Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 917–923
ISBN 978-83-60810-22-4

Search–Based Testing, the Underlying Engine of Future Internet Testing

Arthur I. Baars*, Kiran Lakhotia‡, Tanja E.J. Vos* and Joachim Wegener†
* Centro de Métodos de Producción de Software (ProS)
Universidad Politecnica de Valencia, Valencia, Spain
{abaars, tvos}@pros.upv.es
†Berner & Mattner, Berlin, Germany
joachim.wegener@berner-mattner.de
‡CREST, University College London, London, United Kingdom
k.lakhotia@cs.ucl.ac.uk

***Abstract*—The Future Internet will be a complex interconnection of services, applications, content and media, on which our society will become increasingly dependent. Time to market is crucial in Internet applications and hence release cycles grow ever shorter. This, coupled with the highly dynamic nature of the Future Internet will place new demands on software testing.**

Search–Based Testing is ideally placed to address these emerging challenges. Its techniques are highly flexible and robust to only partially observable systems. This paper presents an overview of Search–Based Testing and discusses some of the open challenges remaining to make search–based techniques applicable to the Future Internet.

***Index Terms*—evolutionary testing; search–based testing; research topics.**

I. Introduction

FUTURE Internet (FI) applications testing will need to be continuous, post-release testing since the application under test does not remain fixed after its initial release. Services and components could be dynamically added by customers and the intended use of an application could change. Therefore, testing has to be performed continuously, even after an application has been deployed to the customer. The overall aim of the European Funded FITTEST project[1] (ICT-257574) is to develop and evaluate an integrated environment for continuous automated testing, which can monitor a FI application and adapt itself to the dynamic changes observed.

The underlying engine of the FITTEST environment, which will enable automated testing and cope with FI testing challenges like dynamism, self-adaptation and partial observability, will be based on Search–Based Testing (SBT).

The impossibility of anticipating all possible behaviours of FI applications suggests a prominent role for SBT techniques, because they rely on very few assumptions about the underlying problem they are attempting to solve. In addition, stochastic optimisation and search techniques are adaptive and therefore able to modify their behaviour when faced with new unforeseen situations. These two properties - freedom from limiting assumptions and inherent adaptiveness - make SBT approaches ideal for handling FI applications testing.

[1]http://www.facebook.com/FITTESTproject

Since SBT is unfettered by human bias, misguided assumptions and misconceptions about possible ways in which components of a system may combine are avoided. SBT also avoids the pitfalls that are found with a humans' innate inability to predict what lies beyond their conceivable expectations and imagination. However, FI applications give users increasingly more power and flexibility in shaping an application, thus placing exactly these requirements on a human tester.

Past research has shown that SBT is suitable for several types of testing, including functional [55] and non–functional [42] testing, mutation testing [10], regression testing [60], test case prioritization [52], interaction testing [12] and structural testing [20], [25], [28], [32], [41], [43], [54]. The relative maturity of SBT means it will provide a robust foundation upon which to build FI testing techniques. However, despite the vast body of previous and ongoing work in SBT, many research topics remain unsolved. Most of them are outside the scope of the FITTEST project and so the aim of this paper is to draw attention to open challenges in SBT in order to inspire researchers and raise interest in this field.

We have divided the research topics into four categories: Theoretical Foundations, Search Technique Improvements, New Testing Objectives and Tool Environment/Testing Infrastructure.

The rest of the paper is organized as follows. Section II provides background information about SBT. Section III highlights the need for research into the theoretical foundations of SBT, before going on to list areas in which SBT may also be improved in the future in section IV. Testing objectives that remain as yet unsolved in the literature are listed in section V. Section VI presents an overview of the demands placed on tools implementing SBT techniques, before section VII concludes.

II. Search–Based Testing

Search–Based Testing uses meta-heuristic algorithms to automate the generation of test inputs that meet a test adequacy criterion. Many algorithms have been considered in the past, including Parallel Evolutionary Algorithms [2], Evolution

978-83-60810-22-4/$25.00 © 2011 IEEE

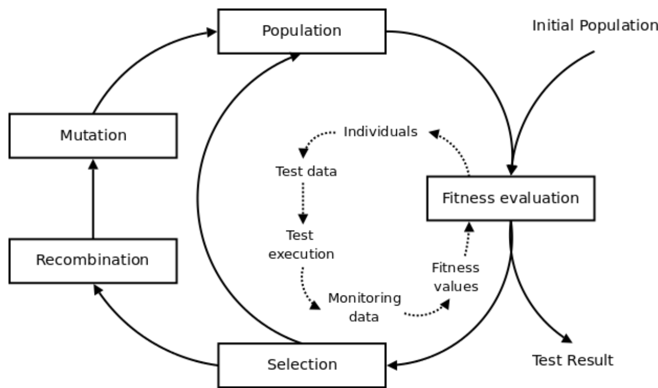

Fig. 1. A typical Evolutionary Algorithm cycle for testing.

