Search based GUI Test Generation in Java
Comparing Code-based and EFG-based Optimization Goals

Mathias Menninghaus, Falk Wilke, Jan-Philipp Schleutker and Elke Pulvermüller
Institute of Computer Science, University of Osnabrück, Wachsbleiche 27, 49090 Osnabrück, Germany

Keywords: GUI Testing, Test Generation, Test Automation, Search based Software Engineering, Genetic Algorithms.

Abstract: Modern software systems often communicate with their users by graphical user interfaces (GUI). While the underlying business logic may be fully covered by unit tests, the GUI mostly is not. Despite the widespread use of capture and replay tools, which leave the test generation of GUI tests to the user, recent research also focuses on automated GUI test generation. From the numerous approaches, which include symbolic execution, model-based generation, and random testing, search based test data generation seems to be the most promising. In this paper, we create GUI tests using hill climbing, simulated annealing and several genetic algorithms which deal differently with the sequence length and use multi or single objective algorithms. These different test data generators are compared in terms of runtime and coverage. All approaches are also compared using different optimization goals which are a high coverage of the event flow graph (EFG) of the GUIs and a high coverage of the underlying source code. The evaluation shows that the genetic algorithms outperform hill climbing and simulated annealing in terms of coverage, and that targeting a high EFG coverage causes the best runtime performance.

1 INTRODUCTION

Graphical user interfaces (GUI) are a common component in todays software systems (Myers et al., 2000). As they handle the human - computer interaction, their correct functionality is a crucial part to fullfill the requirements and to guarantee the quality of a software system. The underlying code base is only accessed indirectly by the interactions on the GUI. Also, often a sequence of interactions is needed to access certain functionalities in the software. For instance, in order to open a file to be edited in a text editor, one may need to open at least one dialog frame to choose the correct file. A GUI and all decisions that are possible through the lifetime of that GUI can be displayed in an event flow graph (EFG). An event flow graph consists of nodes, each representing an event on a GUI component, and edges, one for each possible action which leads from one event on a GUI component, i.e. node in the EFG, to another. One criteria for an appropriate GUI test is a high coverage on the overall software system and the challenge is to determine such tests. With at least one loop in the EFG there are an infinite number of possible paths through the GUI. For instance, consider the example GUI as shown in figure 1. The corresponding EFG is shown in figure 2 and already contains a loop, because the *toggle* button is not dismissed after it has been clicked. Systematically running through all possible paths in an EFG in order to find a minimal set of GUI tests would be too much overhead to compute. Instead, in this paper search based approaches (Harman et al., 2012a; Harman et al., 2012b) are used to create the desired test sets.

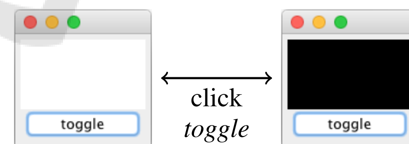


Figure 1: A simple GUI to toggle the color of a panel.

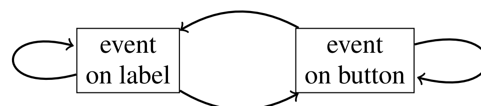


Figure 2: Event flow graph (EFG), the different possible events on the components (e.g. right and left mouse click) already have been merged.

Menninghaus, M., Wilke, F., Schleutker, J-P. and Pulvermüller, E.
Search based GUI Test Generation in Java - Comparing Code-based and EFG-based Optimization Goals.
DOI: 10.5220/0006277801790186
In *Proceedings of the 12th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2017)*, pages 179-186
ISBN: 978-989-758-250-9
Copyright © 2017 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

