Software Engineering Meets Evolutionary Computation

Mark Harman, *University College London*

The concept of evolutionary computation has affected virtually every area of software design, not merely as a metaphor, but as a realistic algorithm for exploration, insight, and improvement.

Software evolves. This fact was recognized early in the history of software engineering.[1] Although the term "software evolution" has come to refer to the process by which successful software installations continually adapt to cater to the changing requirements and environments in which they operate, this is a figurative allusion to Darwinian evolution, not a specifically technical term.

Independently, an entire computer science community has developed that uses the term *evolutionary computation* with a specifically technical meaning: the study of algorithms that incorporate aspects of fitness-guided selection to search a space of candidate solutions for those well-adapted to solving a specific problem. This community has its own conferences and journals that constitute a considerable body of knowledge concerning the best way to develop and apply evolution as a driver for innovation and adaption in an automated metaheuristic optimization process.

Computer scientists have used evolutionary computation to optimize the design of artifacts and processes from an astonishingly wide variety of general engineering disciplines. However, perhaps surprisingly, until the past 10 years, comparatively little work delved into the application of evolutionary computation (and other related search-based optimization) techniques to software engineering. This was the motivation for the foundation of the field now known as *search-based software engineering*, which focuses on the application of search-based optimization techniques to problems in software engineering.

In the past decade, researchers have applied SBSE to a wide range of software engineering topics, including requirements,[2,3] estimation and prediction,[4] design,[5] testing,[6-9] and refactoring.[10,11] Numerous search-based optimization techniques have been used, with a recent comprehensive survey reporting 15 different techniques.[12]

There is no reason why SBSE must be concerned solely with evolutionary computation; other optimization algorithms can and have been used. For example, in the 830 papers in the SBSE repository as of June 2011, 587 use one or more optimization techniques (http://crestweb.cs.ucl.ac.uk/resources/sbse repository). The percentages of papers using each technique are as follows: evolutionary algorithms (no specific style mentioned), 9.0 percent; genetic algorithms, 45.5 percent; genetic programming, 13.5 percent; evolution strategies, 0.6 percent; particle swarm optimization, 1.8 percent; estimation of distribution algorithms, 1.4 percent; and scatter search, 0.8 percent. However, evolutionary computation has been used in 71 percent of all papers on SBSE, and it is the only optimization technique to have been applied to every software engineering application area.[12]

0018-9162/11/$26.00 © 2011 IEEE Published by the IEEE Computer Society

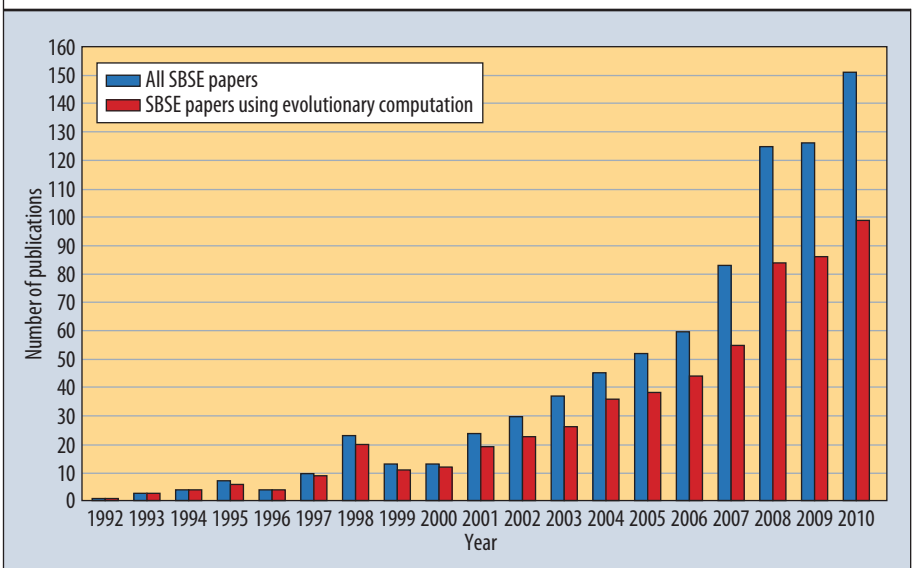

Figure 1. Increase in papers on SBSE and concomitant growth in papers using evolutionary computation for software engineering.

THE GROWTH OF A FIELD

This interest in SBSE in general and evolutionary computation for software engineering in particular has increased rapidly in the past 10 years. Figure 1 shows the growth in publications in SBSE and the concomitant increase in papers within the SBSE field that use evolutionary computation.