Strategies [1], Estimation of Distribution Algorithms [48], Scatter Search [11], Particle Swarm Optimization [57], Tabu Search [13] and the Alternating Variable Method [29]. However, by far the most popular search techniques used in SBT belong to the family of Evolutionary Algorithms in what is known as Evolutionary Testing [22], [56], [55], [39].

Evolutionary Algorithms represent a class of adaptive search techniques based on natural genetics and Darwin's theory of evolution [17], [26]. They are characterized by an iterative procedure that works in parallel on a number of potential solutions to a problem. Figure 1 shows the cycle of an Evolutionary Algorithm when used in the context of Evolutionary Testing.

First, a population of possible solutions to a problem is initialized, usually at random. Starting with randomly generated individuals results in a spread of solutions ranging in fitness because they are scattered around the search–space. This is equal to sampling different regions of the search–space and provides the optimization process with a diverse set of 'building blocks'. However, for the purpose of testing one may want to seed the initial population instead, for example with existing test cases. Seeding allows the optimization process to benefit from existing knowledge about the System Under Test (SUT).

Next, each individual in the population is evaluated by calculating its fitness via a fitness function. The principle idea of an Evolutionary Algorithm is that *fit* individuals survive over time and form even fitter individuals in future generations. This is an analogy to the 'survival of the fittest' concept in natural evolution. A selection strategy is responsible for implementing this behaviour. It selects pairs of individuals from the population for reproduction such that fitter individuals have a higher probability of being selected than less fit ones. Selected individuals are then recombined via a crossover operator. The aim of the crossover operator is to combine good parts from each parent individual to form even better offspring individuals (again analogous to biological reproduction).

After crossover, the resulting offspring individuals may be subjected to a mutation operator. Mutation aims to introduce new information into the gene pool of a population by making random changes to an individual. It is an important operator to prevent the optimization process from stagnation (i.e. crossover operations are not able to produce fitter individuals).

Once offspring individuals have been evaluated, the population is updated according to a re-insertion strategy. For example, one may choose to replace the entire population with the new offspring. More commonly however, only the worst members of a generation are replaced, ensuring fit individuals will always be carried across to the next generation.

An Evolutionary Algorithm iterates until a global optimum is reached (e.g. a test criterium has been satisfied), or another stopping condition is fulfilled. Evolutionary algorithms are generic and can be applied to a wide variety of optimization problems. In order to specialize an evolutionary algorithm for a specific problem, one only needs to define a problem-specific fitness function. For Evolutionary Testing, the fitness function must capture the properties of a test adequacy criterion.

III. Theoretical Foundations

An advantage of meta-heuristic algorithms is that they are widely applicable to problems that are infeasible for analytic approaches. All one has to do is come up with a representation for candidate solutions and an objective function to evaluate those solutions. Despite this flexibililty, meta-heuristic algorithms are not a 'golden hammer' that can be applied to any (testing) problem. Finding a good representation for individuals and designing suitable fitness functions is a hard task and may prove impossible for some problems.

To help a tester overcome these challenges, there is a need for guidelines on what test techniques to use for different testing objectives, different levels of testing or different phases of system development. These guidelines need to extend to how these techniques contribute to the overall reliability and dependability of the SUT, and how efficient and usable their application is. Such guidelines barely exist for traditional testing techniques, and even less is known about SBT. *A good theoretical foundation is missing to tell us which problems can be solved using search–based testing, and for which it is unsuitable*. This problem is not unique to SBT but is experienced throughout the field of Search–Based Software Engineering.