2 RELATED WORK

Previous approaches on general automated test generation include symbolic execution (Khurshid et al., 2004) which is used by the tools KLEE (Cadar et al., 2008) and Barad (Ganov et al., 2008), random tests, either directed and checked (Pacheco et al., 2007) or unchecked and therefore often crashing (Csallner and Smaragdakis, 2004) and model based approaches, which use different models such like activity diagrams (Vieira et al., 2006; Chen et al., 2009) or the event flow graph (Memon, 2007; Yuan and Memon, 2010). The latter approach is used in this paper. Symbolic execution suffers from path explosions and random testing, even if directed, produces a lot of infeasible tests to be discarded. Search based software testing (SBST) (McMinn, 2011) uses metaheuristic search algorithms such as simulated annealing, hill climbing and evolutionary algorithms to systematically scan the search space. Search based automated testing frameworks are CUTE (Sen and Agha, 2006), DART (Godefroid et al., 2005), TestFul (Baresi et al., 2010) and GAMDR (Aburas and Groce, 2016). Search based testing is also successfully used to generate GUI tests (Carino, 2016). Static approaches, such like GUITAR (Nguyen et al., 2014) build the GUI model using a ripping procedure before generating the test cases. Dynamic approaches build the model and generate the test cases dynamically. EXYST (Gross et al., 2012) uses the experiences made with EVOSUITE to generate small GUI test sequences with a high code coverage. Pigdin Crasher (Dan et al., 2014) is developed to find GUI sequences which cause system crashes. Additionally to the mentioned tools, (Soffa et al., 2001) focus on coverage criteria for inter- and intra-component coverage. (Yuan et al., 2011) extend that idea with covering arrays (Chee et al., 2013) to unfold faults more directed by controlling the sequence length, the possible positions of events and certain combination of events. (Arcuri, 2012) investigate the best sequence length of tests regarding to specific configurations. (Fraser and Arcuri, 2011) stress the correct bloat control, with a bloat being the disproportional quickly growth of the length of test sequences. Different appraoches on incorporating the sequence length and bloat control are discussed in section 5.1.

(Ingber and Rosen, 1992) and (Mitchell et al., 1993) compare genetic algorithms with hill climbing and simulated annealing and discuss when and how genetic algorithms may outperform the latter. In this paper, genetic algorithms are compared to simulated annealing and hill climbing in terms of code coverage for GUI tests.

Before presenting our approach, it should be noted, that a test always has to be connected to a corresponding test oracle which defines when the outcome of a test case is correct or not. In a recent survey, (Barr et al., 2015) state that the generation of oracles is still an open problem and "Much work on test oracles remains to be done". In this paper, we focus on the pure generation of GUI tests leaving the oracle problem to the user for now.

3 GUItoolkit

This section roughly describes our GUItoolkit, it is limited to the Java Swing framework and some other limitations will also be described in this section.

The generation of test sequences starts with the detection phase in which all accessible GUI components are detected and their representations are incorporated into an event flow graph. Its algorithm (Figure 3) uses AssertJ [1] to detect all active GUI components and activates them one by one. For instance, it clicks every visible button in a JFrame. Text input is derived from a user defined file with appropriate strings. If no specific file is invoked, GUItoolkit uses a set of short dummy strings. More complex text input generators may use a specific interface and override the file input. After the initial detection of all elements which are visible with the initialization of the system under test (SUT), the detection is repeated over all new elements until no new elements appear. At this point, the detection is limited to elements which are allocated to their parents when the SUT is first invoked. It also expects only one JFrame per SUT and only one pop-up window to be on screen at once. In contrast to GUITAR (Nguyen et al., 2014) the detection only stops when no new elements appear.

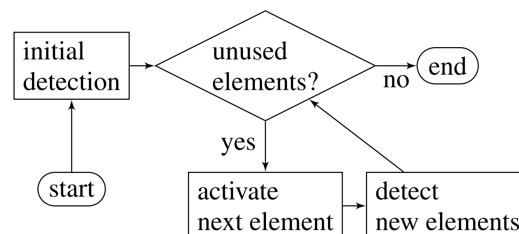


Figure 3: Flow chart of the GUI detection algorithm.

