A FRAMEWORK FOR AUTOMATICALLY REPAIRING GUI TEST SUITES

by

Si Huang

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Myra B. Cohen

Lincoln, Nebraska

August, 2010

A FRAMEWORK FOR AUTOMATICALLY REPAIRING GUI TEST SUITES

Si Huang, M. S.

University of Nebraska, 2010

Adviser: Myra B. Cohen

Testing is an effective means for assuring the quality of software. In programs with Graphical User Interfaces (GUIs), event sequences serve as test cases for executing system tests. To aid in the test generation process, researchers have developed methods that automatically derive graph models from GUIs, which can then be traversed to create sequences for testing. Recent advances using these graph models incorporate combinatorial interaction testing sampling techniques to generate longer GUI test cases, which exercise more event interactions and have been shown to improve fault detection. However, because the models extracted are only approximations of the actual event interactions, the generated test cases might suffer from problems of infeasibility. Specifically, widgets or buttons can become disabled, or the events cannot be dispatched once triggered, causing the test cases to terminate prematurely. These problems are caused by violations of temporal constraints on the event interaction sequences. The lack of awareness of these constraints during GUI test generation means that coverage of the event interactions may be lost, potentially weakening the fault detection effectiveness and reducing the quality of the generated GUI test suites.

In this work, we propose a framework for automatically repairing GUI test suites that contain infeasible sequences. We begin by identifying the set of infeasible patterns that we expect to see in GUIs. We then develop a genetic algorithm for test suite repair. An evaluation on small-sized synthetic GUI subjects demonstrates that our technique is able to cover over 99% of the full combinatorial coverage of the events in these subjects and it outperforms a random algorithm for the same purpose. We

then apply this technique to some non-trivial GUI subjects, and show that it is able to increase the combinatorial coverage of test suites for non-trivial GUIs to 98% of their feasible coverage, and that the types of constraints discovered match those that we have previously identified.

ACKNOWLEDGMENTS

Finally, I can sit down to express my thanks to the people that helped me during my master's study. This work was never easy and required me to learn a lot of things that I had never heard of. So I want to first thank my adviser, Dr. Myra Cohen. She has always been an excellent mentor for my master's study. She thinks for me, guides me, gives suggestions to me, teaches me, and supports me. Without her help, I could never image my master's study, neither this work. I can still remember the time when we worked hard for the paper deadlines. Her dedication to the research and career has set a life-time example for me.

Next, I want to thank the other two members in my thesis committee, Dr. Matthew Dwyer and Dr. Anita Sarma. I want to thank them for spending time to pick out even spelling mistakes in my thesis. I really appreciate their comments, suggestions, and questions for the thesis, as well as their questions and challenges at the thesis defense. All these made me rethink my work and improve the thesis.

I would also like to thank the GUITAR group from University of Maryland, College Park, especially Dr. Atif Memon, Bao N. Nguyen, Dr. Scott McMaster and Dr. Xun Yuan. With their help, my research went much more smoothly than expected. I want to particularly thank Dr. Atif Memon and Bao N. Nguyen for their work on the development of the website for Community-based Event Testing (COMET).

During the past two years, friends in ESQuaReD helped me a lot. I want to thank all of them for their help. Special thanks go to Brady Garvin for helping me on the experiments of my research by providing his CASA tool and solving a lot of system-related issues, Zhihong Xu for her discussion on the search algorithm and implementation, Wayne Motycka for talking about the implementation of the COMET website, Katie Stolee for her cakes and vegetables, Charlie Lucas for inspiring

me starting learning Japanese.

I also want to thank Nancy Lind-Olson, Shelley Everett, and Deb Heckens for their generous help on the administrative operations.

Finally I hope to thank my parents and sister for always standing with me and supporting me.

This work was supported in part by the US National Science Foundation under grants CCF-0747009, CNS-0855139 and CNS-0855055, the Air Force Office of Scientific Research through award FA9550-09-1-0129, and by the Defense Advanced Research Projects Agency.

Contents

Contents **vi**

List of Figures **ix**

List of Tables **xi**

1 Introduction **1**

1.1 Motivation 2

1.2 Contributions 7

2 Background and Related Work **9**

2.1 GUI Testing 9

2.2 Combinatorial Interaction Testing 14

2.3 Search Algorithms and Evolutionary Testing 17

3 Identifying Infeasible Patterns **21**

3.1 Feasibility of GUI Test Cases 21

3.2 Infeasible Patterns: Event Constraints 24

3.3 Summary 29

4 Framework **30**

4.1 GUI Test Suite Repair 30
4.2 Assumptions 32
4.3 GUI Test Suite Repair Framework 33
4.3.1 Discussion on Implementation 35
4.4 A Genetic Algorithm for Repairing GUI Test Suites 37
4.5 Evaluation 43
4.5.1 Subjects 43
4.5.2 Independent Variables 44
4.5.3 Dependent Variables 45
4.5.4 Experimental Methodology 45
4.5.5 Threats to Validity 46
4.6 Results 47
4.6.1 RQ1: Framework Effectiveness 47
4.6.2 RQ2: Comparison with a Random Algorithm 49
4.6.3 RQ3: Scalability of the Genetic Algorithm 52
4.7 Summary 53

5 Application to Non-Trivial GUIs **56**

5.1 Optimizing the Algorithm 57
5.1.1 Reducing the Number of Test Cases to Execute 59
5.1.2 Reducing the Time for the Execution of Individual Test Cases 61
5.1.3 Tuning the Parameters for the Genetic Algorithm 62
5.2 Evaluation 63
5.2.1 Subjects 63
5.2.2 Metrics 64
5.2.3 Experiment Methodology 65

5.2.4 Threats to Validity . . . 66

5.3 Results . . . 67

5.3.1 RQ1: Applicability to Non-Trivial GUIs . . . 67

5.3.2 RQ2: Scalability for Non-Trivial GUIs . . . 68

5.3.3 RQ3: Matching for Discovered Patterns . . . 70

5.4 Summary . . . 73

6 Conclusions and Future Work **75**

6.1 Future Work . . . 76

Bibliography **78**

List of Figures

1.1 An example GUI test case 3
1.2 An example of infeasibility 6
2.1 An example of a GUI and its models: (a) A simple GUI, (b) its EFG, and (c) EIG 12
3.1 An example of event handlers 25
3.2 *Disabled* constraint 27
3.3 *Requires* constraint 27
3.4 *Consecutive* Constraint 28
3.5 *Excludes* Constraint 28
4.1 The framework for GUI test suite repair 34
4.2 Using Genetic Algorithm as Repair Algorithm 37
4.3 Crossover. 2-sets e_1 and e_4 at position 1 and 4 and e'_2 and e'_5 at position 2 and 5 are the new coverage of these two test cases respectively. During crossover, the new coverage of the two test cases are exchanged. 41
4.4 Comparison of Coverage for 2-way Criteria 52
4.5 Comparison of Coverage for 3-way Criteria 53

4.6 Comparison of Numbers of Test Case Executions and Execution Time for GA 55

List of Tables

1.1 Exhaustively listing test cases for events in Figure 1.1. 4

2.1 Some input paramters of “ls”. 15

2.2 A 2-way covering array for the input parameters of “ls”. 17

2.3 Length-4 test cases for testing 2-way combinations of $\{Save, SaveAll, New\}$. 17

4.1 Subject programs 1 44

4.2 Repaired test suites with length-5 test cases (average of five runs) 48

4.3 Comparison of random and genetic algorithms (execution time in minutes (m), days (d) or hours (h)) 50

4.4 Repair for length-15 and -20 test cases; 2-way criteria (execution time in days (d) or hours (h)) 54

5.1 Time for the execution of test cases occupies most of the time for the genetic algorithm. For each group in the table, population size is 100; maximum number of consecutive bad moves is 100; size factor is 1.5; number of test cases that run in parallel is 3. $time_{exe}$ is the time for the execution of test cases; $time_{genetic}$ is the time for the genetic algorithm. Time is in milliseconds. 58

5.2 Subject programs 2 64

5.3 Results of repairing non-trivial GUIs using 2-way criteria. 69

5.4 Number of uncovered t-sets for each type of constraint. 70

5.5 Number of uncovered t-sets due to each type of constraint. 72

Chapter 1

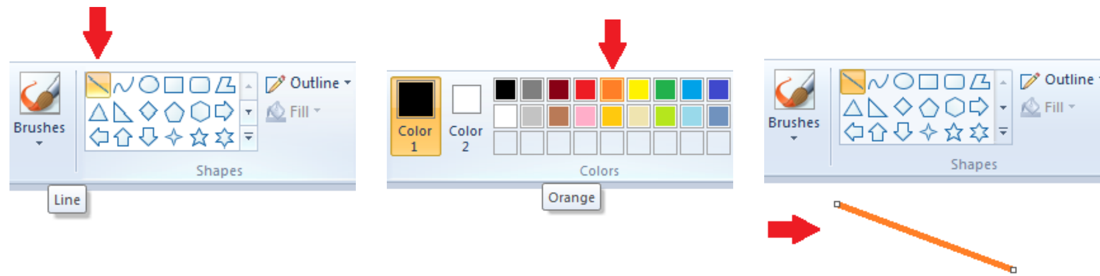
Introduction

Graphical User Interfaces (GUIs) are prevalent in today's software systems, spanning from operating software and networking systems to desktop applications and wireless web interfaces. As its name indicates, a GUI is an interface for users to send commands to and control the underlying business logic modules of the software system; it also passes output information from the computation to the users. It is the face of an application, and a bridge between the users and the core business logic modules of the application. The quality of GUIs directly and largely affects the reliability and usability of the entire software system. Research has shown that faults discovered from underlying modules with the GUIs often lies in the business logic of the application itself [3]. A faulty GUI might compromise the quality of a software product, reducing user satisfaction. In this sense, the quality assurance of the GUIs is very important. Testing, as an effective quality assurance method, has been widely used to assure the dependability of GUIs [36, 15, 24, 1, 10, 14].

1.1 Motivation

Because of the event-driven nature of modern GUI software, the basic black-box approach for GUI testing is to simulate events initiated by users on the Application Under Test (AUT), and then check the correctness of the behavior by examining the GUI states and the output from the GUIs. Unlike white-box techniques, which rely on code coverage of the applications, a black-box approach tests the GUI applications by building test cases according to the specifications and requirements, and checking the correctness of the output from the application. Although it can be used at different levels, a component or system level is usually preferred. For example, to test whether we can draw colored straight lines in the the Windows program `Paint` correctly, we can use a test case containing three events (shown in Figure 1.1). We start the `Paint` application, select "Line" as the shape (step 1), "Orange" as the color (step 2), and then draw a line on the canvas (step 3). If a straight line in orange appears between the starting point and the ending point of the mouse drag, the behavior and the output of the GUIs are considered correct. From this example, we can see that it usually requires several events to finish a meaningful task on a GUI, which means testing separate individual events is not enough. Event sequences are usually employed for testing the GUIs. As the execution of events proceeds, the initial state of the GUIs is changed, and the preceding events produce a state where the following events are executed.

Research has shown that longer event sequences are more likely to discover faults for GUIs [26, 39]. However, generating long event sequences in GUI testing leads to two problems. The first problem is the exponential testing space. For example, for a 10-event GUI, the number of all possible length-10 test cases is 10^{10}. Even when considering possible restrictions on the combinations, the number might still be large.



Step 1: select line   Step 2: select orange   Step 3: draw a line

Figure 1.1: An example GUI test case

For the example of `Paint`, all of the possible test cases consisting of the three events *Select Line*, *Select Orange* and *Draw a Line* are listed in Table 1.1[1]. It is common for even very simple GUIs to have a large number of events. For example, the simple Windows editor `WordPad` contains 325 events [24], and therefore in theory there are 325^{10} length-10 test cases which can be generated for it. Even if we can run a million test cases per second, a thorough walkthough of all the test cases will still cost over 416895303700 years! This means that we need some method to select a subset of sequences for GUI testing. The second problem is constraints on event interactions. The events of a GUI need to follow certain rules for the order of execution. For example, before a modal window is closed, any event in its parental window cannot be executed. Similarly, if clicking on a button A disables another button B, a click on B following a click on A receives no response from the GUIs. This entails that the execution of the events needs to follow particular flows/paths, otherwise some events may not be executed successfully.

Research has been conducted to aid the automated test generation for GUIs. Event-flow graphs (EFGs) and event-interaction graphs (EIGs) [22] divide all events for a GUI program according to modality of its windows, and depict the execution

[1] All the positions in this thesis are zero-based.

Table 1.1: Exhaustively listing test cases for events in Figure 1.1.

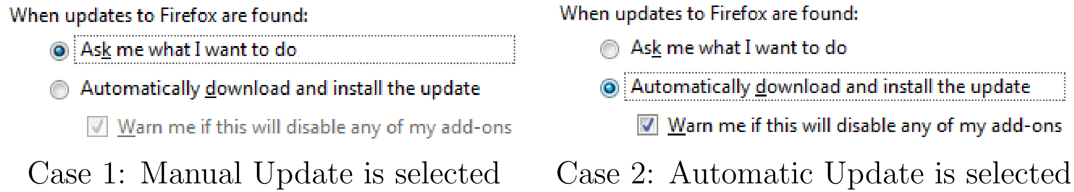
No.	Positions		
	0	1	2
1	*Select Line*	*Select Line*	*Select Line*
2	*Select Line*	*Select Line*	*Select Orange*
3	*Select Line*	*Select Line*	*Draw a Line*
4	*Select Line*	*Select Orange*	*Select Line*
5	*Select Line*	*Select Orange*	*Select Orange*
6	*Select Line*	*Select Orange*	*Draw a Line*
7	*Select Line*	*Draw a Line*	*Select Line*
8	*Select Line*	*Draw a Line*	*Select Orange*
9	*Select Line*	*Draw a Line*	*Draw a Line*
10	*Select Orange*	*Select Line*	*Select Line*
11	*Select Orange*	*Select Line*	*Select Orange*
12	*Select Orange*	*Select Line*	*Draw a Line*
13	*Select Orange*	*Select Orange*	*Select Line*
14	*Select Orange*	*Select Orange*	*Select Orange*
15	*Select Orange*	*Select Orange*	*Draw a Line*
16	*Select Orange*	*Draw a Line*	*Select Line*
17	*Select Orange*	*Draw a Line*	*Select Orange*
18	*Select Orange*	*Draw a Line*	*Draw a Line*
19	*Draw a Line*	*Select Line*	*Select Line*
20	*Draw a Line*	*Select Line*	*Select Orange*
21	*Draw a Line*	*Select Line*	*Draw a Line*
22	*Draw a Line*	*Select Orange*	*Select Line*
23	*Draw a Line*	*Select Orange*	*Select Orange*
24	*Draw a Line*	*Select Orange*	*Draw a Line*
25	*Draw a Line*	*Draw a Line*	*Select Line*
26	*Draw a Line*	*Draw a Line*	*Select Orange*
27	*Draw a Line*	*Draw a Line*	*Draw a Line*

order of the events using graphs. Event semantic interaction graphs (ESIGs) [40, 41] take this a step further. The semantics (i.e., the tasks the events do) are analyzed for the events, and the events are partitioned into groups where the events in one group interact with others in the same group while the events in different groups do not interact with each other. If one event does not interact with another event, they can be tested separately. These models and partition techniques not only reflect the

constraints on the event interaction, but also help restrict the test generation in a more definite space. However, the precision of these models are highly dependent on the techniques used for their extraction. State-of-the-art techniques for obtaining these models rely on dynamic runs of the AUT. That is, the GUI application is started, and the component hierarchies are extracted from the dynamic information to build the models. This step may be repeated several times to achieve a more precise model. Because the dynamic execution still may not show all of the possible aspects of the behavior of the AUT, the resulting models may not contain all the information for the event interactions. The precision of the models is dependent on the information revealed in the dynamic runs of the AUT.

Recent advances in model-based GUI testing leverage the combinatorial interaction testing techniques to sample the testing space derived by the ESIGs [39]. A sampling method based on a covering array is generated for the events of an ESIG, and each row of the covering array becomes a test case for the corresponding ESIG. Because of the properties of covering arrays, all the n-way combinations of the events at every n positions of the event sequences are covered by the test cases. More details on combinatorial interaction testing and covering arrays will be discussed in Chapter 2. This technique has two advantages. First, long test cases for GUIs can be generated systematically at different confidence levels. Second, more context (i.e., states) may be exposed due to the coverage on all the n-way combination at any n positions. This method incorporates more cases where different combinations of events get executed in different contexts, and studies have demonstrated that it is able to find more (and hard) faults than traditional techniques that use shorter test cases [39].

However, in the work of Yuan et al. [39], the generated test cases may contain infeasible event interactions, so that test cases cannot run to completion. For example,



Case 1: Manual Update is selected Case 2: Automatic Update is selected

Figure 1.2: An example of infeasibility

one test case for TerpPresent 3.0 ([39], a program for preparing presentations) is (*New*, *Remove Slide*, *Insert Background Image*, . . .). Because the *New* operation by default creates a file containing only one slide, after *Remove Slide*, *Insert Background Image* cannot be successfully performed because no slide is left any more. The cause of such problem is that the generated test cases do not fully conform to the constraints on the event interaction and follow infeasible paths of event-flows. More specifically, because the construction of the event graph models relies on one or several runs of the GUI program, they may not capture and reflect precisely the actual constraints in GUIs, especially those that require specific context to disclose. For example, considering the events *Undo* and *Redo*, although they interact with each other, if the model does not capture the fact that *Redo* cannot be executed before *Undo*, an infeasible test case (*Redo*, *Undo*) might be generated. Because *Redo* is initially disabled, it obviously cannot be executed. This problem is more severe in long sequences than short sequences, because longer sequences are likely to contain more complicated constraints which involve many events.

