. 1, 2009, pp 49-71.

[26] ISELTA, http://www.iselta.de/.

[27] Complete sets of ESG and EFG models for the case study, http://quebec.upb.de/testbeds2011_models.pdf.

[28] J.H. Andrews, L.C. Briand, Y. Labiche, "Is Mutation an Appropriate Tool for Testing Experiments?," Proc. IEEE Int'l Conf. Software Eng., 2005, pp. 402-411.

[29] A.S. Namin, J.H. Andrews, and D.J. Murdoch, "Sufficient Mutation Operator for Measuring Test Effectiveness," Proc. ICSE, 2008, pp. 351-360.

[30] Complete sets of MSE and MAPE goodness of fit measures for the case study, http://quebec.upb.de/testbeds2011_gof.pdf.