With the detection all necessary information for the following test generation is captured. This includes the components type, the optional text, all registered listeners in order to avoid unnecessary actions

[1] http://joel-costigliola.github.io/assertj/assertj-swing.html

and the optional precondition. In order to identify the detected elements the path through the elements is stored. After detection, the recorded structure of the GUI is transferred into an EFG. The class files are instrumented in order to capture the coverage when the generated test cases (Section 4) are executed. So far, statement, branch and path coverage are supported. During execution, also the coverage on the EFG is recorded. The GUItoolkit is implemented in Java, version 1.8.

4 SEARCH BASED GUI TEST GENERATION

This section describes the test generation techniques used by GUItoolkit. Generally, each of the test cases consists of a sequence of GUI actions which may each include a number of mouse and keyboard events. Therefore, the atomic part of a test is one specific action on a GUI which triggers one event on a component.

4.1 Local Search Algorithms

Two local search algorithms are used to generate the desired test data, hill climbing and simulated annealing (Russell et al., 2003). In contrast to (Korel, 1990), which adapt hill climbing for unit test generation, the original approach (Russell et al., 2003) is used. The next state is simply the maximum of the neigbouring states according to a user defined function h. The major drawback of hill climbing is, that it may get stuck in a local optimum.

In contrast to hill climbing, simulated annealing uses an additional temperature function. It indicates how far the search reaches out for an optimum. The longer the algorithm runs, the colder the temperature gets. The colder the temperature is, the more unlikely it is for the algorithm to choose another state if the current state is an optimum.

Both optimization algorithms choose one transition to the next state in every of their optimization steps. The GUItoolkit employs two different types of transitions. One for starting a new sequence of GUI actions and one for adding a new GUI action to an existing GUI action sequence. By incorporating the EFG into the decision, only valid actions are choosen in order to be added to an existing or new sequence. For the implementation of both heuristics the AIMA3e-Java[2] framework is used and incorporated into GUItoolkit. Other heuristics from AIMA3e could be easily incorporated in future work.

[2] https://github.com/aimacode/aima-java

4.2 Genetic Algorithms

A genetic algorithm starts with an initial population of solutions, or generation. The next generation is created by selecting a set of solutions from the previous generation regarding to a fitness function. The solutions of every generation are then recombined (crossover) and also mutated. In terms of GUI test generation one solution consists of a set of action sequences, with each action representing one event on a certain GUI component. For the selection of new generations binary tournament selection is applied(Goldberg and Deb, 1991), which randomly chooses two solutions and selects the better of the two. Generally, a crossover takes a fraction of two or more solutions and interchanges them. In more detail, for GUI test generation single-point crossover is used. That means, of two parents P_1 and P_2 one point for both is chosen as an index to separate them. Everything in the first part of the first parent P_1 is then combined with the second part of P_2. Vice-versa the first part of P_2 is combined with the second part of P_1. Thus, the action sequences of two solutions are mixed up. A mutation operates on one solution and alters it. Here, a mutation may be the insertion of a new action inside a sequence, the deletion of an action from an existing sequence, or both. Mutation is also applied on the whole solution, i.e. new action sequences are inserted or deleted. With every alteration via crossover or mutation, the validity of the altered sequences is checked. The change is discarded if it causes an invalid action sequence. A genetic algorithm terminates, if either a maximum number of generations has been generated, a solution is found that satisfies a minimal criteria, a fitness plateau is reached, the allocated resources (e.g. time) are consumed, or a combination of these criteria has reached its defined limit.

As not only the achieved coverage on the code or the EFG should be maximized, but also the sequence length minimized, multi objective genetic algorithms are used. In contrast to classic genetic algorithms, multi-objective approaches (Zitzler et al., 2006) use a vector to be minimized or maximized instead of a single value. Along this simple definition a lot of different approaches on genetic algorithms have been proposed. For generating GUI tests only the NSGA-II (Deb et al., 2000), PESA-II (Corne et al., 2001) and SPEA-II (Laumanns et al., 2001) are considered, as they represent a good fraction of the recent approaches on genetic algorithms. All three algorithms

are implemented within the *Multi Objective Evolutionary Algorithm* (MOEA) - Framework [3] which is used by the GUItoolkit.