Many search algorithms are available and it is not clear which is the best for a certain problem or fitness landscape. The choice is often made somewhat ad hoc, based on experience or by trying an arbitrary selection of algorithms. To tackle this problem Harman [19] called for a more concerted effort to characterise the difficulty of Software Engineering problems for which search already produced good results. Such characterisations will aid in selecting the most suitable search technique for a particular class of problems. Since Harman's publication, researchers, leading amongst them Arcuri *et al.*, have taken up this call and started to look at theoretical properties of SBT [3], [25], [4], [5].

However, because of the diversity and complexity of the field, empirical studies remain the main means of evaluting a SBT technique. Empirical studies, if well evaluated [6], thus play an essential part in laying the foundations for guidelines

and hence need to be integrated in a general Test & Quality Assurance strategy. The result of these studies should be used to establish a central repository of examples that can act as a *benchmarking suite* for evaluation of new techniques (for functional as well as non-functional properties). A central benchmark would not only contribute to filling the knowledge gap identified by Harman [19], but it would also allow for much better development of experiments, by enabling a more thorough comparison of different testing techniques, search–based as well as others. It can further be used by the research community to gain insights into the strengths and weaknesses of each technique. This insight is invaluable for industry and enables them to make well-founded decisions on which tool or technique to apply.

Finally, theoretical foundations of SBT need to provide an *assessment of the quality of the output produced by SBT*. How good are the generated tests compared to tests derived using other techniques or developed manually by a tester? Most commonly random testing forms the baseline against which any new SBT technique is evaluated. This hardly seems sufficient in an industrial context. Furthermore, figures are needed to assess the reliability of the test results, given that SBT is based on stochastic algorithms. Such assessments are necessary to determine to which extent SBT could be used as a substitute for manual testing and to which extent as an addition to manual tests.

IV. Search Technique Improvements

Many approaches that look promising in the lab are inapplicable in the field, because they do not scale up. However, making a solution scalable is easier said than done. This section highlights some areas that may enable search–based techniques to scale up in practice.

A. Parallel computing

A great advantage of evolutionary computing is that it is naturally parallelizable. Fitness evaluations for individuals can easily be performed in parallel, with hardly any overhead. Search algorithms in general and SBT in particular therefore offer a 'killer application' for the emergent paradigm of ubiquitous user-level parallel computing. Grid-computing for example is the subject of a great number of EU-projects, so there is an opportunity to team up with these projects and apply the technologies developed in that area in SBT.

Another active research area is about achieving parallelization of evolutionary computation through General Purpose Graphics Processing Unit (GPGPU) cards. In the context of evolutionary computation, GPGPU cards are most commonly used to evolve (parts of) programs through Genetic Programming [47], [46], [18], [8], [38], [34]. Genetic Programming uses trees to represent a program (or part of it) and applies genetic operators such as crossover, mutation and selection to a population of trees. Since trees can grow large in size, and possibly have to be executed (interpreted) a number of times (to account for non-determinism) in order to obtain a fitness value, GP can be very computationally expensive. However, GPGPU cards have also been used to parallelize Particle Swarm Algorithms [15], Evolution Strategies [62], Genetic Algorithms [58], [51], [45], [37], [44] and multi-objective problems [59].

Despite the large body of work on evolutionary computation on GPGPU cards, more research is required to utilize GPGPU cards for SBT. Langdon *et al.* [35] have used GPGPU cards to optimize the search for higher order mutants in mutation testing. The goal of mutation testing is to evaluate the quality of a test suite by generating a set of mutant programs, where each mutant represents a possible fault in the program. A mutant is said to be killed if one or more test cases that pass on the original program, either fail for the mutant program, or produce a different output.

The intuition behind mutation testing is that the more mutants a test suite is able to kill, the better it is at finding real faults. Typically, a mutant program only contains a single change. The concept of higher order mutation testing is to introduce multiple changes into one mutant program, because it is argued that higher order mutants more closely represent real faults in software [23].

B. Combining search techniques

Another way of increasing efficiency is to combine different search techniques in the form of Memetic Algorithms. A study by Harman and McMinn [25] shows that a Memetic Algorithm, combining a global and local search, is better suited to generate branch adequate test data than either a global or local search on its own. Again, more research is needed to find out what combination of search techniques is best for which category of test objective.

Sometimes, a search technique may also be combined with a different testing technique. The work of Lakhotia *et al.* [33] combines different search techniques with Dynamic Symbolic Execution (DSE) [16], [49] to improve DSE in the presence of floating point computations. Inkumsah and Xie combined a Genetic Algorithm (GA) and DSE in a framework called Evacon [27] to improve branch adequate testing of object oriented code.

In both studies, the combination of different test data generation techniques outperforms a pure search–based or dynamic symbolic execution–based approach.

C. Multi-objective approaches

In many scenarios, a single-objective formulation of a test objective is unrealistic; testers will want to find test sets that meet several objectives simultaneously in order to maximize the value obtained from the inherently expensive process of running test cases and examining their output. An added benefit of multi-objective optimization is the insight a Pareto front can yield into the trade-offs between different objectives.

By targeting multiple test objectives at the same time, the value obtained from the expensive process of executing the SUT can be maximized. A case study by Harman *et. al.* [24] shows promising results in this direction. The study investigates the performance of a multi-objective GA for the

twin objectives of achieving branch coverage and maximizing dynamic memory allocation.

Besides the work by Harman *et al.*[24], the field of multi-objective testing has remained relatively unexplored. Regression testing, and in particular test suite minimization [61], is one of the few areas where multi-objective algorithms have been applied. The work mentioned in this sub-section on multi-objective testing suggests that search–based techniques are well suited for multi-objective testing tasks, but more research is needed to maximise their potential.

D. Static parameter tuning

Evolutionary Algorithms are a very powerful tool for many problems. However, to obtain the best performance it is crucial that their parameters are well-tuned in order to peform a particular task. This requires a level of expertise in the area of evolutionary computation. Unfortunately testers usually have very little knowledge in this field.

A solution would be that the testing tool automatically tunes its parameters. One approach is to tune parameters a priori, based on the characteristics of the SUT. These could be obtained, for example, from the tester, as a tester has a lot of knowledge about the SUT. In the case of white box testing this information can also stem from (static) analysis of the SUT.

E. Dynamic parameter tuning

A better approach is to let a search algorithm tune itself, based on how well it is proceeding. In this way the search can automatically adapt to a (possibly changing) fitness landscape. This approach seems very promising for search problems that have many different sub-goals or are very dynamic as in FI applications. Every sub-goal represents a new optimisation problem, thus, parameter settings that work well for one sub-goal, might not work well for others.

F. Testability transformations

A testability transformation [21] is a source-to-source program transformation that seeks to improve the performance of a previously chosen test data generation technique. For structural testing, it is possible to remove certain code constructs that cause problems for SBT by applying transformations. This approach is taken, for example, when removing flag variables. A flag variable is a boolean variable, and the flag problem deals with the situation where relatively few input values exist that make the flag adopt one of its two possible values. As a consequence, flag variables introduce large plateaus in the search space, effectively deteriorating a guided search into a random search.

A possible solution to the flag problem is to apply a testability transformation. Many different transformations have been suggested in the literature to deal with different types of flag variables; simple flags [22], function assigned flags [53] and loop-assigned flags [9].

Testability transformations have also been used to remove nesting of conditional statements from source code for the purpose of test data generation [40]. Nested predicates can have a similar effect on SBT to flag variables. If nested conditional statements are linked through a data dependency, the search is missing crucial information on how to satisfy the nested predicates. Eventually a search will be able to obtain this information, but at that point the required test data has been found. The lack of information available during the optimization process can again lead to plateaus in the fitness landscape.

G. Search space size reduction

Another way to improve efficiency of SBT is to use knowledge about the SUT to restrict the size of the search space. For example, knowledge on value ranges could be used to set parameters of the search, such as step size for variation of integers, doubles, etc. Another example is the seeding of test data with literals extracted from the program code. Such strategies could result in a very significant search space reduction and thus potential speed up of the testing process.

There are many ways to uncover information about a SUT. The models and specifications (on system, software, design or component level) could be analysed for information that can be used to improve the test or the search. Static analysis may be used to determine which input-variables are relevant to the search. Irrelevant variables can be left out, reducing the complexity of the input domain. The effect of input domain reduction via irrelevant variable removal has been investigated by Harman *et al.* [20] on two commonly used search algorithms; the Alternating Variable Method, a form of hill climbing, and a Genetic Algorithm. A theoretical and empirical analysis shows that both test data generation methods benefit from a reduced search space.

The bounds of variables or the control flow are other examples of knowledge that can be used by a fitness function to guide the search. Abstract interpretation may be employed to provide equivalence partitions. Such a partition is a range of values for which the SUT behaves the same. During the search one needs to sample only a single element of the partition to cover the whole range, greatly reducing the search space.

H. Minimizing generated individuals

It is often assumed that a fitness evaluation is the most time consuming task of SBT. However, for time consuming functional testing of complex industrial systems, minimizing the number of generated individuals may also be highly desirable. This might be done using an assumption about the ‘potential’ of individuals in order to predict which individuals are likely to contribute to any future improvement. This prediction could be achieved by using information about similar individuals that have been executed in earlier generations.

I. Seeding of test data

Instead of starting with a completely random population, the search may be initialised using results from previous testing activities. This way the search can benefit from prior knowledge. Different strategies for seeding of test data are investigated by Arcuri *et al.* [7].

J. Other interesting questions

What can we learn about the system under test from the execution of a huge number of test data? Is testing the only thing or could we achieve results for other software engineering activities from that?

V. New testing objectives

The bulk of previous work on SBT focuses on structural test objectives, such as branch coverage [20], [25], [28], [32], [41], [43], [54]. Although the topic of branch coverage is extensively researched, there are still many points for improvements:

- dealing with internal states
- dealing with predicates containing complex types, such as strings, dates, data structures and arrays.
- dealing with loops, especially data dependencies between values calculated in loops and used outside the loop.
- how to improve the calculation of the fitness function for combined conditions (logical and, logical or, etc.)

Besides these points, SBT may also be used to address new testing objectives, both structural as well as functional ones. Research is needed to develop an appropriate representation and fitness function for each new testing objective.

Below we describe a number of possible testing objectives. For some it is clear how to implement them, for others the required representations and fitness functions have not yet been designed and are thus open research topics.

A. Run-time error testing

Some examples of run-time errors are: integer overflow, division by zero, memory leaks. For testing run-time errors the objective is to find inputs that trigger such an error. It should be possible to tackle this area by extending the existing approaches for structural testing. For example to test for memory leaks the fitness function should favour test-inputs on which the subject under test uses more memory. The work by Harman *et al.* [24] provides a starting point in this direction. Equally, the work by Tracey *et al.* [50] on exception testing provides a base on which to build.

B. Testing interactive systems

The test input for interactive systems is a sequence of user actions, such as keystrokes and mouse clicks. This is similar to GUI testing. A possible test criterion is the responsiveness of an application. In this case a fitness function should favour combinations of user actions that take a long time to complete. Another objective could be the coverage of different user actions in various combinations.

C. Integration testing

A system usually consists of a number of modules that are more or less independent from each other. These modules should of course be tested in isolation. However, there are also problems that only occur when integrating the different modules. What should the test goals and corresponding fitness functions be for applying SBT to integration tests?

D. Testing parallel, multi-threaded systems

Testing parallel, multi-threaded systems is hard, especially finding bugs that only occur with a certain interleaving of the processes or threads of the SUT. The need for testing becomes ever larger, systems get more and more complex and multi-processor computers are getting more common. An objective for testing such systems is trying to find deadlock or race conditions. The fitness function should somehow favour executions that are close to a deadlock or race condition. Open questions for this type of testing include the representation of individuals (corresponding to interleaving executions of the SUT) and how to measure the 'closeness' to a deadlock.

Again, a combination of different techniques might prove to be desirable. For example, Bohuslav *et al.* [30] use SBT to optimize the configuration of ConTest [14], a concurrency testing tool for JAVA. Their work shows that SBT can help increase synchronisation coverage of code, and thus increase the chance of finding bugs that appear in commercial software due to errors in synchronization of its concurrent threads.

VI. Tool environment/testing infrastructure

Despite a growing momentum of SBT research in academia, SBT is struggling to find any up-take in industry. One of the reasons is a lack of tooling available. The limited tools that are available are often research prototypes, focused on a subset of a particular programming language. Furthermore, problematic language constructs, such as variable length argument functions, pointers and complex data types (variable size arrays, recursive types such as lists, trees, and graphs) remain largely unsupported. Lakhotia *et al.* [31] made some progress towards this, but many more problems remain, some of which are listed in [32].

Central to any SBT technique is a well designed fitness function. Current tools require a tester to manually write code for a fitness function and integrate that function into the tool. In the research prototypes available, this is often not a straightforward task. Lindlar [36] introduces an approach that aims at simplifying the process of designing fitness functions for evolutionary functional testing in order to further increase acceptability by practitioners. The difficult task of designing a suitable fitness function could further be supported by using a wizard based approach.

Finally, non of the currently available SBT tools provide any visualization that might aid a tester. However, visualisation can provide a user with important insights. There are several aspects of visualisation:

1) Visualisation of testing progress, for example how much was tested, testing effort, test coverage, reliability figures.
2) Visualisation of search progress, e.g. how does the search perform, potential for better results when continuing, identify potentials for improving the search and fitness landscape.

Important questions include, which data is useful to a practitioner, and how to display this data in a concise manner?

Displaying the amount of coverage for a small piece of code is easy, one can simply colour the covered code in the editor, or display a coloured control flow graph of the code. However, for a large system consisting of many lines of code, different techniques need to be developed. Further, visualising the fitness landscape is a challenge. It usually has many dimensions, making it hard to display concisely in 2-D.

VII. Conclusion

Search–Based Testing provides the basis on which the European Funded FITTEST project builds. The goal of the FITTEST project is to perform automated, continuous testing of Future Internet applications. Such testing places new demands on any automated testing technique. This paper has presented an overview of Search–Based Testing and discussed some of the open challenges remaining to make search–based techniques applicable to industry as well as the Future Internet. The aim is to encourage further research into these topics such that users of SBT (like the FITTEST project) may benefit from the results.

Acknowledgement

Many people have contributed to the contents of this paper through personal communications and discussions. We would like to thank Mark Harman from University College London; Youssef Hassoun from King's College London; Marc Schoenauer from INRIA; Jochen Hänsel from Fraunhofer FIRST; Dimitar Dimitrov and Ivaylo Spasov from RILA; Dimitris Togias from European Dynamics; Phil McMinn from University of Sheffield; John Clark from the University of York.

References

[1] Enrique Alba and Francisco Chicano. Software Testing with Evolutionary Strategies. In *RISE*, pages 50–65, 8-9 September 2005.

[2] Enrique Alba and Francisco Chicano. Observations in using Parallel and Sequential Evolutionary Algorithms for Automatic Software Testing. *Computers & Operations Research*, 35(10):3161–3183, October 2008.

[3] A. Arcuri. Full theoretical runtime analysis of alternating variable method on the triangle classification problem. In *Search Based Software Engineering, 2009 1st International Symposium on*, pages 113 –121, may 2009.

[4] Andrea Arcuri. Insight knowledge in search based software testing. In *GECCO*, pages 1649–1656. ACM, 2009.

[5] Andrea Arcuri. Theoretical analysis of local search in software testing. In *Stochastic Algorithms: Foundations and Applications*, volume 5792 of *Lecture Notes in Computer Science*, pages 156–168. Springer Berlin / Heidelberg, 2009.

[6] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE*, pages 1–10. ACM, 2011.

[7] Andrea Arcuri, David Robert White, John Clark, and Xin Yao. Multi-objective improvement of software using co-evolution and smart seeding. In *Proc of the 7th Int Conf on Simulated Evolution And Learning (SEAL '08)*, volume 5361 of *LNCS*, pages 61–70. Springer, December 7-10 2008.

[8] Wolfgang Banzhaf, Simon Harding, William B. Langdon, and Garnett Wilson. Accelerating genetic programming through graphics processing units. In *Genetic Programming Theory and Practice VI*, pages 1–19. 2009.

[9] André Baresel, David Binkley, Mark Harman, and Bogdan Korel. Evolutionary testing in the presence of loop-assigned flags: a testability transformation approach. In *ISSTA*, pages 108–118. ACM, 2004.

[10] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. From genetic to bacteriological algorithms for mutation-based testing. *Softw. Test, Verif. Reliab*, 15(2):73–96, 2005.

[11] Raquel Blanco, Javier Tuya, Eugenia Daz, and B. Adenso Daz. A Scatter Search Approach for Automated Branch Coverage in Software Testing. *International Journal of Engineering Intelligent Systems (EIS)*, 15(3):135–142, September 2007.

[12] Myra B. Cohen, Peter B. Gibbons, Warwick B. Mugridge, and Charles J. Colbourn. Constructing test suites for interaction testing. In *ICSE*, pages 38–48. IEEE Computer Society, May 3–10 2003.

[13] Eugenia Díaz, Javier Tuya, Raquel Blanco, and José Javier Dolado. A Tabu Search Algorithm for Structural Software Testing. *Computers & Operations Research*, 35(10):3052–3072, October 2008.

[14] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.

[15] S. Genovesi, R. Mittra, A. Monorchio, and G. Manara. Particle swarm optimization of frequency selective surfaces for the design of artificial magnetic conductors. Technical report, 2006.

[16] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. *ACM SIGPLAN Notices*, 40(6):213–223, June 2005.

[17] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison Wesley, 1989.

[18] Simon Harding and Wolfgang Banzhaf. Distributed genetic programming on gpus using cuda. In *WPABA'09: Proceedings of the Second International Workshop on Parallel Architectures and Bioinspired Algorithms (WPABA 2009)*, pages 1–10. Universidad Complutense de Madrid, September 12-16 2009.

[19] Mark Harman. The current state and future of search based software engineering. In *2007 Future of Software Engineering*, FOSE '07, pages 342–357. IEEE Computer Society, 2007.

[20] Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, and Joachim Wegener. The impact of input domain reduction on search-based test data generation. In *ESEC/SIGSOFT FSE*, pages 155–164. ACM, 2007.

[21] Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. Testability transformation. *IEEE Trans. Softw. Eng.*, 30:3–16, January 2004.

[22] Mark Harman, Lin Hu, Robert Hierons, André Baresel, and Harmen Sthamer. Improving evolutionary testing by flag removal. In *GECCO*, pages 1359–1366. Morgan Kaufmann Publishers, 9-13 July 2002.

[23] Mark Harman, Yue Jia, and William B. Langdon. A manifesto for higher order mutation testing. In *Mutation 2010*, pages 80–89. IEEE Computer Society, 6 April 2010. Keynote.

[24] Mark Harman, Kiran Lakhotia, and Phil McMinn. A multi-objective approach to search-based test data generation. In *GECCO*, pages 1098–1105. ACM, 2007.

[25] Mark Harman and Phil McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Trans. Software Eng*, 36(2):226–247, 2010.

[26] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[27] Kobi Inkumsah and Tao Xie. Evacon: A framework for integrating evolutionary and concolic testing for object-oriented programs. In *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, pages 425–428, November 2007.

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

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

[30] Bohuslav Krena, Zdenek Letko, Tomás Vojnar, and Shmuel Ur. A platform for search-based testing of concurrent software. In *PADTAD*, pages 48–58. ACM, 2010.

[31] Kiran Lakhotia, Mark Harman, and Phil McMinn. Handling dynamic data structures in search based testing. In *GECCO*, pages 1759–1766. ACM, 2008.

[32] Kiran Lakhotia, Phil McMinn, and Mark Harman. An empirical investigation into branch coverage for C programs using CUTE and AUSTIN. *The Journal of Systems and Software*, 83(12):2379–2391, December 2010.

[33] Kiran Lakhotia, Nikolai Tillmann, Mark Harman, and Jonathan De Halleux. Flopsy: search-based floating point constraint solving

for symbolic execution. In *Proceedings of the 22nd IFIP WG 6.1 international conference on Testing software and systems*, ICTSS, pages 142–157. Springer-Verlag, 2010.

[34] William B. Langdon and Mark Harman. Evolving a CUDA kernel from an nvidia template. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2010.

[35] William B. Langdon, Mark Harman, and Yue Jia. Efficient multi-objective higher order mutation testing with genetic programming. *The Journal of Systems and Software*, 83(12):2416–2430, December 2010.

[36] Felix Lindlar. Search-based functional testing of embedded software systems. In *Doctoral Symposium in conjunction with ICST*, 2009.

[37] Ogier Maitre, Laurent A. Baumes, Nicolas Lachiche, Avelino Corma, and Pierre Collet. Coarse grain parallelization of evolutionary algorithms on gpgpu cards with easea. In *GECCO*, pages 1403–1410. ACM, 2009.

[38] Ogier Maitre, Pierre Collet, and Nicolas Lachiche. Fast evaluation of GP trees on GPGPU by optimizing hardware scheduling. In *Proceedings of the 13th European Conference on Genetic Programming, EuroGP 2010*, volume 6021 of *LNCS*, pages 301–312. Springer, 7-9 April 2010.

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

[40] Phil McMinn, David Binkley, and Mark Harman. Empirical evaluation of a nesting testability transformation for evolutionary testing. *ACM Trans. Softw. Eng. Methodol.*, 18:11:1–11:27, June 2009.

[41] Christoph C. Michael, Gary McGraw, and Michael Schatz. Generating software test data by evolution. *IEEE Trans. Software Eng*, 27(12):1085–1110, 2001.

[42] F. Mueller and J. Wegener. A comparison of static analysis and evolutionary testing for the verification of timing constraints. In *RTAS '98: Proc of the Fourth IEEE Real-Time Technology and Applications Symposium*, page 144. IEEE, 1998.

[43] R. P. Pargas, M. J. Harrold, and R. R. Peck. Test-data generation using genetic algorithms. *Journal of Software Testing, Verification and Reliability*, 9(4):263–282, 1999.

[44] Petr Pospchal, Ji Jaro, and Josef Schwarz. Parallel genetic algorithm on the cuda architecture. In *Applications of Evolutionary Computation*, LNCS 6024, pages 442–451. Springer Verlag, 2010.

[45] Jos L. Risco-Martn, Jos M. Colmenar, and Rubn Gonzalo. A parallel evolutionary algorithm to optimize dynamic memory managers in embedded systems. In *WPABA'09: Proceedings of the Second International Workshop on Parallel Architectures and Bioinspired Algorithms (WPABA 2009)*, pages 21–30. Universidad Complutense de Madrid, September 12-16 2009.

[46] Denis Robilliard, Virginie Marion, and Cyril Fonlupt. High performance genetic programming on GPU. In *Proceedings of the 2009 workshop on Bio-inspired algorithms for distributed systems*, pages 85–94. ACM, 2009.

[47] Denis Robilliard, Virginie Marion-Poty, and Cyril Fonlupt. Genetic programming on graphics processing units. *Genetic Programming and Evolvable Machines*, 10(4):447–471, December 2009. Special issue on parallel and distributed evolutionary algorithms, part I.

[48] Ramón Sagarna, Andrea Arcuri, and Xin Yao. Estimation of Distribution Algorithms for Testing Object Oriented Software. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC '07)*, pages 438–444. IEEE, 25-28 September 2007.

[49] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .NET. In *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2008.

[50] Nigel Tracey, John A. Clark, Keith Mander, and John McDermid. Automated test-data generation for exception conditions. *Software Practice and Experience*, 30(1):61–79, 2000.

[51] Shigeyoshi Tsutsui and Noriyuki Fujimoto. Solving quadratic assignment problems by genetic algorithms with gpu computation: a case study. In *GECCO*, pages 2523–2530. ACM, 2009.

[52] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos. Time aware test suite prioritization. In *ISSTA*, pages 1 – 12. ACM Press, 2006.

[53] Stefan Wappler, Joachim Wegener, and André Baresel. Evolutionary testing of software with function-assigned flags. *The Journal of Systems and Software*, 82(11):1767–1779, November 2009.

[54] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(1):841–854, 2001.

[55] Joachim Wegener and Oliver Bühler. Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system. In *Proc of the Genetic and Evolutionary Computation Conf*, pages 1400–1412, 2004.

[56] Joachim Wegener, Kerstin Buhr, and Hartmut Pohlheim. Automatic test data generation for structural testing of embedded software systems by evolutionary testing. In *GECCO*, pages 1233–1240. Morgan Kaufmann, 2002.

[57] Andreas Windisch, Stefan Wappler, and Joachim Wegener. Applying particle swarm optimization to software testing. In *GECCO*, pages 1121–1128. ACM, 2007.

[58] Man Wong and Tien Wong. Implementation of parallel genetic algorithms on graphics processing units. In *Intelligent and Evolutionary Systems*, pages 197–216. 2009.

[59] Man Leung Wong. Parallel multi-objective evolutionary algorithms on graphics processing units. In *GECCO*, pages 2515–2522. ACM, 2009.

[60] Shin Yoo and Mark Harman. Pareto efficient multi-objective test case selection. In *ISSTA*, pages 140–150. ACM, 2007.

[61] Shin Yoo and Mark Harman. Using hybrid algorithm for pareto effcient multi-objective test suite minimisation. *Journal of Systems Software*, 83(4):689–701, April 2010.

[62] Weihang Zhu. A study of parallel evolution strategy: pattern search on a gpu computing platform. In *GECCO*, pages 765–772. ACM, 2009.