Next, we use another example to show problems that the infeasible test cases can cause. Consider three events, *Manual Update*, *Automatic Update* and *Warn on Incompatibility*, where the first two are exclusive-or options and the third is an option under the second (see Figure 1.2). A constraint here is that selecting *Manual Update* disables *Warn on Incompatibility*. If the model does not reflect this interaction, an infeasible test case (*Automatic Update*, *Manual Update*, *Warn on Incompatibility*)

may be generated. We call the combinations that cause the infeasibility infeasible combinations, and the rest feasbile combinations. The pair (*Manual Update*, *Warn on Incompatibility*) is an infeasible combination in the above test case.

Simply dropping all infeasible test cases can achieve the feasibility of the test suite, but may potentially lose coverage of certain feasible combinations. For example, continuing the above example and assuming we want to cover 2-way combinations of the events, if the test case (*Automatic Update*, *Manual Update*, *Warn on Incompatibility*) is abandoned, the coverage for the combination *Automatic Update* and *Manual Update* at position 0 and 1 may be lost if no other test case covers it. Another question arises: can we instead keep the infeasible test cases and run only the feasible part of them to maintain coverage? The answer is still no. If we consider the above test case, it is obvious that it stops after the execution of *Manual Update* because *Warn on Incompatibility* cannot be executed. Even if the test case remains, the actual coverage for the feasible pair *Automatic Update* and *Warn on Incompatibility* at position 0 and 2 is still missed. As a result, generating new test cases for the uncovered combinations in the test suite is needed to maintain a high coverage.

1.2 Contributions

In this thesis, we propose a technique for repairing the GUI test suites by searching for new feasible test cases to complete the coverage of feasible combinations. We begin with a covering array to generate an initial test suite for all events under consideration, and execute each test case dropping the infeasible ones. We then use a genetic algorithm to search the event sequence space for new feasible test cases that help to increase the combinatorial coverage lost by abandoning the infeasible test cases. This research makes the following contributions:

- We identify patterns of event interactions that may cause infeasibility. These patterns are summarized from event interactions of real GUI applications, and our evaluation demonstrates that they do exist and do cause certain types of infeasibility.

- We present a framework for reparing GUI test suites. Under this framework, a genetic algorithm is designed for searching for new feasible test cases by using the feedback from dynamic execution of infeasible test cases.

- We conduct an evaluation on small-sized synthetic programs to investigate the feasibility of the technique. The results show that the framework is able to recover the coverage of the test suite on feasible combinations to more than 99% while maintaining the feasibility of the test cases.

- We apply our technique to several non-trivial GUI applications. The evaluation shows that our technique can also repair the test suites of non-trivial GUI subjects to cover more than 98% of the feasible combinations. Moreover, the finally uncovered combinations reported by the algorithm help us discover the constraints that exist in the GUIs but were not properly modeled.

The rest of this thesis is organized as follows. We next review the background knowledge and discuss related work (Chapter 2). In Chapter 3, we define and analyze the problems we try to solve in this thesis. With these, we present our framework and algorithms for solving the problems in Chapter 4. After an evaluation for the feasibility of our technique, we apply it for non-trivial GUI applications in Chapter 5. An evaluation for the improved algorithm is then carried out on several non-trivial GUI subjects. Finally, in Chapter 6 we conclude the thesis.

Chapter 2

Background and Related Work

In this chapter, we will review background on model-based GUI testing, combinatorial interaction testing and evolutionary testing. We will also briefly discuss how they are related to our work.

2.1 GUI Testing

GUI testing is an effective means for assuring the quality of GUI programs. It is usually conducted on an integration or a system level using black-box approaches. To perform GUI testing, first, the application to test is identified. After some analysis on the GUI application and necessary preparation, a set of GUI test cases are generated. For each test case, test oracles will also be generated. A test oracle is used for judging whether the test cases run correctly on the GUI application. Then, the test cases are run on the GUIs to exercise the functionalities of the GUI application. The behavior of the GUIs, such as whether a dialog pops up after clicking on a button, whether the text in a label is set correctly after selecting a radio button, and etc., and the output of the GUI application are observed and compared to the corresponding test oracle

for this test case to determine whether the test case run correctly. If not, debugging and fault localization techniques will be used for locating faults in the code of the application. Finally, patches and updates can be developed to fix the bugs. GUI testing can be carried out both manually and automatically. An automated DART process is shown for smoke tests for GUIs in [26]. A lot of tools can be leveraged for GUI testing [1, 10, 14, 31, 12]. In this work, we focus on automated GUI testing, especially the automated GUI test generation.

Due to the event-driven nature of GUIs, the basic elements for constructing GUI test cases are events. So we first define an event of a program.

Definition 1 (Event)**.** *An event is an action disptached to and handled by a program.*

An event may be initiated outside the scope of a program (such as by a human user, the operating system, the network, or other applications, etc.), or by the program itself. The piece of code in the program responsible for handling an event is usually called an event handler. Event handlers are usually implemented as asynchronous callback subroutines. A program is event-driven if it changes its behavior in response to events. A GUI program is not necessarily event-driven, but in this work, we only consider event-driven GUIs. With the definition of events, we can define GUIs considered in this thesis.

Definition 2 (GUI [22])**.** *A GUI, G, is a hierarchical, graphical front-end to a software system that accepts as input user-generated and system-generated events from a finite set of events $E_G = \{e_0, e_1, \ldots, e_{n-1}\}$ and produces deterministic graphical output. A GUI contains a set of graphical objects $O_G = \{o_0, o_1, \ldots, o_{m-1}\}$; each object o_i has a fixed set of properties $P_{o_i} = \{p_{o_i0}, p_{o_i1}, \ldots, p_{o_iq-1}\}$. At any particular time t during the execution of the GUI, these properties have discrete values, the set of which ($P_G = \{o_i \in O_G \mid P_{o_i}\}$) constitutes the state of the GUI.*

To test a GUI, the events that can be performed on it are identified and sent to the GUI. For each event e_i, the GUI changes its state from P_{before} to P_{after}. The states of the GUI are checked against the oracle state for correctness. First, the starting state P_{start} is checked to ensure that the GUI is in a correct starting state. After the execution of an event e_i, the consequent state P_{after} is checked against the oracle state P_{oracle} after e_i. If $P_{after} = P_{oracle}$, the behavior of the event is considered correct. The oracle states can be fetched either manually or automatically [25].

The events for a GUI can be used to construct test cases for it.

Definition 3 (GUI test case)**.** *For a GUI, G, a test case* c[1] *is a finite sequence* $(e_0, e_1, \ldots, e_{k-1})$*, where* $e_i \in E_G, 0 \leq i < k$*, and* k *is the length of the test case.*

A GUI test case is an event sequence. In this work, we use these two terms interchangeably. A randomly generated GUI test case is not guaranteed to be executable due to the interaction among the events. Usually, we generate a set of test cases to form a test suite for testing different aspects and features of a GUI.

For convenience, we use $l(c)$ to denote the length of the test case c. We also use $c_{G,k}$ to denote the set of all length-k test cases for a GUI, G.

Definition 4 (GUI test suite)**.** *For a GUI, G, a test suite* C_k *is a set of GUI test cases of the same length* k *for G. Formally,* $C_k \subseteq c_{G,k}$*.*

While in general, the test cases in a GUI test suite do not need to be of the same length, in this thesis, we consider all the test cases in a GUI test suite to have the same length. Therefore, a GUI test suite C is a $|C| \times k$ array, where k is the length of the test cases in it.

[1] c is used to distinguish from the strength t for a covering array.

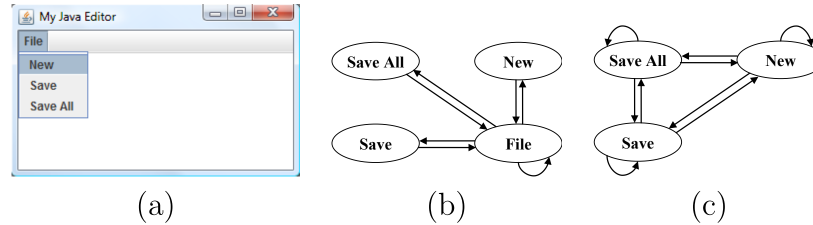


Figure 2.1: An example of a GUI and its models: (a) A simple GUI, (b) its EFG, and (c) EIG

There has already been a lot of work on model-based GUI testing, including those using finite state machines [29], pre- and post-conditions [17], and directed graph models [26]. We talk about graph models because of its relevance to this work.

An EFG is a directed graph that models all possible event sequences that may be executed on a GUI [22]. In EFGs, nodes denote the events in the GUI and edges represent a relationship between events. An edge from node n_x to node n_y means that the event represented by n_y *may be* performed *immediately after* the event represented by n_x *along some execution path*. This relationship is called `follows`. An EFG can be represented by a set of nodes $\mathbf{N}$ representing events in the GUI and a set $\mathbf{E}$ of ordered pairs (e_x, e_y), where $\{e_x, e_y\} \subseteq \mathbf{N}$, representing the directed edges in the EFG; $(e_x, e_y) \in \mathbf{E}$ if and only if e_y `follows` e_x.

Figure 2.1(a) presents a GUI example that consists of four events, *New*, *Save*, *SaveAll*, and *File*. Figure 2.1(b) shows the GUI's EFG; the four nodes represent the four events; the edges represent the `follows` relationships. For example, the event *New* `follows` *File*.

EIGs simplifies EFGs such that they do not include events to open or close menus, or open windows as nodes. The result is a more compact, and hence more efficient, GUI model. An EFG can be automatically transformed into an EIG by using graph-rewriting rules (details are presented in [37]).

Figure 2.1(c) shows the corresponding EIG. Note that the EIG does not contain the menu-opening *File* event. The graph-rewriting rule used to obtain this EIG was to (1) delete *File* because it is a menu-open event, (2) for all remaining events e_x replace each edge (e_x,*File*) with edge (e_x, e_y) for each occurrence of edge $(File, e_y)$, and (3) for all e_y, delete all edges $(File, e_y)$. The GUI's EIG is fully connected with three nodes representing the three events.

The basic motivation of using graph models to represent a GUI is that graph-traversal algorithms may be used to "walk" the graph, enumerating events along visited nodes, thereby generating test cases. An approximation of these models can be constructed automatically using a reverse engineering technique called *GUI Ripping* [21]. A *GUI Ripper* automatically traverses a GUI under test and extracts the hierarchical structure of the GUI and events that may be performed on the GUI. The GUI Ripper is not perfect, i.e., parts of the retrieved information may be incomplete and just for a specific path of execution, which is why we say that it outputs an *approximation* of the EFG. As a result, it is possible that some event sequences generated via these graphs are infeasible.

In more recent work [40], a new feedback-based technique has been developed for GUI testing. This technique requires an initial *seed* test suite to be created and executed on the software. Feedback from this execution is used to augment a model of the GUI and *automatically* generate additional test cases. The seed test suite is generated using the EIG model and executed on the GUI using an automatic test case replayer. During test execution, the run-time state of GUI widgets is collected and used to automatically identify an *Event Semantic Interaction* (ESI) relationship between pairs of events. This relationship captures how a GUI event is related to another in terms of how it modifies the other's execution behavior. The ESI relationships are used to construct a new model of the GUI – ESIG. The ESIG combines

the ESI relationship, dynamic feedback obtained in terms of event execution, into the seed test suite generated from the structural model EIG, and therefore captures certain structural and dynamic aspects of the GUI. The ESIG is used to generate new test cases. These test cases have an important property such that each event is ESI-related to its subsequent event, i.e., it influences the subsequent event during execution of the seed test suite. However, because of the complexity of GUIs, multiple events might contribute to the change of the behavior of another event. So even when using an ESIG, we may still encounter infeasible test cases [39].

A technique for repairing *GUI test cases* for regression testing is developed in [23]. When the structure of a GUI is modified, test cases from the original GUI test suite are either reusable or unusable on the modified GUI. The proposed algorithm (1) automatically determines reusable and unusable test cases from a test suite after a GUI modification, (2) determines the unusable test cases that can be repaired so that they can execute on the modified GUI, and (3) uses *repairing transformations* to repair the test cases. The challenges of repairing sequences are fewer in the context of regression testing because the differences between the two versions' EFGs is used to drive the repair. Our focus on this thesis is repairing GUI test suites rather than GUI test cases.

2.2 Combinatorial Interaction Testing

Combinatorial interaction testing (CIT) is a technique for testing combinations of input parameters to a software system. For example, Table 2.1 shows part of the input parameters the file-listing command "ls". In the table, we show three parameters that can be specified for different behaviors of the command. To ensure the correctness of the interaction on these parameters, we need to test the combinations of the input

Table 2.1: Some input paramters of “ls”.

Color	Block Size	Sort Method
Never	*1KB*	*by status*
Always	*1MB*	*by time*
Auto	*1GB*	*by size*

parameters, such as (*Never*, *1MB block size*, *Sort by time*) and (*Always*, *1KB block size*, *Sort by size*). The number of all the combinations for this is 3*3*3 = 27. Although it is easy to test all the combinations for this example, it might be very hard to apply it to a program with, for example, 20 parameters, where each contains 5 possible values because of the exponential growth of the testing space. In real applications, it is not unexpected for a program to have many input parameters [28].

To this end, covering arrays are used to systematically sample the testing space.

Definition 5 (Covering array [4])**.** *A covering array (written as $CA(N;t,k,v)$) is an $N \times k$ array on v symbols such that any t of the k columns contains all ordered subsets of size t of the v symbols at least once.*

In the definition, t is the strength of the array, k is the number of the columns of the array, and v is the number of possible values for each column of the array. The number of possible values for each column can be different, and this is depicted by another type of covering array named mixed level covering array [4]. In this work, we only consider the type of covering array where the number of values for each column is the same. The v symbols are not necessarily the same for each column; each column can have its own (different) v symbols (or values). One of all ordered subsets of size t of the v symbols appearing on t of the k columns is called a ***t*-set**. The total number of t-sets for $CA(N;t,k,v)$ is $\binom{k}{t}v^t$. A t-set is different from a t-way combination. A t-set does not only consider the combination of t values, but also the positions of these t values at the k columns. For example, a combination (e_1, e_2, e_3) and a combination

(e_1, e_2, e_3) at column 1, 2 and 4 (i.e., a 3-set (e_1, e_2, e_3)) respectively are different. The former does not contain position information and implies that the combination can appear at any three columns, such as (e_1, e_2, e_3) at column 2, 4 and 5, or (e_1, e_2, e_3) at column 1, 3 and 4, etc. The latter contains the position information for the three values, i.e., the three values in this 3-set at only appear at column 1, 2 and 4. This example shows that when a t-way combination is combined with positions, it becomes a t-set. Table 2.2 shows 2-way covering array for the "ls" example. If we pick any two columns from the table, we can see that all the combinations for the values of these two columns appears once. This covering array covers all the 2-sets for the example. If we remove the last row, the 2-sets lost are (*Auto*, *1GB*), (*Auto*, *by status*) and (*1GB*, *by status*) at positions (0, 1), (0, 2) and (1, 2) respectively. However, in general, a t-set might be covered by multiple rows, therefore deleting one of the rows will not necessarily reduce the coverage of a specfic t-set. Next we define this type of coverage.

Definition 6 (Combinatorial coverage)**.** *Given the strength t, the number of columns k, and the number of possible values v, the number of t-sets covered by an arbitrary $M \times k$ array A (M is the number of rows) is called its* **combinatorial coverage***, written as* $\texttt{cov}_{t,k,v}(A)$.

Obviously, $\texttt{cov}_{t,k,v}(CA(N;t,k,v)) = \binom{k}{t}v^t$.

In this work, the concept of covering arrays are leveraged for GUI test generation. For a GUI, G, if we want to test all the t-way combinations of the events using length-k test cases, we can use $CA(N;t,k,|E_G|)$ to generate the test suite. For example, for a GUI containing three events $\{Save, SaveAll, New\}$, we can use $CA(9;2,4,3)$ to generate length-4 test cases to test all the 2-way combinations. Table 2.3 shows all the test cases. In this example, we assume that any event in E_G can appear at any

Table 2.2: A 2-way covering array for the input parameters of “ls”.

Color	Block Size	Sort Method
Never	*1KB*	*by status*
Never	*1MB*	*by size*
Never	*1GB*	*by time*
Always	*1KB*	*by time*
Always	*1MB*	*by status*
Always	*1GB*	*by size*
Auto	*1KB*	*by size*
Auto	*1MB*	*by time*
Auto	*1GB*	*by status*

Table 2.3: Length-4 test cases for testing 2-way combinations of $\{Save, SaveAll, New\}$.

No.	1	2	3	4
1	*Save*	*Save*	*Save*	*New*
2	*Save*	*SaveAll*	*SaveAll*	*Save*
3	*Save*	*New*	*New*	*SaveAll*
4	*SaveAll*	*Save*	*New*	*Save*
5	*SaveAll*	*SaveAll*	*Save*	*SaveAll*
6	*SaveAll*	*New*	*SaveAll*	*New*
7	*New*	*Save*	*SaveAll*	*SaveAll*
8	*New*	*SaveAll*	*New*	*New*
9	*New*	*New*	*Save*	*Save*

position, and one event can be executed multiple times in a row. It is true for the three events in the example, however, for other events, it is not always the case, which means, the generated test cases are not always executable.

2.3 Search Algorithms and Evolutionary Testing

An emerging field of software engineering, called search based software engineering utilizes meta-heuristic search to solve common software engineering problems [11]. Meta-heuristic search techniques can be applied to problems which can be described as an optimization problem, but that cannot be solved through exhaustive methods.

Algorithm 1 Genetic Algorithm for Adding a Single Test Case

Ensure: A optimal solution.
1: Generate initial population
2: Evaluate fitness of and rank the chromosomes in the population
3: **while** Stopping criteria not satisfied **do**
4: Select the best-fit chromosomes for reproduction
5: Breed new chromosomes through crossover and mutation operations, and add them into the population
6: Evaluate the fitness of new chromosomes and rank all the chromosomes
7: Drop off the least-fit chromosomes to maintain population size
8: **end while**

The problem can be specified as a set Σ of feasible solutions (or states) together with a cost $c(S)$ associated with each $S \in \Sigma$. An optimal solution corresponds to a feasible solution with overall (i.e. global) minimum (or maximum) cost. Evolutionary algorithms are a type of meta-heuristic search algorithm that conduct the search in a way inspired by the biological evolutionary process. Usually an evolutionary algorithm contains phases such as reproduction, mutation, recombination and selection, etc., which all exist in the biological evolution [7].

A genetic algorithm is one of the most commonly used evolutionary algorithms. Algorithm 1 shows a framework for a genetic algorithm. In a typical genetic algorithm, all the possible solutions are modeled as **chromosomes** or individuals, which contains a set of **alleles** or genes. A population is composed of many individuals, and is used for the evolution. The population size may be selected according to the nature of the problem. A population at a particular time is call a **generation**. At the beginning of the algorithm, an initial population is generated randomly or through other methods. This serves as the first generation. For each consecutive generation, the individuals in the population are paired as **parents** and a **crossover** or recombination stage takes place for them to exchange and combine information, and breed a set of children. These children are added as new individuals to the population. In this way, the

population size grows after each generation. To maintain a stable population, a **selection** phase is used to select a portion of one successive generation for producing the next generation, while others in the successive generation are dropped.

The selection is based on the evaluation of the **fitness** of the chromosomes. A **fitness function** is designed to judge how good a chromosome is, i.e., how close it is to a desired "optimal" solution. In the selection, all the chromosomes in current population are ranked by the fitness, and then qualified chromosomes are selected. The diversity of a population is important for the search because when the chromosomes are too similar to each other, it is easy for them to converge to a local optimum. If the reached local optimum is not the global optimum, the evolution may slow down or even stop. If the search is trapped, it is hardly able to escape from that because of the similarity of the chromosomes. To avoid this, a **mutation** operator is applied to the population for introducing diversity.

A classic method for doing mutation is changing arbitrary alleles of chromosomes in the population. Although the mutation may harm the fitness of chromosomes, it provides a chance for the trapped chromosomes to escape from local optima. The evolution terminates when the **stopping criteria** are satisfied. The choice of stopping criteria varies. A simple stopping criterion can be that a good enough chromosome (i.e., solution) is found. However, it is not always easy to reach the global optimum, and sometimes a local optimum might be enough. The number of consecutive bad moves is used for this purpose. A **bad move** is a move where the most-fit chromosome in the successive generation is no better than the parental generation. When the evolution keeps making bad moves, it indicates that the search has reached a plateau and is trapped in some local optimum. The more consecutive bad moves, the more likely it is this situation. Under this situation, the search can be stopped because it is unable to make any progress. Similarly, the maximum number of generations can

also serve as a stopping criterion to prevent the search from going too far. Stopping criteria may be also dependent on budget or resources allowed. For example, when the time limit or the maximum allowed memory usage has been reached, the algorithm stops.

One common way to automate test generation/feedback is through the use of evolutionary algorithms such as genetic algorithms [27, 34, 16, 20]. In the work of [27], Pargas et al. leveraged a genetic algorithm for searching test data that can cover certain targets (such as statements, branches, paths, etc.). In the work of [34], Tonella designed a genetic algorithm for generating unit tests for objects. It considers both the input data as well as method invocation sequences, but suffers from potentially infeasible test cases. Other work using meta-heuristic search to generate test sequences such as that of Marchetto and Tonella [19] is related in that they generate long sequences for testing web applications. Their focus is to provide test case diversity, but they may also suffer from infeasibility. We expect that our work may also benefit this type of test sequence generation.

Genetic algorithms have been used for generating CIT samples [32]. Our work is closely related to this thread of research, but genetic algorithms have only been used to generate single CIT samples without temporal constraints. More recent work by Arcuri and Yao [2] and Weimer et al. [35] present genetic programming solutions to repair faulty source code. Although they too focus on repair, their goal is quite different from ours in that they are targeting source code and using genetic programming to evolve new non-faulty statements.

Chapter 3

Identifying Infeasible Patterns

Infeasible test cases are not desired during GUI test generation. They reduce the confidence and usefulness of the test suite in terms of combinatorial coverage. To avoid generating infeasible GUI test cases, we first define what is considered to be infeasible in this work. After that, we analyze why and in what forms the infeasibility occurs. We learn and identify several patterns from the event interactions of real GUIs, and these patterns summarize the possible cases where the infeasibility may occur. Some of the material in this chapter has been published in [13].

3.1 Feasibility of GUI Test Cases

Definition 7 (Feasibility)**.** *For a GUI, G, and any $k > 0$, $c = (e_0, e_1, e_2, \ldots, e_{k-1})$ is a test case in $c_{G,k}$. If all $e_i \in c$ can be triggered, dispatched to and responded to by G, c is feasible; otherwise, c is infeasible.*

Infeasibility occurs when any of triggering, dispatching or handling the event cannot be successfully performed. For example, in Java Swing, if a menu item is disabled, the click event on it can be triggered, however, the event will not be dispatched to

the event handler by the library, and thus the event cannot be handled by the corresponding event handler and there will no response from the GUI. Similarly, if a modal dialog is not closed before attempting to send an event to its parental window, the parental window rejects the event because it cannot get the GUI focus of the current application (because usually only one window can hold the focus at one time). In either case, the event cannot be successfully passed to the corresponding event handler, so is not processed by the GUI.

In an infeasible GUI test case, there must be at least one event that cannot be executed successfully. Because the events are executed in order, when the first event that cannot be executed successfully is encountered, we say that the GUI test case is infeasible.

Definition 8 (Failure point)**.** *For a GUI, G, and* $c = (e_0, e_1, \ldots, e_{k-1})$*, a length-k test case for G,*

- *if c is feasible, its failure point* $f(c) = k$*.*
- *if c is infeasible, the failure point* $f(c) = i$*, where* $0 \leq i < k$*, and the sequence* $(e_0, e_1, \ldots, e_{i-1})$ *is a feasible test case for G.*

Although the first event not run to completion is treated as the failure point, there can be more events that are infeasible behind the event at the failure point. However, because the first infeasible event prevents the following events to exhibit their infeasibility, the infeasible events beyond the event at the failure point are ignored. Using the failure point, we are able to determine whether a GUI test case is feasible or not.

Infeasibility appears to be failures on the event execution using the GUI testing tools, which are very similar to those caused by faults, however, they are caused by different reasons. Next, we discuss infeasibility and faults.

Infeasibility vs. Faults: Faults are violations of the specifications of GUIs. They may cause failures during the interaction between the GUI application and human users. For example, when the *Redo* event is executed, if the event *Undo* is not redone, there is a fault in the application. Infeasibility is different. In specifications, infeasibility is described as exclusion of certain event interactions. That means, these event interactions are not allowed in the execution of GUI applications. For example, if the application is currently at the first page of a document, the event *Previous Page* cannot be executed because there is no page before the first page. During the development, such event interactions are usually avoided by disabling or hiding GUI components.

To the human users that interact with the GUIs, infeasibility means they cannot execute certain events while faults mean they cannot get the expected output or behavior from the GUI. Typically, when a user interacts with a GUI application, the infeasibility does not cause failure. They are simply forbidden to execute these events. But faults usually cause failures that prevent the users from obtaining correct information from the GUI.

The reason why we cannot easily distinguish between infeasibility and faults is because we are using automated tools to run test cases. The tool sends events to the application, and judges whether the event can be executed successfully. The failure occurring here can be either caused by the fact that the event cannot be executed or that the event is not executed correctly. Further, since a fault can be anything that violates the specifications, it can "pretend" to be an infeasible event interaction. For example, executing the event *Undo* should enable the event *Redo*. A fault can keep *Redo* disabled after executing *Undo*. Then the event *Redo* cannot be executed. It appears to have an infeasible event interaction here, but actually it is caused by a fault. So when we generate test cases, we make several assumptions to avoid these

situations. We leave the thorough analysis of failures and infeasibility as our future work.

3.2 Infeasible Patterns: Event Constraints

Infeasible test cases appear to cause failures during the execution because of the violation of the constraints on the event interaction. Now we analyze the reasons for the infeasibility at a finer granularity. Because the execution of events is actually the execution of the code of their corresponding event handlers, the constraints on the event interaction are actually the constraints on the code elements. Figure 3.1 shows an example code fragment which is simplified from real GUI applications for handling events *Undo* and *Redo*. The command pattern [6] is used in it. Two buttons, `undoButton` and `redoButton` (line 4), are used in the GUI for events *Undo* and *Redo* respectively. They are initially disabled (lines 5–10). When the events are triggered, methods `undoPerformed` (lines 18–26) and `redoPerformed` (lines 27–36) will be used to handle events *Undo* and *Redo* respectivley. Method `undoableActionPerformed` performs an undoable action (lines 11–17). From the code, we can see that after an undoable event is performed, the `undoButton` will always be enabled (line 15) while the `redoButton` will always be disabled (line 16). This means that the *Redo* event can never follow an *Undoable* event. Because `redoButton` and `undoButton` are both disabled at the startup of the application, they need to be enabled to perform the *Redo* and *Undo* events respectively. The way to enable the `undoButton` is to perform an *Undoable* event, because after invoking the event handler of an *Undoable* event, the `undoButton` is always enabled. The way to enable the `redoButton` is to perform an *Undo* event, because the `redoButton` is always enabled after performing an *Undo* event. Meanwhile, the *Redo* event also enables the `undoButton`, however, the *Redo*

```
 1. class ExampleGUIApplication {
 2.     List<Action> actions;
 3.     int pos = 0;
 4.     JButton redoButton, undoButton;

        ...

 5.     public void initialize() { // both buttons are first disabled
 6.         redoButton = new JButton("Redo");
 7.         redoButton.setEnabled(false);
 8.         undoButton = new JButton("Undo");
 9.         undoButton.setEnabled(false);
            ...
10.     }

11.     public void undoableActionPerformed(Action event) {
12.         event.execute();
            ... // remove actions after ‘‘pos’’
13.         actions.add(event);
14.         pos = actions.size() - 1;
15.         undoButton.setEnabled(true);
16.         redoButton.setEnabled(false);
17.     }

18.     public void undoPerformed() {
19.         Action currentAction = actions.get(pos);
20.         currentAction.undo();
21.         pos = pos - 1;
22.         redoButton.setEnabled(true);
23.         if (pos == 0) {
24.             undoButton.setEnabled(false);
25.         }
26.     }

27.     public void redoPerformed() {
28.         pos = pos + 1;
29.         Action currentAction = actions.get(pos);
30.         currentAction.redo();
31.         undoButton.setEnabled(true);
32.         if (pos == actions.size() - 1) {
33.             redoButton.setEnabled(false);
34.         }
35.     }
36. }
```

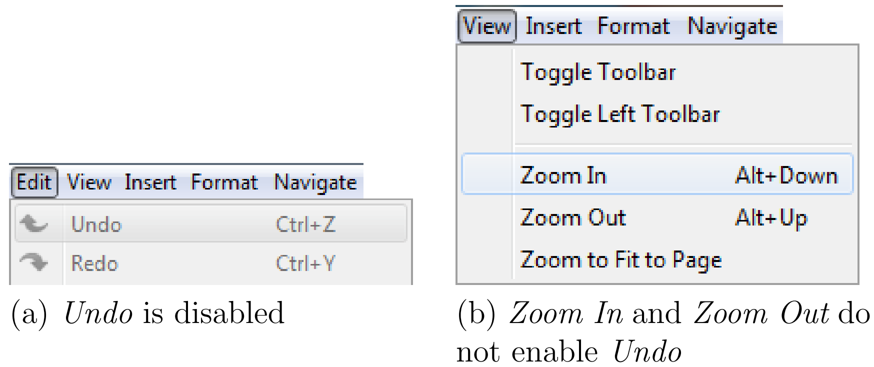
Figure 3.1: An example of event handlers

event also needs to be enabled before it can be performed. So using a *Redo* event to enable the `undoButton` needs events *Undoable* and *Undo* to be executed before it in order.

Although at the code level, the events may interact with each other through the elements of programs, such as variables and methods, etc., at a higher level, the event interactions show interesting patterns. After an informal study on the behavior of the GUIs, we find several patterns for the constraints on event interactions and classify them into four broad categories, *disabled*, *requires*, *consecutive* and *excludes*. This study is preliminary, and the classification is based on the author's experience and observation. In this work, we consider this as a starting point, and leave a deeper and thorough exploration of constraints on GUI applications as future work.

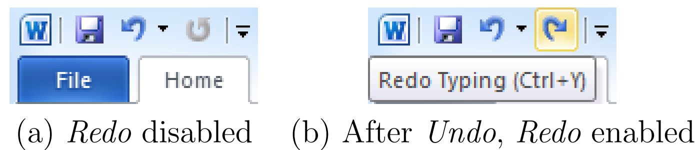
All of the examples shown are illustrated with short sequences for simplicity, however we can find examples for most of these constraints that are longer. For instance, we may have two, three or more events that are excluded or two or more events that require another event. We return to the issue of arity of constraints in our evaluation (Chapter 4) where we have designed synthetic programs to mimic each of these patterns of infeasibility.

Disabled Event Constraint: This type of constraint occurs when an event is always disabled. A menu item or widget exists for the event, but it will never be visible or enabled. The existence of this constraint might signal an error in the GUI, or we may encounter it during in-house development or in a program provided as a beta-release for preview. For instance, commercial software companies may release a beta version of a product for end-user testing, however, some features in the software might not have yet passed internal testing, or may not be suitable for early release. These may be turned off or disabled. This type of constraints can also appear within a specific set of events. For example, considering events *Undo*, *Zoom In* and *Zoom Out*, because



(a) *Undo* is disabled (b) *Zoom In* and *Zoom Out* do not enable *Undo*

Figure 3.2: *Disabled* constraint



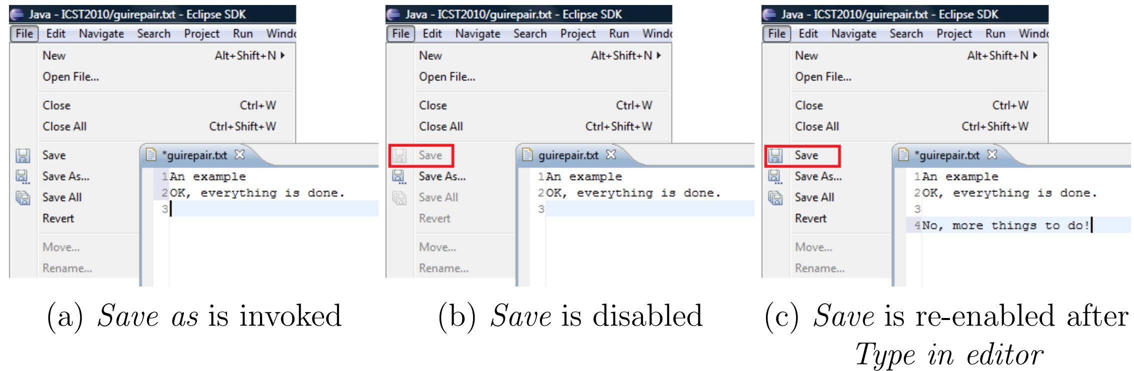
(a) *Redo* disabled (b) After *Undo*, *Redo* enabled

Figure 3.3: *Requires* constraint

Zoom In and *Zoom Out* are not undoable events, and *Undo* is disabled at startup, combinations of these events never enable *Undo*. In this case (within the range of these events), *Undo* can also be considered as a disabled event. Figure 3.2 shows such an example.

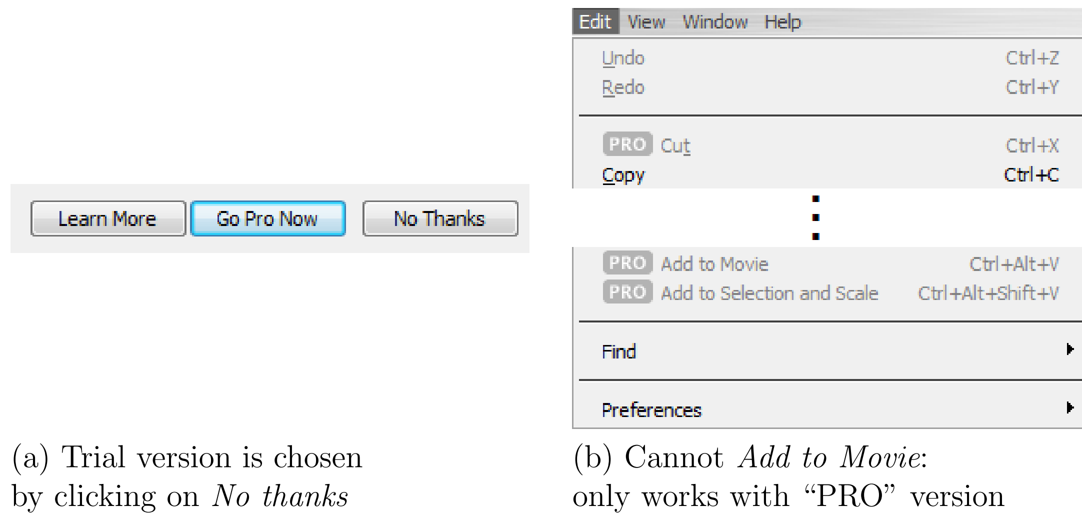
Requires Constraint: This constraint indicates that some event needs another event to be executed before it is enabled. An example of this type of constraint is illustrated by the *Redo* operation. Before one can execute *Redo* they must first execute *Undo*. Figure 3.3 shows an example of this sequence in Microsoft Office 2010.

Consecutive Constraint: This constraint means that two events cannot be executed consecutively. Usually, in this type of constraint, the execution of the first event disables the second event, making it unexecutable. The second event is re-enabled if another event occurs between them. An example of this type of constraint is the sequence *Save as*, *Save*. When these are executed sequentially, the *Save* event may be



(a) *Save as* is invoked (b) *Save* is disabled (c) *Save* is re-enabled after *Type in editor*

Figure 3.4: *Consecutive* Constraint



(a) Trial version is chosen by clicking on *No thanks* (b) Cannot *Add to Movie*: only works with "PRO" version

Figure 3.5: *Excludes* Constraint

disabled. But another event such as *Type in editor* may re-enable the second event. We show an example of this sequence from Eclipse 3.5 in Figure 3.4.

Excludes Constraint: This type of constraint is similar to the last one, however once the first event has been enabled there is no way to re-enable the second event within the current group of events. An example of this type of constraint can be seen in QuickTime 7. After QuickTime is downloaded and installed, it is by default an evaluation version. A window asks whether the user wants to select the professional version (see Figure 3.5(a)). If the user chooses to stay in the evaluation version, then

features only provided in the professional version are disabled (see Figure 3.5(b)). As a result, for instance, the sequence *No Thanks*, *Add to Movie* is always infeasible whether or not events occur between them.

Compound Constraint: This type of constraint is a combination of multiple constraints above. Real GUI programs may contain many constraints rather than a single one. For example, Eclipse 3.5 contains the *Requires* constraint for *Redo* and *Undo*, as well as the *Event Consecutive* constraint for *Save*, *Save as* and *Type in editor*.

3.3 Summary

In this chapter, we presented a study for the feasibility of GUI test cases. The study shows that infeasible event interactions do exist in test cases for real applications. We summarize our observation for these and catergorize the discovered infeasible patterns into four types of constraints. These constraints reflect the most commonly seen infeasibility that occurs in real GUI applications. Based on these, we next propose a framework for generating and repairing GUI test suites when these infeasible patterns occur.

Chapter 4

Framework

In this chapter, we present a framework for repairing GUI test suites when infeasibility is detected. First, we formally define what GUI test suite repair is, and set up the goals for this process. A repair algorithm is the core part of our framework. With the repair algorithm as the central element of our framework, the entire process is elaborated step by step. Some of the work in this chapter appeared in [13].

4.1 GUI Test Suite Repair

In non-trivial GUI applications, event interactions tend to be complicated. To thoroughly model constraints on the event interactions of GUIs requires a lot of manual effort. Some events are even only triggered in a special situation. The user usually needs to follow specific (sometimes very special) operations to set up contexts where certain events can run. If the users do not know the "hidden" event flows to follow, it is very likely that they will miss constraints when modeling the GUIs, especially when modeling without prior knowledge or assistance for the GUIs.

The GUI ripper from [21] automatically rips the component hierarchies for GUI

programs. However, the state-of-the-art techniques for doing so rely on one or more dynamic runs of the GUI programs to discover the structures of the GUIs. The problem for this is that one run (or even more) does not necessarily show all the possible event flows existing in the GUI, so the models extracted from the GUI may not contain all the possible event interactions. Then, the constraints on the event interactions may not be fully reflected by the models derived from the GUI. Although manual work can also be employed to help this effort, as we show earlier, it is very hard to gaurantee that all the possible event flows can be found by human.

Static analysis does not always thoroughly identify the constraints on the event interactions either. Modern GUIs are usually implemented with the help of GUI libraries, and in these libraries, polymorphism, callback methods, reflection and multi-threading are widely used. This may impede precise analysis on the programs [30, 18]. Moreover, the source code of the libraries may not be available and the GUIs and the underlying business logic may be implemented in different programming languages. All of these pose big challenges on static analysis for GUIs. When static analysis techniques are used, they may need to dive into the libraries. However, the cost of analyzing the libraries can introduce a large overhead, even for very small GUIs.

Summarizing all of the above, modeling all the constraints in a GUI can be extremely difficult. In this work, instead of extracting constraints from the GUIs and generating GUI test suites according to the constraints, we propose a method to generate GUI test suites without knowing the exact constraints in the GUIs and use the feedback from the actual execution of the generated test cases to repair the test suite. Before describing the framework, we define the GUI test suite repair and then discuss assumptions that must hold.

Definition 9 (GUI test suite repair)**.** *For a GUI, G, and C_G, a test suite for G, a*

GUI test suite repair for C_G is a transformation from C_G to another test suite C'_G, where every test case $c \in C'_G$ is feasible, and the lengths of the test cases in C_G and in C'_G are the same.

If all the test cases in C_G are feasible, the GUI test suite repair for C_G is trivial, because we can let $C'_G = C_G$. The same C_G can be repaired to obtain different C'_Gs.

In this work, our goal is to design a GUI test suite repair such that given a strength t, a GUI, G, and a length-k GUI test suite C_G, the test suite C'_G obtained after repair satisfies $\texttt{cov}_{t,k,v}(C'_G) \geq \texttt{cov}_{t,k,v}(C_G)$, where $v = |E_G|$.

4.2 Assumptions

This work makes several assumptions.

- No prior knowledge of the constraints on the event interaction: When we repair a given GUI test suite, we assume no knowledge on the constraints that exist in the GUIs. Rather, we learn the feasibility of a GUI test case by directly executing it. This assumption is based on the observation that learning constraints from the GUI can be expensive and error-prone. During the repair, we keep a record for the covered t-sets, and after that, the uncovered ones can be extracted. These uncovered t-sets are caused by the constraints on GUIs. From them, we can derive constraints that cause infeasiblity.

- Deterministic GUI behaviors: Real GUIs might provide functionalities related to random conditions, such as time, randomly generated numbers, etc. The randomness introduced in the GUIs can change the constraints in the GUIs on the fly. For example, when the traffic of the network is busy, some functionalities might be disabled, while when the traffic returns to normal, these functionalities

are enabled. Because we use the information from the execution of test cases to determine whether they are feasible, we want the information to be consistent regardless of random conditions. Otherwise, the information from one run of the test case cannot be directly used to determine whether it is feasible. In this work, we do not consider such non-deterministic behavior of GUIs, and assume that AUTs consistently provide determinstic behaviors.

- Assuming that we have a fault-free version for determining the feasibility of GUI test cases: In the work of Memon et al. [26], fault-free versions are used to generate oracles for the GUI test cases. Similarly, in this work, we also need fault-free versions of the AUT to determine whether a GUI test case is feasible. As we discussed, it can be very difficult to distinguish between a fault and infeasibility. If we do not have such versions, the failures occurring during the execution of test cases might be caused by faults in the AUTs rather than infeasibility. And in test generation, event sequences that reveal these faults will be treated as infeasible and will be avoided. We leave the issue of differentiating faults and infeasibility as future work.

4.3 GUI Test Suite Repair Framework

Figure 4.1 provides an overview of our GUI test suite repair framework. The input to the framework is an ESIG (or other similar graph model) (1) and the GUI structure extracted during ripping. The main controller (2) passes the graph and GUI structure to the test case assembler (3) which sends the ESIG model to a covering array generator (5) with the desired strength of testing (i.e., 2-way, 3-way, etc.). The covering array generator returns an initial set of event sequences for testing. In between the framework and the covering array generator is a base component interface (4). This

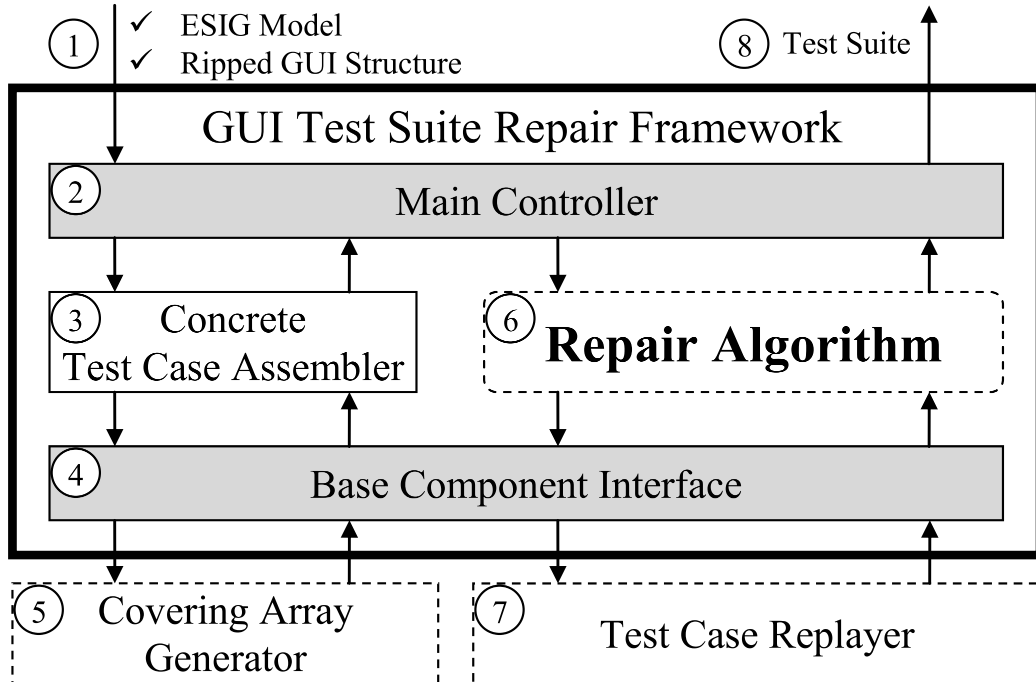


Figure 4.1: The framework for GUI test suite repair

serves as an adapter for the external tools. Once the test case assembler has a covering array, it assembles this into concrete test cases using the GUI structural information. These are passed back to the controller and the test suite repair phase begins (6). The repair algorithm interfaces through the base component interface with a test case replayer (7). When the repair phase is complete the framework returns a test suite containing only feasible test cases.

There are three points in this framework that can be fit into by difference implementations. We use two plug-ins to achieve 1) the initial covering array generation and 2) test case replaying. The third point is the component for the repair algorithm. For our current instantiation of this framework, we use a simulated annealing algorithm developed by Garvin et al. [9] for the covering array generation and a modified version of GUITAR [10] for test case execution. We have modified GUITAR by adding exception handling that detects when events become unavailable during replay and

to report that the test case is infeasible and at which point in the sequence.

4.3.1 Discussion on Implementation

We made several design decisions for the implementation of the framework and discuss these below.

Test Case Execution: Usually, the test case replayer needs to run a set of test cases for the repair algorithm at one time. To improve the performance of execution, the framework is designed to run test cases in parallel. In the base component interface, one guard thread is created for the process of running each test case. The thread starts a process for the replayer to run the test case, monitors the running status, and finally collects and reports the results. Here we choose to run test cases in separated processes because running different test cases in the same process may unexpectedly change program states. For example, if two test cases are run within the same process, the global states (such as static variables) of the AUT may be changed after the first test case finishes, and since the global states are accessible by any threads within the process, the execution of the second test case is very likely to be affected. An extreme example is that if there is a `System.exit(0)` in an event handler of the AUT and the first test case contains an event processed by this event handler, the process might terminate prematurely after it reaches that event without running the second test case. Although techinques might be used to deal with these issues (such as state recovery, etc.), in order to keep the tool simple and clean, we decide to run test cases on the AUT in separated processes. Since we assume the behavior of the GUIs we test is deterministic, which means each run of the same test case under the same context must give identical results, we keep a list of executed test cases in the base component interface to avoid running the same test case twice.

Execution Result Collection: Because we run the test cases in separated processes, the way our framework communicates with the test case replayer turns to communication between processes. As a result, the test case replayer needs to support this feature to pass results into our framework. Files are used for this purpose. The test case replayer outputs information and results of the execution into a file before it ends, and the guard thread for each test case read in the content of the file for analysis and use in the repair algorithm. A unique path is generated for the result file of each test case so that no result overwrites others.

Recognition of Feasibility: In the results of execution, the test case replayer is required to provide the feasibility of the test case run. If an event in the event sequence is infeasible, we can discover that by checking the accessibility and certain properties of the related components. For example, if a window or a component is not accessible because it has disappeared, or it has not been created, or it cannot get the focus of current GUI, the events on them cannot be executed successfully, and therefore these events are infeasible. Likewise, if a button is disabled, the event clicking on it cannot be executed, and we can call `isEnabled()` on the button to see if it can be executed. The method `isVisible()` is also used for such checking. Before the execution of an event, related properties are checked to see whether it can be successfully executed. If not, the test case is infeasible. At the same time the failure point where the test case stops is also recorded.

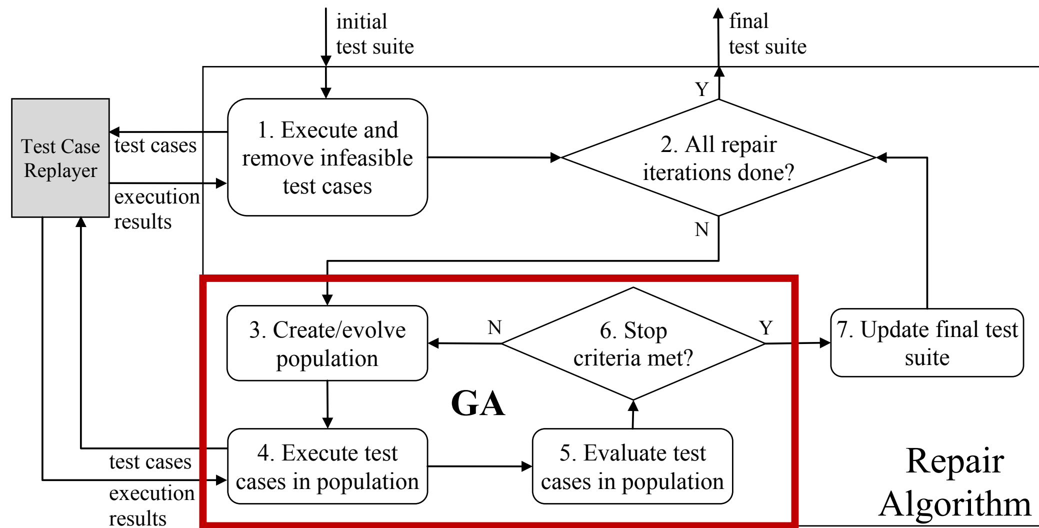


Figure 4.2: Using Genetic Algorithm as Repair Algorithm

4.4 A Genetic Algorithm for Repairing GUI Test Suites

The core part of our framework is the repair algorithm. Figure 4.2 shows an overview of our repair algorithm ((6) in Figure 4.1). Algorithm 2 shows the details for the repair algorithm. While different types of algorithms for repair are possible, in this work, we have used a genetic algorithm (shown in Algorithm 3) because the problem of repair is an optimization problem; we want to generate a minimal set of new test cases that complete the feasible coverage.

First, we walk through Algorithm 2. The first part of the repair is to execute the initial set of test cases and remove any tests that fail due to constraints (step 1, lines 4–5[1]). If all test cases are feasible (step 2, lines 6–8), it exits. If there are any infeasible ones, it begins the repair phase. We set a number of iterations for our algorithm and for each iteration the algorithm adds at most one test case. The number of iterations chosen is based on an estimate of how large we will allow the

[1]We show the steps in figures and lines in algorithms together.

Algorithm 2 GUI Test Suite Repair Algorithm

Require: M: an event-flow model
Require: G: a GUI structure model
Require: t: the strength for the test cases
Require: s: the size factor
Require: k: the length of test cases
Ensure: A GUI test suite T

```
1: E ← {event e | e ∈ M} {Set of all events}
2: Generate a covering array C_I = CA(N; t, k, |E|)
3: Generate the initial test suite I using C_I
4: Execute each test case in I with the help of G to evaluate feasibility
5: T ← {feasible test cases in I}
6: if T = I then
7:    return T
8: end if
9: Rounds ← s · |I| − |T|
10: for i ← 1 to Rounds do
11:    best = Invoke the genetic algorithm with I.
12:    if best ≠ NULL then
13:       T ← T ∪ {best}
14:    end if
15: end for
16: return T
```

repaired test suite to grow. It is set to the maximum number of test cases that can be added (line 9). The constant s is the **size factor**, and typically it is greater than one. Because we can add at most one test case per iteration, the size factor indicates how many times of the size of the initial test suite the final test suite can grow to. For example, if the size of the initial test suite is 10, the size factor 2, and 3 feasible test cases are already added into the final test suite, then we may use at most 17 iterations to complete the final test suite.

For each iteration we run the genetic algorithm (steps 3–6, line 11). The algorithm returns the best test case and adds this to the final test suite. It is possible that the genetic algorithm does not converge in some iterations on a test case that increases coverage. In this case no test cases are added and the final test suite will be smaller

Algorithm 3 Genetic Algorithm for Adding a Single Test Case

Require: $P_{initial}$: a GUI test suite
Ensure: A feasible test case with the best new coverage, or NULL

1: $Population \leftarrow P_{initial}$
2: $nGens \leftarrow 0$ {Experienced number of generations}
3: $nBadMoves \leftarrow 0$ {Experienced number of bad moves}
4: $LastBest \leftarrow -\infty$ {Best fitness from last generation}
5: **while** $nGens <$ MAX_GENS and $nBadMoves <$ MAX_BAD_MOVES **do**
6:   Execute each test case in *Population* and calculate fitness
7:   Sort *Population* in the ascending order of fitness from the best to the worst
8:   $best \leftarrow$ best test case of *Population*
9:   **if** fitness($best$) is not better than $LastBest$ **then**
10:     $nBadMoves \leftarrow nBadMoves + 1$
11:   **else**
12:     $nBadMoves \leftarrow 0$
13:   **end if**
14:   $LastBest \leftarrow$ fitness($best$)
15:   Crossover($Population$)
16:   Select($Population$)
17:   Mutate($Population$)
18:   $nGens \leftarrow nGens + 1$
19: **end while**
20: $best \leftarrow$ best test case of *Population*
21: **if** fitness($best$)> 0 **then**
22:   return *best*
23: **else**
24:   return NULL
25: **end if**

than the maximum possible size (line 12–14).

Next, we discuss the genetic algorithm designed for the repair. Algorithm 3 shows the detailed genetic algorithm for searching for one test case. It accepts an initial GUI test suite as the initial population, and returns the best feasible test case if any is found in the search. In the algorithm, lines 1–4 do the initialization. The loop from line 5 to 19 is the main part for the evolution. In the loop, it first executes all the test cases in the current population (line 6) and rank them according to the fitness calculated (line 7). If the fitness of the best test case in the population is not better

than that of the one in the previous population, the current generation is considered as a bad move (line 8 –14). After that, crossover and mutation are done to generate the next population, and selection is used to eliminate worst individuals in the current population (lines 15–17). When the stopping criteria are met, the evolution stops. If the best individual in the final population is good enough, then return it as the test case to add; otherwise, NULL is returned to indicate no good enough test case is found for this iteration (lines 20–25). We discuss the parameters used in the genetic algorithm next.

Chromosome and Population: The chromosome for this algorithm is a test case (or an event sequence), where the alleles are the events that form the sequence. The population is a list of test cases. The initial population is a set of test cases generated randomly.

Stopping Criteria: We use three criteria as our stopping criteria. First, a maximum number of generations is used to ensure that the algorithm always stops. Second, a maximum number of bad moves helps predict whether the algorithm has converged. If the best fitness of current population is worse than that of the previous one, it is considered a bad move. Third, if the best test case of the population already covers the maximum number of t-way combinations one test case can cover, the algorithm stops. It is worth noting that in real applications, we do not know the exact constraints ahead of time so we cannot know whether a test case completes 100 percent of feasible coverage; therefore, in practice, this will not be a realistic stopping criterion.

Fitness Function: We consider two factors in the fitness function. One is the feasibility of the test case, and the other is the new coverage a test case can contribute based on the coverage already achieved. The feasibility information is achieved through test case execution. We define the failure point of an execution for a test

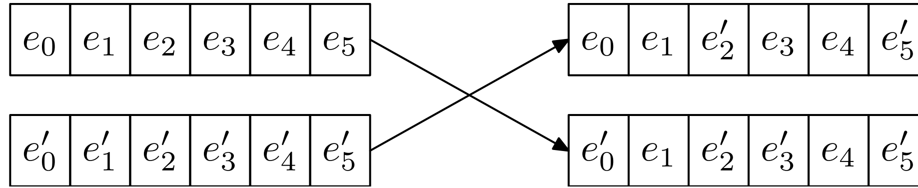


Figure 4.3: Crossover. 2-sets e_1 and e_4 at position 1 and 4 and e'_2 and e'_5 at position 2 and 5 are the new coverage of these two test cases respectively. During crossover, the new coverage of the two test cases are exchanged.

case to be the position of the event which is not successfully executed. If a test case is feasible, its failure point is equal to the length of the test case. For example, the failure point of the test case (e_1, e_2, e_3) which fails at e_2 is 1. Given a length-k test case c, the failure point is $f(c)$, and the number of newly covered t-sets that c contributes is $\texttt{cov}(c)$, the fitness function is defined as

$$fitness(c) = b \cdot \texttt{cov}(c) - p \cdot (k - f(c)),$$

where b and p are both non-negative numbers. The factor b assigns a bonus to new t-way combinations which can be introduced by c. The definition of $\texttt{cov}(c)$ in the fitness function can be adapted to other coverage criteria to generalize the framework for test suites other than those derived from CIT (i.e., $\texttt{cov}_{t,k,v}(c)$). The factor p is used to penalize infeasibility. In our implementation $b = 10$ and $p = 100,000$. This very large value of p ensures that infeasible test cases are more likely to be thrown out. In fact, we could have used a binary value for p, i.e., *feasible/infeasible*. However, we have left this generic for now.

Crossover and Mutation: For crossover we rank the test cases in descending order and pair consecutive chromosomes. The chromosomes are crossed over in pairs. For example, if the population contains test cases 1, 2, 3, 4, 5 and 6 in order, the pairs for crossover are 1 and 2, 3 and 4, and 5 and 6. For each pair, the events providing

new coverage of the test cases are exchanged. Figure 4.3 shows the crossover for two test cases $e = (e_0, e_1, \ldots, e_5)$ and $e' = (e'_0, e'_1, \ldots, e'_5)$. The new t-sets they can cover are e_1 and e_4 at position 1 and 4, and e'_2 and e'_5 at position 2 and 5 respectively. The crossover exchanges the two t-sets to generate two new test cases (the right part of the Figure 4.3). After this crossover, we can see both of the new test cases have these two t-sets, and are better than the original ones. The two new test cases are added into the current population for selection. It should be noted that when there are many new t-sets in the original test cases, the exchange may overwrite some of them. However, because we do not drop the original test cases, we will lose the new coverage discovered in the crossover.

Mutation ensures diversity in our population. Given a mutation rate m_r, the number of events mutated is calculated against the total number of events in the current population p using $m_r \times p$. The positions to mutate are chosen randomly from all of the chromosomes. Events in these positions are replaced by an event that is randomly chosen respectively.

Selection: We use a linear selection, picking the best S chromosomes where S is the population size.

Final Test Suite: After the stopping criteria is met, the evolution of test cases ends and the one with highest fitness is returned. Because we impose a large penalty for infeasible test cases, they will have a negative fitness value. Therefore, if the fitness of a test case is zero or positive, it is feasible. However, if the fitness is zero, it contributes no new combinations. As a result, we only add test cases with a positive fitness to the suite; if there are no test cases with a positive fitness in an iteration, none are added.

4.5 Evaluation

We have designed a set of experiments to determine the feasibility of our framework. Although our ultimate goal is to apply this to large scale GUI software, we have chosen to first experiment on a set of synthetic subjects. The advantages of this approach are that we can control the types of constraints, we know the target feasibility for coverage, and we do not suffer from any non-determinism or native faults that might appear in a real environment. This will allow us to evaluate the potential effectiveness and performance of our approach in isolation before moving to real subjects. The evaluation presented here is not trivial, however. The experiments constitute a total effort of 363 machine-days of computational time.

We have developed three research questions that we aim to answer in this study:

- RQ1: Can the framework generate test sequences that increase feasible coverage using a genetic algorithm?
- RQ2: How does a genetic algorithm compare with a random approach?
- RQ3: How does the framework scale to longer test sequences?

4.5.1 Subjects

We have created seven subject programs that contain the types of constraints described in Section 3. Table 4.1 describes the details of each program. The programs are written in Java and each event is of the type *Button Click*, with no functionality other than the enabling/disabling of other events defined by the specified constraints. The first six programs each contain a single constraint. The last program (Comp) contains a combination of constraints taken from three of other subjects. The first four programs, (Disb, Reqs, 2Cons, 2Excl), each contain three events. The next two,

Table 4.1: Subject programs 1

No.	Full Name	No. Events	Abbreviated Name	Constraint Description
1	Disabled Event Constraint	3	Disb	One event is always disabled
2	Requires Constraint	3	Reqs	One event requires another event to be executed before it
3	Consecutive Constraint (2-way)	3	2Cons	A pair of events is infeasible when executed sequentially
4	Excludes Constraint (2-way)	3	2Excl	A pair of events is infeasible if executed (possibly non-consecutively) in sequence
5	Event Consecutive (3-way)	4	3Cons	A sequence of three events is infeasible when executed
6	Excludes (3-way)	5	3Excl	A (possibly non-consecutive) sequence of three events are infeasible
7	Compound Constraints	5	Cmpd	Includes constraints found in Subject 2, 3 and 5

(3Cons, 3Excl), contain four events and the last program, (Comp), contains five. The additional events in the last three programs allow for the more complex/longer constraints to be defined. Since constraints may be of differing arity, for the consecutive and excludes constraints we have included two versions. One has constraints between only two events (2Cons and 2Excl) and the other has 3-way constraints (3Cons and 3Excl).

4.5.2 Independent Variables

Our independent variables are the seven subject programs, the length of the test sequences and the target coverage of our test suites, defined by the CIT sample strength (e.g., 2-way, etc.). For 2-way coverage we run only one of the 3-way constrained programs (3Cons) since the 3-way constraints will not reduce the coverage of any pairs of events, although they may still render certain test cases infeasible. For the 3-way coverage we drop Disb, the subject with only one disabled constraint, since we expect very similar behavior as is seen in the 2-way coverage; a single event is removed from the pool. Our preliminary results confirm these observations.

4.5.3 Dependent Variables

We examine three aspects of the repair to determine its success and quality. The first metric is concerned with coverage of the test suites after repair. We calculate the number and percentage of t-sets (pairs or triples of events) that are feasible given the program constraints. We examine both the original test suites and the repaired ones and compute the increase in CIT coverage as the increase in the ratio of feasible t-sets divided by covered t-sets. Second we quantify the size of the original and final (repaired) test suite and calculate the percentage increase measured by the number and percentage of new test cases. Finally we consider both the time to execute and the number of test cases executed during the repair of the algorithm. For this metric we do not consider running the original test suite since that is a constant factor in our experiments.

4.5.4 Experimental Methodology

The experiments are carried out on a computing cluster with AMD 2.4GHz dual-core 64-bit processors, 16GB shared memory, Linux 2.6.18, and Java 1.6. For each experiment, we ran five trials to reduce bias due to the randomness in our algorithms. We report averages of the results.

Random Algorithm: For RQ2 we developed a random algorithm to gauge the difficulty of incidentally covering all feasible t-sets. We use the maximum test suite size that is used for the genetic algorithm and then try to iteratively generate a set of random test cases of that size. At each stage we keep the set of test cases that gives us the highest new coverage. Since we expect that some test cases will be infeasible using random generation, we calculate coverage for *all* test cases, regardless of feasibility. If a test case is infeasible we calculate the coverage up until the point where it failed,

giving preference to the random algorithm. (The genetic algorithm will discard the coverage for the entire test case). At the end of all iterations, the random algorithm returns the set of tests cases that cover the greatest number of new t-sets.

Algorithm Parameters: In our implementation of the genetic algorithm we use the following parameters. We derived these heuristically, but leave a systematic tuning of the algorithm for future work. The maximum number of generations for the genetic algorithm is 10^6, the maximum number of consecutive bad moves is 100, the population size is set at 100 and the mutation rate is set to 0.03. For both the genetic algorithm and the random algorithm we set the size factor to be 1.5 for 2-way coverage, and 1.3 for 3-way coverage. For the random algorithm we set the number of iterations to 10^6. The timeout for the random algorithm is approximately 1.5 as long as the genetic algorithm on average giving the random algorithm more time to obtain new coverage when competing with the genetic algorithm.

4.5.5 Threats to Validity

We describe the main threats to validity that we have identified. First we wrote the programs that are used for experimentation and seeded the constraints. While a threat, we believe that these are realistic small samples of the types of infeasibility that are seen in practice. We have used a small number of events in each program (3-5), and these events do not contain any real functionality. However, in some systems where we group interacting events in an ESIG we think that the number of events may be realistic. We also believe that the lack of functionality provides better determinism in the test harness since we do not have to contend with problems related to thread ordering during replay. We have used a single set of parameters for the genetic algorithm and random algorithm which may impact their final results. We wrote many programs to implement this framework and cannot be one hundred percent

certain that they are fault free, but we have validated our results with a different set of tools and have made every attempt to confirm that the numbers are reported correctly. Finally, we realize that there are other metrics that we may have collected, we feel that coverage, execution time, and test suite size are a legitimate starting set for this work.

4.6 Results

We examine the results for each research question next[2].

4.6.1 RQ1: Framework Effectiveness

Table 4.2 shows the results for repairing test suites with length-5 test cases. For each subject we provide the number of t-sets in the original model, followed by the number that are feasible given the constraints. We then show the average initial size of the test suite, the average number of feasible test cases from within that test suite and the final size followed by the percentage increase. We then show the initial, final and average final coverage for the repaired test suite using our genetic algorithm. The last column shows the percentage increase in coverage. We point out a few results from this table. First, in all subjects except the last, we reach 100% feasible coverage. The last subject has a goal of 3-way coverage and has compound constraints. We see a range of increased coverage from 4.0% in the 3-way, 3Excl subject to 233% for 2-way coverage the Disb subject. This subject has one event that is always disabled therefore its initial test suite had only a single feasible test case.

[2] Full experimental results and artifacts are available for download at: `http://www.cse.unl.edu/~myra/artifacts/icst2010`

Table 4.2: Repaired test suites with length-5 test cases (average of five runs)

Parameters and Target Coverage				Avg. Size				Avg. Final Coverage				
Strength (t)	Subject	Total t-sets	Feasible t-sets	Init. Size	Feasible Test Cases	Final Size	%Size Increase	Init. Cov.	%Init. Cov.	Final Cov.	%Final Cov.	%Cov. Increase
2-way	Disb	90	40	11.0	1.2	6.0	**-45.5%**	12.0	30.0%	40.0	100.0%	233.3%
	Reqs	90	77	11.0	6.2	11.2	1.8%	54.8	71.2%	77.0	100.0%	40.5%
	2Cons	90	86	11.0	5.2	13.4	21.8%	48.4	56.3%	86.0	100.0%	77.7%
	2Excl	90	80	11.0	5.2	13.8	25.5%	47.8	59.8%	80.0	100.0%	67.4%
	3Cons	160	160	16.0	15.2	17.6	10.0%	152.0	95.0%	160.0	100.0%	5.3%
	Cmpd	250	223	25.8	10.8	33.8	31.0%	106.6	47.8%	223.0	100.0%	109.2%
3-way	Reqs	270	206	33.0	17.8	34.0	3.0%	154.4	75.0%	206.0	100.0%	33.4%
	2Cons	270	234	33.0	19.0	40.2	21.8%	169.0	72.2%	234.0	100.0%	38.5%
	2Excl	270	200	33.0	14.0	31.6	**-4.2%**	126.0	63.0%	200.0	100.0%	58.7%
	3Cons	640	637	64.0	61.0	69.2	8.1%	610.0	95.8%	637.0	100.0%	4.4%
	3Excl	1250	1240	153.0	144.0	163.4	6.8%	1192.0	96.1%	1240.0	100.0%	4.0%
	Cmpd	1250	987	153.0	68.0	177.6	16.1%	598.0	60.6%	985.0	99.8%	64.7%

For each subject we compare the size increase of the final test suite. Although we provide an upper bound for our final test suite that is as high as 1.5 times that of the original, we see that in five experiments the number of test cases increases by less than 10%. In two cases (bold) we have reduced the size of the test suite from the original size. This is due to constraints that remove a large number of feasible combinations. For the other six subjects we see a range of increased sizes but only one reaches the maximum (2-way coverage with compound constraints).

From this data we answer RQ1 by concluding that we can increase feasible coverage with our approach.

4.6.2 RQ2: Comparison with a Random Algorithm

We now compare the results of our genetic algorithm against a random algorithm. We do this to validate the need for a guided search and to infer the difficulty of the problem. The results of RQ2 are shown in Table 4.3.

In this table we first show the subject parameters and target coverage for length-5 and length-10 sequences. We show the space size of each subject which represents the total number of unique sequences in the search space. We then show the total number of t-sets and the number of feasible ones. The next set of columns provides data from the covering array before repair. The last two sections show data first for the random algorithm and then for the genetic algorithm after repair. We present data for the final size of the test suite, the percentage size increase, the final missed coverage (represented as the number of t-sets). We then show the percentage of target feasible coverage for each problem. Finally, we show the number of test cases executed during repair and the time in minutes(m) hours(h) and days(d).

We show the final coverage percentage for the genetic algorithm in bold when it

Table 4.3: Comparison of random and genetic algorithms (execution time in minutes (m), days (d) or hours (h))

Parameters and Target Coverage						Before Repair					After Repair											
						Covering Array					Random Algorithm						Genetic Algorithm					
Strength (t)	Subject	Length	Space Size	Total t-sets	Feasi. t-sets	Init. Size	Feasi. T. C.	Init. Cov.	Init. Missed	%Init. Cov.	Final Size	%Size Inc.	Final Missed	%Final Cov.	No. Executed	Time	Final Size	%Size Inc.	Final Missed	%Final Cov.	No. Executed	Time
2-way	Disb	5	3^5	90	40	11.0	1.2	12.0	28.0	30.0%	17.0	54.5%	0.0	100.0%	243.0	16.68m	6.0	-45.5%	0.0	100.0%	230.6	10.13m
		10	3^{10}	405	180	15.0	0.2	9.0	171.0	5.0%	23.0	53.3%	53.2	70.4%	5990.2	8.34h	9.0	-40.0%	0.0	**100.0%**	7745.6	6.02h
	Reqs	5	3^5	90	77	11.0	6.2	54.8	22.2	71.2%	17.0	54.5%	5.0	93.5%	238.6	16.76m	11.2	1.8%	0.0	**100.0%**	228.2	10.28m
		10	3^{10}	405	377	15.0	8.0	275.4	101.6	73.1%	23.0	53.3%	20.8	94.5%	5803.6	8.35h	17.2	14.7%	0.0	**100.0%**	7548.2	5.98h
	2Cons	5	3^5	90	86	11.0	5.2	48.4	37.6	56.3%	17.0	54.5%	0.0	100.0%	205.0	16.68m	13.4	21.8%	0.0	100.0%	236.0	10.08m
		10	3^{10}	405	396	15.0	5.2	203.0	193.0	51.3%	23.0	53.3%	43.0	89.1%	6940.8	9.73h	17.2	14.7%	0.0	**100.0%**	9115.8	7.20h
	2Excl	5	3^5	90	80	11.0	5.2	47.8	32.2	59.8%	17.0	54.5%	1.8	97.8%	234.0	16.69m	13.8	25.5%	0.0	**100.0%**	236.6	10.46m
		10	3^{10}	405	360	15.0	1.8	75.4	284.6	20.9%	23.0	53.3%	85.4	76.2%	8766.0	12.51h	20.6	37.3%	0.0	**100.0%**	10917.4	8.62h
	3Cons	5	4^5	160	160	16.0	15.2	152.0	8.0	95.0%	24.0	50.0%	1.0	99.4%	285.6	16.73m	17.6	10.0%	0.0	**100.0%**	185.4	7.82m
		10	4^{10}	720	720	25.8	22.0	661.6	58.4	91.9%	39.0	51.2%	7.0	99.0%	5778.6	8.35h	28.6	10.9%	0.0	**100.0%**	7038.0	4.77h
	Cmpd	5	5^5	250	223	25.8	10.8	106.6	116.4	47.8%	38.0	47.3%	46.4	79.2%	2886.2	4.17h	33.8	31.0%	0.0	**100.0%**	2604.8	2.13h
		10	5^{10}	1125	1068	38.8	11.6	449.4	618.6	42.1%	57.0	46.9%	209.6	80.4%	52680.0	3.47d	44.8	15.5%	0.0	**100.0%**	55351.6	1.97d
3-way	Reqs	5	3^5	270	206	33.0	17.8	154.4	51.6	75.0%	43.0	30.3%	11.4	94.5%	227.7	20.01m	34.0	3.0%	0.0	**100.0%**	231.4	11.89m
		10	3^{10}	3240	2891	54.0	26.0	2203.0	688.0	76.2%	71.0	31.5%	180.6	93.8%	25607.3	1.45d	59.4	10.0%	0.0	**100.0%**	20806.2	20.67h
	2Cons	5	3^5	270	234	33.0	19.0	169.0	65.0	72.2%	43.0	30.3%	25.2	89.2%	201.0	20.21m	40.2	21.8%	0.0	**100.0%**	232.8	12.04m
		10	3^{10}	3240	3024	54.0	18.8	1682.2	1341.8	55.6%	71.0	31.5%	429.0	85.8%	32428.0	1.74d	65.8	21.9%	0.0	**100.0%**	25473.4	1.08d
	2Excl	5	3^5	270	200	33.0	14.0	126.0	74.0	63.0%	43.0	30.3%	21.4	89.3%	198.0	20.39m	31.6	-4.2%	0.0	**100.0%**	241.0	11.70m
		10	3^{10}	3240	2400	54.0	6.0	640.0	1760.0	26.7%	71.0	31.5%	624.4	74.0%	28468.7	1.51d	59.8	10.7%	0.0	**100.0%**	22513.2	22.94h
	3Cons	5	4^5	640	637	64.0	60.8	608.0	29.0	95.4%	84.0	31.3%	9.4	98.5%	318.3	33.50m	69.2	8.1%	0.0	**100.0%**	347.0	17.04m
		10	4^{10}	7680	7672	132.0	117.8	7325.2	346.8	95.5%	172.0	30.3%	115.4	98.5%	43594.7	2.14d	145.0	9.8%	0.0	**100.0%**	32219.0	1.29d
	3Excl	5	5^5	1250	1240	153.0	144.0	1192.0	48.0	96.1%	199.0	30.1%	19.8	98.4%	925.3	1.67h	163.4	6.8%	0.0	**100.0%**	966.8	57.63m
		10	5^{10}	15000	14880	261.0	173.8	12412.4	2467.6	83.4%	340.0	30.3%	900.0	94.0%	59586.0	1.51d	305.4	17.0%	0.0	**100.0%**	221286.4	13.93h
	Cmpd	5	5^5	1250	987	153.0	68.0	598.0	389.0	60.6%	199.0	30.1%	205.6	79.2%	3066.7	5.56h	177.6	16.1%	2.0	**99.8%**	3054.4	3.54h

exceeds that of the random algorithm. There are only two cases where this does not occur. The first is for the first subject, of length 5, where one event is disabled. Both the genetic algorithm and the random algorithm reach 100% coverage on average. The second case occurs in the length-5 test sequences for 2Cons where both algorithms again reach 100% coverage.

To examine the coverage further we graph the percent coverage in the initial test suite, and after repair for both the random algorithm and the genetic algorithm. We show this data in Figures 4.4 and 4.5. The `x-axis` shows the subject and length while the `y-axis` shows the percent coverage. We can see that in all cases both of the repair algorithms improve coverage, but that the genetic algorithm outperforms the random algorithm in most subjects.

We next look at the size of the final test suites. The random suites are consistently larger than the genetic algorithm. This is not unexpected given our implementation, but even in cases such as 2Excl, 3-way which has a 4.2% reduction or a 10.7% increase (for length-5 and length-10 sequences respectively) for the genetic algorithm, a 30% increase in test cases does not necessarily improve coverage. The random suites, with up to 35% more test cases, have only 89% and 74% coverage compared to 100% for the genetic algorithm.

Finally we examine the run time and number of executed test cases. Since we set the timeout of the random algorithm to be approximately 1.5 times of the time used by the genetic algorithm, the repair time for each of the groups using the random algorithm is longer than that using genetic algorithm. However, we can see that except Disb and 2Cons, 2-way for length 5, all the groups using the random algorithm have a lower final coverage than the genetic algorithm.

From this data we answer RQ2 by concluding that the genetic algorithm outperforms the random algorithm.

4.6.3 RQ3: Scalability of the Genetic Algorithm

Our last research question examines scalability. To answer this, we examine length 15 and 20 sequences. We only show data for 2-way coverage due to resource limitations. The results of this experiment are shown in Table 4.4. In this table we show the coverage of pairs of events, the number of executed test cases and the time in hours and days. We show missing coverage in bold. As can be seen, we have achieved 100% coverage in all cases but two. Both cases of missing coverage occur on the 3Excl subject where we have to exclude any combination of a specific 3-event sequence. Although we have not achieved 100% coverage we are only missing on average 3.0 and 9.0 pairs respectively for length 15 and 20 which is a minor percentage of the final coverage. We have also increased our original coverage by more than 90% since none of the initial test cases ran to completion. The time data is not as encouraging. The shortest running repair takes slightly less than one day to complete, while most repairs take from one to two days of computational time. The longest running repair, the compound constraint of length 20 takes almost 3 weeks to converge.

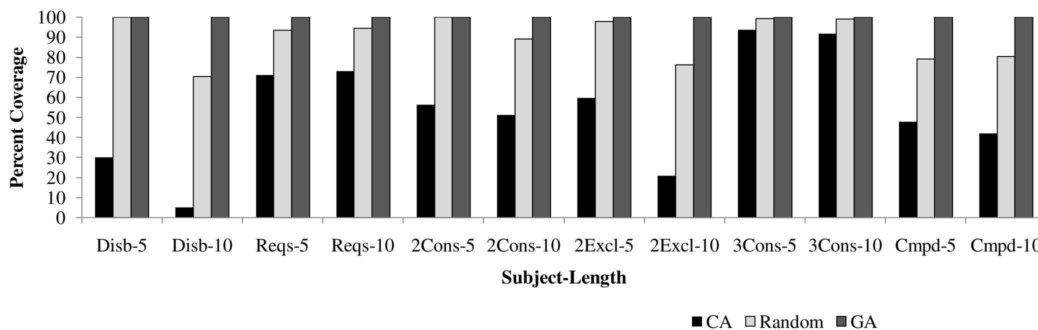


Figure 4.4: Comparison of Coverage for 2-way Criteria

We examine this further in Figure 4.6. In the two graphs we plot first the number of executions and then the time in hours for each program. We examine data for length 5, 10, 15 and 20. The curves of the lines are similar between graphs, indicating that

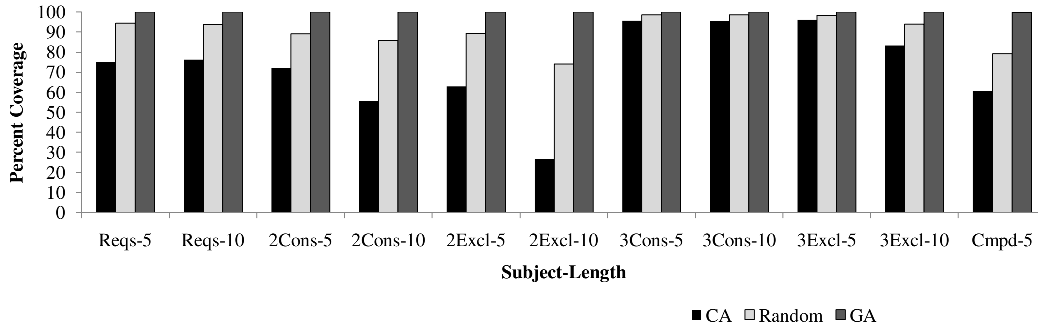


Figure 4.5: Comparison of Coverage for 3-way Criteria

the overriding factor in execution time is the number of test cases executed. This is consistent with other research that points out that setup time to execute a test case is more important than the length of the sequence [38]. These programs have dummy events so this may not always be true in real systems. More notably, these graphs point out that there is a big jump in both executions and run time moving from length 10 to 15.

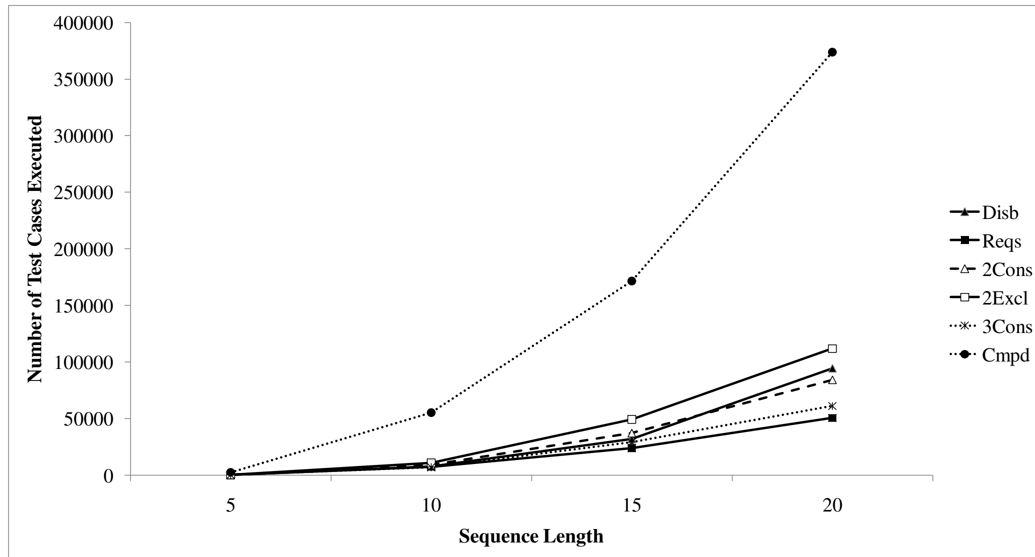
This data leads us to answer RQ3 as follows. From the perspective of coverage the algorithm appears to scale well. However, execution time does not. We believe that optimizations and other heuristics to terminate the genetic algorithm and to tune the parameters are needed before this can be applied to large systems. We believe that an adaptive method that repairs test suites incrementally during testing may be effective.

4.7 Summary

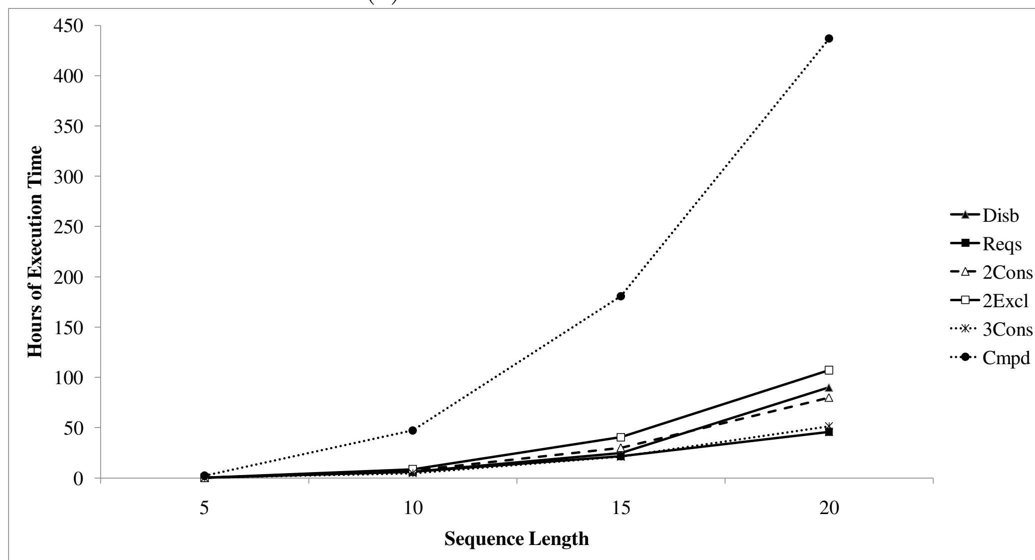
In this chapter, we presented a framework for GUI test suite repair. We discussed the process and algorithms used for the repair in detail. An evaluation was also conducted for our tool implementation for the framework. The results show that our technique can repair the GUI test suite for the synthetic programs. When compared

Table 4.4: Repair for length-15 and -20 test cases; 2-way criteria (execution time in days (d) or hours (h))

Subject	Length	Total t-sets	Feasible t-sets	Init. Cov.	Init. Missed	Final Cov.	Final Missed	% Final Cov.	No. Executed	Time
Disb	15	945	420	0.0	420.0	420.0	0.0	100.0%	31978.4	1.04d
	20	1710	760	0.0	760.0	760.0	0.0	100.0%	94409.6	3.75d
Reqs	15	945	902	660.6	241.4	902.0	0.0	100.0%	24016.8	21.43h
	20	1710	1652	1212.8	439.2	1652.0	0.0	100.0%	50657.4	1.91d
2Cons	15	945	931	197.8	733.2	931.0	0.0	100.0%	37173.2	1.25d
	20	1710	1691	292.6	1398.4	1691.0	0.0	100.0%	84244.8	3.33d
2Excl	15	945	840	42.0	798.0	837.0	**3.0**	**99.6%**	49363.2	1.69d
	20	1710	1520	0.0	1520.0	1511.0	**9.0**	**99.4%**	111919.4	4.47d
3Cons	15	1680	1680	1482.8	197.2	1680.0	0.0	100.0%	29131.6	21.04h
	20	3040	3040	2697.6	342.4	3040.0	0.0	100.0%	61170.6	2.14d
Cmpd	15	2625	2538	829.6	1708.4	2538.0	0.0	100.0%	171514.4	7.53d
	20	4750	4633	1200.2	3432.8	4633.0	0.0	100.0%	373747.8	18.21d



(a) Number of executions



(b) Time for GA in hours

Figure 4.6: Comparison of Numbers of Test Case Executions and Execution Time for GA

to a random algorithm, we demonstrate that the genetic algorithm is necessary and can achieve better results for GUI test suite repair. In the next chapter, we optimize our algorithm for the application to non-trivial GUIs.

Chapter 5

Application to Non-Trivial GUIs

In this chapter, we apply our framework to a few non-trivial GUIs used in previous work applying CIT to GUI test generation. These non-trivial GUI subjects contain more constraints than the synthetic programs, and the event interactions and states in these applications are more complicated than those in the synthetic programs. For example, a test case for a non-trivial application may contain a combination of 3-way, 4-way and 5-way constraints. When a 2-way criterion is used, all the 2-sets in the test case are feasible, however, the test case will still not be selected because the combination of the above constraints in it make it infeasible. This shows that in non-trivial applications the search might be more likely to encounter infeasible test cases due to the interference of constraints, so it may need to cover a larger space in order to find feasible test cases. Although such situations can also be found in the synthetic programs, it is more prevalent in these non-trivial GUIs. Also, the execution of non-trivial GUIs are more difficult to control than that of the synthetic programs. In the synthetic programs, the button clicks do nothing, while in the non-trivial GUIs, it may take a relatively long time to process a button click event, etc. Therefore, the synchronization for the execution of the events need to be carefully considered,

otherwise the states of the GUIs may err. Moreover, more GUI components (such as tabs, tables, etc.) need to be supported for the execution of test cases. In this chapter, we analyze the bottleneck of the algorithms based on the observation from last chapter, and figure out several methods to overcome it. After that, we try an optimized algorithm on several non-trivial GUI subjects. The evaluation shows that although our technique still cannot escape from the exponential complexity in the worst case, it is able to repair the test suites for these GUI subjects with reasonable efficiency.

5.1 Optimizing the Algorithm

From our evaluation in Chapter 4, our algorithm suffers from scalability problems. The time for the genetic algorithm is the main contributor of the total time for the GUI test suite repair. Naturally, a subsequent question is: what is the main contributor to the time for genetic algorithm? A brief study showed that the time used for the execution of test cases exceeds 90% of the total time for the genetic algorithm. Table 5.1 shows the results for two groups of the subject “Cmpd”. To shorten the time for the entire execution, we set a timeout for each execution. Our tool checks if it is timed out at the beginning of each round. Because it always finishes the ongoing round before each check, it might take a longer time than the specified timeout. Because the test cases are run in parallel rather than back to back, the sum of the time for the execution of individual test cases is not the total execution time. Instead, we time the execution of groups of test cases run at one time separately, and add of all the times for groups together to achieve an estimation of the time for the execution of test cases.

Compared to the time for the execution of GUI programs, the manipulation of

Table 5.1: Time for the execution of test cases occupies most of the time for the genetic algorithm. For each group in the table, population size is 100; maximum number of consecutive bad moves is 100; size factor is 1.5; number of test cases that run in parallel is 3. $time_{exe}$ is the time for the execution of test cases; $time_{genetic}$ is the time for the genetic algorithm. Time is in milliseconds.

Strength	Length	Timeout	$time_{exe}$	$time_{genetic}$	$\frac{time_{exe}}{time_{genetic}}$
2-way	20	5 days	437131401 (5.06 days)	444592413 (5.15 days)	98.3%
3-way	10	7 days	590403510 (6.83 days)	634244802 (7.34 days)	93.1%

the data structures in memory for the genetic algorithm only counts for a very small part. This is because running the GUI test cases usually requires initialization of the display and settings, which costs a relatively long time (usually in seconds rather than milliseconds) than performing events and terminating themselves.

Since the bottleneck is found, the main goal for optimizing the genetic algorithm is to reduce the time used on the execution of test cases. There are several directions that we can take to reach this goal, and we believe the following three would be good ones. First, if the number of test cases to execute is reduced, the time for the execution will be reduced. Second, if the time for the execution of individual test cases can be reduced, the total time for the execution will also be reduced. In [38], it is found that the time for the setup of the execution of the test cases takes a large portion of the total execution time, and therefore reducing time for these may be helpful. Third, we can adjust the parameters of the genetic algorithm to make the criteria for the search slightly weaker, so that the search may take a shorter time. We discuss each next, but only focus on the first one when applying the optimization since we believe it has the greatest impact. The other two are left as our future work.

5.1.1 Reducing the Number of Test Cases to Execute

The genetic algorithm presented in Chapter 4 tries to find the best test case for each round and adds it into the final test suite. Due to our selection of the parameters for the fitness function, the best test case found has two properties: (1) feasible; (2) with an optimal new coverage. The genetic algorithm first determines whether all the test cases in the current population are feasible by executing all of them, and then ranks them according to their new coverage. This strategy actually executes the test cases that will never be selected (those test cases that are infeasible and with little new coverage). Skipping executing these test cases may help reduce the number of test cases to execute. But we still need to find out the best test case according to the feasibility, which is determined through execution. Picking the best test case with limited execution of test cases is the key of the optimization. The new method is described next.

For each generation, we first calculate the potential new coverage each test case can contribute assuming it is feasible. Because the calculation of the potential new coverage can be fulfilled by checking the covered t-sets, it is much cheaper than the execution. After that, all of the test cases are sorted in decreasing order by potential new coverage, and executed one by one (or several by several) until a feasible one is found. The first feasible test case must be the best one for this generation because it is feasible and has an optimal new coverage in the population of this generation. In this way, we can keep the best test case for each generation, and the best test case in the final generation is treated as the best test case for this round. This new method swaps the two steps in the old method, and uses a selective approach for the execution of test cases.

This improvement only requires a minor modification to the genetic algorithm.

We replace line 6–8 of Algorithm 3 with Algorithm 4. For those test cases that are not executed in one generation, the failure point is set to 0 by default. This means they are given death penalties and will never be selected as the best test case for that round. However, in the selection phase of the genetic algorithm, the "dead" test cases may still be selected for next generation because they might contain feasible t-sets that are not covered. The selection is based on the potential new coverage those test cases can introduce. Although the new method is likely to improve the performance, it should also be noted that when the constraints are complicated, especially when only very few test cases are feasible, it might not outperform the previous genetic algorithm because it may still execute all the test cases in one generation in order to find a feasible one.

Algorithm 4 Improvement on test case execution (replacing line 6–8 of Algorithm 3)

1′: Calculate the potential new coverage for each test case in *Population* assuming it is feasible
2′: Sort *Population* in the descending order of the potential new coverage of the test cases
3′: $best \leftarrow -\infty$
4′: **for** $i \leftarrow 1$ **to** $|Population|$ **do**
5′: Execute the test case c with ith rank in *Population*
6′: **if** c is feasible **then**
7′: $best \leftarrow$ fitness(c)
8′: **break**
9′: **end if**
10′: **end for**
11′: Calculate fitness for each test case in *Population*
12′: Sort *Population* in the ascending order of fitness from the best to the worst

5.1.2 Reducing the Time for the Execution of Individual Test Cases

We examined the main contributor to the time of the execution of the test cases, and found that delays used in the replayer significantly lengthen the time for the execution. The delays are used to give enough time for the execution of the event handlers to finish and for the GUIs to be synchronized. The delays cannot be eliminated in general. For example, if one event hander starts another thread, which subsequently starts even more threads, the time for handling the event might be much longer than the time merely for the execution of the code in the event handler. And it is very hard to the predict the time for the execution because the scheduling for the thread execution can vary for different runs. A possible idea for eliminating the delays is to let the events notify the replayer when they finish (this may require notification chains from the threads and other things started by the event handler), but the GUI libraries we use do not support explicit notification at the end of the execution of the events on the library level, therefore doing this would require modification in the libraries. Another thing to mention is that the execution time for an event is application-specific, because different applications need to perform different computations for handling events. Actually, even events in the same application are very likely to use different execution times. However, we may adjust the delays by studying the applications, and set them to a reasonable value within which most of the events can finish. For exceptional cases, such as the case that the execution of the test cases is lengthened due to unexpected interruption on the system, the test case replayer will rerun the test cases automatically.

While we leave accelerating the execution of individual test cases as our future work, we leverage the execution of test cases in parallel to make full use of the machine.

Because the test case replayer needs to do initialization for the test cases, including I/O and system settings, etc., parallelization can greatly maximize the usage of the machine time. But we need to balance the number of test cases to execute in parallel, because too many outstanding running instances (such as five) may lead to too much competition in the thread and process scheduling.

5.1.3 Tuning the Parameters for the Genetic Algorithm

The third method is tuning the parameters of the gentic algorithm. In the experiments, we found that because we do not know the constraints in the GUIs when generating the test suites, the maximum number of generations and the maximum number of consecutive bad moves serve as the stopping criteria. If the maximum number of generations and the maximum number of consecutive bad moves are decreased, the search may reach the stopping criteria more quickly. However, the selection of these two numbers varies for different applications. For applications with fewer constraints (such as the synthetic programs), these two numbers can be selected smaller, while for others, the numbers need to be larger. If we choose too small numbers for them, the search may not try hard enough so that it may miss a lot of coverage; on the other hand, if we choose too large numbers for them, the search may try too hard so that it takes too long to finish.

The population size also affects the number of test cases to execute. For example, a population size of 50 is more likely to require less test cases to execute than a population size of 100, although sometimes it might not be the case due to generated duplicate test cases (because we only execute a test case once regardless of duplication). The mutation rate may affect the progress of the search as well. Mutation not only improve the variety of the population, it may also destroy the new coverage by

replacing an event with another. In the latter case, the search might be lengthened. We leave all these as our future work for more detailed study.

5.2 Evaluation

We have designed a set of experiments to evaluate our techniques on non-trivial GUIs. A group of GUI subjects that were used to evaluate GUI testing techniques are selected from [39]. These subjects have been used in a lot of research on GUI testing [26, 40, 37, 39], and are well understood. Also, according to the experiments in [39], many of the generated test cases for these groups are infeasible despite careful modeling. This is why we select them to apply our technique to repair the test suites generated for each group in this experiment. Unlike the subjects used in Chapter 4, the subjects here were developed by others, so we do not know the constraints in them. Rather than validating the result with known infeasible t-sets, we derive constraints from the finally uncovered t-sets and validate these constraints on the corresponding GUIs manually.

We answer the following research questions in this study:

- RQ1: Can the repair framework increase the combinatorial coverage of test suites for non-trivial GUIs?
- RQ2: Does test repair scale to non-trivial GUIs with respect to execution time?
- RQ3: Do the discovered patterns match the constraints/patterns in Chapter 3?

5.2.1 Subjects

We select nine groups of six subjects from [39]. The selected subjects include four applications from TerpOffice series and four open source GUI programs from Source-

Table 5.2: Subject programs 2

No.	Program Name	LOC	Group	No. Events	Abbreviated Name	Task Description
1	TerpPaint 3.0	13315	4	11	TPa-G4	Clipboard Operations
2	TerpPresent 3.0	44591	5	14	TPr-G5	Content
3	TerpWord 3.0	22806	1	14	TW-G1	Table Operations
4	TerpSpreadSheet 3.0	6337	1	14	TS-G1	Format Cell
5			5	8	TS-G6	Table Format
6	CrosswordSage 0.35	3220	4	14	CS-G4	Preference Settings
7	FreeMind 0.80	24665	1	11	FM-G1	Map Operations
8			2	18	FM-G2	Format
9			4	10	FM-G4	Clipboard Operations

Forge. As the names indicate, TerpPaint, TerpPresent, TerpWord and TerpSpreadSheet are respectively for graphics painting, presentation, word processing and spreadsheet processing. They are developed by students in computer science from University of Maryland, College Park [33]. CrosswordSage is a tool for creating professional looking crosswords with powerful word suggestion capabilities [5], and FreeMind is a mind-mapping software [8]. They are both written in Java.

For each subject, we select one or more ESIG groups. These groups are partitioned according to the event semantic interactions in the subjects. The numbers of events in each group are listed in Table 5.2. Each group focuses on one specific task. For example, events in group 1 of TerpWord 3.0 (abbreviated as TW-G1) are all related to Table operations; events in group 2 of FreeMind 0.80 (abbreviated as FM-G2) are all used for formatting the map and displays. We also list lines of code for each subject in the table.

5.2.2 Metrics

For the three research questions, we design metrics for them respectively. The first metric is the increase on the combinatorial coverage obtained by repair. The initially and finally covered t-sets are calculated for the initial and the final test suite respectively. After that, the increase on the number of covered t-sets can be calculated by

subtracting the initial coverage from the final coverage. This says whether our technique is able to repair the GUI test suites. Second, the time for the repair is recorded. We time the test execution in the genetic algorithm, the entire genetic algorithm and the entire repair respectively. These three times can be used for the analysis of the performance and further analysis for improvement. Third, we learn from the finally uncovered t-sets for constraints, and extract constraints from them. The learned constraints are quantified and compared to the formerly identified infeasible patterns.

5.2.3 Experiment Methodology

The experiments are carried out on a computing cluster with AMD 2.4GHz dual-core 64-bit processors, 16GB shared memory, Linux 2.6.18, and Java 1.6. Our tool implementation is run for each group on an individual processor. We believe this assures more precise timing. In the experiments, we reuse existing covering arrays from [39], and then rebuild all the models for the programs using the new GUITAR framework [10]. First, an initial test suite is constructed using the existing covering array. After that, all of the test cases in the initial test suite are run on the corresponding subject to determine the feasibility of the test cases and the initial combinatorial coverage. The feasible test cases are directly added into the final test suite, and the repair starts.

The initial test suite is used as the initial population for the genetic algorithm. In the experiments, we choose the following parameters for the genetic algorithm. The maximum number of generation is 10^6; the max number of consecutive bad moves is 100; the population size is 100; the mutation rate is 0.03; the size factor is set to 1.5 times. In the genetic algorithm, the execution of the test cases is conducted on the new GUITAR replayer [10]. Because the new GUITAR framework switches

its event handling to the Java accessibility package and is still under development, some of the old events that can be replayed using the old replayer cannot be ripped and replayed in the new GUITAR framework. In the experiments, instead of deleting these events, we use a NULL event as a placeholder for them each. The NULL event does not carry out any GUI operation, and is used as if the original event can succeed. We replace one event (out of 11) in TPa-G4 and one event (out of 14) in TPr-G5 with a NULL event respectively in our experiments. All the events in the other groups are reproducible. After the repair, instead of validating the results with known constraints, we learn constraints manully from the finally uncovered t-sets and validate them on the corresponding subjects.

5.2.4 Threats to Validity

In this evaluation, although we tried to make our experiments objective and reasonable, we still have several threats to validity. First, because the new GUITAR framework cannot deal with some events in the old artifacts, we replace them with NULL events. However, because NULL events do nothing, we might miss some interactions between the original event and other events. If these interactions cause infeasibility, our experimental results may not reflect them correctly. To correct possible errors, we use a manual method to check the results against the AUT.

Second, because the subjects are not developed by us, the constraints in them are not known to us. When we check whether the uncovered t-sets cannot really be covered, we use a manual method, that is, we try event combinations on the AUT to test if they can be be executed successfully. Because the number of possible combinations is very large, besides the manual execution, we also use the constraints learned from other combinations to help reduce the space. Although we try to, we

still cannot ensure that the constraints learned are 100% correct.

5.3 Results

We discuss results of the experiments and answer the research questions.

5.3.1 RQ1: Applicability to Non-Trivial GUIs

Table 5.3 shows the results for repairing the subjects. For each group, we first list the number of all t-sets and the number of feasible t-sets manually learned from the remaining uncovered t-sets. We examine the sizes of initial and final test suites, as well as the initial coverage and the final coverage for the *feasible* t-sets. Next, we provide data for the number of executed test cases and times used for each parts. "Time for Execution" is the time used for the execution of test cases during the evolution of the genetic algorithm; "Time for GA" is the total time for the genetic algorithm (excluding the time for the execution of the initial test suite); and "Total Time" is the time from the tool is started until it ends.

From the table, comparing the columns for the initial combinatorial coverage ("Init. C. C.") and for the final combinatorial coverage ("Final C. C."), or those for the corresponding percentages, we can see increases between about 10% (for TPa-G4) and 90% (for TS-G5) on the initial coverage. For all of the groups, our algorithm is able to repair the test suite to cover more than 98% of the feasible t-sets. For group TPa-G4 and TPr-G5, because one event in each is replaced by a NULL event, both of their final test suites cover one 2-set that should not be covered. In the table, the calculation of the final combinatorial coverage in percentage (column "%Final C. C.") excludes this 2-set respectively for the two groups. Otherwise, for example, the

combinatorial coverage for TPa-G4 would be $\frac{5438}{5437}$, which is greater than 100%. We discuss this in more details later.

While the combinatorial coverage is almost fully regained for all the groups, the sizes of the final test suites are not significantly increased. Comparing the columns for the sizes of the initial test suite ("Init. Size") and the final test suite ("Final Size"), we can see the maximum increase on the number of test cases is 70 (for TW-G1). Also we discovered that the number of test cases for some groups are even reduced, such as TS-G1 (reduced by 50), TS-G5 (reduced by 30) and FM-G2 (reduced by 1). However, the sizes of the final test suites do increase compared to the numbers of initially feasible test cases for all the groups.

5.3.2 RQ2: Scalability for Non-Trivial GUIs

Table 5.3 also shows the results for the second research question. We answer the second research question using the number of executed test cases and the time used for the execution (shown in the last four columns). We can see that the total time for the repair still remains at a relatively high level. For some groups, it still needs more than ten days to finish the repair. However, at the same time, we can see that most of the time is still used for the execution of the test cases (see column "Time for Execution"). The time for the test execution still takes the more than 90% of the time for the genetic algorithm for most of the groups. So reducing the time used on the execution of test cases will remain the main focus on future improvement.

Also, we find that the speeds for the execution of different groups are different. For example, for group TW-G1, it takes 33.97 days to run 126648 test cases while for group CS-G4, it takes only 15.36 days to finish 129779 test cases. This demonstrates that the time for the execution of events in different programs are different, so even

Table 5.3: Results of repairing non-trivial GUIs using 2-way criteria.

Group	Total C. C.	Feasi. C. C.	Init. Size	Feasi. T. C.	Final Size	Init. C. C.	% Init. C. C.	Final C. C.	% Final C. C.	Missed Feasi. C. C.	No. Executed	Time for Execution	Time for GA	Total Time
TPa-G4	5445	5437	171	148	188	4950	91.04%	5438	100.00%	0	2526	0.99d	1.00d	1.04d
TPr-G5	8820	8782	280	157	326	5738	65.34%	8778	99.94%	5	20071	9.47d	9.60d	9.67d
TW-G1	8820	8361	280	24	350	1037	12.40%	8250	98.67%	111	126648	33.97d	35.43d	35.46d
TS-G1	8820	6480	280	64	230	2568	39.63%	6480	100.00%	0	89190	31.60d	32.37d	32.42d
TS-G5	2880	1620	92	4	62	175	10.80%	1620	100.00%	0	57093	18.12d	18.29d	18.30d
CS-G4	8820	7780	280	23	316	998	12.83%	7757	99.70%	23	129779	15.36d	17.23d	17.25d
FM-G1	5445	5105	171	52	202	2050	40.16%	5081	99.53%	24	37295	8.95d	9.18d	9.20d
FM-G2	14580	13005	463	270	462	9648	74.19%	13005	100.00%	0	80029	23.58d	24.43d	24.52d
FM-G4	4500	4201	144	23	164	966	22.99%	4154	98.88%	47	38344	9.12d	9.34d	9.36d

- C. C. denotes combinatorial coverage;
- T. C. denotes test case.

Table 5.4: Number of uncovered t-sets for each type of constraint.

Subjects	Constraint Types					Actual No. Uncov.	Feasi. but Uncov.	Infeasi. but Cov.	Expected No. Uncov.
	Disb	Reqs	2Cons	2Excl	Other				
TPa-G4	0	0	8	0	0	7	0	1	8
TPr-G5	0	0	38	0	0	42	5	1	38
TW-G1	0	392	67	0	0	570	111	0	459
TS-G1	2340	0	0	0	0	2340	0	0	2340
TS-G5	1260	0	0	0	0	1260	0	0	1260
CS-G4	0	1040	0	0	0	1063	23	0	1040
FM-G1	0	317	23	0	0	364	24	0	340
FM-G2	1575	0	0	0	0	1575	0	0	1575
FM-G4	0	293	7	0	0	346	46	0	300

if some group executes less test cases than others, it still can take a longer time.

5.3.3 RQ3: Matching for Discovered Patterns

Table 5.4 shows the number of uncovered t-sets for each type of constraint found at the end of the repair. Because our experiments are for 2-way criteria, the constraints Consecutive and Excludes are 2-way Consecutive and 2-way Excludes respectively. We list the actual number of uncovered t-sets, number of feasible t-sets but not covered, number of infeasible t-sets but covered and expected number of uncovered t-sets. Among these four numbers, only the actual number of uncovered t-sets is directly from the results of the experiments, while the other three are achieved through manual analysis.

Groups TPa-G4, TPr-G5, TS-G1, TS-G5, CS-G4 and FM-G2 contain only one type of constraint, while groups TW-G1, FM-G1 and FM-G4 contain more than one type of constraint. However, for each type of constraint, we only list the total number of t-sets that are not covered by the final test suite, without considering the exact reasons. That means even though for some groups, there is only one type of constraint, they may be caused by different event interactions. For example, in CS-G4, among the 1040 uncovered t-sets, eight event interactions cause the infeasibility. They are that events *Proxy Address*, *Proxy Port*, *User Name* and *Password* in both

Word and Crossword modes all require the checkbox *Use Proxy* to be selected before them.

We next examine the uncovered t-sets for each group. We start with the uncovered t-sets (the "Actual No. Uncov." column). First, we check whether the t-sets can be covered by certain sequences. This is carried out by adding events at other positions to manually form test cases. The test cases are performed manually on the AUT to learn the behavior, and then we manually guess and try more combinations that might cover these t-sets. Those that are really covered by some combinations are categorized as feasible t-sets but uncovered (the "Feasi. but Uncov." column). After that, we use the t-sets that are found not covered to explore whether any t-sets that cannot be covered are covered. This is achieved in several ways, including constructing different t-sets for the same uncovered t-way combination, replacing the events in the uncovered t-way combinations with similar events, etc. Then, the events in these new t-sets are compared and analyzed against those in the actually uncovered t-sets to see whether they are feasible or one. For example, when a pair of events appears to be infeasible at all of the pairs of the positions except several, it is reasonable to suspect whether they should also be infeasible in the exceptions. If these t-sets are really infeasible, they are catergorized as infeasible but covered (the "Infeasi. but Cov." column). We also refer to the logs for the acutal execution of related test cases during the repair to confirm the reasons that cause (or do not cause) the infeasibility.

Next, we show the process of manual learning for the constraints through an example, group TPa-G4. Table 5.5 shows the uncovered 2-sets after repair for TPa-G4. From the table, we can see an obvious pattern: the combination (*Select All*, *Copy To*) cannot appear consecutively. After trying the combinations on the AUT, we found that because *Copy To* following *Select All* opens a dialog that is not captured by the model, the replayer cannot execute the succeeding events so the test case is

Table 5.5: Number of uncovered t-sets due to each type of constraint.

No.	Positions									
	0	1	2	3	4	5	6	7	8	9
1	Select All	Copy To								
2		Select All	Copy To							
3			Select All	Copy To						
4				Select All	Copy To					
5					Select All	Copy To				
6						Select All	Copy To			
7							Select All	Copy To		

infeasible. Interestingly, the pair is covered at all the positions except 7 and 8, and 8 and 9. After referring to the logs, we found that one test case (…, *Select All*, *Copy To*, *NULL*) covers the pair at position 7 and 8. Because the NULL event does nothing, it can still be executed even if a unknown dialog is opened after *Copy To*. If the original event *Draw on Canvas* is used here, the test case will be infeasible. The coverage at position 8 and 9 is because when the replayer successfully executes the last event, it programmatically closes the window (by calling `dispose()` method) rather than closes the application by clicking the close button. So the termination will not be impeded by the unknown dialog, and the execution of the test case is considered successful. After the above analysis, we find that the coverage at position 7 and 8 should not have been covered if we use the original event, while the coverage at position 8 and 9 is due to the way the replayer controls the execution and can be covered. As a result, the 2-set (*Select All*, *Copy To*) at position 7 and 8 is infeasible but covered, while that at position 8 and 9 is feasible. In summary, we learned a Consecutive constraint on the event interactions: event *Select All* and event *Copy To* cannot appear consecutively except that they appear at the end of a test case.

The 2-set that is infeasible but covered for group TPr-G4 is also caused by the use

of NULL events. Other groups do not have infeasible but covered t-sets. Compared to the finally covered t-sets (column "Final C. C." in Table 5.3), the numbers of feasible but uncovered t-sets are relatively small (column "Feasi. but Uncov." in Table 5.4). This again demonstrates that our technique is able to discover uncovered feasible t-sets and generate proper test cases to cover most of them.

From the table, we find that there are disabled events in group TS-G1, TS-G5 and FM-G2. For example, in FM-G2, the event *Cloud Color* is disabled no matter how the events in the group are arranged. It is only enabled by another event named *Cloud*, which is not included in the group. So within the range of the events of this group, *Cloud Color* is disabled, and whenever it appears, the test case becomes infeasible. We also see that there are no Excludes constraints in these groups. This is normal, because the Excludes constraints entail very strong conditions – wherever the combination of events in it occur, the test case becomes infeasible. In real GUIs, Excludes constraints may rarely occur, especially for the most commonly used functionalities of a GUI application, that when some events are executed, certain events are not longer able to be executed.

In summary, the infeasible patterns discovered from the non-trivial GUI application match those presented in Chapter 3. Also, the constraints learned from the uncovered t-sets present reasonable restrictions on the event interactions of the applications. They are all constraints widely seen in common GUI applications.

5.4 Summary

In this chapter, we optimized our algorithm and applied the technique to several non-trivial GUI subjects. The evaluation shows that our approach can repair the test suites for non-trivial GUIs. The time used for the execution of test cases in

the algorithm is still the target for future improvement. Also, we manually learn constraints for the GUIs through the uncovered t-sets. As we expected, the learned constraints match those identified from real GUI programs. Our technique not only generates and repairs GUI test suite, but can also serve as a guide for the modeling of the constraints in the GUIs.

Chapter 6

Conclusions and Future Work

Today's GUIs are event-driven. Event sequences serve as typical GUI test cases in model-based GUI testing techniques. Research has also shown that longer sequences are more likely to generate fault-detection effective test cases, so recent advances combine the model-based method and the CIT sampling method for a systematic generation of GUI test cases. However, because of the constraints on the event interactions, the generated test cases suffer from infeasibility problems. Specifically, the generated test cases cannot run to completion because some events in them cannot execute successfully.

In this work, we analyzed this problem and identified infeasible patterns from real GUI applications. We then proposed a framework for generating and repairing GUI test suites. This framework first uses a covering array tool to generate the initial test suite, and then, an approach that incorporates feedback from execution of test cases is used to avoid infeasible patterns in test cases. While different ways can be leveraged for the feedback approach, we designed a genetic algorithm to evolve the test suite. The algorithm ensures not only that the test cases in the final test suite are all feasible, but also the combinatorial coverage of the final test suite.

We carried out an evaluation for our implementation of the framework on a set of synthetic programs. The results show that our technique can significantly increase the coverage of the test suite while maintaining a reasonable size for it. We also compared the genetic algorithm with a random algorithm. Even with a shorter execution time, the genetic algorithm provides a higher final coverage. The study on the scalibility shows that the time used for the execution of test cases is the bottleneck of the repair process and does not scale well.

To improve our implementation for the framework, we tried to reduce the time used on the execution of test cases. We then applied the improved implementation to non-trivial GUIs. An evaluation shows that the improved implementation is able to repair the test suites for non-trivial GUIs. We tried to learn and summarize the constraints from the finally uncovered t-sets manually. The learned constraints match the types we discovered from real GUI applications.

6.1 Future Work

This work suggests several directions of future work. First, a study on the fault-detection effectiveness of the repaired test suites should be conducted to see they improve the ability of fault detection for non-trivial GUIs since ultimately the reason for higher coverage is to detect more faults. Second, more effective feedback approaches for the repair may be discovered. Now the feedback is information from pure test execution, which is quite expensive. In the future, information from direct dynamic and static analysis may also be fed back to improve the performance. Third, automated approaches for learning the constraints from the finally uncovered t-sets can be designed. This may involve in learning techniques which automatically extract patterns from a large amount of data. Fourth, after the patterns are learned, they

can be used for improving the precision of the models for the GUIs. In this way, the test generation can have a much better pre-knowledge of the constraints on the event interaction. Finally, besides repairing the test suites, repairing individual test cases can also be studied to see if they help the fault detection.

Bibliography

[1] Abbot Java GUI test framework. http://abbot.sourceforge.net/.

[2] Andrea Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In *IEEE Congress on Evolutionary Computation*, pages 162–168, 2008.

[3] Penelope Brooks, Brian Robinson, and Atif M. Memon. An initial characterization of industrial graphical user interface systems. In *International Conference on Software Testing, Verification and Validation*, pages 11–20, 2009.

[4] Myra B. Cohen, Peter B. Gibbons, Warwick B. Mugridge, and Charles J. Colbourn. Constructing test suites for interaction testing. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 38–48, 2003.

[5] Crosswordsage 0.35. http://crosswordsage.sourceforge.net/.

[6] Ralph Johnson Erich Gamma, Richard Helm and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1st edition, 1994.

[7] S. Forrest. Genetic algorithms: principles of natural selection applied to computation. *Science*, 261(5123):872 – 878, 1993.

[8] Freemind 0.80. http://freemind.sourceforge.net/.

[9] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. An improved meta-heuristic search for constrained interaction testing. In *International Symposium on Search Based Software Engineering*, pages 13–22, 2009.

[10] GUITAR – a GUI Testing frAmewoRk. http://guitar.sourceforge.net/.

[11] Mark Harman. The current state and future of search based software engineering. In *Future of Softare Engineering*, pages 342–357, 2007.

[12] HP QuickTest Professional. http://en.wikipedia.org/wiki/HP_QuickTest_Professional.

[13] Si Huang, Myra B. Cohen, and Atif M. Memon. Repairing GUI test suites using a genetic algorithm. In *ICST '10: Proceedings of the Third Conference on Software Testing, Verification and Validation*, pages 245–254, 2010.

[14] Jemmy Module. https://jemmy.dev.java.net/.

[15] David J. Kasik and Harry G. George. Toward automatic generation of novice user test scripts. In *CHI '96: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 244–251, 1996.

[16] B. Korel. Automated software test data generation. *IEEE Transaction on Software Engineering*, 16(8):870–879, 1990.

[17] Ping Li, Toan Huynh, Marek Reformat, and James Miller. A practical approach to testing GUI systems. *Empirical Software Engineering*, 12(4):331–357, 2007.

[18] Donglin Liang, Maikel Pennings, and Mary Jean Harrold. Evaluating the precision of static reference analysis using profiling. In *ISSTA '02: International Symposium on Software Testing and Analysis*, pages 22–32, 2002.

[19] Alessandro Marchetto and Paolo Tonella. Search-based testing of Ajax web applications. pages 3–12, 2009.

[20] Phil McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[21] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, 2003.

[22] Atif M. Memon. An event-flow model of GUI-based applications for testing. *Software Testing, Verification and Reliability*, 17(3):137–157, 2007.

[23] Atif M. Memon. Automatically repairing event sequence-based GUI test suites for regression testing. *ACM Transactions on Software Engineering and Methodology*, 18(2):1–36, 2008.

[24] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Using a goal-driven approach to generate test cases for GUIs. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*, pages 257–266, 1999.

[25] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Automated test oracles for guis. *SIGSOFT Software Engineering Notes*, 25(6):30–39, 2000.

[26] Atif M. Memon and Qing Xie. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Transactions on Software Engineering*, 31:884–896, 2005.

[27] Roy P. Pargas, Mary Jean Harrold, and Robert R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9(4):263–282, 1999.

[28] Xiao Qu, Myra B. Cohen, and Gregg Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *ISSTA '08: Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 75–86, New York, NY, USA, 2008. ACM.

[29] Brian Robinson and Lee White. Testing of user-configurable software systems using firewalls. In *International Symposium on Software Reliability Engineering*, pages 177–186, 2008.

[30] Atanas Rountev, Scott Kagan, and Michael Gibas. Evaluating the imprecision of static analysis. In *PASTE '04: Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 14–16, New York, NY, USA, 2004. ACM.

[31] Selenium. http://seleniumhq.org/.

[32] John Stardom. Metaheuristics and the search for covering and packing arrays. Master's thesis, Simon Fraser University, 2001.

[33] Terpoffice 3.0. http://www.cs.umd.edu/users/atif/TerpOffice/.

[34] Paolo Tonella. Evolutionary testing of classes. *SIGSOFT Software Engineering Notes*, 29(4):119–128, 2004.

[35] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 364–374, 2009.

[36] Lee J. White. Regression testing of GUI event interactions. In *ICSM '96: International Conference on Software Maintenance*, pages 350–358, 1996.

[37] Qing Xie and Atif M Memon. Using a pilot study to derive a GUI model for automated testing. *ACM Transactions on Software Engingeering and Methodology*, 18(2):1–35, 2008.

[38] Xun Yuan, Myra Cohen, and Atif M. Memon. Covering array sampling of input event sequences for automated GUI testing. In *ASE '07: Proceedings of the twenty-second IEEE/ACM International Conference on Automated software engineering*, pages 405–408, 2007.

[39] Xun Yuan, Myra B. Cohen, and Atif M. Memon. GUI interaction testing: Incorporating event context. *IEEE Transactions on Software Engineering*, 99(PrePrints):to appear, 2010.

[40] Xun Yuan and Atif M. Memon. Using GUI run-time state as feedback to generate test cases. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 396–405, 2007.

[41] Xun Yuan and Atif M. Memon. Generating event sequence-based test cases using GUI runtime state feedback. *IEEE Transactions on Software Engineering*, 36:81–95, 2010.