5 EVALUATION

This section describes the evaluation on the presented GUItoolkit and the applied search based optimization algorithms. Simulated annealing, hill climbing and three genetic algorithms are compared. The main goal of the algorithms is to achieve a maximized branch coverage. Therefore two different optimization goals, a high branch coverage in the code and a maximized EFG coverage are compared.

5.1 Setup

The setup is derived from previous work (Section 2), personal experience and in order to achieve a maximum coverage on the possible settings. The evaluation is performed on two example GUIs, a rather simple and a complex one. Since mostly all configurations cause a very high coverage on the simple GUI, only the results for the more complex GUI, depicted in Figure 4, are shown.

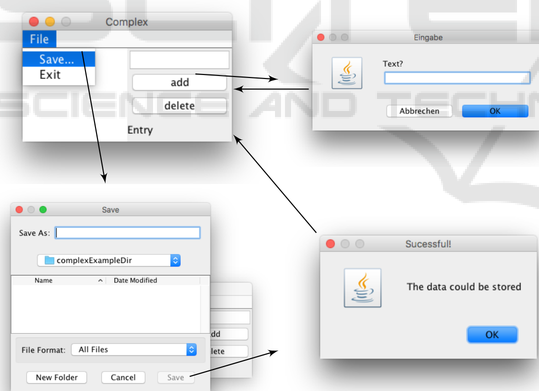


Figure 4: Excerpt from a more complex GUI, creating and deleting entries in a list, and saving the list in a certain file. The standard actions on the `JFileChooser` (bottom left) are not shown.

5.1.1 Local Search Algorithms

In contrast to simulated annealing, the presented approach on hill climbing is strictly deterministic. Therefore, less evaluation runs are needed. As the evaluation is very time consuming, the maximum number of iterations is limited according to Table 1, third column. Also, the maximum number of attempts in using the same action in one sequence is limited in order to avoid loops and local optima (fourth column). Taking bloat control into account (Fraser and Arcuri, 2011), the maximum sequence length is also limited according to the fifth column in Table 1. An automated process may set the limitations after pre - evaluating the EFG of a specific GUI.

5.1.2 Genetic Algorithms

Three different genetic algorithms (NSGA-II, SPEA-II and PESA-II) are compared together with a random search algorithm[4]. All of them use binary tournament selection, single point crossover on test sets, and mutate test sets and action sequences by inserting or deleting GUI actions from sequences and sequences from test sets, respectively. For both, mutation and crossover, a probability of 0.2 and 0.8 is investigated, resulting in 4 different probability setups. The settings of the genetic algorithms are set to the standard values used in the MOEA framework. The branch coverage of the underlying code and the coverage on nodes in the EFG are used as fitness functions. Using the EFG coverage as fitness function only requires one goal, whereas when using the branch coverage of the code the overall coverage on the complete code, the average coverage on the methods and the median coverage on the methods is used. The average and median coverage on methods are used in order to not only cover the methods with a big control flow graph but cover all methods equally. Initially, three sequences with three vertices each are created. The population size is set to 100 for EFG-based approaches, and to 10 for code-based approaches. For EFG-based approaches, the maximum number of evaluations is set to 5000, and for code-based approaches to 150. The selected values are set due to the corresponding computation costs, as the branch coverage calculation costs are higher than EFG coverage calculation costs.

The length is added to both types of fitness functions to minimize it. In order to achieve a greater impact of the length also every configuration is run by not only using the length as additional objective, but by using length categories. The fitness is set to 0.1 for lengths < 6, 0.2 for lengths < 12 and 0.3 otherwise when using the EFG-based approach. For code-based coverage the thresholds are set to 8 and 16, respectively. As for the limitation on local search algorithms, the length categories might be set after automatically pre - evaluating the EFG.

