2015

Automated blackbox GUI specifications enhancement and test data generation

Mohammad Ali Darvish Darab
Iowa State University

Automated blackbox GUI specifications enhancement and test data generation

by

Ali Darvish

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:

Carl K. Chang, Major Professor

Morris Chang

Samik Basu

Simanta Mitra

Xiaoqiu Huang

Iowa State University

Ames, Iowa

2015

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my sincere gratitude to those who helped me during the course of my PhD studies. First and foremost, I would like to thank my major advisor Dr. Carl K. Chang for his guidance, patience and support throughout my research and the writing of this dissertation. His insights and words of encouragement have often inspired me and renewed my hopes for completing my graduate education.

I would also like to thank Dr. Simanta Mitra who not only provided insightful comments on my research work, but also taught me a lot about teaching and mentorship; I had the chance to assist him in a number of courses at ISU and I learned many things from him. I thank Dr. Samik Basu for agreeing to write a recommendation letter for me and always being supportive and understanding. His office door was always open to me and he provided me with many helpful hints and guidance about my career plans. I had a number of fruitful technical discussions with Dr. Morris Chang and I would like to thank him for his constructive comments and constant support. I also extend my gratitude to Dr. Xiaoqiu Huang, another invaluable member of my PhD committee.

My acknowledgement section would be incomplete without mentioning Dr. Myra Cohen of the University of Nebraska-Lincoln. She was the one who initially taught me the basics of GUI testing and search-based software engineering research during the time I spent at the ESQuaRed laboratory at UNL. This dissertation is largely based on that training. So, thank you very much Dr. Cohen.

Last but not least, I would like to thank my dad, my sister, and my mom whom I lost forever while I was thousands of miles away from home. Mom, you were, are and will remain the dearest in my heart. Dad, you are my hero. Words cannot express what you have done for me and the family; thanking you here is the least of things I can do.

There are many other people and organizations who have helped me over the years to get to where I am today. While their names do not appear here, I would still like to express my sincere gratitude to them.

ABSTRACT

Applications with a Graphical User Interface (GUI) front-end are ubiquitous nowadays. While automated model-based approaches have been shown to be effective in testing of such applications, most existing techniques produce many infeasible event sequences used as GUI test cases. This happens primarily because the behavioral specifications of the GUI under test are ignored. In this dissertation we present an automated framework that reveals an important set of state-based constraints among GUI events based on infeasible (i.e., unexecutable or partially executable) test cases of a GUI test suite. *GUIDiVa*, an iterative algorithm at the core of our framework, enumerates all possible constraint violations as potential reasons for test case failure, on the failed event of an infeasible test case. It then selects and adds the most promising constraints of each iteration to a final set based on the *Validity Weight* of constraints. The results of empirical studies on both seeded and nine non-trivial open-source study subjects show that our framework is capable of capturing important aspects of GUI behavior in the form of state-based event constraints, while considerably reducing the number of insfeasible test cases. The second part of this dissertation deals with the problem of automatic generation of relevant test data for parameterized GUI events (i.e., events associated with widgets that accept user inputs such as textboxes and textareas). Current techniques either manipulate the source code of the application under test (AUT) to generate the test data, or blindly use a set of random string values. We propose a novel way to generate the test data by exploiting the information provided in the GUI structure to extract a set of key identifiers for each parameterized GUI widget. These identifiers are used to compose appropriate online search phrases and collect relevant test data from the Internet. The results of an empirical study on five GUI-based applications show that the proposed approach is applicable and results in execution of some hard-to-cover branches in the subject programs. The proposed technique works from a black-box perspective and is entirely independent from GUI modeling and event

sequence generation, thus it does not require source code access and offers the possibility of being integrated with existing GUI testing frameworks.

CHAPTER 1. INTRODUCTION

Most modern software systems, including web and mobile applications, have a Graphical User Interface (GUI) front-end. Automated functional system testing [1] of these applications presents new challenges to software testing community. One of the primary challenges to deal with is the huge, undetermined and complex input space of such systems. Model-based approaches [2] have put forward a practical solution for automated GUI testing and have been shown more efficient and cost-effective compared to the traditional capture-then-replay techniques [3]. The primary idea [4] in model-based approaches is to create a model of the GUI under test and use it for GUI testing purposes.

A GUI test case is a sequence of *events* to be run on GUI *widgets*. This, in fact, corresponds to the way a human user interacts with a GUI by invoking sequences of events on the widgets. GUITAR [5] is a notable work in this area that facilitates the testing process in four steps: 1) capturing the entire GUI structure 2) constructing graph models from the GUI structure 3) test case generation from the constructed models, and 4) test case replaying (i.e., execution). In the GUITAR approach, test case generation is achieved by walking on Event Flow Graph models (EFGs) where vertices of the graph correspond to GUI events and edges correspond to flow among the events. These graphs provide lightweight approximations of the GUI under test, compared to bulkier models such as Finite State Machines (FSMs). However, a primary issue with these models is that they do not fully capture the dynamics and run-time characteristics of the GUI. A GUI is a stateful and context-sensitive system where changes can occur during the course of its execution depending on the user actions and inputs. But, these models are constructed based on limited runs of the GUI, they are unable to capture a comprehensive picture of the GUI input space. As a result, the search for executable paths within static models such as EFGs (i.e., test case generation) typically leads to deficient test generation.

Furthermore, not only do executable event sequences have to be generated, but also appropriate input test data has to be supplied for parameterized events (i.e., the events associated with widgets that accept input values such as textboxes and textareas). Failing to generate relevant test data also hinders exercising crucial parts of the business logic code, degrading the overall testing adequacy as well as the fault detection capability.

In this dissertation, we deal with two problems in GUI testing: 1) ruling out infeasible (i.e., unexecutable or partially executable) test cases in GUI test suites, and 2) generating relevant test data. A test case is infeasible if an event in the event sequence is not available at some point during execution based on the state of the GUI. The point and the event at which an infeasible test case fails are referred to as "failure point" (denoted by *fp*) and "failed event" (denoted by *fe*), respectively.

Infeasibility of a test case can be due to different reasons such as faults in the application under test (AUT) or violation of some constraint among GUI events (i.e., event constraints). In this work, we propose a new way of discovering event constraints. Examples of common event constraints are when a certain widget is always disabled making all its associated events inaccessible or when an event requires another event before it can be run (e.g., *Redo* operation typically requires an *Undo* operation before it). Knowing about these constraints not only enhances our understanding of how different parts of a GUI work, but also are helpful to exclude infeasible test cases during test case generation.

We devise a *"GUI Invariant Discovery and Validation"* framework that detects common types of state-based event constraints [6] which occur frequently in many GUI applications. The novelty of our approach lies in the fact that we accomplish this using an iterative algorithm, called *GUIDiVa*[1], assuming we are only provided with a combinatorially coverage-adequate test suite[2][7]. In each iteration, *GUIDiVa* finds the most promising constraints on failed events of the test suite and adds them to a final set of event constraints. For each infeasible test case, all possible constraint violations (from among the considered constraint types) on the failed event are enumerated as potential causes of failure. Next, each enumerated constraint receives

[1] GUI Invariant Discovery and Validation

[2] Although we use t-way covering test suites to discover the constraints more efficiently, *GUIDiVa* can take any GUI test suite as input consisted of feasible and infeasible test cases.

a *validity weight* representing its potential validity level. This number is calculated using two other numbers: 1) number of test cases that fail at an identical event and violate a certain event constraint, and 2) number of feasible test cases in the test suite that include the failed event and violate the same constraint. The infeasible test cases with event combinations that violate the final discovered constraints are excluded from the test suite and new test cases are generated and added to achieve feasible desired coverage. The new test suite is used in the next iteration of the algorithm. *GUIDiVa* stops when there are either no more infeasible/new test cases left in the test suite or the algorithm has iterated for a user-specified number of times.

We report on the effectiveness, accuracy, and efficiency of *GUIDiVa* framework by conducting a set of experiments on both seeded and non-trivial subject projects. The results show that our approach is capable of revealing all seeded constraints with no error and only misses a total of seven unseeded constraints in nine real-world study subjects. Furthermore, we are able to considerably reduce the number of insfeasible test cases when we take the discovered constraints into account for test generation.

To address the second problem (i.e., generating relevant test data for parameterized widgets) discussed above, we propose a novel blackbox approach that makes use of the information exposed to the user in the GUI to produce suitable data for testing. We extract a set of key identifiers for each parameterized event from relevant parts of the GUI structure and use them to find and collect concrete string values from the web. The identifiers extracted from the GUI structure provide us clues about the type of values a parameterized event expects as input. We use the extracted identifiers to find appropriate regular expressions as well as valid and invalid concrete values.

We conduct a preliminary experiment on five GUI-based applications and report on the effectiveness and efciency of our approach. The results show that our approach is capable of improving the code coverage compared to using random values, a commonly used approach in the GUI testing research community. Also, it is shown that the amount of time required to produce the test data is easily negligible compared to the test execution time. The main contributions of this dissertation can be listed as follows:

- *GUIDiVa*: We design and implement a GUI Invariant Discovery and Validation framework to automatically detect common types of state-based event constraints in order to enhance both the GUI specifications and the quality of generated test suites.

- Empirical evaluation of *GUIDiVa*: We evaluate the *GUIDiVa* framework by running a set of experiments on both seeded and non-trivial subjects and report the results.

- Automated blackbox test data generation for GUI testing: As opposed to mainstream approaches for test data generation, such as symbolic execution [8] and meta-heuristics [9] which manipulate the source code of the AUT, we propose to make use of the information provided in the GUI to generate relevant test data. We discuss a novel way of producing test data for parameterized GUI widgets based on extracting and refining keywords from GUI structure and using them in an online search to collect the required data from the Internet.

- Preliminary empirical study of the proposed test data generation approach: We carry out a preliminary empirical evaluation of the proposed test data generation approach in regard to source code coverage metrics by running an experiment on five small to medium sized GUI applications. .

CHAPTER 2. BACKGROUND

2.1 Event-Driven GUIs

A modern GUI system is an instance of event-driven software [10], where it has the ability to detect and react to *events*. An event is a change in the state of the system that deserves attention. It is a specific signal triggered by an external pulse [11]. In event-driven GUIs, this external pulse typically corresponds to user interactions such as mouse clicks or keyboard strokes. A GUI is a hierarchical, graphical front-end to a software system that accepts as input user-generated and system-generated events from a fixed set of events and produces deterministic graphical output [10].

GUIs are comprised of a set of visual objects called *widgets* (a.k.a controls). A widget is a building block for a GUI, and it corresponds in appearance to a visual object that can be manipulated by the user [12]. Windows, labels, menu bars, menu items, toolbars, buttons and textboxes are among common GUI widgets. Features of a widget can be adjusted through a set of *properties* associated with it. Each of these properties accepts a certain value from a predefined range of values at a specific state during GUI execution. For instance, width, height, color, text color and enable are all among properties of a button widget that can change throughout the GUI execution time. Furthermore, each widget has zero or multiple events associated with it. Window-close, window-minimize and maximize are examples of events associated with a window where button-click is the most common event of a button. Label has no events associated with it. Every widget has a boolean property called "enable". Only the events of an enabled widget (i.e., enable=true) can be triggered.

2.2 Model-based GUI Testing Using Graph Models

Different approaches to model-based GUI testing have been proposed over recent years [13] [14][15]. But, because of the focus of this work, we limit ourselves to GUI testing based on graph models using GUITAR [16], a framework consisted of four basic tools that facilitate automatic model-based GUI testing.

The primary step in model-based testing is to extract a model that approximates the system under test (SUT) [4]. From this extracted model, abstract test cases can be generated. GUITAR utilizes graph models as an intuitive way to model user-interaction with a GUI; GUI test cases are then generated from these graph models. A GUI test case is a sequence of events to be run on the widgets.

Take an example scenario in a graphical text editor application as shown in Figure 2.2. First, a user Types in *Hello World!* into the editor and highlights it, then opens the *Edit* menu from the menu bar, selects *Copy*, opens the *Edit* menu again, selects *Paste*, and finally closes the window. This scenario corresponds to the event sequence <*Type In "Hello World!"*, *Select "Hello World!"*, *Edit*, *Copy*, *Edit*, *Paste*, *Close*>, which can be run as a test case.

The first step in the GUITAR approach is to "rip" a GUI using the GUITAR GUI Ripper tool. Ripping is the reverse engineering process of discovering the entire GUI structure by opening and recording all the GUI windows, the widgets, and their associated properties and values at the time of execution [3]. From the output of the ripping phase, "GUITAR Model Constructor" builds an Event Flow Graph (EFG). In an EFG, each vertex corresponds to a GUI event while directed edges between the vertices determine the temporal flow among the events. Thus, an edge in an EFG between two events, say e_1 and e_2, implies that e_2 is executable immediately after e_1. Figure 2.4 shows the EFG model for the *Edit* menu of the text editor application.

Formally, an EFG is defined as a 4-tuple $< \mathbf{V}, \mathbf{E}, \mathbf{B}, \mathbf{I} >$ where:

1. $\mathbf{V}$ is a set of vertices representing all the events. Each $v \in \mathbf{V}$ represents an event.

2. $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$ is a set of directed edges between vertices. Event v_j **follows** v_i iff v_j may be performed immediately after v_i. An edge $(v_x, v_y) \in \mathbf{E}$ iff the event represented by v_y **follows** the event represented by v_x.

