Repairing GUI Test Suites Using a Genetic Algorithm

Si Huang, Myra B. Cohen
Dept. of Computer Science & Engineering
University of Nebraska-Lincoln
Lincoln, NE 68588-0115
{shuang, myra}@cse.unl.edu

Atif M. Memon
Dept. of Computer Science &
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742
atif@cs.umd.edu

***Abstract*—Recent advances in automated functional testing of Graphical User Interfaces (GUIs) rely on deriving graph models that approximate all possible sequences of events that may be executed on the GUI, and then use the graphs to generate test cases (event sequences) that achieve a specified coverage goal. However, because these models are only approximations of the actual event flows, the generated test cases may suffer from problems of infeasibility, i.e., some events may not be available for execution causing the test case to terminate prematurely. In this paper we develop a method to automatically repair GUI test suites, generating new test cases that are feasible. We use a genetic algorithm to evolve new test cases that increase our test suite's coverage while avoiding infeasible sequences. We experiment with this algorithm on a set of synthetic programs containing different types of constraints and for test sequences of varying lengths. Our results suggest that we can generate new test cases to cover most of the feasible coverage and that the genetic algorithm outperforms a random algorithm trying to achieve the same goal in almost all cases.**

I. Introduction

Black-box testing of programs with Graphical User Interfaces (or GUIs) can be achieved by executing sequences of events based on a model of the GUI [1], [2]. Test cases define sequences of behavior; i.e., these are subsets of the specifications of behavior for the program. Research has shown that testing a GUI from this perspective will find faults related not only to the GUI and its glue code, but to the underlying business logic of the application as well [3].

A recent approach developed to obtain such a model is to execute the program under test and analyze (or *rip*) the encountered events and relationships between them [1]. These events can then be modeled as a graph approximating their flow (EFG) or more abstractly as a directed graph of events which interact (EIG). These models are then used to combine and test sequences of events for program validation.

Such models are effective at generating short sequences of events for testing; length two or three [1]; such sequences can be executed quickly and automatically using a test harness. If a GUI has five events, where every event can be executed after every other event, then there are only 5^2 or 25 length two sequences and only 125 length three sequences.

Research has shown that longer event sequences can expose faults undetectable by short ones [1], [4]. These longer sequences provide more context and reach more complex code in the program. However, there are two primary limitations of testing GUIs using long test sequences.

First, the number of sequences grows exponentially with length. In the five-event GUI, if we generate length 10 sequences, we have a potential pool of 5^{10} or 9,765,625 unique sequences and for length 20 this grows to 9.5×10^{13}. To control the exponential growth and yield the same coverage, a sampling technique derived from *combinatorial interaction testing* (CIT) can be applied to GUI sequences [5], [6]. In this sampling technique (discussed in Section II) all t-way combinations (t-sets) of events are combined in all possible combinations of locations in the sequence. The results of this work shows, that when applied to multiple benchmarks, CIT can detect many more faults than shorter exhaustive sets of sequences. However the technique suffers from the second limitation – infeasible test cases.

This second limitation is the focus of this paper. We take an existing CIT test suite and *repair* it by removing infeasible test cases and inserting new feasible ones to provide additional coverage. We say that a test case is infeasible if at least one of its events that is expected to be available during test execution is, in fact, unavailable. In such a situation, the test harness may hang or fail. Test case infeasibility may be due to a defect in the program or missing information in the model used to obtain the test case. Although the problem of infeasibility is also relevant to short sequences, research has shown that it is much more severe for long sequences [7].