In contrast to the multi objective approaches, the

[3] http://moeaframework.org

[4] http://moeaframework.org/javadoc/org/moeaframework/algorithm/RandomSearch.html

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Table 1: Limitations for the generation of GUI tests with simulated annealing and hill climbing. Coverage is always meant to be branch coverage on the given code.

Algorithm	Coverage goal	Max Iterations	Max Tries	Max Length
Hill climbing	EFG	40	30	3
Simulated annealing	EFG	8	5	3
Hill climbing	code	40	30	3
Simulated annealing	code	8	5	3

Table 2: Evaluation setups for genetic algorithms. The dashed path shows one example configuration choosing the multi objective setup with coverage and minimized length as optimization goal using the PESA-II algorithm, the EFG as coverage goal and a mutation and crossover rate of 0.8. The random setup is not depicted here.

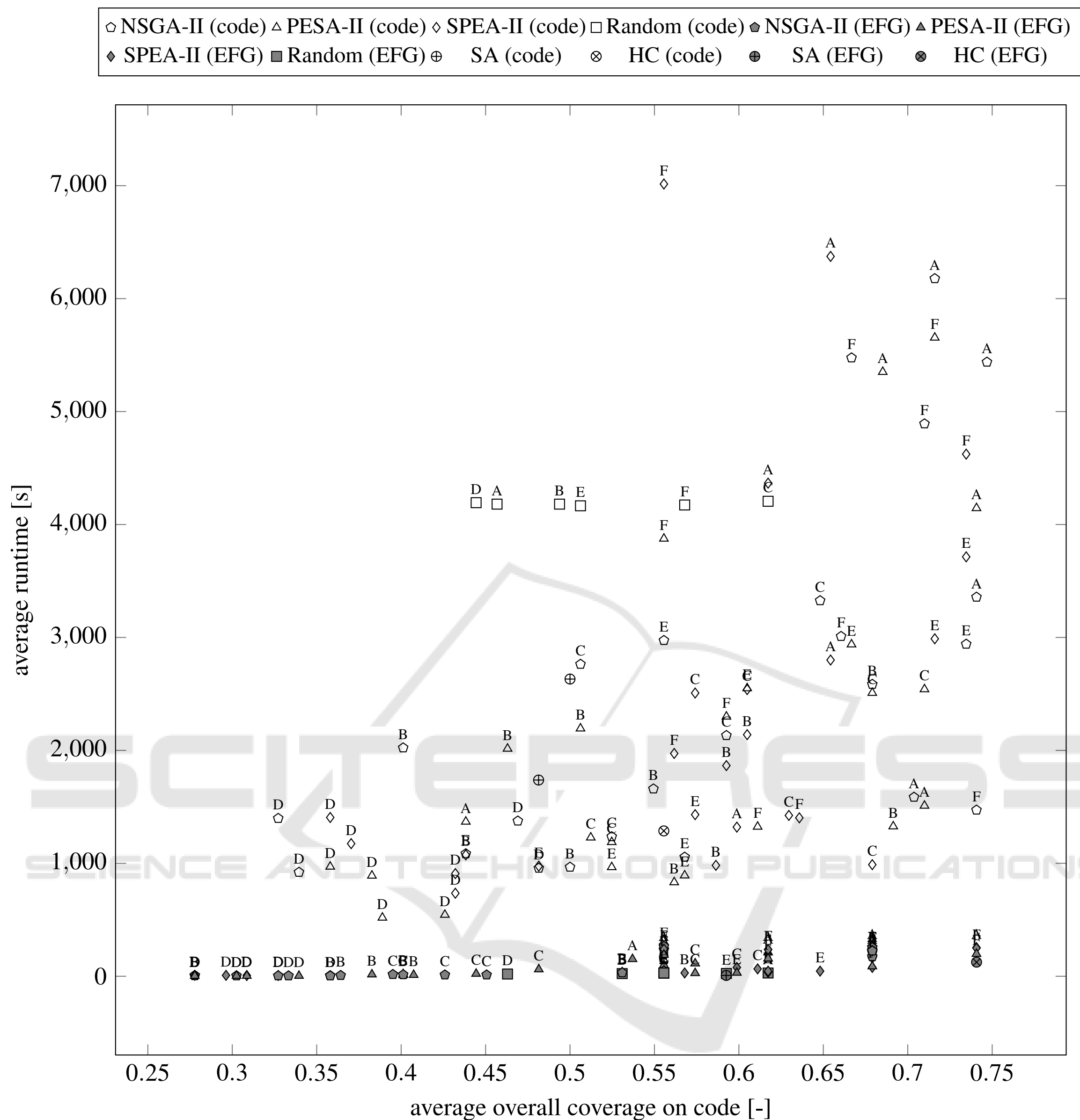
Basic variant	Algorithm	Coverage goal	Mutation Rate	Crossover Rate
A: no length	NSGA-II	EFG	0.2	0.2
B: with minimized length				
C: with minimized length in categories	SPEA-II			
D: single additive objective				
E: single additive objective with normalized length	PESA-II	code	0.8	0.8
F: single objective				