3. $\mathbf{B} \subseteq \mathbf{V}$ is a set of vertices representing events that are available to the user when the GUI is first invoked.

4. $\mathbf{I} \subseteq \mathbf{V}$ is the set of restricted-focus events. A restricted-focus event opens another window which monopolizes the user's focus [3]. (i.e. user cannot interact with any other window while the window is open)

If vertices that represent structural events, used solely to open and close menus and windows, are removed from the EFG, the resulting graph model is called Event Interaction Graph (EIG) [17] (i.e., a graph model that only includes events that interact with the underlying application's code). An EFG can be transformed into an EIG following a set of graph-rewriting rules presented in [18]. EIGs provide a more compact and efficient model and test cases generated from them are shown to be more effective in revealing system faults [18]. Figure 2.5a shows the EIG model for the *Edit* menu of the text editor application, where the vertex representing *Edit* is removed from the graph. Events excluded from the EIG-based test cases are later on added to make them executable [18].

From the constructed graph models, GUITAR Test Case Generator generates sequences of events as test cases. An event sequence of length l is defined as a vector of events (e_0, e_1, e_2, ..., e_{l-1}) where $l > 0$ and e_0, e_1, e_2, ..., e_{l-1} are all vertices in an EFG or EIG model. The number of events in a sequence *es*, determined by *l(es)*, is the event sequence length. For instance, *l*(<*File, New, File*>)=3. GUITAR test case generator can be asked to generate event sequences of different lengths (e.g., l=3, l=5, l=10, etc) as test cases for a GUI under test. It does this by applying graph traversal algorithms on the graph model [19]. For example, enumerating vertices and edges generate test cases of length-1 and length-2, respectively.

Finally, GUITAR Test Case Replayer tool runs the generated test cases on the GUI one by one. For each test case, the test case replayer launches the application, runs the event sequence on the GUI, records the results and closes the application.

2.2.1 Infeasible Test Cases

Many of test cases generated using the graph models can be infeasible. These models are built only based on a single run of AUT and do not capture all dynamics and run-time characteristics of a GUI. A test case is infeasible if one of the events in the event sequence fails during execution by the test case replayer such that no subsequent events can be run on the GUI. For example, $<$*Undo, Copy, Paste*$>$ event sequence in the text editor application shown in Figure 2.5 is an infeasible test case because *Undo* is disabled when the application is launched first; the *Undo* operation requires an undoable operation like *TypeInText* or *Cut* before it. For each infeasible test case, two values are recorded:

1. **Failure Point [*fp*]**: the index (zero-based) of the last event in an event sequence which has successfully been executed by the test case replayer. Failure point of event sequence *es* is determined by $fp(es)$. For example, $fp(<Undo, Copy, Paste>) = 0$ in Figure 2.5, because the sequence fails at the first event. A feasible event sequence has a failure point equal to its length. (i.e., $fp(es)=l(es)$)

2. **Failed Event [*fe*]**: the event in an event sequence that the test case replayer fails to run. Failed event of event sequence *es* is determined by $fe(es)$. For example, $fe(<Undo, Copy, Paste>) = Undo$ in Figure 2.5, because the sequence fails at the first event.

Huang et al. in [20] identify four types of state-based event constraints. Currently, our framework is limited to discovering constraints of these four types. These constraint classes are:

1. **Disabled:** It occurs when an event is always disabled. A menu item or widget exists for the event, but it will never be accessible.

2. **Requires:** It indicates that some event needs another event to be executed before it is enabled. An example is *Undo* operation in an editor application. One needs to carry out an undoable operation before being able to execute *Undo*.

3. **Non-Consecutive:** It means that a certain event has to be non-consecutive with a specific sequence of events. In other words, a sequence of events makes another event inaccessible. The disabled event is re-enabled if another event occurs between the disabling sequence and the disabled event. An example would be doing two *Save* operations in a row. After one does a *Save*, a change should occur before another *Save*, making $<Save, Save>$ an infeasible sequence.

4. **Exclusive:** It is similar to the last one, with the difference that once a particular event or sequence of events are run, it disables a specific set of events permanently.

There are several constraints among the events of *Edit* menu in Figure 2.5. 1) *Undo* and *Redo* cannot be the first event in an event sequence (i.e. they are disabled when the application is launched first). 2) *Redo* cannot be the second event in an event sequence. 3) *Undo* requires an undoable operation (e.g., *Paste*). 4) *Paste* and *Cut* cannot be followed by *Redo*. 5) *Redo* requires *Undo*. These constraints make abstract test cases <*Undo, Copy*>, <*Redo, Undo*>, <*Copy, Undo, Copy*>, <*Copy, Paste, Redo*>, and <*Copy, Cut, Paste, Cut, Redo*> infeasible, all generated using the EIG model.

2.3 Covering Arrays

The number of generated event sequences grows rapidly as the sequence length and number of events in a GUI increase. For example, a GUI with 10 events has 10^5 length-5 test cases alone. Needless to say, real-world GUIs include a much greater number of events. This brings about the need for sampling techniques. A well-known sampling technique called covering array (CA), mainly used for combinatorial software testing, has been successfully applied to GUI test case generation [21]. Moreover, previous research has shown that longer event sequences generated by CAs are capable of detecting more faults compared to shorter exhaustive sequences [22].

Table 2.1: Full and intermediate-level coverage CAs with and without constraints

row#	**CA(27; 3, 3, 3)**			**row#**	**CA(9; 3, 3, 2)**			**row#**	**CA(9; 3, 3, 2)**		
1.	e1	e1	e1	**1.**	e1	e1	e1	**1.**	e1	e2	e3
2	e1	e1	e2	**2.**	e1	e2	e3	**2.**	e1	e2	e1
3.	e1	e1	e3	**3.**	e1	e3	e2	**3.**	e2	e1	e1
4.	e1	e2	e1	**4.**	e2	e1	e2	**4.**	e2	e3	e2
5.	e1	e2	e2	**5.**	e2	e2	e1	**5.**	e2	e3	e1
6.	e1	e2	e3	**6.**	e2	e3	e3	**6.**	e2	e1	e2
7.	e1	e3	e1	**7.**	e3	e1	e3	**7.**	e2	e1	e3
...	...			**8.**	e3	e2	e2	**8.**	e2	e3	e3
27.	e3	e3	e3	**9.**	e3	e3	e1	-	-		

A CA, written as $CA(N; t, k, v)$), is a $N \times K$ array on v symbols with the property that every $N \times t$ sub-array contains all ordered subsets of size t of the v symbols at least once [23]. These parameters are defined as follows:

1. ***k:*** is the CA degree. It corresponds to the number of columns in the array.

2. ***v:*** is the number of possible values each element in the array can take. In the GUI testing context, v corresponds to the number of events.

3. ***t:*** is the CA or test strength. $t = 0$ means no coverage, while $t = k$ means full coverage. When any of t of the k columns are chosen, all v^t of the possible t-tuples must appear among the rows. In other words, any sub-set of t-columns of the array must contain all t-way combinations of the symbols.

We use CAs to generate so-called coverage-adequate GUI test suites; test suites comprising event sequences that satisfy a certain CA strength. For instance, if we were to generate event sequences of length-3 for a GUI with three events $e1, e2, e3$, we would get a total of 27 test sequences. However, for strength 2 (t=2), this number can be reduced to 9, where all *2*-way combinations in all locations are still covered at least once. These are shown as the first and second arrays in Table 2.1.

Furthermore, CA constraints provide a way to exclude invalid combinations from the resulting arrays [24]. We use ACTS CA generator [24] in this work from the National Institute of Standards and Technology (NIST). Constraints in ACTS are specified in the form of restricted

first-order logical formulas[1]. These formulas are taken into account during test generation such that the resulting testset would cover combinations that satisfy these constraints. We translate our event constraints discussed in section 2.2.1 into CA constraints and input them to ACTS. In our three-event GUI, suppose we had two constraints: 1) $<e2, e2>$ is an infeasible sequence (i.e., $e2$ and $e2$ cannot run consecutively), and 2) $e3$ requires $e2$. These constraints are translated into CA constraints using boolean and relational operators as follows:

1) !((C1=$e2$ && C2=$e2$) || (C2=$e2$ && C3=$e2$))

2) !(C1=$e3$) && (C2=$e3$ $\implies$ C1=$e2$) && (C3=$e3$ $\implies$ (C1=$e2$ || C2=$e2$))

where C1, C2, and C3 refer to the first, second and third columns (i.e., parameters) of the array, respectively. The first constraint expresses that either the first and second or the second and third columns cannot take the value $e2$ in one row. The second constraint says that if there is an $e3$ in a row, there must be an $e2$ before it. The resulting array that satisfies these two constraints is the right-most array in Table 2.1.

[1]Boolean, relational and arithmetic operations are supported, but not quantification.

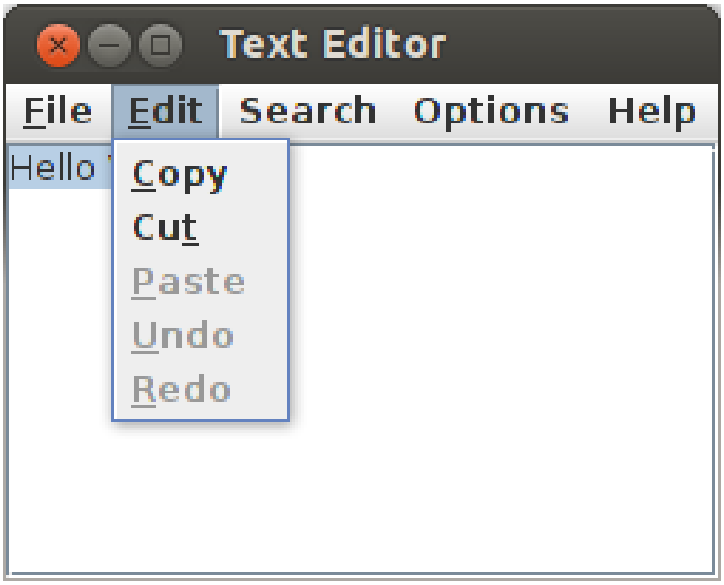


Figure 2.2 Edit menu snapshot

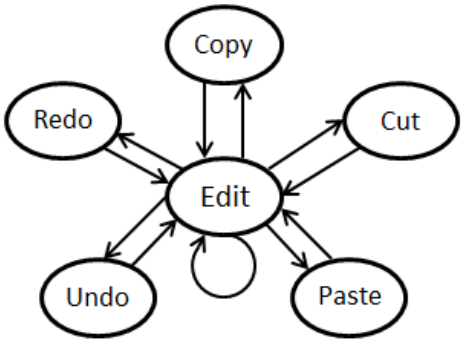
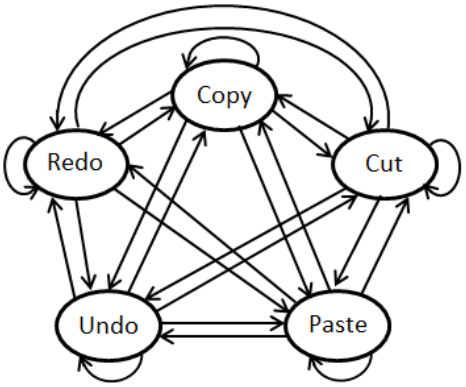


Figure 2.4 EFG model for the edit menu



(a) EIG model for the edit menu

Figure 2.5: Edit menu in a sample text editor application and corresponding graph models

CHAPTER 3. GUI INVARIANT DISCOVERY AND VALIDATION FRAMEWORK

In this chapter, we present our framework and *GUIDiVa* to discover a set of state-based event constraints.

3.1 Constraint Discovery and Validation Framework

Figure 3.1 shows the overall structure of our framework. GUI application, its ripped structure and constructed graph models, as well as appropriate values for CA parameters are inputs to the framework. Main Coordinator is responsible for orchestrating the entire process by interacting with different components of the framework. Based on supplied values of t (test strength), k (test case length), and v (number of events), the CA generator is asked to put together a coverage-adequate array. Test Case Assembler interfaces with the CA generator by passing the CA parameters and constraints into it. It then maps the generated arrays (i.e., abstract test sets) into concrete test suites by replacing array symbols with GUI events. Afterwards, the test suite is passed on to *GUIDiVa*, which runs the set of generated test cases on the GUI using the GUITAR Test Case Replayer to determine the feasibility or infeasibility of each test case. Then, in each iteration, *GUIDiVa* outputs a set of event constraints. These event constraints are translated into first-order logical formulas, readable by the ACTS tool, through Constraint Translator unit. The detailed procedure is described in the following section.

3.1.1 GUIDiVa

Algorithm 21 shows *GUIDiVa* pseudocode. A t-way covering GUI test suite (i.e., $\mathbf{TS_t}$) is input to the algorithm and the output is a final set of possible state-based event constraints

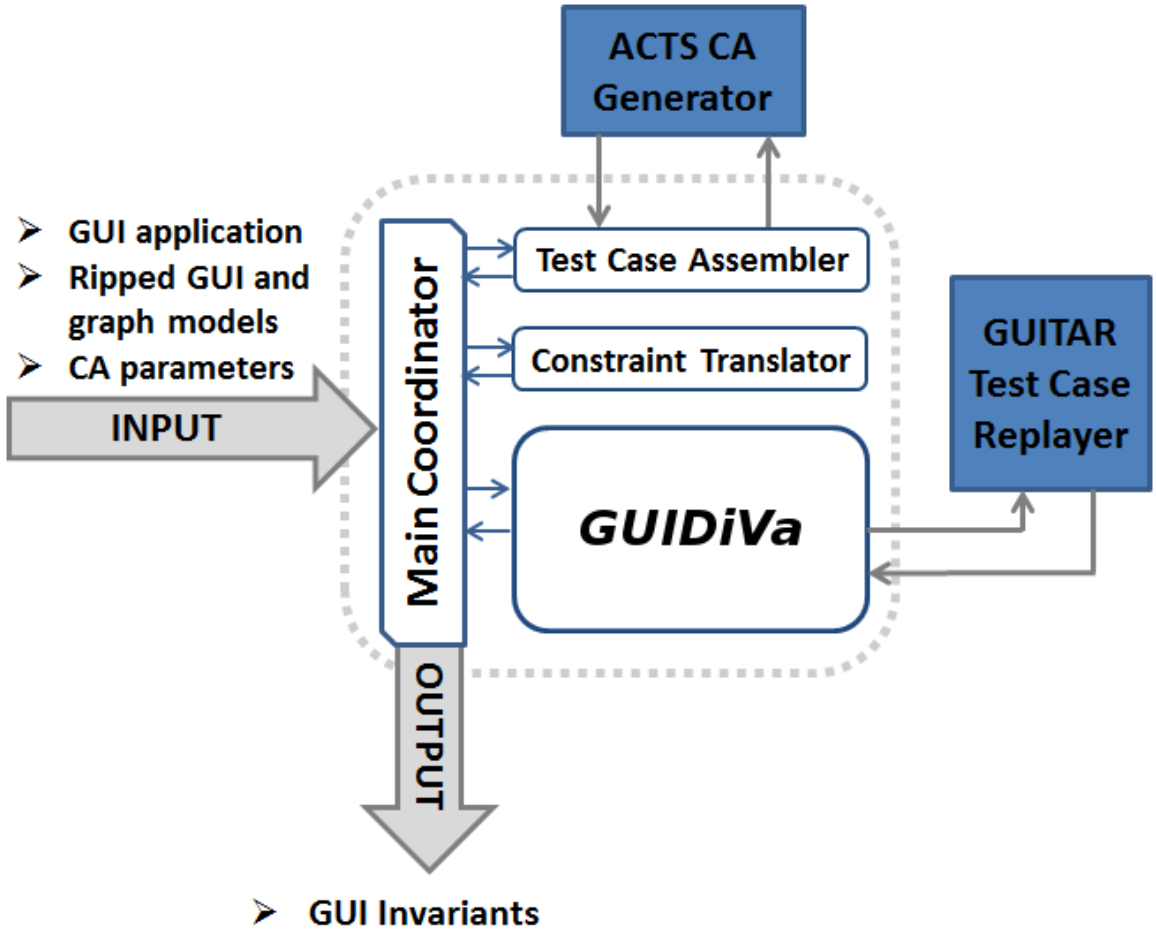


Figure 3.1: GUI Constraints Discovery and Validation Framework

(i.e., $\mathbf{\Pi}$) as well as an updated test suite (i.e., $\mathbf{TS'}$) that satisfies the constraints in $\mathbf{\Pi}$. Now, we describe the algorithm in details:

3.1.2 Initialization and Test Case Replaying

$\mathbf{TS'}$ is initialized with the input test suite $\mathbf{TS_t}$, and $\mathbf{\Pi}$ is initialized as an empty constraint set. All new test cases in $\mathbf{TS'}$ are replayed on the GUI to decide failure point and failed event of each test case. In the first iteration, this includes all test cases in $\mathbf{TS'}$ since they have not been replayed before. If there are no infeasible test cases in the test suite (i.e., $\forall\ tc\ \in \mathbf{TS'}$, $fp(tc)$=$l(tc)$), the algorithm returns. We use combinatorially coverage-adequate test suites to include all possible t-way event combinations in the input test suite.

Input : TestSuite TS_t (t-way covering test suite)
Output: Set Π (set of constraints)
TestSuite TS' (test suite that satisfies constraints in Π)

1 $TS' = TS_t$
2 **Set** $\Pi = \emptyset$
3 **while** *true* **do**
4 replay newly added test cases in TS' to decide their *fp* and *fe*
5 **if** *no infeasible or new test cases in* TS' **then**
6 **return**;
7 **end**
8 **repeat**
9 $\pi = \emptyset$
10 TC = next set of infeasible test cases in TS' with an identical *fe* e
11 **foreach** *test case* $tc \in TC$ **do**
12 $\pi = \pi \cup$ (**consViolations**(tc, *fp*(tc)) - Π)
13 **end**
14 **foreach** *constraint* $c \in \pi$ **do**
15 calculate $vw(c)$ based on values of $s(c)$ and $r(c)$
16 **end**
17 add constraint(s) in π with a positive maximum vw value to Π
18 **until** *all infeasible test cases in* TS' *are visited*;
19 remove conflicts from Π
20 remove test cases of TS' that violate constraints in Π and add new test cases to achieve feasible t-way coverage
21 **end**

Algorithm 1: *GUIDiVa* Algorithm

This maximizes the chances of revealing infeasible combinations, hence constraints among the events. However, any GUI test suite can be used as input.

3.1.3 Validity Weight Calculation

In each iteration of the *repeat* loop, a set of test cases with an identical failed event (i.e., all test cases that fail at a certain event) are selected from $\mathbf{TS'}$ and added to a test case set named **TC**. For each of the test cases in **TC**, all possible constraint violations on the failed event, which could potentially have caused the test case failure, are enumerated. Note that this is limited to the 2-way and 3-way constraint types discussed in section 2.2.1. **consViolations**() auxiliary function on line 12 enumerates all these possible constraint violations given a test case and its failure point. It does so by considering all the present as well as missing event combinations (from among a predefined events set) in a given event sequence up to its failure point. Then, all the enumerated constraints, excluding those that are already in the final constraints set $\mathbf{\Pi}$, are added to the constraints set $\boldsymbol{\pi}$. Note that we only need to enumerate the constraint violations on the failed event up to the test case failure point (i.e., *fp(tc)*). This is because events after the the failed event do not get to execute, thus they can have no influence on the failure of the test case. Afterwards, for each enumerated constraint c on the failed event e in the set $\boldsymbol{\pi}$, two numbers are calculated:

- **Support** [***s(c)***]: Number of infeasible test cases that fail at e and violate c. Thus, $\forall\ c\ \in \boldsymbol{\pi}, 1 \leq \mathbf{s}(c) \leq |\mathbf{TC}|$.
- **Reject** [***r(c)***]: Number of feasible test cases in $\mathbf{TS'}$ that include e and violate c. Thus, $\forall\ c\ \in \boldsymbol{\pi}$,
  $0 \leq \mathbf{r}(c) \leq |\text{feasible test cases in } \mathbf{TS'} \text{ that include } e|$.

The idea is, the higher the number of event sequences that fail at e and violate c, the more likely c is valid. On the other hand, the higher the number of times e successfully executes while violating c, the more likely c is invalid. Using values of s and r, a so-called *Validity Weight* (*vw*) value is calculated for each constraint in $\boldsymbol{\pi}$. *vw* value for a constraint provides a measure on

how valid the constraint is w.r.t. relevant feasible and infeasible test cases across a test suite. Validity weight for a constraint c is defined as:

$$vw(c) = \begin{cases} \frac{s(c)}{|TC|} & \text{if } \frac{r(c)}{s(c)} < \delta \\ \\ 0 & \text{if } \frac{r(c)}{s(c)} \geq \delta \end{cases}$$

where δ is an adjustable error threshold value. By definition, $vw(c) \in [0, 1]$, because $vw(c)$ is either 0 or $\frac{s(c)}{|TC|}$, where $\forall\ c$, $0 \leq s(c) \leq |\mathbf{TC}|$. The $s(c)$ numerator gives higher validity weights to constraints that have more infeasible test cases to support them and no (or very few) feasible test cases to reject them. Ideally, if $r(c) > 0$ for a constraint c, we would want to discard c (i.e., set $vw(c)$ to 0), because an event constraint is a GUI invariant that must always hold. However, in practice, due to timing and replayer-related issues [6], we might have few feasible test cases that violate a valid constraint. This is why we allow a small amount of error using δ.

One might think it is more efficient to consider all infeasible test cases with an identical failed event together for the purpose of generating candidate constraints. The reason *GUIDiVa* considers each infeasible test case individually is that it can be the case that there are more than one constraint on a specific event, which means a single constraint violation may not be sufficient to justify all the failures on an event in an iteration. For this reason, the $s(c)$ value for a constraint c is not decreased if the infeasibilty is not justifiable by c.

After calculating the validity weights of all constraints in $\boldsymbol{\pi}$, the one(s) with a positive maximum vw value is/are added to $\mathbf{\Pi}$.

We follow these rules to decide if a feasible test case violates a constraint:

1. $\mathbf{e_i}$ **is disabled:** the event sequence includes e_i.

2. $\mathbf{e_i}$ **requires** $\mathbf{e_j}$**:** the first occurrence of e_i in the event sequence is before the first occurrence of e_j, if any. Note that "requires" constraints on a certain event are disjunctive. For instance, in the editor example, the "*Undo* requires *TypeInText*, *Cut*, or *Paste*" constraint makes any sequence that has any of *TypeInText*, *Cut*, or *Paste* before *Undo* a feasible test

case. Thus, a feasible event sequence rejects either all the requires constraints in $\boldsymbol{\pi}$ or none.

3. **$\mathbf{e_i}$ cannot run after <es>:** the event sequence includes the sequence $<<es>, e_i>$.

3.1.4 Removing Conflicts

If there are conflicts between any two constraints in $\mathbf{\Pi}$, the one with a lower *vw* value is removed from $\mathbf{\Pi}$. If they have equal *vw* values, both are removed from $\mathbf{\Pi}$ and recorded in an external log file with their associated *vw* values. This is done to minimize the chances of excluding new feasible combinations; infeasible combinations may always be detected and removed in subsequent iterations of the algorithm. From our experience, conflicts occur rarely but they may happen due to coverage inadequacies of the input test suite, which adversely affect the *s(c)* and *r(c)* scores. Examples for conflicting constraints are "e_i requires e_j and e_j requires e_i", or "e_i requires e_j and e_j is always disabled".

3.1.5 Updating Test Suite

Once the new constraints with the maximum *vw* value on each failed event are added to $\mathbf{\Pi}$, all constraints in $\mathbf{\Pi}$ are translated into first-order logical formulas and are fed into the CA generator[1]. All constraints in $\mathbf{\Pi}$ are conjunctive except "requires" constraints on identical events which are disjunctive.

3.1.6 Stopping Criteria

There are three stopping conditions. The first condition is that there are no infeasible test cases left in $\mathbf{TS'}$. The second condition is that there are no new test cases to replay. This situation happens when no new constraints is added to $\mathbf{\Pi}$ in an iteration, leaving $\mathbf{TS'}$ unchanged. This means there are no more constraints that can effectively justify failure of infeasible test cases in the test suite. The third one is when the algorithm has iterated for a user-specified number of times.

[1]In our current implementation, we use the "extend" mode of ACTS. It automatically removes test cases that violate the given constraints from the existing test set and adds new ones to achieve the feasible desired test strength.

3.2 Complexity Analysis

Let n be the number of test cases in $\mathbf{TS_t}$, l the length of test cases, e_f the number of unique failed events, a the considered constraint arity[2] (for *non-consecutive* and *exclusive* constraints), and f the number of feasible test cases.

First, we replay all the new test cases to record *fp* and *fe* and decide if there are any infeasible test cases (*O(n)*). In practice, this is the most time-consuming step, since for each test case the replayer tool launches the application to run the event sequence on it. Afterwards, for each group of infeasible test cases (i.e., **TC**), we enumerate all possible constraint violations by making at most n calls to **consViolations**. In the worst case, **consViolations** performs *(2+l+(e-1)+2a)* operations (*O(e) + O(l)*). All constraint types except *requires* take constant time (i.e., 2 for *disabled* and a for either *non-consecutive* or *exclusive)*. Generating all possible *requires* constraints has an upper bound of *(l+e)*, since we go over the sequence up to its failure point *(fp* $\leq$ *l)* and generate at most $e-1$ *requires* constraints. Calculating *vw* for each generated constraint in $\boldsymbol{\pi}$ needs ($|\mathbf{TC}|$ * l) + (f * l) operations. To calculate *s(c)* for a constraint, we go over all infeasible test cases in a certain group ($|\mathbf{TC}|$) and check each for the possible violations (l). Similarly, we need to check the possible violations in each feasible test case (f * l) in order to calculate *r(c)*. Thus, the validity weight calculation step is done in *O(n*l)* since both $|\mathbf{TC}|$ and f are bounded by n. Removing possible conflicts is also upper bounded by $O(e_f{}^2)$. Finally, the constraint translation time complexity for each constraint is *O(l)*. Putting all these together, *GUIDiVa* is a polynomial algorithm linear to the number of test cases (*O(n)*), and quadratic ($O(e_f{}^2)$) to the number of unique failed events in the worst case.

3.3 Example

Take the example from section 2.3. Assume there are three events $e1$, $e2$, and $e3$ and two constraints: 1) $<e1, e1>$ is infeasible, and 2) $e1$ *requires* $e2$. Now, suppose we input a length-3 strength-2 test suite into the algorithm. It is the left array in Table 3.1. For the purposes of simplicity, we do not consider >2-way constraints in this example.

[2] As explained earlier, we consider constraints with arities of up to 3.

First, all 9 test cases are replayed on the GUI application using the GUITAR replayer tool to decide failure point and failed event of each test case. Test cases #4, #5, #6 and #8 are identified as the feasible ones. In the first iteration of the *repeat* loop, test cases #1, #2, #3, #7, and #9, which all fail at $e1$, are selected and added to **TC**. Thus, $|\mathbf{TC}| = 5$. Then, all possible constraint violations (i.e., possible reasons for failure) are enumerated for each test case in **TC**. For instance, for test case #1, constraints "$e1$ is disabled", "$e1$ requires $e2$" and "$e1$ requires $e3$" are generated, and for test case #9 constraints "$e1$ is disabled", "$e1$ requires $e2$" and "$<e3, e1>$ infeasible" are generated. Note that "$e1$ requires $e3$" is not generated for #9, since there is an $e3$ before $e2$. These give $s($"$e1$ requires $e2$"$) = 5$ because it is supported by all the (infeasible) test cases in **TC**, and $s($"$<e3, e1>$ infeasible"$) = 1$ because it is only supported by test case #9. On the other hand, $r($"$e1$ requires $e2$"$) = 0$ because there is no feasible test case that rejects this constraint, while $r($"$e1$ is disabled"$) = 2$ as the feasible test cases #4 and #5 include $e1$. At the end of the first *foreach* loop, we have:

$$\boldsymbol{\pi} = \{e1 \text{ is disabled}, e1 \text{ requires } e2, e1 \text{ requires } e3, <e3, e1> \text{ is infeasible}\}$$

Next, validity weight of each enumerated constraint in $\boldsymbol{\pi}$ is calculated:

$$vw(e1 \text{ requires } e2) = \frac{s(\text{“}e1 \textit{ requires } e2\text{”})}{|TC|} = \frac{5}{5} = 1$$

$$vw(e1 \text{ requires } e3) = \frac{s(\text{“}e1 \textit{ requires } e3\text{”})}{|TC|} = \frac{3}{5} = 0.60$$

$$vw(<e3, e1> \text{ infeasible}) = \frac{s(<e3, e1> \textit{ infeasible})}{|TC|} = \frac{1}{5} = 0.20$$

$$vw(e1 \text{ is disabled}) = 0, \text{ (i.e., } r(\text{e1 } \textit{is disabled}) = 2)$$

Thus, in the first iteration of the algorithm for the failed event $e1$, the constraint "$e1$ requires $e2$" with the highest vw value of 1 is added to $\mathbf{\Pi}$. Next, the CA generator is asked to remove test cases that violate this constraint and add new test cases to cover missing feasible 2-way combinations (the array shown on the right side of Table 3.1). The first four test cases are the

Table 3.1: Input test suites (failed event in bold font)

row#	**First Iter**			**row#**	**Second Iter**		
1.	**e1**	e1	e1	**1.**	e2	e1	e2
2.	**e1**	e2	e3	**2.**	e2	e2	e1
3.	**e1**	e3	e2	**3.**	e2	e3	e3
4.	e2	e1	e2	**4.**	e3	e2	e2
5.	e2	e2	e1	**5.**	e3	e3	e2
6.	e2	e3	e3	**6.**	e2	e1	**e1**
7.	e3	**e1**	e3	**7.**	e2	e3	e1
8.	e3	e2	e2	**8.**	e2	e1	e3
9.	e3	e3	**e1**	**9.**	e3	e2	e3
10.	**-**			**10.**	e3	e2	e1

feasible ones from the initial test suite, but test cases #5−#10 are new. After replaying the newly generated test cases, only test case #6 fails at event $e1$. The same procedure as iteration 1 is followed in the second iteration, resulting in:

$\mathbf{\Pi} = \{(e1 \text{ requires } e2,\ 1),\ (<e1, e1> \text{ infeasible},\ 1)\}$

Thus, test case #6 gets removed from the test suite. The algorithm terminates at the beginning of the third iteration after successfully finding both constraints. The algorithm stops because there are no more infeasible test cases left in the test suite.

CHAPTER 4. EXPERIMENTAL STUDIES WITH GUIDiVa

We conduct a set of experimental studies to determine the applicability of our approach. Our goal is to answer three research questions:

Research Question I (RQ I): Is our framework capable of discovering constraints with pre-formulated templates among events of a GUI effectively and efficiently?

Research Question II (RQ II): How useful are the set of discovered constraints for generating test suites with a smaller number of infeasible test cases?

Research Question III (RQ III): How does our framework perform compared to alternative approaches?

4.1 Experiment Setup and Assumptions

In order to answer these questions, we perform three experiments discussed in the subsequent sections. Our framework interfaces with ACTS-2.8 through ACTS API and executes the GUITAR-1.1.1 replayer tool externally. The models used are built using GUITAR-1.1.1 ripper and model constructors. An application is said to have passed a test case if it did not crash (unexpected termination or uncaught exceptions) [25]. For this purpose, we developed a crash monitor to record the result of each replayed test case. Also, the unique event IDs generated by the ripper are used instead of the event names to prevent any confusions. In all experiments, we have considered 2-way *requires* and both 2-way and 3-way *non-consecutive* and *exclusive* constraints. The δ (i.e., error threshold value) is set to 0.05 for all experiments. The experiments are done on a 32-bit machine equipped with an Intel 2.4GHz-4MB cache dual-core processor and 4GB of physical memory running Ubuntu 12.10 and Java 1.6.

Table 4.1: Experiment results on UNL.TOY.2010 artifacts

Artifact	CA params			|CA|	Constraint Description	#i	#t	#M	#I	T(sec)
	#e	k	t							
1. Dis	3	2	2	9	e1 is always disabled	1	9	0	0	64.42
2. Req	3	2	2	9	e2 requires e1	1	9	0	0	66.66
3. Con2	3	2	2	9	<e2,e3> infeasible	1	9	0	0	71.12
4. Exc2	3	2	2	9	e1 disables e2 permanently	1	9	0	0	56.64
5. Con3	4	3	3	64	<e1,e2> disables e3	1	64	0	0	443.36
6. Exc3	5	3	3	125	<e1,e2> disables e3 permanently	1	125	0	0	821.21
7. Com	5	3	3	125	Constraints 2, 3, and 5	1	125	0	0	903.31

4.2 Experiments

In this section, we conduct a set of experimental studies on both seeded and non-trivial applications in order to answer the three research questions.

4.2.1 UNL.TOY.2010

First, we experiment with a set of synthetic applications with seeded constraints. For this study, we use UNL.TOY.2010 from COMET group[1][26]. All the artifacts are non-faulty and the set of constraints are known beforehand. Table 4.1 shows the results. All artifacts include one constraint, except "Compound" which has three constraints of different types. |**CA**| is the size of initial test suite. For the first four artifacts length-2 strength-2 test suites are used, and length-3 strength-3 test suites are used for the last three ones. This is to make sure that all 3-way combinations are included in the initial test suite, otherwise the 3-way constraints might go undetected.

For this experiment, we do not manually specify any maximum value for the number of iterations allowed, so *GUIDiVa* stops when there are no more infeasible test cases left in the test suite. #**i** shows the number of iterations *GUIDiVa* takes to finish. #**t** shows the total number of test cases executed. #**M** shows the number of valid constraints that *GUIDiVa* cannot find (i.e., missed constraints), and #**I** is the number of invalid constraints found.

In all the artifacts, the framework discovers all the exact constraints without any misses or

[1]Community Event-based Testing (COMET) is a joint effort between E2 laboratory at UNL and EDSL group at UMD.

errors. The most time consumed is slightly above a total of 15 minutes for "Compound". In all cases, the bulk of time is spent by the replayer tool to run the test cases. The results of this experiment show that our approach is effective and accurate in finding seeded constraints.

4.2.2 Non-trivial Applications

As a second case study, we experiment with five real-world Java GUI applications that are categorized into nine study subjects. Unlike the seeded applications, the constraints among GUI events are unknown this time. The five selected applications are:

- **TerpWord** [27]: TerpWord is a word processing application and part of an office application developed at UMD.
- **TerpPresent** [27]: TerpPresent is a slide presentation application and part of an office application developed at UMD.
- **CrosswordSage** [28]: CrosswordSage is a crossword building and solving application.
- **Freemind** [29]: FreeMind is a mind-mapping application. A mind-mappinf application is useful to represent information in form of diagrams that shows relationships between concepts, ideas or other information.
- **Rachota** [30]: Rachota is a time-tracking tool with the ability to record tasks, times spent on those tasks and setting reminders for them.

Given the large number of events in these applications, the events are categorized into a set of groups based on their functionality [22]. For example, all events related to manipulating tables in TerpWord application fall within the *Table Operations* group. This would include events like *Insert Table*, *Insert Row*, *Delete Column*, etc. Since a group of events include all events that contribute to a specific functionality, a single event might be part of multiple event groups. For each of the five applications, we focused on specific functionalities to form event-groups that are likely to involve different types of constraints. Table 4.2 shows the specifications of these applications as well as the event groups studied for each of them.

Table 4.2: Non-trivial applications and event groups

#	GUI Application Name	LOC	#events	Group	
				Description	#events
1	**TerpWord 3.0**	22933	157	File Operations	8
2				Clipboard Operations	9
3				Table Operations	14
4	**TerpPresent 3.0**	45201	322	Content	14
5	**CrosswordSage 0.35**	3220	98	Preference Settings	9
6	**FreeMind 0.80**	24689	973	Clipboard Operations	10
7				Map Operations	11
8	**Rachota 2.4**	14330	168	System Settings	18
9				Task Manipulations	19

4.3 Research Questions

In this section, we answer the three research quesions we had devised given the experiments results.

4.3.1 RQ I

We use length-5 test cases (k=5) with strength-3 (t=3) for all the nine subjects. We find these values adequate enough to discover constraints effectively and efficiently[2]. Also, the maximum number of iterations allowed for *GUIDiVa* is set to 10 for all the subjects.

First, a skilled human oracle is asked to extract as many constraints as possible for each studied event-group. The human oracle only extracts the constraint types that are considered in our framework. Afterwards, for each subject, we run the framework five times to discover and output a set of constraints for that event-group and we record the intersection of all the five runs in $\mathbf{\Pi}$. Note that for each run, a different initial test suite generated by the CA generator is used, otherwise the results would be the same. In the end, the oracle is asked to verify the validity of the constraints in $\mathbf{\Pi}$ by sorting out valid constraints from the invalid ones for each event-group.

Table 4.3 shows the outcome of this study. An excerpt of constraints found by the framework for each subject is shown as well as the number of constraints that the human oracle finds, but are missed by the framework (#M). No constraints of type *exclusive* is found in any of the

[2]In our side experiments, we also examined k=10, k=20, and t=4 for several subjects, but no significant improvements were observed.

Table 4.3: (D)isabled, (R)equires, (N)on-Consecutive, (I)nvalid, #(M)issed. #M is the number of constraints found by human oracle that were not discovered by the framework

#	**Constraints Discovered By The Framework**	#M
1	**D**) *Save* **disabled** at index 0 **R**) *Save* **requires** *TypeInText* **N**) <*Close All, Close All*>, <*New, Save*>, <*Save, Save*> **infeasible** & **6** more **I**) *Save* **requires** *Open*, <*Close, Close, Close*>**infeasible**	0
2	**D**) *Undo* & *Redo* **disabled** at index 0, *Redo* **disabled** at index 1 **R**) *Redo* **requires** *Undo*, *Undo* **requires** *TypeInText* **N**) <*TypeInText, Redo*>, <*Cut*, *Redo*>, <*Paste, Redo*> **infeasible** **I**) <*Select All, Paste, Redo*> **infeasible**	1
3	**D**) *Undo* and *Redo* **disabled** at index 0, *Redo* **disabled** at index 1 **R**) *Redo* **requires** *Undo*, *Undo* **requires** *InsertTable* & **12** more **N**) <*InsertTable, Redo*>, <*InsertRow, Redo*> <*DeleteRow, AppendRow*> **infeasible** **14** more **I**) -	3
4	**D**) - **R**) - **N**) <*Open, Open*>, <*Open, Save*>, <*Save, Save*>, <*Redo, Save*> **infeasible** & **2** more **I**) -	0
5	**D**) *Proxy Address, Proxy Port, User Name & Password* **disabled** at index 0 **R**) *Proxy Address, Proxy Port, User Name, & Password* **requires** *Use Proxy Server* **N**) - **I**) <*Use Proxy Server, Use Proxy Server, Proxy Address*> **infeasible**	1
6	**D**) *Undo, Redo* **disabled** at index 0, *Redo* **disabled** at index 1 **R**) *Undo* **requires** either *Paste Format, Cut, Paste* or **2** more, *Paste Format* **requires** *Copy Format*, *Redo* **requires** *Undo* **N**) <*Paste, Redo*>, <*Cut, Redo*> **infeasible** **I**) -	1
7	**D**) *Undo, Redo* **disabled** at index 0, *Redo* **disabled** at index 1 **R**) *Undo* **requires** *Show Icon*, *Automatic Layout*, *Blinking Node* & **5** more, *Redo* **requires** *Undo* **N**) <*Automatic Layou, Redo*>, <*Blinking Node, Redo* > **infeasible** & **1** more **I**) -	0
8	**D**) *proxy server, port, inactivity time,* & *inactivity action* **disabled** at index 0 **R**) *proxy server* and *port* **requires** *Report weekly activity, inactivity time, inactivity action* **requires** *inactivity detection* **N**) - **I**) -	0
9	**D**) *Hour increase, Hour decrease, Min increase, Min decrease, Start automatically* **disabled** at index 0 **R**) *Hour increase, Hour decrease, Min increase, Min decrease, Start automatically* **requires** *Notification of task*, *Edit, Select, Remove* **requires** *Task selection,* and **3** more **N**) <*Select, Select*> and <*Remove, Remove*> **infeasible** **I**)<*Notification of task at, Notification of task at, Start Automatically*> **infeasible**	1

Table 4.4: Average results of five runs

#	#iter	#executed tc	Time(h)
1	6	1215	6.18
2	5.2	1598.2	7.61
3	6	5421	19.61
4	3.2	4682.5	16.71
5	3.8	1491	6.76
6	5.6	1962	10.45
7	5.8	3379.4	14.91
8	1	8299.2	36.42
9	8.8	13411.6	52.98

subjects. In four of the subjects, one or more invalid constraints are found. These happen in cases where there are only one or two infeasible test cases that support the constraints, whereas no feasible test cases exist to reject them. In fact, except for "*Save* **requires** *open*" in subject #1, all invalid constraints have only 1 infeasible test case to support them and no single feasible test case to reject them. Moreover, in four of the subjects no single constraint is missed. The other five subjects miss constraints, all due to the fact that there are no infeasible test cases at the intersection of the five initial test suites that fail due to violation of any of the missed constraints.