SBSE is not only an academic research area—it increasingly provides a set of methods, tools, and techniques that are finding widespread industrial application. The first (and still the most widely) studied area of research targets the application of SBSE to automated test data generation.[12] Known as *search-based software testing*, this widely surveyed area has its own coherent body of literature, and SBST serves as the topic area for a dedicated annual workshop.[6-9]

One of the earliest industrial examples of the application of SBST in industrial practice was at Daimler Chrysler, where Joachim Wegener and his research team implemented a system for evolutionary testing.[13] This system used a genetic algorithm to search for branch-adequate test data, returning a set of test data and associated coverage metrics to the developer. Daimler also experimented with search-based techniques for the functional testing of a parking system[14] and the temporal testing of air bag controllers.[15]

More recently, Microsoft incorporated search-based techniques for incorporating floating-point computation into its PeX software testing tool,[16,17] while Google incorporated multiobjective regression test optimization into its test process.[18] NASA,[19] Motorola,[20] and Ericsson[21] have experimented with SBSE for requirements analysis and optimization, while Ericsson has also used genetic programming (GP) to predict fault slip-through on two large projects.[22]

EVOLUTIONARY COMPUTATION

As with so much of significance in computer science, Alan Turing[23] was the first to introduce the idea that Darwin's theory of evolution might find a sympathetic counterpart in computation, although this very initial formulation was arguably more Lamarckian than Darwinian. John Holland's subsequent experiments with evolutionary computation played an important role in popularizing the field.[24]

It is hard to overstate the impact of evolutionary computation on computational thinking. To give some quantifiable indication of this impact, on 22 June 2011, Holland's book had attracted 24,597 citations (according to Google Scholar), while David E. Goldberg's more recent account had 38,706 citations.[25] These citations come from varied fields of engineering and design, not just from software engineers and computer scientists. There is hardly any aspect of design that has not been profoundly affected by the concept of evolutionary optimization, not merely as a metaphor, but as a realistic algorithm for exploration, insight, and improvement.

With such exceptional and far-reaching impact, it is not surprising that software engineering would also fall under the evolutionary spell. The only surprising aspect of this history is that it took so long. With the benefit of hindsight, it is extraordinary that what is essentially a *software* technique could have failed to find application to *software* engineering for so many years. Perhaps this is simply a reflection of the time it took for the research community to recognize that activities associated with software development are, indeed, essentially *engineering* activities and that, consequently, *optimization* was a natural approach.

Fortunately, despite its slowness to take up evolutionary computation, the software engineering community has recently made up for this sluggishness. Recently, it has even been argued that software's "virtual" nature renders it the most suitable of all engineering materials for optimization techniques.[26]

The formulation of the generic genetic algorithm shown in Figure 2 is taken from a 2003 survey of SBSE.[27] Although there might be variations on the theme in the extensive literature on the topic, the central principles of fitness com-

putation and selection can be found in most formulations of evolutionary computation.

The most widely studied variations on this theme are evolution strategies[28] (which have been used in test data generation[29]) and GP,[30] in which the artifact to be optimized is not a list but a tree—the abstract syntax tree of some programming notation. Of these, GP has been more widely used in SBSE.

The first authors to suggest evolutionary algorithms for software engineering were S. Xanthakis and colleagues,[31] who advocated the use of genetic algorithms for software test data generation. Soon after, Carl Chang and colleagues developed SPMNet, a tool for software project planning based on evolutionary algorithms.[32] In an editorial that presaged the advent of SBSE as an integral field of study, Chang argued for the more widespread application of evolutionary computation in software engineering.[33]

The term SBSE was coined in 2001 to capture the general application of search-based optimization techniques, including evolutionary computation, to software engineering problems.[34] This paper first articulated a vision for a research field of SBSE, arguing that search-based optimization offered a potentially generic, unifying, and potent approach to the spectrum of software engineering activities and products. A recent 10-year retrospective study confirms the growing importance of SBSE research activity.[35]

THE ROLE OF TESTING

Of all the areas of software engineering activity to which researchers have applied SBSE techniques, software testing is both the first area to be tackled and the area that has received the most widespread study.

The general idea behind all approaches to search-based test data generation is that the set of test cases forms a search space and that the test adequacy criterion is coded as a fitness function. For example, to achieve branch coverage, the fitness function assesses how close a test input comes to executing an uncovered branch. In contrast, to find worst case execution time, fitness is simply the duration of execution for the test case in question.

Researchers have used search to attack a variety of testing goals, including structural testing,[36,37] functional testing,[14] safety testing,[38] security testing,[39] robustness testing,[40] stress testing,[41] integration testing,[42] Web application testing,[43] and quality-of-service testing.[44] Most of this work has concerned the problem of generating inputs that provide a test suite that meets a test adequacy criterion. Although the problem of test input generation is often called automated test data generation (ATDG), strictly speaking, without an oracle, only the input is generated. Figure 3 illustrates the generic form of the most common approach in the literature, in which test inputs are generated according to a test adequacy criterion. The test adequacy criterion, the human input to the process, determines the testing goal.

Set generation number, $m := 0$
Choose the initial population, $P(0)$
Evaluate fitness $P(0)$, $F(P_i(0))$
loop
Recombine: $P(m) := R(P(m))$
Mutate: $P(m) := M(P(m))$
Evaluate: $F(P(m))$
Select: $P(m + 1) := S(P(m))$
$m := m + 1$
exit when goal or stopping condition is satisfied
end loop;

Figure 2. A generic genetic algorithm.

The adequacy criterion can be almost any form of testing goal that can be defined and assessed numerically. For example, it can be structural (covering branches, paths, statements), functional (covering scenarios), temporal (finding worst/best case execution times), and so on. SBST's generic nature is a considerable advantage and one reason why many researchers have been able to adapt it to different testing problems.

A human-defined fitness function must capture the adequacy criteria. Once the fitness function for a test adequacy criterion, C, has been defined, the generation of C-adequate test inputs can be automated using SBSE. The SBSE tools that implement different forms of testing all follow the broad structure outlined in Figure 3. They code the adequacy as a fitness, using it to assess the fitness of

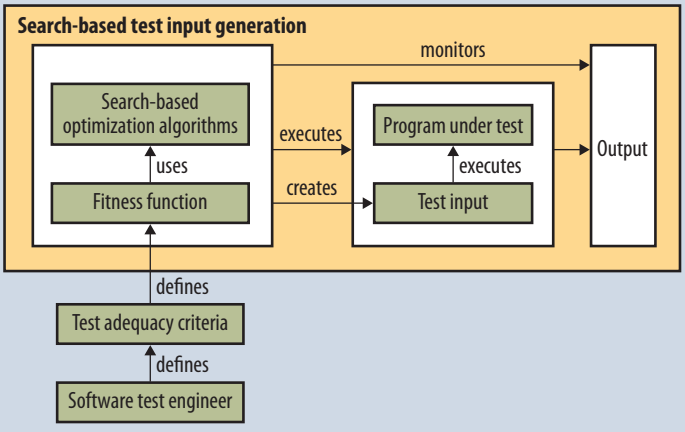

Figure 3. A generic search-based test generation scheme.

candidate test inputs. To assess fitness, the ATDG system must cause the program to be executed for the candidate inputs. The ATDG system then monitors the execution to assess fitness: how well does the input meet the test adequacy criterion?

GENETIC PROGRAMMING

Since program notation is one of the engineering materials with which software engineering is principally concerned, we might expect that GP would have found widespread application in software engineering. Indeed, there has been much work in GP for software engineering, but not always to create the program code for the software system itself. Program construction remains a largely and stubbornly human-centric activity. One goal of the SBSE research agenda is the creation of optimization techniques that can maximally automate the program construction.

There has been much work in genetic programming for software engineering, but not always to create the program code for the software system itself.

This remains an open grand challenge for SBSE, as it does for several other related research communities. Although full achievement of this grand challenge remains some way off, there are many highly effective ways to use GP in SBSE.

Predictive modeling

It is not necessary for GP to evolve a complete program for it to be useful; it merely needs to generate a set of rules that capture some important property of a problem. GP is very good at this. It was first used in software engineering to capture the equations that define good software cost-estimation models.[45] This is a natural fit because these models are often highly nonlinear and exhibit piecewise behavior. The equations that define them are nevertheless relatively small and there is a natural fitness function: the degree of "fit" of the equation to the observed data for a set of projects. Consequently, the space of all possible software cost models forms a natural target for GP work.

Mutation testing

Recent work has shown how GP can be used to search for subtle software faults.[46] A strongly typed GP system uses a restricted C grammar to generate complex faults. The system computes fitness for two objectives: one syntactic and the other semantic.

The syntactic fitness function measures the syntactic similarity of the original program and the faulty version to minimize the difference between the two. The semantic fitness uses a set of test cases to measure the degree of semantic similarity between the two programs, seeking a faulty version that is almost (but not quite) identical to the original program. In this manner, the approach seeks faults that are hard to detect either syntactically or semantically.

Automated bug fixing

Most work on SBSE applied to testing has been concerned with the discovery of faults in software. However, more recently, authors have also turned their attention to the use of SBSE to patch software and fix bugs.[47,48]

Wesley Weimer and colleagues[45] used GP to evolve patches that fix real bugs, winning a Gold Medal at the 2009 Genetic and Evolution Computation Conference for the achievement of human competitive results. The patching process uses what might be called "plastic surgery" to scavenge fragments of code for fault fixing from elsewhere within the program under test. The patches are evolved using GP, for which fitness is computed in terms of passing and failing test cases, thereby fixing the fault and avoiding regression.

OPEN PROBLEMS AND REMAINING CHALLENGES

In previous work on evolutionary computation for software engineering, the evolution model was an extremely simple one. Whether the technique used was a genetic algorithm or a variant such as genetic programming, there was but a single population, evolving according to a single fitness function. Real-world evolution is far more complex because multiple populations interact with one another, effectively solving multiple objective problems with several different fitness functions. Furthermore, in the real world of coevolution, the fitness of one population can critically impact the fitness of another.

Coevolutionary computation

In coevolutionary computation, two or more populations evolve simultaneously, with the fitness of each depending upon the current population of the other. Coevolution takes two forms: *cooperative* and *competitive*.

In cooperative coevolution, the two populations are sympathetic to each other, and the overall system seeks a symbiotic balance between the two. Biology offers many examples of this, such as the symbiotic relationship between pollinating insects and the plants they pollinate. By contrast, in competitive coevolution, one population plays the role of predator and the other is its prey. This coevolution model's goal is to create an "arms race" between the two populations so that each continually improves its fitness with respect to the other.

Researchers have applied both cooperative and competitive coevolution to software engineering problems,

with the competitive form the first to be studied. Konstantinos Adamopoulos and coauthors[49] used it to evolve sets of mutants and sets of test cases, where the test cases act as predators and the mutants as their prey.

Andrea Arcuri and Xin Ya[47] also developed a competitive coevolutionary model of bug fixing, in which one population essentially seeks out patches that can pass test cases, while an oracle produces cases to test the current population of potential patches. In this way, the patch is the prey, while the test cases act as predators.

In the work of both Adamopoulos and colleagues and Arcuri and Yao, the two populations consist of code fragments and test cases. The primary difference is that for Adamopoulos and colleagues, the goal is to find models of faults, whereas in the work of Arcuri and Yao, the goal is to find patches that fix faults.

Any situation in which code and test cases coevolve offers a natural candidate for a competitive, predator-prey model of coevolution. We can expect much more work on this model because it is well-suited to the way software testing takes place. Indeed, many software developers will already be familiar with the feeling that they are the prey for some demanding "testing predator." In software security, too, a similar predator-prey culture is prevalent, and we can expect competitive coevolution to make this more than mere analogy.

Many other aspects of software engineering problems lend themselves to a coevolutionary optimization model because software systems are complex and rich in potential populations that could be productively coevolved using both competitive and cooperative coevolution. While software testing and security problems naturally fit a competitive instantiation, many other software engineering problems are better suited to a cooperative model. For example, components, agents, stakeholder behavior models, designs, cognitive models, requirements, test cases, use cases, and management plans are all important aspects of software systems for which optimization is an important concern.

If they can find a suitable fitness function, researchers could use SBSE to constructively and sympathetically coevolve collaborating subsolutions. For example, in recent work on cooperative coevolution in SBSE, Jian Ren and coauthors[50] explored the way in which different aspects of a software project could be cooperatively coevolved.

Although work on coevolution in software engineering is in its infancy, it seems likely that this area will be the subject of much interest over the coming years because of the close fit to complex software engineering problems.

Scalability

Scalability is probably the biggest generic challenge facing software engineering. Although processing power is increasing, the scale and complexity of software systems is increasing at least as fast. Techniques for software development, testing, and verification are required that can scale to meet this challenge.

Fortunately, SBSE techniques such as evolutionary computation are highly scalable because they can be easily parallelized. Genetic algorithms involve a population to which the same fitness function is applied repeatedly. There is no reason why the entire population's fitness cannot be evaluated on an SIMD architecture in a single step. Several authors have demonstrated that this parallelism can be exploited in SBSE work to obtain scalability through distributed computation.[51-53]

If they can find a suitable fitness function, researchers could use SBSE to constructively and sympathetically coevolve collaborating subsolutions.

Recent work has shown how this parallelism can be exploited on general-purpose graphical processing units (GPGPUs). Shin Yoo and coauthors[54] showed how to map the problem of regression testing onto a GPGPU, solving multiobjective regression test selection problems up to 20 times faster than conventional architectures. The results also outperformed multicore CPU computation and made possible large-scale, real-world smoke-testing optimization that would have been impossible without the scalability offered by GPGPUs.

This work opens up the prospect of an inexpensive and effective means of unleashing the parallelism latent in evolutionary computation and harnessing its scalability for software engineering applications. The importance of scalability and the rapid industry-wide migration to multicore computing indicate that parallel SBSE is a topic likely to receive considerable attention in the coming years.

Interactive evolution

Although researchers have successfully applied SBSE to the automation of many areas of software engineering, some aspects of this discipline cannot be automated. The designs we create are ultimately intended for human consumption, and human judgment will always play a critical role in almost every design process. It is difficult to capture some design aspects in a fitness function, and software engineering design challenges are no exception.

Interactive evolution offers an approach that places the human in the fitness computation loop.[55] SBSE researchers have used interactive evolution to allow humans to play a role in defining a software engineering design's fitness. For example, Christopher L. Simons and coauthors used interactive evolution to search for good designs for an object-oriented class diagram.[56]

This work is intended to support early life-cycle design activity in which human judgment is critical. Since software engineering design involves many aspects of human judgment and remains a labor-intensive engineering activity, we can expect much future interest in interactive forms of evolutionary computation for software engineering.

Characterization of landscapes and problems

As in any field, SBSE generated initial excitement that led to widespread uptake and development of a broad range of applications. Researchers attacked many different aspects of software engineering using evolutionary computation, with early successes stimulating further work and wider development. However, after the initial "gold rush," there comes a time for consolidation.

In fact, this closely mirrors the behavior of the evolutionary compilation process itself. That is, the early stages of the evolutionary process tend to favor exploration of new parts of the search space, while the later phases tend to favor the exploitation of productive areas, seeking improved fitness within them.

Seeking optimal and near-optimal solutions is natural because it lies at the core of all engineering approaches and pervades all engineering activity.

There is evidence for the emergence of this second phase of activity in the software engineering research community. Although work continues on developing exciting new areas for applying SBSE for the first time, a consistent effort is emerging that seeks to develop a deeper understanding and scientific basis for the results obtained thus far. Both the theoretical study of algorithm performance and the theoretical and empirical analysis of problems is now common in the well-explored areas of SBST.[36,37,57] But as the field matures, we can expect that there will be an increasing proportion of activity in the area of theoretical characterization of problems and algorithms that can be used to resolve them.

Hyperheuristic optimization

Seeking optimal and near-optimal solutions is natural because it lies at the core of all engineering approaches and pervades all engineering activity. We have seen from SBSE's widespread application that its generic and highly applicable nature has led to many tailor-made optimization algorithms for problem instances across the spectrum of software engineering activities.

These algorithms have proved to be astonishingly effective, and researchers and practitioners alike are using them. However, custom-made development will always remain a slow and expensive process compared to the potential offered by hyperheuristics—algorithms devised by combining simpler heuristics to efficiently solve computational search problems—making *hyperheuristic software engineering* a natural next step for the research community.[58]

A primary goal of SBST work is test input generation. However, this work does not address the problem of the generation of test cases, which are, in their most abstract form, input-output pairs. SBST tends to generate inputs that achieve various test adequacy criteria but not the expected outputs; checking the test cases therefore typically remains an unautomated aspect of the overall test activity. Determining the correct output for a given input is the "oracle problem" in software testing. While there has been work on using SBST to reduce the cost of human oracle checking,[59] little work has combined SBST with oracle construction. One promising recent approach is the work by Gordon Fraser and Andreas Zeller,[60] which uses SBST to generate test data to kill mutants and simultaneously seeks to construct a partial oracle.

Applications in emerging areas

Much of the SBSE literature focuses on what might be termed "traditional" software engineering application areas, such as requirements, project planning, design, implementation, testing, and maintenance. However, there is no reason why SBSE cannot be applied to other forms of software development. Any software engineering problem in which there are objectives to be optimized offers opportunities for search-based optimization.

For example, because researchers could optimize many competing objectives for agile computing paradigms, the application of evolutionary computation to these paradigms cannot be far off. In addition, initial work has been reported formulating the challenges related to the emerging cloud-based paradigm.[61] It is also likely that the increasing use of mobile and embedded systems will lead to increased nonfunctional properties, for which SBSE has already been applied.[6] Even in the comparatively "blue sky" field of quantum computation, SBSE has proved to be a potential source of both solutions to potential challenges[62] as well as a beneficiary of developments in quantum computation.[63]

WHY IS EVOLUTIONARY COMPUTATION SO POPULAR?

Evolutionary algorithms have proved to be the most widely applied of all SBSE techniques.[12] However, this raises the question of why evolutionary search algorithms should have proved so popular. Is there a particular technical reason for this prevalence?

It is true that these algorithms are widely applicable because they can cater to single and multiple objectives,

they can incorporate human fitness evaluation, and they can be easily parallelized. These important technical considerations provide compelling motivations for the consideration of evolutionary computation as a natural technique for optimizing software engineering. However, evolutionary algorithms also offer many interesting variations and numerous parameters to be tuned and explored. These can prove to be as enticing to researchers as they are frustrating to practitioners.

We must be wary of the unquestioning adoption of evolutionary algorithms merely because they are popular and widely applicable or because, historically, other researchers have adopted them for SBSE problems; none of these are scientific motivations for adoption. Part of the importance of the second phase of research activity concentrating on characterization and algorithmic complexity analysis lies in the way in which it can help determine the best algorithm for a particular software engineering problem. This will not always be an evolutionary algorithm. For example, there is evidence to suggest that, for branch-adequate test data generation, local search algorithms may be better suited to fast and effective coverage.[37]

PREVIOUS RESEARCH

Since the term SBSE was coined in 2001, there has been an explosion of interest and activity in this area, creating a considerable body of literature.

Surveys covering requirements,[3] predictive modeling,[4] design,[5] and testing[6-9] provide an excellent analysis of the application of SBSE to these topic-specific areas. Other surveys of specific software testing topics include sections devoted to search-based techniques, such as recent reports covering mutation testing[64] and regression test optimization.[65]

Some reports focus on specific aspects of SBSE, for example, the use of metrics as fitness functions,[66] structured reviews of the empirical evidence concerning test data generation,[7] nonfunctional testing,[6] and predictive modeling.[4] Other reports identify open problems and future SBSE research agendas for program comprehension,[67] predictive modeling,[68] and testability transformation.[69]

The systematic reviews provide a convenient summary of the current empirical evidence base for SBSE, while the agenda-defining papers may provide potential topics and ideas for future research projects and PhD theses. Students interested in finding a more introductory tutorial might consider a forthcoming summary that provides a self-contained introduction to SBSE and offers practical guidance and advice.[70] Finally, some surveys seek to cover the entire area of SBSE. A 2003 survey that covered project planning, design, maintenance, and testing,[27] and the overview and introductory paper from ICSE 2007,[71] are both widely cited sources of general information about SBSE. A comprehensive 2009 survey mapped the entire field, providing the first data on emerging SBSE trends,[12] and a 2011 retrospective provides a bibliometric analysis of the SBSE literature.[35]

Researchers can use evolutionary computation algorithms to optimize any artifact of design for which a suitable fitness function can be defined. Software engineering provides a rich and varied source of such artifacts together with metrics that can be readily adapted to their new role as fitness functions. This has led to 10 years of rapid developments in search-based software engineering. Evolutionary computation for software engineering forms a substantial part of this overall growth in SBSE research and practice. There are also many exciting emerging developments in evolutionary computation for software engineering for which we can expect a great deal of future work. C

Acknowledgments

As of July 2011, SBSE research had rapidly grown to include publications by more than 800 authors from 270 institutions in more than 40 countries. Although I cannot acknowledge each person here by name, I am grateful to this community for the many stimulating conversations about SBSE over the past 10 years. In particular, I thank Bill Langdon, Richard Torkar, Wes Weimer, and Xin Yao for their comments on an earlier draft of this article. I also thank Yuanyuan Zhang for many discussions on SBSE and her assistance in the production of Figures 1 and 3 as well as her tireless work on the SBSE repository, which provides an excellent resource for this rapidly growing community.

References

1. M.M. Lehman, "On Understanding Laws, Evolution and Conservation in the Large Program Life Cycle," *J. Systems and Software*, vol. 1, no. 3, 1980, pp. 213-221.
2. A.J. Bagnall, V.J. Rayward-Smith, and I.M. Whittley, "The Next Release Problem," *Information and Software Technology*, Dec. 2001, pp. 883-890.
3. Y. Zhang, A. Finkelstein, and M. Harman, "Search-Based Requirements Optimisation: Existing Work and Challenges," *Proc. Int'l Working Conf. Requirements Eng.: Foundation for Software Quality* (REFSQ 08), LNCS 5025, Springer, 2008, pp. 88-94.
4. W. Afzal and R. Torkar, "On the Application of Genetic Programming for Software Engineering Predictive Modeling: A Systematic Review," *Expert Systems Applications*, vol. 38, no. 9, 2011, pp. 11984-11997.
5. O. Räihä, "A Survey on Search-Based Software Design," *Computer Science Rev.*, vol. 4, no. 4, 2010, pp. 203-249.
6. W. Afzal, R. Torkar, and R. Feldt, "A Systematic Review of Search-Based Testing for Non-Functional System Properties," *Information and Software Technology*, vol. 51, no. 6, 2009, pp. 957-976.
7. S. Ali et al., "A Systematic Review of the Application and Empirical Investigation of Search-Based Test-Case Generation," *IEEE Trans. Software Eng.*, vol. 36, no. 6, 2010, pp. 742-762.

8. M. Harman, "Automated Test Data Generation Using Search-Based Software Engineering," *Proc. 2nd Int'l Workshop Automation of Software Test* (AST 07), IEEE CS Press, 2007, p. 2.
9. P. McMinn, "Search-Based Software Test Data Generation: A Survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, 2004, pp. 105-156.
10. M. O'Keeffe and M. Ó Cinnéide, "Search-Based Software Maintenance," *Proc. Conf. Software Maintenance and Reengineering* (CSMR 06), IEEE CS Press, 2006, pp. 249-260.
11. M. Harman and L. Tratt, "Pareto Optimal Search-Based Refactoring at the Design Level," *Proc. 9th Ann. Conf. Genetic and Evolutionary Computation* (GECCO 07), ACM Press, 2007, pp. 1106-1113.
12. M. Harman, A. Mansouri, and Y. Zhang, *Search-Based Software Engineering: A Comprehensive Analysis and Review of Trends, Techniques, and Applications*, tech. report TR-09-03, Dept. of Computer Science, King's College London, 2009.
13. J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary Test Environment for Automatic Structural Testing," *Information and Software Technology*, vol. 43, no. 14, 2001, pp. 841-854.
14. J. Wegener and O. Bühler, "Evaluation of Different Fitness Functions for the Evolutionary Testing of an Autonomous Parking System," *Proc. Genetic and Evolutionary Computation Conf.* (GECCO 2004), LNCS 3103, Springer, 2004, pp. 1400-1412.
15. J. Wegener and F. Mueller, "A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints," *Real-Time Systems*, vol. 21, no. 3, 2001, pp. 241-268.
16. C. Cadar et al., "Symbolic Execution for Software Testing in Practice: Preliminary Assessment," *Proc. 33rd Int'l Conf. Software Eng.* (ICSE11), ACM Press, 2011, pp. 1066-1071.
17. K. Lakhotia et al., "FloPSy—Search-Based Floating Point Constraint Solving for Symbolic Execution," *Proc. 22nd IFIP Int'l Conf. Testing Software and Systems* (ICTSS 10), LNCS 6435, Springer, 2010, pp. 142-157.
18. S. Yoo, R. Nilsson, and M. Harman, "Faster Fault Finding at Google Using Multiobjective Regression Test Optimsation," *Proc. 8th European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations of Software Eng.* (ESEC/FSE 11), ACM Press, Sept. 2011; http://2011.esec-fse.org/industrial-track.
19. S.L. Cornford et al., "Optimizing Spacecraft Design—Optimization Engine Development: Progress and Plans," *Proc. IEEE Aerospace Conf.* (AeroConf 03), IEEE Press, 2003, pp. 3681-3690.
20. P. Baker et al., "Search-Based Approaches to Component Selection and Prioritization for the Next Release Problem," *Proc. 22nd Int'l Conf. Software Maintenance* (ICSM 06), IEEE Press, 2006, pp. 176-185.
21. Y. Zhang et al., "Today/Future Importance Analysis," *Proc. ACM Genetic and Evolutionary Computation Conf.* (GECCO 10), ACM Press, 2010, pp. 1357-1364.
22. W. Afzal et al., "Search-Based Prediction of Fault-Slip-Through in Large Software Projects," *Proc. 2nd Int'l Symp. Search-Based Software Eng.* (SSBSE 10), IEEE CS Press, 2010, pp. 79-88.
23. A.M. Turing, "Computing Machinery and Intelligence," *Mind*, Jan. 1950, pp. 433-460.
24. J.H. Holland, *Adaption in Natural and Artificial Systems*, MIT Press, 1975.
25. D.E. Goldberg, *Genetic Algorithms in Search, Optimization & Machine Learning*, Addison-Wesley, 1989.
26. M. Harman, "Why the Virtual Nature of Software Makes It Ideal for Search-Based Optimization," *Proc. 13th Int'l Conf. Fundamental Approaches to Software Eng.* (FASE 10), IEEE CS Press, 2010, pp. 1-12.
27. J. Clark et al., "Reformulating Software Engineering as a Search Problem," *IEE Proceedings—Software*, vol. 150, no. 3, 2003, pp. 161-175.
28. H.-P. Schwefel and T. Bäck, "Artificial Evolution: How and Why?" *Genetic Algorithms and Evolution Strategy in Engineering and Computer Science*, D. Quagliarella et al., eds., John Wiley & Sons, 1998, pp. 1-19.
29. E. Alba and F. Chicano, "Observations in Using Parallel and Sequential Evolutionary Algorithms for Automatic Software Testing," *Computers & Operations Research*, Oct. 2008, pp. 3161-3183.
30. J.R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, 1992.
31. S. Xanthakis et al., "Application of Genetic Algorithms to Software Testing," *Proc. 5th Int'l Conf Software Eng.* (ICSE 92), IEEE CS Press, 1992, pp. 625-636.
32. C.K. Chang et al., "SPMNet: A Formal Methodology for Software Management," *Proc. 18th Ann. Int'l Computer Software and Applications Conf.* (COMPSAC 94), IEEE CS Press, 1994, p. 57.
33. C.K. Chang, "Changing Face of Software Engineering," *IEEE Software*, vol. 11, no. 1, 1994, pp. 4-5.
34. M. Harman and B.F. Jones, "Search-Based Software Engineering," *Information and Software Technology*, Dec. 2001, pp. 833-839.
35. F.G. Freitas and J.T. Souza, "Ten Years of Search-Based Software Engineering: A Bibliometric Analysis," *Proc. 3rd Int'l Symp. Search-Based Software Eng.* (SSBSE 11), Sept. 2011; www.springerlink.com/content/q2tr783534pj4444.
36. A. Arcuri, "It Does Matter How You Normalise the Branch Distance in Search-Based Software Testing," *Proc. Int'l Conf. Software Testing* (ICST 10), IEEE CS Press, 2010, pp. 205-214.
37. M. Harman and P. McMinn, "A Theoretical and Empirical Study of Search-Based Testing: Local, Global and Hybrid Search," *IEEE Trans. Software Eng.*, vol. 36, no. 2, 2010, pp. 226-247.
38. A. Baresel, H. Sthamer, and J. Wegener, "Applying Evolutionary Testing to Search for Critical Defects," *Proc. Conf. Genetic and Evolutionary Computation* (GECCO 04), LNCS 3103, Springer, 2004, pp. 1427-1428.
39. C. Del Grosso et al., "Improving Network Applications Security: A New Heuristic to Generate Stress Testing Data," *Proc. Conf. Genetic and Evolutionary Computation* (GECCO 05), ACM Press, 2005, pp. 1037-1043.
40. A.C. Schultz, J.J. Grefenstette, and K.A. De Jong, "Test and Evaluation by Genetic Algorithms," *IEEE Expert* (also *IEEE Intelligent Systems and Their Applications*), vol. 8, no. 5, 1993, pp. 9-14.
41. L.C. Briand, Y. Labiche, and M. Shousha, "Stress Testing Real-Time Systems with Genetic Algorithms," *Proc. Conf. Genetic and Evolutionary Computation* (GECCO 05), ACM Press, 2005, pp. 1021-1028.

42. L.C. Briand, J. Feng, and Y. Labiche, "Using Genetic Algorithms and Coupling Measures to Devise Optimal Integration Test Orders," *Proc. 14th Int'l Conf. Software Eng. and Knowledge Eng.* (SEKE 02), ACM Press, 2002, pp. 43-50.
43. N. Alshahwan and M. Harman, "Automated Web Application Testing Using Search-Based Software Engineering," *Proc. IEEE/ACM Int'l Conf. Automated Software Eng.* (ASE 11), Nov. 2011, to appear.
44. M. Di Penta et al., "Search-Based Testing of Service Level Agreements," *Proc. 9th Ann. Conf. Genetic and Evolutionary Computation* (GECCO 07), ACM Press, 2007, pp. 1090-1097.
45. J.J. Dolado and L. Fernandez, "Genetic Programming, Neural Networks and Linear Regression in Software Project Estimation," *Proc. Int'l Conf. Software Process Improvement, Research, Education and Training* (INSPIRE III), British Computer Soc., 1998, pp. 157-171.
46. W.B. Langdon, M. Harman, and Y. Jia, "Efficient Multi-objective Higher Order Mutation Testing with Genetic Programming," *J. Systems and Software*, vol. 83, no. 12, 2011, pp. 2416-2430.
47. A. Arcuri and X. Yao, "A Novel Co-evolutionary Approach to Automatic Software Bug Fixing," *Proc. IEEE Congress on Evolutionary Computation* (CEC 08), IEEE CS Press, 2008, pp. 162-168.
48. W. Weimer et al., "Automatically Finding Patches Using Genetic Programming," *Proc. Int'l Conf. Software Eng.* (ICSE 09), IEEE CS Press, 2009, pp. 364-374.
49. K. Adamopoulos, M. Harman, and R.M. Hierons, "How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-Evolution," *Proc. Conf. Genetic and Evolutionary Computation* (GECCO 04), LNCS 3103, Springer, 2004, pp. 1338-1349.
50. J. Ren, M. Harman, and M. Di Penta, "Cooperative Co-evolutionary Optimization on Software Project Staff Assignments and Job Scheduling," *Proc. 3rd Int'l Symp. Search-Based Software Eng.* (SSBSE 11), Sept. 2011; www.springerlink.com/content/a611526179255p80.
51. F. Asadi, G. Antoniol, and Y.-G. Guéhéneuc, "Concept Location with Genetic Algorithms: A Comparison of Four Distributed Architectures," *Proc. 2nd Int'l Symp. Search-Based Software Eng.* (SSBSE 10), IEEE CS Press, 2010, pp. 153-162.
52. K. Mahdavi, M. Harman, and R.M. Hierons, "A Multiple Hill Climbing Approach to Software Module Clustering," *Proc. IEEE Int'l Conf. Software Maintenance* (ICSM 03), IEEE CS Press, 2003, pp. 315-324.
53. B.S. Mitchell, M. Traverso, and S. Mancoridis, "An Architecture for Distributing the Computation of Software Clustering Algorithms," *Proc. Working Conf. Software Architecture* (WICSA 01), IEEE CS Press, 2001, pp. 181-190.
54. S. Yoo, M. Harman, and S. Ur, "Highly Scalable Multi-Objective Test Suite Minimisation Using Graphics Cards," *Proc. 3rd Int'l Symp Search-Based Software Eng.* (SSBSE 11), Sept. 2011; www.springerlink.com/content/5381g43g1pp41811.
55. P. Funes et al., "Interactive Multiparticipant Task Allocation," *Proc. IEEE Congress on Evolutionary Computation* (CEC 04), IEEE Press, 2004, pp. 1699-1705.
56. C.L. Simons, I.C. Parmee, and R. Gwynllyw, "Interactive, Evolutionary Search in Upstream Object-Oriented Class Design," *IEEE Trans. Software Eng.*, vol. 36, no. 6, 2010, pp. 798-816.
57. P.K. Lehre and X. Yao, "Runtime Analysis of Search Heuristics on Software Engineering Problems," *Frontiers of Computer Science in China*, vol. 3, no. 1, 2009, pp. 64-72.
58. E.K. Burke et al., "A Graph-Based Hyper-Heuristic for Educational Timetabling Problems," *European J. Operational Research*, vol. 176, no. 1, 2007, pp.177-192.
59. M. Harman et al., "Optimizing for the Number of Tests Generated in Search-Based Test Data Generation with an Application to the Oracle Cost Problem," *Proc. 3rd Int'l Workshop Search-Based Software Testing* (SBST 10), IEEE CS Press, 2010, pp. 182-191.
60. G. Fraser and A. Zeller, "Mutation-Driven Generation of Unit Tests and Oracles," *Proc. 19th Int'l Symp. Software Testing and Analysis* (ISSTA 10), ACM Press, 2010, pp. 147-158.
61. H. Wada et al., "Evolutionary Deployment Optimization for Service-Oriented Clouds," *Software Practice and Experience*, vol. 41, no. 5, 2011, pp. 469-493.
62. P. Massey, J.A. Clark, and S. Stepney, "Human-Competitive Evolution of Quantum Computing Artefacts by Genetic Programming," *Evolutionary Computation*, vol. 14, no. 1, 2006, pp. 21-40.
63. R.J. Hall, "A Quantum Algorithm for Software Engineering Search," *Proc. Automated Software Eng.* (ASE 2009), IEEE CS Press, 2009, pp. 40-51.
64. Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Trans. Software Eng.*, 2011, doi:10.1109/TSE.2010.62.
65. S. Yoo and M. Harman, "Regression Testing Minimisation, Selection and Prioritisation: A Survey," *J. Software Testing, Verification and Reliability*, 2011, doi:10.1002/stvr.430.
66. M. Harman and J. Clark, "Metrics Are Fitness Functions Too," *Proc. 10th Int'l Software Metrics Symp.* (Metrics 04), IEEE CS Press, 2004, pp. 58-69.
67. M. Harman, "Search-Based Software Engineering for Program Comprehension," *Proc. 15th Int'l Conf. Program Comprehension* (ICPC 07), IEEE CS Press, 2007, pp. 3-13.
68. M. Harman, "The Relationship Between Search-Based Software Engineering and Predictive Modeling," *Proc. 6th Int'l Conf. Predictive Models in Software Eng.* (PROMISE 10), 2010; www.cs.ucl.ac.uk/staff/mharman/promise-keynote.pdf.
69. M. Harman, "Open Problems in Testability Transformation," keynote, *Proc. 1st Int'l Workshop Search-Based Testing* (SBT 08), 2008; http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4567008.
70. M. Harman et al., "Search-Based Software Engineering: Techniques, Taxonomy, Tutorial," *Empirical Software Engineering and Verification: LASER 2009-2010*, B. Meyer and M. Nordio, eds., Springer, 2012.
71. M. Harman, "The Current State and Future of Search-Based Software Engineering," *Proc. Future of Software Eng.* (FOSE 07), IEEE CS Press, 2007, pp. 342-357.

Mark Harman *leads the Software Systems Engineering Group and is director of the Centre for Research on Evolution Search and Testing in the Department of Computer Science at University College London. His research focuses on source code analysis and testing, and he was instrumental in founding the field of search-based software engineering. Contact him at mark.harman@ucl.ac.uk.*