genetic algorithms are also computed with two other types of optimization criteria. First, the multiple objectives are transferred into a single additive objective, meaning that all objectives are weighted and summed up to a single objective. For the EFG-based approaches, the coverage is weighted with 0.9 and the length is weighted with 0.1. For the code-based approaches the three coverage types are weighted with 0.3 each and the length is weighted with 0.1. As the results are not promising, the length is also normalized with the maximum length, in order to be comparable to the coverage which always lies within the range $[0,1]$. Second, only one objective is used. For the EFG-based approaches only the node coverage on the EFG is used and for the code-based approaches only the overall coverage is used.

Summing up, six basic variants are created, each of them is computed with all three genetic algorithms plus the random approaches and each of them is computetd with four different probability setups and two different major optimization goals, EFG-based or code based. An overview of all variants for the genetic algorithms is given in Table 2, resulting in $6 \times 3 \times 2 \times 2 \times 2 = 144$ variants. Note, that every basic variant is also computed using each coverage goal with the random algorithm, thus adding another $6 \times 2 = 12$ variants.

5.2 Results

The results are presented in the scatter plot in Figure 5. Using the coverage on the EFG as optimization goal instead of the code improves the runtime. All EFG-based computations have a smaller runtime than any code-based computations. Speaking of genetic algorithms, variants A, F and E, i.e. the variants not minimizing the length, using a single additive objective with normalized length and using only a single objective, perform best in terms of coverage. Variant A has the highest average coverage for the code based approaches (0.65), whereas variant F has the highest average coverage for the EFG-based approaches (0.64). A also shows the best performance for both approaches (0.7469 code-based and 0.7407 EFG-based), with the best results from F being only slightly worse (0.7407 code-based and 0.7407 EFG-based). Variant F also produces the best test set in terms of runtime, even when achieving higher coverage (see lower right of Figure 5). The simple hill-climbing approach is able to compete with variant F, but only for the EFG-based approaches. The performance of the three different genetic algorithms can not be divided as clearly as the performance of the different variants. Because of the high diversity within and between the different genetic algorithms, none performs significantly better or worse than the others. Also, mutation and crossover rate show either best and worse performances in every configuration. The distinction between overall, average and median coverage as optimization goal only shows an isolated effect on the generated approaches. Summing up, using the EFG as optimization goal may not lead to a smaller coverage on the overall code and always performs better in terms of runtime. Additionally, local search algorithms are able to compete against genetic algorithms in terms of coverage and runtime, and the genetic algorithms perform best when using a single objective.



SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

Figure 5: Scatter plot of the average overall code coverage and average runtime on every configuration. EFG-based approaches are filled in gray, code-based approaches are not filled. The variants are identified with respect to Table 2.

6 CONCLUSION AND OUTLOOK

In this paper, we investigated the automatic generation of test cases for graphical user interfaces. The literature review emphasized the usage of search-based techniques to create GUI tests which have a good coverage of the underlying code base. We used hill-climbing, simulated annealing and several genetic algorithms to produce the GUI tests. Instead of using the EFG only as model for the test generation, the coverage of the EFG was also used as optimization goal. After a detailed evaluation, it seems that the genetic algorithms using only a single objective perform best in terms of runtime and coverage. Moreover, hill climbing was able to compete with these approaches. Using the EFG coverage as optimization goal leads to a coverage of the underlying code base which is equal to the coverage generated with the coverage of the code as optimization goal. Additionally, the EFG-based approaches have a much smaller runtime.

For the future, the employment of the EFG cover-

age as optimization goal has to be studied further. The evaluation presented in this paper has to be performed on more and especially more complex GUIs. Therefore, the presented test generation framework, GUI-toolkit, has to be optimized. It also may be beneficial to incorporate GUITAR for future projects with the presented algorithms and their configurations. The generated test cases may also be used for fair performance tests which depend on highly covering test cases (Menninghaus and Pulvermüller, 2016).

REFERENCES

Aburas, A. and Groce, A. (2016). A Method Dependence Relations Guided Genetic Algorithm. In *International Symposium on Search Based Software Engineering*, pages 267–273. Springer International Publishing.

Arcuri, A. (2012). A Theoretical and Empirical Analysis of the Role of Test Sequence Length in Software Testing for Structural Coverage. *IEEE Transactions on Software Engineering*, 38(3):497–519.

Baresi, L., Lanzi, P. L., and Miraz, M. (2010). TestFul: An Evolutionary Test Approach for Java. In *Third International Conference on Software Testing, Verification and Validation*, pages 185–194. IEEE.

Barr, E. T., Harman, M., McMinn, P., Shahbaz, M., and Yoo, S. (2015). The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering*, 41(5):507–525.

Cadar, C., Dunbar, D., and Engler, D. R. (2008). KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX conference on Operating systems*, pages 209–224.

Carino, S. (2016). *Dynamically Testing Graphical User Interfaces*. PhD thesis.

Chee, Y. M., Colbourn, C. J., Horsley, D., and Zhou, J. (2013). Sequence Covering Arrays. *SIAM Journal on Discrete Mathematics*, 27(4):1844–1861.

Chen, M., Qiu, X., Xu, W., Wang, L., Zhao, J., and Li, X. (2009). UML Activity Diagram-Based Automatic Test Case Generation For Java Programs. *The Computer Journal*, 52(5):545–556.

Corne, D. W., Jerram, N. R., Knowles, J. D., Oates, M. J., and J, M. (2001). PESA-II: Region-based Selection in Evolutionary Multiobjective Optimization. In *Proceedings of the genetic and evolutionary computation conference*.

Csallner, C. and Smaragdakis, Y. (2004). JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050.

Dan, H., Harman, M., Krinke, J., Li, L., and Marginean, A. (2014). Pidgin crasher: searching for minimised crashing GUI event sequences. *SSBSE 2014 - Symposium on Search -Based Software Engineering*.

Deb, K., Agrawal, S., Pratap, A., and Meyarivan, T. (2000). A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-objective Optimization: NSGA-II. In *International Conference on Parallel Problem Solving From Nature*, pages 849–858. Springer Berlin Heidelberg.

Fraser, G. and Arcuri, A. (2011). It is Not the Length That Matters, It is How You Control It. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST)*, pages 150–159. IEEE.