Table 4.4 shows the average results of the five runs. For each subject, the number of iterations, the number of executed test cases, and the amount of time consumed (in hours) is reported. Subject #9 takes the most amount of time and number of iterations since it includes 19 events with 13 constraints on 8 of them. Also, it is interesting to see that subject #8 finishes in only one iteration in all the five runs. This is because there are only 4 "requires" constraints on 4 events out of 18, and all of them can be discovered in a single iteration. Putting together the outcome of this and previous experiments, we can answer RQ I: our framework is effectively capable of discovering both seeded and unseeded pre-formulated state-based constraints.

4.3.2 RQ II

To answer RQ II, for subject studies #1−#9, we compared the number of feasible test cases against the total test suite size in both initial test suites and final test suites, where the final test suites satisfy all the discovered constraints in $\mathbf{\Pi}$. The initial test suites ($\mathbf{TS_t}$) consist of

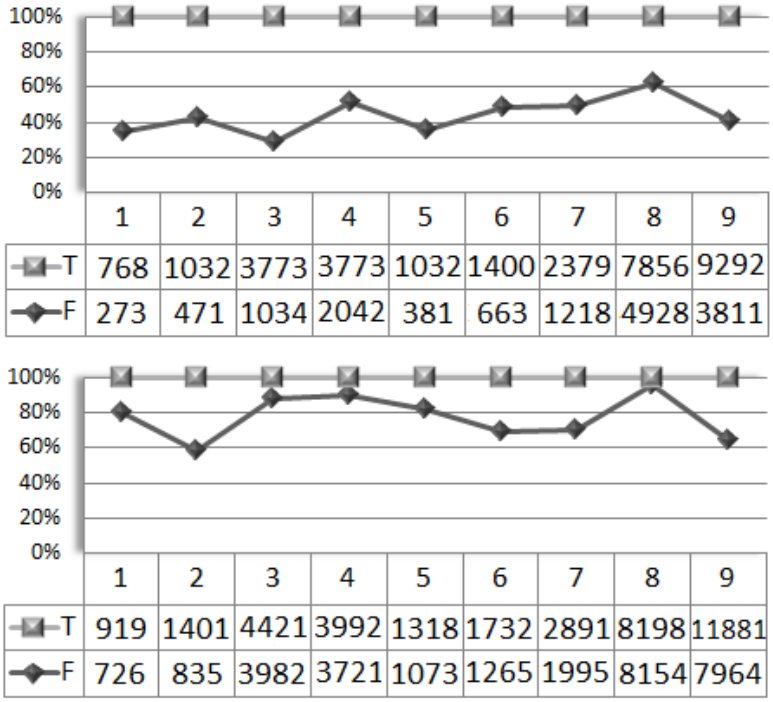


Figure 4.1: (T)otal: Total number of test cases in the test suite. (F)easible: Number of feasible test cases in the test suite

length-5 strength-3 test cases. The final test suites (**TS′**) also use the same parameters, except that event combinations that are not allowed by the discovered constraints are excluded, and new combinations are added to meet the feasible desired coverage criteria.

Figure 4.1 shows the average results of the five runs for this study. As can be seen, the number of feasible test cases in each test suite is increased considerably for all subjects when the discovered constraints are taken into account. In all cases, the number of feasible test cases is increased by at least more than 16%. In fact, subject #8 sees an improvement of about 37%, leaving only 44 infeasible test cases in a test suite of size 8198. With these results, we answer RQ II: Discovered event constraints using *GUIDiVa* can improve generated test suites by significantly reducing the number of infeasible test cases in them compared to when no event constraints are assumed.

Table 4.5: *GUIDiVa* v.s. AutoInSpec

#	***GUIDiVa***			***AutoInSpec***		
	#cons	#M	#I	#cons	#M	#I
0	9	0	0	8	1	NA
3	31	3	0	23	11	NA
4	6	0	0	4	3	NA
5	4	1	1	4	0	NA
6	9	1	0	7	2	NA
7	12	0	0	9	1	NA

4.3.3 RQ III

To answer this research question, we compare our results with those of AutoInSpec [6], which is based on [20]. AutoInSpec uses Prolog queries to infer GUI invariants based on missing coverage in a combinatorially-adequate feasible test suite.

Table 4.5 summarizes the results of this study[3]. #0 refers to UNL.TOY.2010. Also, since five of the non-trivial subjects (#3−#7) are the same in both studies and the constraints types considered are similar, it is straightforward to compare the results. However, it is not possible to do a comprehensive comparison because the oracles used in the two studies are not synchronized and no numbers are reported by AutoInSpec for invalid constraints found in each subject. Thus, we only report the number of missed (#M) and invalid (#I) constraints for each framework. Also, a technical comparison of the two approaches is given in section 6. Our results show that for all subjects, *GUIDiVa* is able to discover all constraints that AutoInSpec does. Moreover, it finds 1 seeded constraint and 15 extra valid constraints in subjects #0, #3, #4, #6 and #7 which AutoInSpec misses. All the 15 unseeded constraints found by *GUIDiVa* are valid but are missed by AutoInSpec. An interesting observation is that *GUIDiVa* finds several *3*-way *non-consecutive* constraints in subject #3 which not only are missed by AutoInSpec, but also by the human oracles in both studies. These constraints correctly suggest that no table operations can follow the <*Select All, Cut*> sequence. The outcome of this comparative study enhances our confidence that *GUIDiVa* can effectively discover state-based constraints and confirms that it outperforms, or at least performs as good as, alternative approaches.

[3] "disabled at index 0 or 1" is not counted as an individual constraint since it is a product of the "requires".

4.4 Threats to Validity

We identify a number of factors as potential threats to the validity of the results. To reduce the human oracle bias, we asked two persons who were not involved in this research work to extract the constraints and used the intersection of their outputs. They were both conversant with using the subject applications and were familiar with the constraint classes. Moreover, we did all the experiments on non-trivial subjects using k=5 t=3 test suites. We found these values adequate to show the effectiveness of our proposed approach. To better evaluate *GUIDiVa*, we incorporated the invalid constraints (as determined by the oracle) and their implications on the final test suites. The invalid constraints are either the product of the event-groupings or insufficiencies in the initial test suites, and may prevent inclusion of new feasible event combinations. We are working to eliminate the event-groupings and use larger values of t and k, if needed, in our further studies. Also, to make sure that all the results are consistent and reproducible, we ran each experiment five times and reported the average results.

We implemented the proposed framework and made use of ACTS and GUITAR tools. We made changes to the replayer tool to record failure point and failed event of each test case. We also developed a customized crash monitor to record the results of the replayed test cases. We tested our implementations by manual and run-time checks, but we cannot be certain that they are all fault-free. Finally, a threat to external validity is that we only experimented with limited parts of five Java applications. These applications have been used in other studies and we think they are reasonable representatives for mainstream GUIs.

CHAPTER 5. BLACKBOX TEST DATA GENERATION FOR GUI TESTING

In this chapter, we present our novel way of producing relevant test data for GUI testing [31].

5.1 Motivation

In recent years, there has been a noticeable amount of work on devising various automated techniques to 1) capture and model all events of the GUI under test, and 2) generate feasible event interactions to detect faults effectively and efficiently [32][33][34][35][36][37][22][15][38][39][14]. However, these approaches act naively or inefficiently on generating relevant input data for the parameterized events. The two prevalent strategies are 1) to use a constant set of random strings (e.g., {negative number, real number, zero, long random string, empty string, string with special characters}) for all parameterized events of a GUI, and 2) to manipulate the underlying code using symbolic execution (SE) [15] or search-based [34] techniques and generate data for structural coverage (e.g., branch coverage). The first approach is very unsophisticated due to a very limited set of values it offers and the latter ones are inefficient due to the complexity and scalability issues as well as their known limitations on handling complex constraints and operations of string data type. In fact, finding test data that achieves structural coverage under all circumstances is still an open research avenue [40].

Take the example shown in Figure 5.1, a user registration form which collects and registers user information. Eleven of the widgets in the GUI window are textboxes which accept input values. Such windows that include a number of parameterized widgets can be encountered in most GUI-based applications. Unfortunately, applying existing GUI testing techniques that mainly focus on generating feasible event combinations and neglect test data generation are unable to get important parts of the application's code to run. For instance, it seems sensible to test both valid and invalid values for each of the email address, zip code, date of birth,

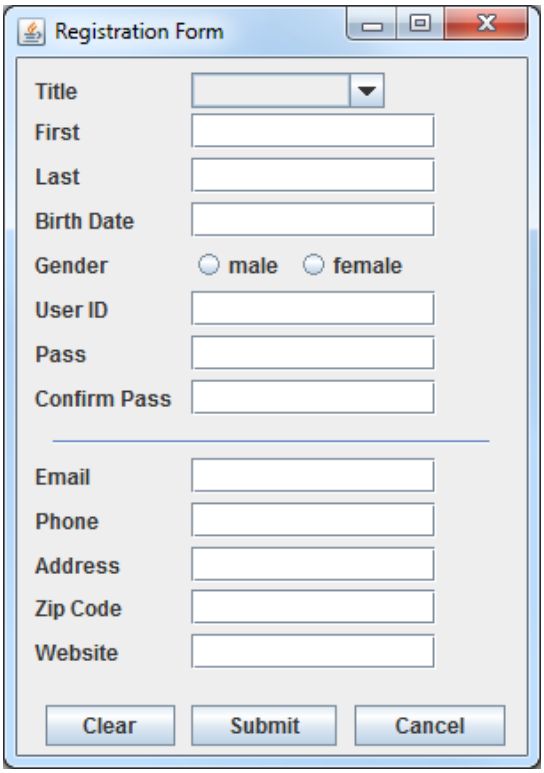


Figure 5.1: Registration form

User ID, phone number, and website fields in the example shown in Figure 5.1. There exist complex formatting restrictions as well as limitations on the range of possible values for each of these fields; sophisticated checks are carried out at the business logic layer (i.e., application's code), oftentimes with the help of third-party libraries [41]. To fully exercise these parts of the code, we need to be mindful of the input values we use when testing at the GUI/system level. Figure 5.2 shows the associated code that checks the validity of the email address field using Apache Common Validator library [41] when the `Submit` button is pressed. The naive approach of using a set of random strings and the techniques based on SE/search all fail to get the true branch of the `if` condition to execute.

The key idea here is to make use of the information exposed to the user, instead of using random values blindly or manipulating the source code, to produce suitable test data. We propose a novel way to generate the data for GUI testing by borrowing some of the ideas from [42]. We devise a novel black-box approach and implement it as a prototype which extracts

```
btnSubmit.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent arg0) {
        ...
        EmailValidator emailValidator = EmailValidator.getInstance();
        if (emailValidator.isValid(txtEmail.getText())) {
            //email check passed
        }
        else {
            //email check not passed
            JOptionPane.showMessageDialog(null, "Please provide a " +
            "valid email address.", "Error", JOptionPane.ERROR_MESSAGE);
        }
        ...
    }
});
```

Figure 5.2: Event Listener of the `Submit` Button

a set of key identifiers for each parameterized event from relevant parts of the GUI structure and uses those keywords to find and collect concrete values from the web. The identifiers extracted from the GUI structure provides us clues about the type of values a parameterized event expects as input. We use the extracted identifiers to find appropriate regular expressions as well as valid and invalid concrete values. For instance, the labels on the left side of the textboxes in Figure 5.1 instantly reveal what are expected inside the boxes. Valid and invalid values based on the kind of values expected inside each box seem to be very reasonable test data choices and very likely to get the corresponding parts of the source code to execute.

5.2 Blackbox Test Data Generation

Unlike the approach presented in [42] which manipulates the source code of the AUT to identify and extract the key identifiers, we make use of the GUI structure file produced by GUITAR ripper. The GUI structure file records all the GUI windows, widgets, their type, their properties (e.g., background-color, opaque), and values (e.g., blue, false). We utilize this rich file to locate all the parameterized widgets and extract a set of corresponding identifiers for each of them.

Our approach takes the the following steps: A) identify parameterized widgets of the GUI, B) extract key identifiers for each parameterized widget, C) conduct a set of processing steps on the extracted identifiers D) Using the refined identifiers, find valid and invalid test data.

Parts of our process replicates the approach presented in [42]. Here we only provide an overview of each step and highlight the modifications and adjustments we have made. We refer the interested readers to [42] for detailed explanations.

5.2.1 Identify parameterized widgets

Information about each widget, including its type, is recorded in the GUI structure file. In this work, we consider all subclasses of JTextComponent of Java Swing as text-dependent widgets except JPasswordField. This includes JTextField, JFormattedTextField, JTextArea, JEditorPane and JTextPane classes. We scan the GUI structure file and record the widget ID of all the text-dependent widgets with their associated set of property names and values.

5.2.2 Extract key identifiers

For each text-dependent widget, we extract information from the GUI that can potentially provide useful clues about the type of values the widget expects:

- **Name of the widget:** The value of this property is set by calling "setName" method of the widget class to give the widget an identification. The words used in this string can be highly related to the type of contents expected in the widget.

- **Tooltip:** The tooltip property gives hints about a widget [43]. This is the piece of text that gets shown when the user hovers the mouse cursor over a widget. The words used in a tooltip can be a useful source of information about the widget and the type of expected values.

- **Default text:** Sometimes, text-dependent widgets are initialized with a default piece of text. Usually, this string is an instance of a valid input for the box and can be used as a concrete test value.

- **Descriptive label:** It is a common practice in GUI design to place a label adjacent to textboxes/textareas to indicate what is expected inside them [43]. We call such labels "descriptive labels", because they make a description about the text-dependent widget in front of them. The caption value of these labels provide invaluable information about

the widgets and the type of data they expect. In the example shown in Figure 5.1, each textbox has a label to its left which provides a direct reference to the type of values expected inside the box.

All the above information are directly available from the GUI structure file except the descriptive label. Finding the descriptive label for a given text-dependent widget is an instance of a polynomial-time problem called nearest neighbor problem (NNP), also known as post office problem [44]. Coordinates of each widget in the GUI are recorded in the GUI structure file. To find the descriptive label, we need to locate all the labels that share the same top-level container[1]with the widget and are positioned in its proximity. For each such neighboring label, we calculate the Euclidean distance between the positions of the widget and the label using the Pythagorean metric (i.e., $\sqrt{(\Delta X)^2 + (\Delta Y)^2}$ where ΔX and ΔY refer to the differences between X and Y axes of the widget and those of the label, respectively. The closest label, if any, with a distance of less than an adjustable number is regarded as the "descriptive label" for that widget. Our current implementation uses a linear search to find the descriptive label which runs in O(n) time, where n is the number of widgets in the container; it is possible to improve this time complexity by utilizing other methods such as the ones using space partitioning [44].

5.2.3 Processing key identifiers

The extracted identifiers from previous section are first tokenized since they are sometimes formed from concatenation of a number of words. In addition to camel-case and underscore tokenization, we also do dash tokenization. Underscore and camel-case concatenations are popular naming conventions at the source code level, but dash concatenation is more used at the GUI level. Underscores and dashes are replaced with whitespaces. Also, a whitespace is inserted before each upper letter word. The complete tokenization step is only done for "name of the widget". For the values of the other properties, only dash tokenization is done if needed. The value of the default text property is an exception to the discussed procedure. Oftentimes,

[1]In Java Swing, every GUI widget has to be part of a containment hierarchy [45]. There are three container classes in Java Swing: JFrame, JDialog, and JApplet. In this work, we consider JFrame (the frame that contains the widgets, if any) and JDialog (the window that contains the widgets) classes.

the default text value is a concrete example for a valid test data and can be directly used in the testing process.

In the second phase of the processing step, Part-of-Speech tagging is done to remove articles (e.g., a, an, the, to) and non-nouns (e.g. verbs, adverbs). We do this step with the help of Stanford Log-linear Part-Of-Speech Tagger [46] tool using its default settings. Finally, non-words (e.g., numbers, special characters, abbreviations) are removed from the set of extracted key identifiers by JAZZY [47] tool and looking up SCOWL word list [48]. Detailed explanations about each of these steps can be found in [42].

5.2.4 Finding valid and invalid test data

Once the key identifiers are tokenized and refined, they are used to find appropriate regular expressions. We use the processed identifier names as search phrases to find relevant regular expressions by looking up RegExLib [49]. RegExLib is an online website indexing about 4000 regular expressions for different string types. The regular expressions are rated from 0 (poor) to 5 (excellent). We make HTTP requests to the website and download the results. We only consider regular expressions with a rating of 4 and above. All the regular expressions found are validated by making a call to Pattern.Compile() method of Java and the malformed ones are discarded. For each regular expression, RegExLib also provides two example sets called "Matches" and "Non-matches", corresponding to the valid test data and invalid test data, respectively.

As an example, for the identifier "Email", we search the RegExLib to find and download the top-rated regular expressions with their associated "Matches" and "Non-Matches" examples. These examples are used directly in the testing process as valid and invalid test data.

To generate an invalid value using the default text property, we replace at most three characters of the string value: 1) if there are any special characters, we replace one of them randomly with an alphabetical or numerical character, 2) if there exist any numbers in the string, we replace one of them randomly with a special or an alphabetical character, and 3) if there are any alphabetical characters, we replace one of them randomly with a number or a special character.

5.3 Evaluation

We experiment with five Java GUI-based applications to evaluate the effectiveness and efficiency of our approach. The subject applications used in this study and their specifications are listed in Table 5.1. '#w' denotes the number of windows/tabs in the GUI. '#e' and '#p' refer to the total number of events and the total number of text-dependent events in the GUI of the subject application, respectively. These numbers are derived from the EFG models and the GUI structure files of these applications which are constructed using the GUITAR ripper and model constructor tools. 'text types' refers to the types of strings expected inside the text-dependent widgets of each study subject. If there is no specific and well-defined type for a widget, it is listed with its more general type (e.g., alphabetical, alpha-numeric).

User Registration and MyFinance are developed by the authors of the paper. The other three subjects are open-source projects obtained from Github [50]. User Registration is the application used in the example shown in Figure 5.1. It collects users information and writes them into a database. MyFinance is an application that allows its users to organize and maintain their creditcard and banking information in a secure way. FTPClient is a simple FTP client. ImageGetter is used to download images in batches from given URLs, and Addressbook allows its users to sort, write, delete, and update their contacts information.

5.3.1 Experiment Setup and Assumptions

The experiments are carried out on a 32-bit machine equipped with an Intel 2.4GHz-4MB cache dual-core processor and 4GB of physical memory running Ubuntu 12.10 and Java 1.6. We use GUITAR-1.1.1 ripper, models constructor and replayer tools. The maximum distance to locate the descriptive labels is set to 50 pixels on a "1280 $\times$ 1024" screen resolution. We correctly locate the descriptive labels using this value 100% of the time for all the subjects. To measure the code coverage, Cobertura [51] tool is used. For each identifier, up to three regular expressions with their associated sets of "matches" and "non-matches" are downloaded from RegExLib. For each downloaded regular expression, one instance of valid and invalid strings are used in the testing.

Table 5.1: Subject applications

Subject	LOC	#w	#e	#p	text types
User Registration	462	1	17	9	alphabetical, email, phone, URL, US zip, address, date, integer
MyFinance	1425	3	36	12	creditcard no., US account no. CVC, US zip, US state, address, alphabetical, RTN, integer
FTPClient	3238	2	16	3	integer, IPv4, alpha-numeric
ImageGetter	5412	4	32	11	integer, URL, alpha-numeric, date
Addressbook	8210	1	51	10	alphabetical, email US state, US zip, phone

5.3.2 Preliminary Experiment and Results

First, for each study subject, the GUI structure file and the EFG model are constructed using GUITAR. Further, for each subject, length-2 test cases from the EFG model are generated. The choice of length-2 test cases is motivated by previous works in this area [21][35]. The GUI structure file and the EFG models are next taken by our prototype implementation as input to extract text-dependent widgets and find concrete test data values for them based on the presented approach. The values produced for each text-dependent event are stored into a file which is accessed and read by the GUITAR test case replayer tool when executing the test cases.

For the random approach, one random string is generated for each of the elements of the following set: {*negative real number, positive real number, negative integer, positive integer, zero, empty string, alpha-numeric string, string with special characters, long string with alphabetical characters*}. The choice of these elements is also motivated by previous works [21][33]. It may happen that no identifiers are extracted for a text-dependent widget because there is no relevant information available in the GUI (i.e., all the relevant property values are null or non-existent) or no regular expressions can be found for the identifiers. For such cases, we also

use the random values. This is why we call the first approach "Blackbox-Random". Note that the same initial test suites (i.e., EFG-based length-2 test cases) are used for both approaches, thus the only difference between the results in code coverage stems from the string values used with each test suite. Also, note that the textboxes that expect a password or a filename are excluded from this study. Such data have to be provided by the test engineer.

Table 5.2: Results of the Blackbox-Random approach

Subject	***Blackbox-Random***						
			Coverage		***Time***		
	#tc-EFG	#tc-EX	line%	branch%	t1	t2	v
User Registration	200	2201	**99**	**96**	17s	3.8h	10.2
MyFinance	424	3891	**69**	**63**	19s	5.5h	8.2
FTPClient	172	944	**51**	**43**	5s	2.3h	14
ImageGetter	428	2702	58	56	21s	4.7h	7
Addressbook	841	6952	**57**	**47**	24s	10.4h	11

Table 5.3: Results of the Random approach

Subject	***Random***					
			Coverage		***Time***	
	#tc-EFG	#tc-EX	line%	branch%	t2	v
User Registration	200	1970	**81**	**79**	3.2h	9
MyFinance	424	4012	**43**	**32**	6.2h	9
FTPClient	172	680	**45**	**39**	1.8h	9
ImageGetter	428	3320	59	58	5.4h	9
Addressbook	841	6456	**52**	**38**	10.1h	9

Tables 5.2 and 5.3 show the results of this experiment. '#tc-EFG' shows the size of the initial parameterized test suite which includes EFG-based length-2 test cases. '#tc-EX' refers to the number of test cases actually executed using the produced test data for the text-dependent events. 't1' reports the amount of time taken to produce the test data using our approach and 't2' is the amount of time the test case replayer takes to replay all the instantiated test cases on the GUI. As can be seen, the time spent to produce the test data is negligible compared to the amount of time it takes to execute the test suite. Note that 't1=0' for the random approach, since it takes less than a second to generate the random data. The line and branch coverages are reported for each approach as well.

'v' refers to the average number of concrete test data values produced by each approach for the text-dependent widgets of each subject (i.e., v = $\frac{the\ total\ number\ of\ values\ produced}{the\ number\ of\ text-dependent\ widgets}$). 'v' is 9 for all the subjects using the random approach, since a fixed set of test data values is used. However, it varies for the blackbox-random approach because depending on the number of identifiers and regular expressions for each text-dependent widget, a different number of values are produced.

In all the subjects except ImageGetter, both line and branch coverages achieved by the blackbox-random approach are higher than the random approach. In MyFinance, the branch coverage is improved by 31%, which is a considerable number. This happens because the regular expressions that formalize a credit-card number, CVC, RTN, account number, and zip code are successfully found in RegExLib. In ImageGetter, the random approach does slightly better. The reason is that a number of identifiers are extracted for three textboxes, but the regular expressions that are found using those identifiers do not match the format expected in the application. For instance, one of the widgets expects a date in dd/mm/yy format, but the regular expressions found only match mm/dd/yyyy, dd-mm-yy, and dd-mm-yy strings. Also, for four of the widgets, no identifiers are extracted, thus the same random values are used.

CHAPTER 6. RELATED WORK

This chapter provides an overview of previous work on the general areas of GUI-level system testing and automated test data generation. First, an overview of GUI testing tools are provided. Next, the related works on GUI testing are presented. Finally, a review of previous work on automated data generation for software testing is given.

6.1 GUI Testing Tools

A traditional approach to testing software systems at the GUI level (or GUI testing for short) is based on capture then replay methods. The idea is to capture GUI events during the capture phase and record them in a test script. Then, a test engineers can edit it and paly it back to test the system. There are many tools and libraries out there (offered as both free and commercial software) that provide various degrees of automation for GUI testing on different platforms (e.g., web, mobile, desktop etc) such as JFCUnit [52], Abbot [53] Jemmy [54], Pounder [55], Selenium [56], GUITAR [5], UISpec4J [57], Sahi [58], Squish [59], Marathon [60], Rational Functional Tester [61], GUIdancer [62], IcuTest [63], iMacros [64], and Watir [65] only to name a handful of them. JFCUnit, Abbot, Jemmy, UISpec4J and IcuTest are used for unit testing of GUIs. Pounder and Marathon support the capture-then-replay style of GUI testing. Selenium, Sahi, iMacros and Watir are browser automation tools used for web-based GUI testing. Some frameworks such as Squish and Rational Functional Tester support both functional and regression testing of GUIs on a variety of platforms. Although there are many GUI testing tools available, there exist no single one that has been adopted universally by the industry and the process is still carried out manually for the most part. Part of the reason is testing of GUI-based applications is complex and costly by nature, compared to testing of Command-Line-Interface-based systems. This is primarily due to the huge, undetermined, and

context-sensitive input space of non-trivial GUIs. There is no single testing tool avaible today that is effective and efficient to reveal faults in, or even applicable to, all different types of GUI applications and technologies. Another major challenge is automation of GUI oracles [25].

6.2 GUI-level System Testing

Over recent year, there has been a large body of research work to automate GUI testing [5][14][15][66][67][68][69][70][71][72][34][73][74][75][36][76][32]. Among the proposed approaches, model-based GUI testing [14][66][67][69][70][71][72][34][73][32] has gained the most attention. The primary idea is to create a model that approximate the entire or some part of the GUI which and use it for testing purposes. Finite States Machines [14][66][67][68], UML diagrams [69], task models [70], use cases [71], semantic models [72] and graphs [5] are some of the models that have been successfully used in the past. AutoBlackTest [36] and EXSYST [34] are two interesting recent works in the area that in addition to creating static models, manipulate the source code as well. AutoBlackTest [36] uses Q-Learning to learn how to interact with a given GUI application and produces test cases. A test case selector then filters redundant test cases following additional statement coverage prioritization approach. EXSYST uses a genetic algorithm to evolve GUI test suites for branch coverage while incrementally creating a GUI model based on state machines. Both of these approaches treat GUI test generation as an optimization problem and utilize FSM-based models. Bauersfeld et al. [75] utilize the ant colony algorithm to derive test cases based on optimization of Maximum Call Tree metric. Schulz's work [76] builds on the EXSYST approach by hybridizing static slice analysis and the dynamic evolutionary approach to generate minimized test suites for branch coverage; the result of the static slice analysis is used in the mutation operation to include events that are more likely to cover uncovered branches. Ganov et al. [15] identify and generate GUI event sequences as well as input data by extracting event listeners from the source code based on symbolic execution and branch coverage.

Perhaps graphs are the most well-known models used in automated GUI testing. The primary idea of the GUITAR approach [5] revolves around creating an event-flow graph (EFG) to approximate GUI events and the flow among them [77]. In an EFG, a node corresponds to

an event and an edge between two nodes corresponds to the temporal flow between them. Due to the fast explosion of the number of possible test cases generated using event-flow models and to improve the results, various techniques have been proposed to identify/refine test cases for higher fault detection capabilities [22][38][35][33]. Yuan et al. [38] proposed to create event semantic interaction (ESI) relationships based on run-time effects of event executions on GUI widgets properties. In [22], ideas of combinatorial testing to include unique event combinations in a test suite were successfully applied to GUI test case generation. In [35], a lightweight static analysis of the bytecode was carried out to determine data-dependency between event listeners and the results were used to create event dependency graph (EDG) models. The test cases generated using EDGs were shown to be more effective and efficient at revealing faults. In a recent work, Boa et al. [33] extended the GUI model construction phase to test execution; they iteratively incorporated the results of test executions into the initial model in order to develop a carry out a more complete and accurate testing of the GUI.

6.3 Avoiding/Repairing Infeasible GUI Tests

Many of the model-based approaches may generate infeasible (i.e., unexcecutable or partially executable) test cases, becuase they take a static/blackbox perspective on the GUI. In general, a test case is infeasibile if it does not comply with the specifications of the software under test. Infeasible test cases are regarded as invalid since they may trigger false failures [34].

Previous research works have been done on avoiding or repairing infeasible GUI test cases [78][79][80][20][81]. In [80], a prediction technique is proposed to classify test cases automatically into feasible and infeasible classed based on two supervised machine learning methods. Zhang et al. [81] use dynamic profiling, static analysis, and random testing to repair infeasible test workflows. Memon [78] and Grachanik et al. [79] propose automated approaches to repair test GUI suites and scripts in the context of regression testing. Huang et al. [20] propose a framework based on genetic algorithms to repair infeasible test cases [20]. They also identify several classes of state-based event constraints which we used in *GUIDiVa* framework. AutoInSpec [6] uses a by-product of the repair tool (i.e., missed coverage) to infer the constraints using Prolog queries. The work is different from ours in that AutoInSpec needs missing t-sets

as input, thus it is dependent on both combinatorially-adequate test suites and the repair process [20]. *GUIDiVa*, however, can improve any GUI test suite and discover the constraints at the same time. Moreover, AutoInSpec does not report on its performance to compute the missed coverage. This is why we unfortunately could not compare the efficiency of the two techniques.

6.4 Combinatorial GUI Testing

In this dissertation, we devised a black-box approach to detect a set of state-based GUI invariants to enhance both GUI specifications and test suites. For better efficiency, we utilized covering arrays (CAs) [24], commonly used mathematical objects in combinatorial software testing, to include as many unique combinations as possible in the initial test suite. CAs have been successfully applied to GUI testing [22] to show that longer combinatorially-adequate event sequences have higher fault detection rate compared to shorter exhaustive ones.Covering arrays have been successfully applied to GUI test generation [22]. In this work, we also made use of covering arrays [24] to generate coverage-adequate GUI test suites that satisfy the discovered event constraints [82]. We proposed an approach to discover possible state-based GUI event constraints from infeasible test cases of a test suite. Related works [83][84] in the combinatorial testing area identify and rank failure-inducing combinations (i.e., combinations that cause failures) from among the set of present combinations in the failed test cases. Finding event constraints in GUI test suites, however, not only depends on the present combinations, but also on the missing ones. Furthermore, examining the failure point and failed event of a test case is essential to generate and select the most promising constraints based on their validity weight values.

6.5 Automated Data Generation for Software Testing

Test data generation is a central activity to software testing, and the adequacy of the test data is oftentimes assessed by structural code coverage [85] and fault detection capability [86]. Current test data generation methods fall into two broad categories: static methods and dynamic methods. Static methods utilize static code analysis and may use Symbolic Execution

(SE) [87] to generate test data that achieves certain structural coverage (e.g., branch coverage). Dynamic methods, on the other hand, manipulate the information gained from program execution and can be based on random [88] or search-based [9] testing.

System testing of applications at the GUI level presents new challenges to software testing. One such challenge is that the input spaces of GUI-based applications are undetermined, two dimensional (i.e., event dimension and data dimension) and even infinite. Over recent years, many approaches have been proposed to 1) capture and model this huge space, and 2) generate relevant event sequences as test cases. The problem of generating appropriate test data, however, has gained less attention. SE-based [15] and search-based [34] techniques have been successfully applied to GUI testing. However, both of these approaches have limited abilities when handling the complex operations of the string data type. Moreover, it is not clear how the proposed techniques perform on the data generation since they tackle both problems (i.e., event sequence and data generation) simultaneously.

The technique presented in this dissertation gets the branches that involve complex string operations to execute by inferring input formats from identifier names (i.e., finding appropriate regular expressions from the Internet). The produced test data using this approach are more human-readable compared to the machine-generated values. Previous research has shown that the human-readability of the test data considerably reduces the human oracle cost [89]. This is specially important since devising full-fledged automated oracles for GUI testing is challenging and costly [90]. Additionally, the test data generation is done completely independent from GUI modeling and event sequence generation, thus the presented approach can be integrated with the existing GUI testing models (e.g. ESIGs [38], EDGs [35], Covering Arrays [22]). In fact, the idea of exploiting GUI information to generate the test data could be used in any GUI testing process and on any platform (e.g., web and mobile platforms). Extracting and using identifiers to search for relevant test data on the Internet was first proposed in [91] and further refined in [42]. However, unlike those works which extract the identifiers from the source code and use them to generate unit tests, we extract the key identifiers from the GUI and use them to find relevant test data for GUI testing.

CHAPTER 7. FUTURE WORK

In this chapter, some possible directions for future work are discussed. First, the potential improvements and further steps to event sequence generation and GUI specifications/test suite enhancement are discussed. Then, the ideas related to automated blackbox test data generation for GUI testing are given.

7.1 GUI Specifications and Test Suite Enhancement

GUIDiVa framework can be improved and extended along the following paths:

- **Capturing more of GUI behavior**: One way to acheive this is through identifying relevant event interaction patterns and integrating more complex constraint types into *GUIDiVa*: In this work, we only considered a limited number of constraint templates given by [20]. We need more sophisticated and fine-grained constraint classes to enable a more precise and thorough GUI modeling. As an example, in subjects II, IV, and V in section 4.2.2 we have a constraint "*Redo* requires *Undo*", which only partially expresses how *Redo* and *Undo* events are related. In fact, the precise behavior is: "Each *Redo* event requires one *Undo* and no undoable events should occur between the *Undo* and the *Redo*". Thus, <*TypeInText*, *Undo*, *Redo*, *Redo*> and <*TypeInText*, *Undo*, *TypeInText*, *Redo*> test cases fail at their last event, although both of them satisfy the "*Redo* requires *Undo*" constraint. Considering constraints that involve interaction of more than three events is also something that can be considered.

- **Compromised constraint v.s. fault**: Infeasibility of a test case can be due to a fault in AUT and not because of a compromised constraint. One way of improving the performance and precision of *GUIDiVa* is to adopt the framework to carefully differentiate

between the two, possibly by leveraging the proposed notion of validity weight and GUI fault models [92].

- **Case Studies**: Conducting larger case studies to further evaluate the performance of *GUIDiVa* can result in new findings as well as higher confidence in the proposed method. In particular, this can be done by either developing larger event-groups in each subject or ideally eliminating them altogether.

- **Code Coverage and/or Fault Detection Capability**: For more grounded results, it is possible to calculate the source code coverage when we take the constraints into account for test generation. Even though our framework works from blackbox perspective completely, reflecting on the coverage of the underlying application's source code provides a more relaiable measure compared to blackbox measures such as CA coverage. Even better is to compare the fault detection rate of test suites with and without considering constraints. However, for the latter one, we need to devise appropriate oracles [25].

7.2 Test Data Generation for GUI Testing

In regard to the presented test data generation technique, we identify three immediate routes for future work.

- **Data Intensive GUI Testing**: Constructing minimized parameterized test suites that are suitable for data-intensive GUI testing seems a very promising path for further exploration. In particular, we feel that studying the actual user proles and/or an analysis of data-dependency between the parameterized events and event listeners can be helpful.

- **Further Experimental Studies** Conducting larger case studies is desirable. Also, it would be interesting to integrate the proposed approach with the existing GUI testing frameworks that have limited test data generation abilities, and then measure the likely improvements of code coverage and fault detection rates.

- **Comparison with Alternative Approaches** A comparative study between the presented approach and Symbolic-Execution-based/Search-based techniques in the context of GUI testing is also an interesting direction for future work.

CHAPTER 8. CONCLUSION

Software testing is an integral part of software development process and accounts for more than half of software development costs [93]. Studies show that software bugs cost the U.S economy $59.5 billion annually and better software testing could save more than one third of this cost [94]. Most modern software products nowadays come with a graphical user interface (GUI) at their front-end and can run on a variety of platforms such as desktop, web, mobile etc. Industrial-scale automated testing of such systems, however, still remains an ad-hoc process. In recent years, model-based approaches have put forward a promising solution to automate testing of GUI applications, but there are still limitations and challenges that prevent a high-quality fully-automated testing cycle.

In this dissertation, we designed and implemented an automated framework to enhance GUI specifications and test suites utilizing model-based GUI testing and combinatorial testing. *GUIDiVa* at the core of our framework discovers and validates an important set of GUI invariants (as part of GUI specificaitons) in the form of state-based event constraints. $GUIDiVa$ is an iterative algorithm that discovers the most promising constraints based on the proposed notion of validity weight. The results of empirical studies on both seeded and five Java applications showed that *GUIDiVa* is effective and reasonably accurate in discovering GUI invariants. *GUIDiVa* was able to find all seeded constraints and only missed seven unseeded ones among about ninety constraints.

We also presented a novel blackbox approach to produce relevant test data for GUI testing. Our approach used the GUI information to extract key identifiers for the parameterized widgets (i.e., widgets that accept input values such as textboxes) and found appropriate valid and invalid test data using an online search. The preliminary experiments with five applications showed that the proposed technique is feasible and applicable.

BIBLIOGRAPHY

[1] B. Beizer, *Software System Testing and Quality Assurance.* Van Nostrand Reinhold Co., 1984.

[2] S. Arlt, C. Bertolini, S. Pahl, and M. Schäf, "Trends in model-based gui testing," *Journal of Advances in Computers*, vol. 86, pp. 183–222, 2012.

[3] A. M. Memon, "A comprehensive framework for testing graphical user interfaces," Ph.D. dissertation, University of Pittsburgh, 2001, aAI3026063.

[4] M. Utting and B. Legeard, *Practical Model-Based Testing - A Tools Approach.* Morgan Kaufmann, 2007.

[5] B. N. Nguyen, B. Robbins, I. Banerjee, X. Yuan, Q. Xie, and A. Nagarajan. (2001) Guitar website. [Online]. Available: http://guitar.sourceforge.net

[6] M. B. Cohen, S. Huang, and A. M. Memon, "Autoinspec: Using missing test coverage to improve specifications in guis," in *IEEE International Symposium on Software Reliability Engineering*, 2012, pp. 251–260.

[7] R. Kuhn, Y. Lei, and R. Kacker, "Practical combinatorial testing: Beyond pairwise," *IT Professional*, vol. 10, no. 3, pp. 19–23, 2008.

[8] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[9] P. McMinn, "Search-based software test data generation: A survey: Research articles," *Softw. Test. Verif. Reliab.*, vol. 14, no. 2, pp. 105–156, Jun. 2004.

[10] R. C. Bryce, S. Sampath, and A. M. Memon, "Developing a single model and test prioritization strategies for event-driven software," *IEEE Transactions on Software Engineering*, vol. 37, no. 1, pp. 48–64, 2011.

[11] L. P. F. M. Hugh Taylor, Angela Yochem, *Event-Driven Architecture: How SOA Enables the Real-Time Enterprise*. Addison-Wesley Professional, 2009.

[12] R. E. Eberts, *User interface design*. Prentice Hall, 1994.

[13] P. Li, T. Huynh, M. Reformat, and J. Miller, "A practical approach to testing gui systems," *Journal of Empirical Software Engineering*, vol. 12, no. 4, pp. 331–357, Aug. 2007.

[14] F. Belli, "Finite-state testing and analysis of graphical user interfaces," in *IEEE International Symposium on Software Reliability Engineering*. IEEE Computer Society, 2001, pp. 34–43.

[15] S. Ganov, C. Killmar, S. Khurshid, and D. E. Perry, "Event listener analysis and symbolic execution for testing gui applications," in *International Conference on Formal Engineering Methods*. Springer-Verlag, 2009, pp. 69–87.

[16] A. M. Memon and Q. Xie, "Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 884–896, 2005.

[17] Q. Xie and A. M. Memon, "Model-based testing of community-driven open-source gui applications," in *nternational Conference on Software Maintenance*. IEEE Computer Society, 2006, pp. 145–154.

[18] X. Qing and A. Memon, "Using a pilot study to derive a gui model for automated testing," *ACM Transactions on Software Engineering Methodology*, vol. 18, pp. 7:1–7:35, 2008.

[19] D. R. Hackner and A. M. Memon, "Test case generator for guitar," in *IEEE International Conference on Software Testing, Verification and Validation Companion*, 2008, pp. 959–960.

[20] S. Huang, M. B. Cohen, and A. M. Memon, "Repairing gui test suites using a genetic algorithm," in *IEEE International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, 2010, pp. 245–254.

[21] X. Yuan, M. Cohen, and A. Memon, "Covering array sampling of input event sequences for automated gui testing," in *IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2007, pp. 405–408.

[22] X. Yuan, M. B. Cohen, and A. M. Memon, "Gui interaction testing: Incorporating event context," *IEEE Transactions on Software Engineering*, vol. 37, pp. 559–574, 2011.

[23] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn, "Constructing test suites for interaction testing," in *IEEE International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, 2003, pp. 38–48.

[24] L. Yu, Y. Lei, R. Kacker, and R. Kuhn, "Acts: A combinatorial test generation tool," in *IEEE International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, 2013, pp. 370–375.

[25] Q. Xie and A. M. Memon, "Designing and comparing automated test oracles for gui-based software applications," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 1, Feb. 2007.

[26] M. Cohen and A. Memon, "Comet website." [Online]. Available: http://comet.unl.edu

[27] "Terpoffice website." [Online]. Available: http://www.cs.umd.edu/~atif/Benchmarks/UMD2006b.html

[28] "Crosswordsage website." [Online]. Available: http://crosswordsage.sourceforge.net

[29] "Freemind website." [Online]. Available: http://freemind.sourceforge.net

[30] "Rachota website." [Online]. Available: http://rachota.sourceforge.net

[31] A. Darvish and C. K. Chang, "Black-box test data generation for gui testing," in *QSIC '14*. IEEE Computer Society, 2014, pp. 133–138.

[32] A. Darvish and C. Chang, "Guidiva: Automated discovery and validation of state-based gui invariants," in *COMPSAC '14*, 2014, pp. 65–74.

[33] B. N. Nguyen and A. Memon, "An observe-model-exercise* paradigm to test event-driven systems with undetermined input spaces," *IEEE Trans. Softw. Eng.*, vol. 40, no. 3, pp. 216 – 234, 2014.

[34] F. Gross, G. Fraser, and A. Zeller, "Exsyst: Search-based gui testing (demo paper)," in *IEEE International Conference on Software Testing, Verification and Validation*. IEEE Press, 2012, pp. 1423 – 1426.

[35] C. B. M. S. I. B. A. M. Stephan Arlt, Andreas Podelski, "Grey-box gui testing: Efficient generation of event sequences," in *IEEE International Symposium on Software Reliability Engineering*. IEEE Computer Society, 2012, pp. 301–310.

[36] L. Mariani, M. Pezze, O. Riganelli, and M. Santoro, "Autoblacktest: Automatic black-box testing of interactive applications," in *ICST '12*, 2012, pp. 81–90.

[37] S. Bauersfeld, S. Wappler, and J. Wegener, "A metaheuristic approach to test sequence generation for applications with a gui," in *Symposium on Search-Based Software Engineering*. Springer-Verlag, 2011.

[38] X. Yuan and A. M. Memon, "Generating event sequence-based test cases using gui runtime state feedback," *IEEE Transaction on Software Engineering*, vol. 36, no. 1, pp. 81–95, Jan. 2010.

[39] A. Memon, M. E. Pollack, , and M. L. Soffa, ""hierarchical gui test case generation using automated planning"," *"IEEE Transaction on Software Engineering."*, vol. 27, no. 2, pp. 144–155, 2001.

[40] I. Ghosh, N. Shafiei, G. Li, and W.-F. Chiang, "Jst: An automatic test generation tool for industrial java applications with strings," in *ICSE '13*. IEEE Press, 2013, pp. 992–1001.

[41] Apache, "Apache common validator." [Online]. Available: http://commons.apache.org/proper/commons-validator/

[42] M. Shahbaz, P. McMinn, and M. Stevenson, "Automatic generation of valid and invalid test data for string validation routines using web searches and regular expressions," *Elsevier Journal of Science of Computer Programming*, 2014.

[43] W. O. Galitz, *The Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques*. Wiley, 2007.

[44] A. Andoni, "Nearest neighbor search: the old, the new, and the impossible," Ph.D. dissertation, Massachusett Institute of Technology, 2009.

[45] O. Corporation, "Java swing library documentation." [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/swing/

[46] K. Toutanova, "Stanford log-linear part-of-speech tagger." [Online]. Available: http://nlp.stanford.edu/software/tagger.shtml

[47] Jazzy, "Jazzy." [Online]. Available: http://sourceforge.net/projects/jazzy

[48] K. Atkinson, "Spell checking oriented word lists (scowl)." [Online]. Available: http://wordlist.sourceforge.net/

[49] RegExLib, "Regexlib library." [Online]. Available: http://regexlib.com/

[50] "Github website." [Online]. Available: https://github.com

[51] "Cobertura code coverage tool website." [Online]. Available: http://cobertura.github.io/cobertura/

[52] "Jfcunit website." [Online]. Available: http://jfcunit.sourceforge.net/

[53] "Abbot website." [Online]. Available: http://abbot.sourceforge.net/doc/overview.shtml

[54] "Jemmy website." [Online]. Available: https://jemmy.java.net/

[55] "Pounder website." [Online]. Available: http://pounder.sourceforge.net/

[56] "Selenium website." [Online]. Available: http://www.seleniumhq.org/

[57] "Uispec4j website." [Online]. Available: http://www.uispec4j.org/

[58] "Sahi website." [Online]. Available: http://www.sahipro.com

[59] "Squish website." [Online]. Available: http://www.froglogic.com/squish/gui-testing/

[60] "Marathon website." [Online]. Available: http://marathontesting.com/

[61] "Rational functional tester website." [Online]. Available: http://www-03.ibm.com/software/products/en/functional

[62] "Guidancer website." [Online]. Available: http://www.bredex.de/guidancer_jubula_en.html

[63] "Icutest website." [Online]. Available: http://www.nxs-7.com/icu/

[64] "imacros website." [Online]. Available: http://imacros.net/overview/web-testing

[65] "Watir website." [Online]. Available: http://www.watir.com

[66] R. K. Shehady and D. P. Siewiorek, "A method to automate user interface testing using variable finite state machines," in *FTCS '97*. IEEE Computer Society, 1997, pp. 80–.

[67] L. White and H. Almezen, "Generating test cases for gui responsibilities using complete interaction sequences," in *ISSRE '00*. IEEE Computer Society, 2000, pp. 110–.

[68] L. White, H. Almezen, and N. Alzeidi, "User-based testing of gui sequences and their interactions," in *ISSRE '01*. IEEE Computer Society, 2001, pp. 54–63.

[69] M. Vieira, J. Leduc, B. Hasling, R. Subramanyan, and J. Kazmeier, "Automation of gui testing using a model-driven approach," in *AST '06*. ACM, 2006, pp. 9–14.

[70] J. L. Silva, J. C. Campos, and A. C. R. Paiva, "Model-based user interface testing with spec explorer and concurtasktrees," *Electron. Notes Theor. Comput. Sci.*, vol. 208, pp. 77–93, Apr. 2008.

[71] P. L. M. Navarro, D. S. Ruiz, and G. M. Pérez, "A proposal for automatic testing of guis based on annotated use cases," *Adv. Soft. Eng.*, vol. 2010, pp. 5:1–5:13, Jan. 2010.

[72] N. R. Krishnaswami and N. Benton, "A semantic model for graphical user interfaces," *SIGPLAN Not.*, vol. 46, no. 9, pp. 45–57, Sep. 2011.

[73] E. S. G. Hassan Reza, Sandeep Endapally, "A model-based approach for testing gui using hierarchical predicate transition nets," in *International Conference on Information Technology : New Generations*. IEEE Computer Society, 2007, pp. 366–370.

[74] T.-H. Chang, T. Yeh, and R. C. Miller, "Gui testing using computer vision," in *CHI '10*. ACM, 2010, pp. 1535–1544.

[75] S. Bauersfeld, S. Wappler, and J. Wegener, "A metaheuristic approach to test sequence generation for applications with a gui," in *SSBSE '11*, 2011, pp. 173–187.

[76] T. Schulz, "Automatic Evolutionary GUI Testing Assisted by Static Analysis," Master's thesis, Hamburg University of Technology, Hamburg, Germany, 2013.

[77] A. M. Memon, "An event-flow model of gui-based applications for testing: Research articles," *Softw. Test. Verif. Reliab.*, vol. 17, no. 3, pp. 137–157, Sep. 2007.

[78] A. Memon, "Automatically repairing event sequence-based gui test suites for regression testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 2, pp. 4:1–4:36, Nov. 2008.

[79] M. Grechanik, Q. Xie, and C. Fu, "Maintaining and evolving gui-directed test scripts," in *ICSE '09*. IEEE Computer Society, 2009, pp. 408–418.

[80] R. Gove and J. Faytong, "Machine learning and event-based software testing: Classifiers for identifying infeasible gui event sequences." *Advances in Computers*, vol. 86, pp. 109–135, 2012.

[81] S. Zhang, H. Lü, and M. D. Ernst, "Automatically repairing broken workflows for evolving gui applications," in *ISSTA '13*. ACM, 2013, pp. 45–55.

[82] L. Yu, Y. Lei, M. Nourozborazjany, R. N. Kacker, and D. R. Kuhn, "An efficient algorithm for constraint handling in combinatorial test generation," in *IEEE International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, 2013, pp. 242–251.

[83] C. Nie and H. Leung, "The minimal failure-causing schema of combinatorial testing," *ACM Transactoins on Software Engineering Methodology*, vol. 20, no. 4, pp. 15:1–15:38, Sep. 2011.

[84] L. S. G. Ghandehari, Y. Lei, T. Xie, R. Kuhn, and R. Kacker, "Identifying failure-inducing combinations in a combinatorial test set," in *IEEE International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, 2012, pp. 370–379.

[85] N. Tracey, J. Clark, K. Mander, and J. McDermid, "An automated framework for structural test-data generation," in *ASE '98*. IEEE Computer Society, 1998, pp. 285–288.

[86] R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Trans. Softw. Eng.*, vol. 17, no. 9, pp. 900–910, Sep. 1991.

[87] J. C. King, "A new approach to program testing," *SIGPLAN Not.*, vol. 10, no. 6, pp. 228–233, Apr. 1975.

[88] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," *SIGPLAN Not.*, vol. 40, no. 6, pp. 213–223, Jun. 2005.

[89] S. Afshan, P. McMinn, and M. Stevenson, "Evolving readable string test inputs using a natural language model to reduce human oracle cost," in *ICST '13*. IEEE Computer Society, 2013, pp. 352–361.

[90] Q. Xie and A. M. Memon, "Designing and comparing automated test oracles for gui-based software applications," *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 1, p. 4, 2007.

[91] P. McMinn, M. Shahbaz, and M. Stevenson, "Search-based test input generation for string data types using the results of web queries," in *ICST '12*. IEEE Computer Society, 2012, pp. 141–150.

[92] J. Strecker and A. M. Memon, "Accounting for defect characteristics in evaluations of testing techniques," *ACM Trans. on Softw. Eng. and Method.*, vol. 21, no. 3, 2012.

[93] R. Patton, *Software Testing (2Nd Edition)*. Indianapolis, IN, USA: Sams, 2005.

[94] NIST(Report), "The economic impacts of inadequate infrastructure for software testing," 2002.