Building upon our work on using CIT for GUI testing, we have developed a framework to automatically *repair a test suite* by adding feasible test cases through an evolutionary process. The framework first builds CIT samples and executes them. Then infeasible test cases are discarded and a genetic algorithm is used to generate new sequences that improve CIT coverage and avoid infeasible sequences. Genetic algorithms have been successfully used before in test case generation [8] and genetic programming has been used for program statement repair [9], [10]. We have experimented with this technique on a large scale study, consuming over one machine-year of time on a set of seven programs that

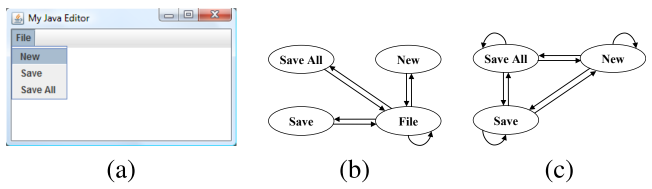


Figure 1. (a) A Simple GUI, (b) its EFG, and (c) EIG

contain realistic types of infeasible patterns. We show that our genetic algorithm increases CIT coverage to more than 99% for a small increase (and in some cases a decrease) in the final CIT sample size. Compared with a random algorithm that covers as little as 74% of the feasible coverage in the worst case, our technique generates smaller test suites with better coverage on the longer test sequences.

The rest of this paper is laid out as follows. In Section II we present some background and related work on GUI testing, CIT and genetic algorithms. In Section III we present a set of example constraints we have identified in real programs and that we will mimic for our experimentation. In Section IV we present our framework for repair. In Section V we present a case study on seven programs modeled after the patterns seen in Section III. Finally we conclude and present future work in Section VII.

II. Background and Related Work

We now provide an overview of GUI testing, CIT, genetic algorithms and other related work.

A. *GUI Testing*

There has been a large body of work on model-based GUI testing, including using finite state machines [11], pre- and post-conditions [12], and directed graph models [13]. Because of their relevance to this work, we will discuss only graph models.

An *event-flow graph* (EFG) models all possible event sequences that may be executed on a GUI. It is a directed graph with nodes (one for each event in the GUI) and edges that 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`. The EFG is 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; $(e_x, e_y) \in \mathbf{E}$ iff e_y `follows` e_x.

Figure 1(a) presents a GUI that consists of four events, `New`, `Save`, `SaveAll`, and `File`. Figure 1(b) shows the GUI's EFG; the four nodes represent the four events; the edges represent the `follows` relationships. In this EFG, the event `New follows File`.

Event-interaction graph (EIG) nodes, on the other hand, do not represent events to open or close menus, or open windows. 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 [1]).

Figure 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.

There are two points to note. First is that graph-traversal algorithms may be used to "walk" the graph, enumerating events along visited nodes, thereby generating test cases. A technique to generate test cases, each corresponding to an EIG edge has been developed; these test cases are called *smoke tests* [13]. Two examples of length 2 smoke test cases for Figure 1(c) are (`New`, `Save`) and (`Save`, `SaveAll`). There are a total of 9 such tests – one for each EIG edge. It is easy to see that the number will grow exponentially with sequence length. The second advantage is that an approximation of these models can be constructed automatically using a reverse engineering technique called *GUI Ripping* [13]. 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/incorrect, which is why we say that it outputs an *approximation* of the EFG. It is possible that some event sequences generated via these graphs are infeasible.

In more recent work [4], we developed a new feedback-based technique 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. The seed suite is 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, called the *Event Semantic Interaction Graph* (ESIG). Because the seed suite is generated from the EIG (a structural model) and the ESI relationship is obtained in terms of event execution (a dynamic activity), the ESIG 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 – each event is ESI-related to its subsequent event, i.e., it was shown to influence the subsequent event during execution of the seed suite. It turns out that because of the strict conditions associated with the ESI relationship, the ESIG is actually a collection of smaller sub-graphs; events within a sub-graph are all ESI-related. Even when using an ESIG, we may still encounter infeasible test cases.

We have repaired *test cases* before, albeit for regression testing [7]. When the structure of a GUI is modified, test cases from the original GUI's suite are either reusable or unusable on the modified GUI. We developed algorithms to (1) automatically determine the usable and unusable test cases from a test suite after a GUI modification, (2) determine the unusable test cases that can be repaired so that they can execute on the modified GUI, and (3) use *repairing transformations* to repair the test cases. The challenges of repairing sequences were fewer in the context of regression testing because we used the differences between the two versions' EFGs to drive the repairs.

B. Testing GUIs Using CIT

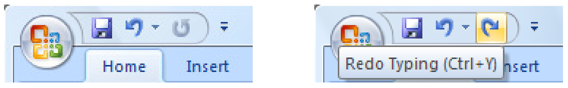
The basis for combinatorial interaction testing is a *covering array* (written as $CA(N; t, k, v)$), an $N \times k$ array on v symbols with the property that every $N \times t$ sub-array contains all ordered subsets of size t of the v symbols *at least* once [14]. In other words, any subset of t-columns of this array will contain all t-way combinations of the symbols. We use this definition of a covering array to define the GUI event sequences by viewing the same event in different positions in the sequence as different events. Suppose we want to test sequences of length four and each location in this sequence can contain exactly one of three events (`Save`, `SaveAll`, `New`). Testing all combinations of these sequences requires 81 test cases. We can instead sample this system, including all sub-sequences of shorter size, perhaps two. We model this sequence as a $CA(9; 2, 4, 3)$; we have 9 test sequences and we cover all 2-way combinations in all locations at least once. There are 54 pairs that should be covered in this sample. The *strength* of testing is t. For instance we set $t = 2$ in the example and include all pairs of events between all four locations. If we examine any two columns of the covering array, we will find all nine combinations of event sequences at least once.

C. Search Algorithms for Testing

One common way to automate test generation/feedback is through the use of evolutionary algorithms such as genetic algorithms [8], [15], [16]. Genetic algorithms are a type of meta-heuristic search algorithm called evolutionary algorithms. These model the biological evolutionary process. A population is composed of many individuals from the set of all possible solutions each represented as a *chromosome* containing a set of *alleles* or genes. Pairs of solutions (*parents*) are selected based on their fitness. The fitness is a function that evaluates how good a chromosome is; i.e., how close it is to an "optimal" solution. A crossover and recombination stage take place during which parents exchange and combine information to generate a set of children. A mutation rate is applied to the new population to diversify individuals and the fittest solutions from this new population are then selected and the process is repeated.

Other work using meta-heuristic search to generate test sequences such as that of Marchetto and Tonella [17] 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 [18]. Our work is closely related to this thread of research, but it has only been used to generate single CIT samples without constraints. More recent work by Arcuri and Yao [9] and Weimer et al. [10] 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.



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

Figure 2. *Requires* Constraint

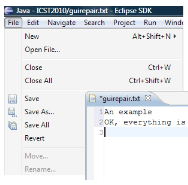
III. Infeasible Patterns: Event Constraints

We begin with a description and classification of some types of infeasible sequences that we have identified. We term these *event constraints* or simply constraints. We classify constraints into four broad categories, *disabled*, *requires*, *consecutive* and *excludes*. These are based on constraints that can be found in real GUI applications.

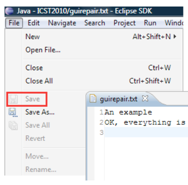
All of the examples shown are illustrated with short sequences for simplicity, however we can find examples for most of these constraints that are longer. For instance, we may have two, three or more events that are excluded or two or more events that require another event. We return to the issue of arity of constraints in our evaluation (Section V).

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.

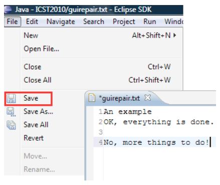
Requires Constraint: This constraint indicates that some event needs another event to be executed before it is enabled. An example of this type of constraint is illustrated by the `Redo` operation. Before one can execute `Redo` they must



(a) `Save as` is invoked

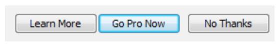


(b) `Save` is disabled

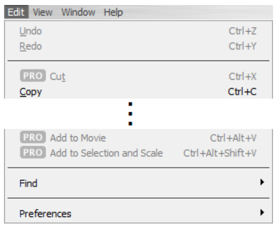


(c) `Save` is re-enabled after `Type in editor`

Figure 3. *Event Consecutive* Constraint



(a) Trial version is chosen by clicking `No thanks`



(b) Cannot `Add to Movie`: only works with "PRO" version

Figure 4. *Excludes* Constraint

first execute `Undo`. Figure 2 shows an example of this sequence in Microsoft Office 2007.

Event 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 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.

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 4(a)). If the user chooses to stay in the evaluation version, then features only provided in the professional version are disabled (see Figure 4(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`.

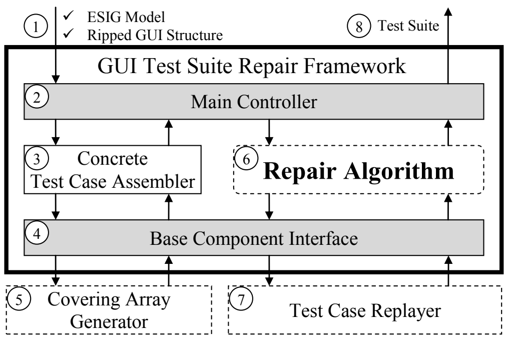

1
✓ ESIG Model
✓ Ripped GUI Structure
8 Test Suite
GUI Test Suite Repair Framework
2 Main Controller
3 Concrete Test Case Assembler
6 Repair Algorithm
4 Base Component Interface
5 Covering Array Generator
7 Test Case Replayer

Figure 5. Organization of GUI Test Suite Repair Framework

IV. Overview of Test Suite Repair

Figure 5 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 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 of variation in this framework. We use two plug-ins to achieve 1) the initial covering array generation and 2) test case replaying. The third point of variation is the repair algorithm. For our current instantiation of this framework, we use a simulated annealing algorithm developed by Garvin et al. [19] for the covering array generation and a modified version of GUITAR [20] 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. We discuss our repair algorithm next, which has been instantiated as a genetic algorithm.

A. Repair Algorithm

The core part of our framework is the repair algorithm. Figure 6 shows an overview of our repair algorithm ((6) in Figure 5). While different types of algorithms for repair are possible, in this work, we have used a genetic algorithm 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.

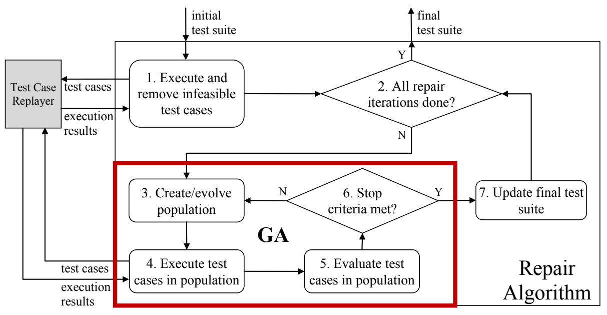

initial test suite
final test suite
Test Case Replayer
test cases
execution results
1. Execute and remove infeasible test cases
2. All repair iterations done?
Y
N
3. Create/evolve population
6. Stop criteria met?
N
Y
7. Update final test suite
GA
4. Execute test cases in population
5. Evaluate test cases in population
test cases
execution results
Repair Algorithm

Figure 6. Using Genetic Algorithm as Repair Algorithm

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). If all test cases are feasible (step 2), it exits and is done. 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 repaired test suite to grow. It is set to the maximum number of test cases that can be added. For example, if the maximum final size is 15, and 3 feasible test cases are already added into the final test suite, then we use 12 iterations to complete the final test suite.

For each iteration we run the genetic algorithm (steps 3–6). 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 than the maximum possible size. Next, we discuss the genetic algorithm in detail.

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 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 test case s, assume the length of s is l, the failure point of s is f_s, and the number of newly covered t-sets that s contributes is cov_s, the fitness function is defined as

$$fitness(s) = b \cdot cov_s - p \cdot (l - f_s),$$

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 s. The definition of cov_s in the fitness function can be adapted to other coverage criteria to generalize the framework for test suites other than those derived from CIT. 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; having numerical values allows some infeasible test cases that add coverage to be included in the final test set. We leave fine tuning of b and p to future work.

Crossover and Mutation: For crossover we rank the test cases in descending order and pair consecutive chromosomes. A one-point crossover is used. 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 a randomly chosen event.

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. 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.

V. 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?

A. Subjects

We have created seven subject programs that contain the types of constraints described in Section III. Table I 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, (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).

B. 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, 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.

C. 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.

D. 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.

E. 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

Table I
SUBJECT PROGRAMS

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 occur before it
3	Event 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 are infeasible if they occur (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

some systems where we group interacting events into 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.

VI. RESULTS

We examine the results for each research question next[1].

A. RQ1: Framework Effectiveness

Table II 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.

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.

B. 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 III. In this table we first show the subject parameters and target coverage for length 5 and 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 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 7 and 8. 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.

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

Table II
REPAIRED TEST SUITES WITH LENGTH 5 TEST CASES (AVG OF 5 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%

Table III
COMPARISON OF RANDOM ALGORITHM AND GENETIC ALGORITHM (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

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 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.

C. *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 IV. 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.

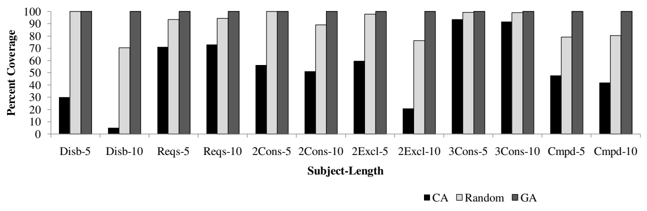


Figure 7. Comparison of Coverage for 2-way Criteria

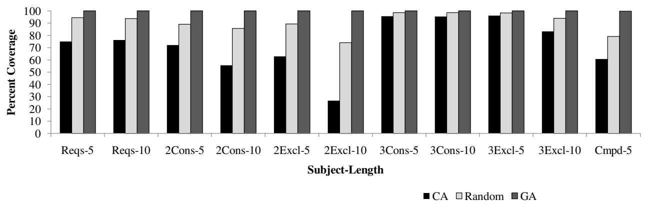


Figure 8. Comparison of Coverage for 3-way Criteria

We examine this further in Figure 9. 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 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 [5]. 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.

VII. Conclusions and Future Work

In this paper we have presented a framework for GUI test suite repair. We generated test suites to cover all pairs or three-way combinations of event sequences for length 5, 10, 15, and 20 in seven synthetic programs. The programs contain event dependencies that may not be discovered during GUI ripping, and that will cause test execution to fail. We examined test suite repair using both a genetic algorithm and a random algorithm. From our experiments we conclude that we can successfully increase the feasible coverage of our test suites and that the genetic algorithm outperforms the random algorithm. This is consistent for both 2- and 3-way coverage. When we run longer test sequences of length 15 and 20 we conclude that our algorithm still scales for the coverage metric; we increase our coverage in all cases. However the execution time increases and may prevent this from scaling to large applications.

In future work we plan to examine the issue of scalability by tuning the genetic algorithm more finely for execution time and to examine an adaptive approach where we incrementally repair. We plan to apply this framework to real GUI applications to study both effectiveness of coverage as well as fault detection effectiveness of the repaired test cases. We will also examine different types of coverage other than CIT. In this paper we have only considered the test input and assume the existence of a global, generic test oracle. In the future we plan to consider the issue of oracle repair as well. Finally, we plan to develop an automated method to classify the missing coverage so that we can provide automated feedback to the graph models as the repair proceeds.

Acknowledgments

We would like to thank Scott McMaster for providing us with the replayer modified for our experiments, and Mary Lou Soffa for early discussions on this work. This work was partially supported by the US National Science Foundation under grants CCF-0747009, CCF-0447864, CNS-0855139 and CNS-0855055, the Air Force Office of Scientific Research through award FA9550-09-1-0129, the Office of Naval Research grant N00014-05-1-0421 and by the Defense Advanced Research Projects Agency.

References

[1] Q. Xie and A. M. Memon, "Using a pilot study to derive a GUI model for automated testing," *ACM Transactions on Software Engineering and Methodology*, pp. 1–35, 2008.

[2] T. Tuglular, C. A. Muftuoglu, O. Kaya, F. Belli, and M. Linschulte, "GUI-based testing of boundary overflow vulnerability," in *Annual International Computer Software and Applications Conference*, 2009, pp. 539–544.

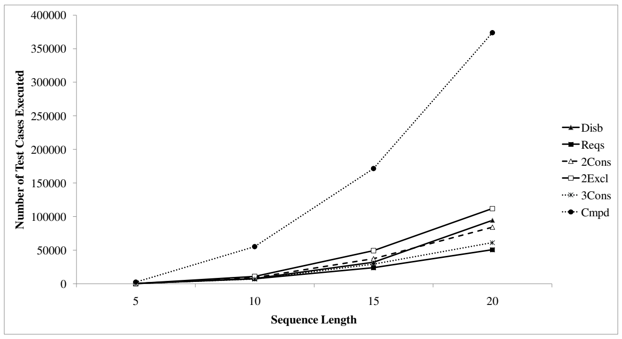
[3] P. Brooks, B. Robinson, and A. M. Memon, "An initial characterization of industrial graphical user interface systems," in *International Conference on Software Testing, Verification and Validation*, 2009, pp. 11–20.

[4] X. Yuan and A. M. Memon, "Using GUI run-time state as feedback to generate test cases," in *International Conference on Software Engineering*, 2007, pp. 396–405.

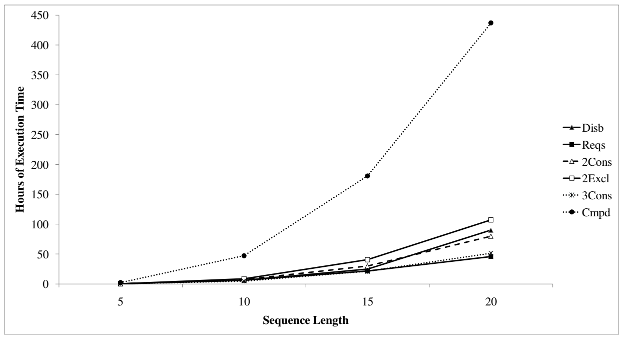
[5] X. Yuan, M. Cohen, and A. M. Memon, "Covering array sampling of input event sequences for automated GUI testing," in *International Conference on Automated Software Engineering*, 2007, pp. 405–408.

Table IV
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 9. Comparison of Numbers of Test Case Executions and Execution Time for GA

[6] X. Yuan, M. B. Cohen, and A. M. Memon, "GUI interaction testing: Incorporating event context," *IEEE Transactions on Software Engineering*, 2010, to appear.

[7] A. M. Memon, "Automatically repairing event sequence-based GUI test suites for regression testing," *ACM Transactions on Software Engineering and Methodology*, pp. 1–36, 2008.

[8] R. Pargas, M. J. Harrold, and R. Peck, "Test-data generation using genetic algorithms," *Software Testing, Verification and Reliability*, vol. 9, no. 3, pp. 263–282, 1999.

[9] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *IEEE Congress on Evolutionary Computation*, 2008, pp. 162–168.

[10] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *International Conference on Software Engineering*, 2009, pp. 364–374.

[11] B. Robinson and L. White, "Testing of user-configurable software systems using firewalls," in *International Symposium on Software Reliability Engineering*, 2008, pp. 177–186.

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

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

[14] M. B. Cohen, C. J. Colbourn, P. B. Gibbons, and W. B. Mugridge, "Constructing test suites for interaction testing," in *International Conference on Software Engineering*, 2003, pp. 38–48.

[15] B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870–879, 1990.

[16] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[17] A. Marchetto and P. Tonella, "Search-based testing of Ajax web applications," in *International Symposium on Search Based Software Engineering*, 2009, pp. 3–12.

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

[19] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "An improved meta-heuristic search for constrained interaction testing," in *International Symposium on Search Based Software Engineering*, 2009, pp. 13–22.

[20] "GUITAR – a GUI Testing frAmewoRk," website, 2009, http://guitar.sourceforge.net.