Ganov, S. R., Killmar, C., Khurshid, S., and Perry, D. E. (2008). Test generation for graphical user interfaces based on symbolic execution. In *Proceedings of the 3rd international workshop on Automation of software test*, pages 33–40. ACM.

Godefroid, P., Klarlund, N., and Sen, K. (2005). DART: directed automated random testing. *ACM SIGPLAN Notices*, 40(6):213–223.

Goldberg, D. E. and Deb, K. (1991). A Comparative Analysis of Selection Schemes Used in Genetic Algorithms. *Foundations of Genetic Algorithms*, pages 69–93.

Gross, F., Fraser, G., and Zeller, A. (2012). EXSYST: search-based GUI testing. In *ICSE Proceedings of th International Conference on Software Engineering*, pages 1423–1426. IEEE Press.

Harman, M., Mansouri, S. A., and Zhang, Y. (2012a). Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, 45(1).

Harman, M., McMinn, P., de Souza, J. T., and Yoo, S. (2012b). Search Based Software Engineering: Techniques, Taxonomy, Tutorial. In *Empirical Software Engineering and Verification*, pages 1–59. Springer Berlin Heidelberg.

Ingber, L. and Rosen, B. (1992). Genetic Algorithms and Very Fast Simulated Reannealing: A comparison. *Mathematical and computer modelling*, 16(11):87–100.

Khurshid, S., Visser, W., Păsăreanu, C. S., and Khurshid, S. (2004). Test input generation with java PathFinder. *ACM SIGSOFT Software Engineering Notes*, 29(4):97–107.

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

Laumanns, M., Thiele, L., and Zitzler, E. (2001). SPEA2: Improving the strength Pareto evolutionary algorithm. *Eurogen*, 3242(103):95–100.

McMinn, P. (2011). Search-Based Software Testing: Past, Present and Future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 153–163. IEEE.

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

Menninghaus, M. and Pulvermüller, E. (2016). Towards Using Code Coverage Metrics for Performance Comparison on the Implementation Level. In *the 7th ACM/SPEC*, pages 101–104. ACM.

Mitchell, M., Holland, J. H., and Forrest, S. (1993). When will a genetic algorithm outperform hill climbing? *Ann Arbor*.

Myers, B., Hudson, S. E., and Pausch, R. (2000). Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 7(1):3–28.

Nguyen, B. N., Robbins, B., Banerjee, I., and Memon, A. (2014). GUITAR: an innovative tool for automated testing of GUI-driven software. *Automated Software Engineering*, 21(1):65–105.

Pacheco, C., Lahiri, S. K., and Ernst, M. D. (2007). Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*.

Russell, S. J., Norvig, P., Canny, J. F., Malik, J. M., and Edwards, D. D. (2003). *Artificial intelligence: a modern approach*, volume 2. Prentice hall Upper Saddle River.

Sen, K. and Agha, G. (2006). CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In *International Conference on Computer Aided Verification*, pages 419–423. Springer Berlin Heidelberg.

Soffa, M. L., Pollack, M. E., and Memon, A. M. (2001). Coverage criteria for GUI testing. *ACM SIGSOFT Software Engineering Notes*, 26(5):256–267.

Vieira, M., Leduc, J., Hasling, B., Subramanyan, R., and Kazmeier, J. (2006). Automation of GUI testing using a model-driven approach. In *Proceedings of the 2006 international workshop on Automation of software test*, pages 9–14. ACM.

Yuan, X., Cohen, M. B., and Memon, A. M. (2011). GUI interaction testing: Incorporating event context. *IEEE Transactions on Software Engineering*, 97(4):559–574.

Yuan, X. and Memon, A. M. (2010). Generating Event Sequence-Based Test Cases Using GUI Runtime State Feedback. *IEEE Transactions on Software Engineering*, 36(1):81–95.

Zitzler, E., Deb, K., and Thiele, L. (2006). Comparison of Multiobjective Evolutionary Algorithms: Empirical Results. *Evolutionary computation*, 8(2):173–195.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS