Search Based Software Engineering for Program Comprehension

Mark Harman
CREST,
King's College London
Strand, London, WC2R 2LS
United Kingdom

Abstract

Search Based Software Engineering (SBSE) is an approach to software engineering in which search based optimization algorithms are used to identify optimal or near optimal solutions and to yield insight. SBSE techniques can cater for multiple, possibly competing objectives and/or constraints and applications where the potential solution space is large and complex. Such situations are common in software engineering, leading to an increasing interest in SBSE. This paper provides a brief overview of SBSE, explaining some of the ways in which it has already been applied to program–comprehension related activities. The paper also outlines some possible future applications of and challenges for the further application of SBSE to Program Comprehension.

1. Introduction

Search Based Software Engineering (SBSE) aims to apply search based optimization algorithms to problems drawn from software engineering. This approach to optimization is a natural one, because so many engineering applications are characterised by many complex and competing objectives in large search spaces. In these situations, automated optimization techniques are natural candidates.

Search based optimization techniques are widely used in other engineering domains, for example mechanical engineering [46], chemical engineering [12], biomedical engineering [67, 69, 79], civil engineering [4, 10, 27, 42] and electronic engineering [9, 22, 63]. Software engineering is coming of age as a mature engineering discipline. It is, therefore, natural to ask:

> "Why not apply search based optimization to software engineering?"

This question was posed by Harman and Jones in 2001 [34], in a 'manifesto paper' that drew attention to the huge potential of automated search based optimization for software engineering problems and also in an initial survey of early work on Search Based Testing, Modularisation and Cost Estimation by Clark et al. [19]. However, work on search based approaches to software engineering problems dates back much further, with work on optimization for testing starting as early as 1976, when Miller and Spooner [60] used classical optimization techniques for test data generation. In 1992, Xanthakis et al. were the first[1] to apply meta heuristic search to software engineering, when they used search based optimization for test data generation [81].

Since the 2001 manifesto paper, there has been a great deal of activity in this area, with SBSE being increasingly applied to a wide range of diverse areas within software engineering. There have also been several workshops, conferences and special issues on SBSE.

Search based optimization techniques have proved to be highly applicable in software engineering. For example, in the past five years, Search Based Software Engineering has been applied to requirements engineering [5, 82], project planning and cost estimation [2, 3, 44], testing [6, 7, 13, 16, 33, 48, 58, 78], automated maintenance [14, 26, 32, 61, 62, 66, 71, 72], service-oriented software engineering [18], compiler optimization [20] and quality assessment [15, 43]. The application of optimization techniques to software testing has recently witnessed intense activity. In 2004 there was sufficient material to warrant a survey paper on this sub-area of activity [57]. However, as the list of application areas above indicates, optimization can be applied right across the spectrum of software engineering activity.

This paper accompanies the author's keynote at the 15^{th} International Conference on Program Comprehension. It explores ways in which SBSE ideas and techniques can be applied to problems in Program Comprehension. Following the explosion of work in search based optimization for software engineering in general, this paper asks the more specific question:

> "Why not apply search based optimization to Program Comprehension?"

[1] So far as the author is aware.

The paper is organised into three principal sections. Section 2 provides a brief introduction to SBSE to make the paper self–contained. A more detailed treatment can be found in the author's FoSE paper [30]. Sections 3 and 4 present existing work on SBSE within the area of program comprehension and possible future applications of SBSE to Program Comprehension.

2. A Brief Introduction to SBSE

The very simplest search of all is a random 'search', in which solutions are simply generated purely at random. However, such a search is not really a proper *search*, because it is not guided by the fitness function and so it cannot display any intelligence. The simplest intelligent search techniques are local search techniques, such as hill climbing. Hill climbing starts with a random candidate solution and considers simple mutations that produce similar 'near neighbour' solutions, moving to those that appear to be more promising according to the fitness function.

Previous work on SBSE has gained considerable value from simple search based optimization techniques such as hill climbing [44, 51, 61]. Pure hill climbing has no mechanism to avoid the obvious problem of local maxima. That is, the search starts at a randomly chosen starting point and proceeds to ascend the nearest hill. However, while the top of this nearby hill is locally maximal, there is no guarantee that it will also be globally maximal. To some extent, this 'local maxima problem' can be ameliorated by re–starting the hill climbing process many times, thereby climbing many different hills in the hope that one will be closely approximate the height of a globally maximal hill.

Other more sophisticated meta heuristic search techniques have also been widely used, such as simulated annealing [15, 36, 61], genetic algorithms [7, 29, 32, 78] genetic programming [8, 23, 24, 77] and multi objective search [35, 38, 43, 66, 73, 82]. These techniques (and other search based optimization techniques) are likely to find application in Search Based Program Comprehension. This section provides a brief overview of these meta heuristic search techniques to make the paper self contained.

2.1. Key Ingredients

There are only two key ingredients [30] for the application of search-based optimization to software engineering problems:

1. The choice of the representation of the problem.

2. The definition of the fitness function.

This simplicity and ready applicability has led to a dramatic increase in research in this area. With these two simple ingredients, it is possible to apply search techniques to a chosen area of software engineering and to obtain interesting and potentially important results with relative ease. The two ingredients are also often found in many software engineering applications; we often have representations because this is a starting point for any work on automated software engineering support. We are also typically able to find plenty of advice on the choice of fitness function in the form of proposed software metrics [31].

All of the fitness functions so far considered in the literature on SBSE have been fully automated. However, for other branches of engineering, there has been extensive work on fitness functions that incorporate human judgement [28]. This form of search is known as 'interactive optimization'. For Program Comprehension, with its inherent human-centric focus, it would seem likely that interactive optimization would find many applications.

2.2. Simulated Annealing

Simulated annealing [59] can be thought of as a variation of hill climbing that seeks to avoid the local maxima problem by permitting moves to less fit individuals. Simulated annealing is a simulation of metallurgical annealing, in which a highly heated metal is allowed to reduce in temperature slowly, thereby increasing its strength. As the temperature decreases the atoms have reduced freedom of movement. However, the greater freedom in the earlier (hotter) stages of the process allows the atoms to 'explore' different energy states.

A simulated annealing algorithm will move from some point x_1 to a *worse* point x'_1 with a probability that is a function of the drop in fitness and a 'temperature' parameter that models metal temperature in metallurgical annealing. The effect of 'cooling' on the simulation of annealing is that the probability of following an unfavourable move is reduced. In the final stages of the simulated annealing algorithm, the effect is that of pure hill climbing.

2.3. Genetic Algorithms and Genetic Programming

A generic genetic algorithm template is presented in Figure 1. An iterative process is executed, initialised by a randomly chosen population. The iterations are called generations and the members of the population are called chromosomes, because of their analogs in natural evolution. The process terminates when a population satisfies some pre-determined condition.

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

Figure 1. A Generic Genetic Algorithm

At each generation, some members of the population are recombined, crossing over elements of their chromosomes. A fraction of the offspring of this union are mutated and, from these offspring and the original population, a selection process is used to determine the new population. Crucially, recombination and selection are guided by the fitness function; fitter chromosomes having a greater chance to be selected and recombined.

Genetic programming (GP) [45] is a variation of the evolutionary model of computation embodied in genetic algorithms. For genetic algorithms, the individuals that are optimised are lists. For GP, the individuals are trees, denoting the abstract syntax trees of simple programs.

There is a widely held misunderstanding that the goal of GP is to evolve realistic large scale programs. While this would be extremely attractive, as software engineers, we know only too well what a demanding, and perhaps utopian, goal this would be. However, it should be noted that the applications of GP gain considerable value from comparatively simple programs that, despite their simplicity, capture important and insightful computational processes.

2.4. Multi Objective Search Based Software Engineering

A natural approach to many problems starts with the consideration of the objectives to optimise and how they are to be measured. In software engineering problems it is not uncommon to find that there are several objectives to be optimised. In this situation, it is valuable to consider the concept of Pareto optimality, which can be used to explore the tradeoffs between multiple objectives. More recent work on SBSE has moved in the direction of multi objective search and it is likely that this trend will continue as the area matures. Program Comprehension research is often characterised by the need to balance competing objectives and so it is likely that Pareto optimal formulations of SBSE problems are likely to find application in Program Comprehension.

To see how Pareto optimality works, suppose a problem is to be solved that has n fitness functions, $f_1, \ldots, f_n$ that take some vector of parameters $\overline{x}$. Using Pareto optimality, the fitness functions are combined to form an aggregated fitness F as follows:

$$\begin{gathered} F(\overline{x_1}) > F(\overline{x_2}) \\ \Leftrightarrow \\ \forall i . f_i(\overline{x_1}) \geq f_i(\overline{x_2}) \quad \wedge \quad \exists i . f_i(\overline{x_1}) > f_i(\overline{x_2}) \end{gathered}$$

Using Pareto optimality, one solution is better than another if it is better according to at least one of the individual fitness functions and no worse according to all of the others. Under the Pareto interpretation of combined fitness, "no overall fitness improvement occurs no matter how much almost all of the fitness functions improve, should they do so at the slightest expense of any one of their number" [30].

When searching for solutions to a problem using Pareto optimality, the search yields a set of solutions that are non–dominated. That is, each member of the non-dominated set is no worse than any of the others in the set, but also cannot be said to be better. Any set of non–dominated solutions forms a Pareto front. Assuming convergence, the longer the search algorithm is run, the better the approximation becomes to the real Pareto front. The Pareto front can be used to improve understanding of the problem at hand. For instance, the nature of the trade offs between objectives are represented by the shape of the front. This allows the engineer to identify 'sweet spots' in which a little reduction on the value of one objective can yield a significant improvement in others.

3. Previous SBSE work Relevant to Program Comprehension

This section briefly reviews previous work on SBSE at the code level and the design level which have an application to Program Comprehension.

3.1. Optimising Source Code for Comprehension

SBSE techniques have been applied to the problem of optimising code. Sometimes, the goal is to improve execution time by re-ordering compiler optimizations [21] or finding transformations that optimise for parallel execution [65, 68, 80]. However, there has also been work on transformation to improve source code from a *program comprehension* point–of–view.

For example, Fatiregun et al. [25, 26] showed how search based transformations could be used to reduce code size and to construct amorphous program slices. Several authors have also considered the problem of refactoring object oriented code to improve metrics [38, 66, 73], using the approach that 'metrics are fitness functions too' [31]. The search considers a search space of, either all possible refactored versions of the program, or all possible sequences of refactoring steps.

This is a naturally multi objective problem; there are typically many candidate metrics that can be applied and not all of these metrics will be sympathetic to one another. Using weighted multi objective search, the several differing metrics are combined into a single agglomerated super-metric that aims to combine the relative strengths of each. However, where the metrics are in conflict, the alternative Pareto optimal approach allows the engineer to consider the trade offs, without forcing potentially arbitrary choices of weights.

Search based approaches have also been applied to the concept assignment problem [11, 52, 55]. In this problem, the goal is to identify sections of code that correspond to high–level domain concepts. Gold et al. [29] showed that search techniques are well suited to scenarios in which boundaries between concept bindings are not sharp. They used a fitness function to optimise the 'signal to noise' ratio in bindings, while allowing overlapping concept boundaries within the code.

3.2. Optimising Designs for Comprehension

Mancoridis et al. introduced the concept of software modularization as a search based clustering problem [53, 54]. The goal of this work is to re-draw module boundaries to increase cohesion and reduce coupling. In so doing, the work effectively improves the understandability of the design, based on the widely studied software engineering factors of cohesion and coupling. The seminal work of Mancoridis et al. led to several other studies of search based approaches to software modularisation [32, 37, 51], and to the application of search based clustering in other areas of software engineering [20]. A recent survey of work on search–based modularisation is presented by Mitchell and Mancoridis [62].

With the exception of the work of Lutz [50], work on this problem has used a weighted combination of cohesion and coupling as a fitness function. Lutz adopted a different approach that may well find further application in Program Comprehension. He considered the problem of hierarchical decomposition of software, using a fitness function based upon an information-theoretic formulation inspired by Shannon [74]. The goal was to reduce the amount of information used to describe the software design. The conjecture was that reducing the information content denoted by a design would make it easier to understand.

4. Possible Future Applications of SBSE in Program Comprehension

This section introduces some possible applications of SBSE in Program Comprehension. In each case the goal is to capture some aspect of interest with a fitness function so that a search based approach can be used to optimise the property of interest. In each case, the nature of the problem tends to suggest a certain form of search based approach. The aim of this section is to provide a set of challenges for the wider extension of SBSE research to Program Comprehension.

By tackling problems in Program Comprehension, SBSE researchers will be addressing some of the most complicated (but potentially rewarding) aspects of software engineering. Achieving the full potential of SBSE for comprehension raises the issues of human involvement in fitness computation (interactive optimisation), multi objective search and complex fitness functions. This set of demanding, but nonetheless achievable, goals will provide a further impetus and stimulus to research on SBSE. The techniques developed to address these problems are also likely to find application in other areas of software engineering, and possibly to other engineering domains.

4.1. Optimised Semantic Pretty Printing

Pretty printing [17] produces code that is more readable, by performing lexical transformations to improve layout. This may be one of the first published approaches to Program Comprehension [56]. A more semantic approach to pretty printing transforms the program into a style most suited to the programmer reading the code. For example, tail recursion can be transformed to iteration, `while` loops with compile time bounds to `for` loops, and nested conditionals to `case` statements.

As with lexical pretty printing, the developer of such a semantic pretty printer may believe they know best; "surely it is obvious that recursion should be removed — many students find it hard to understand". However, there is no *guarantee* that all programmers follow the same cognitive model. Indeed, there is much evidence from the Program Comprehension literature to show that programmers have *different* cognitive styles [47, 64, 76].

Using a search based approach, it would be possible to use human performance in cognitive tasks to guide a fitness function that would tailor the pretty printer to the cognitive style of the programmer. Such a flexible approach would provide a 'semantic lens' through which the programmer could view the software in order to make its algorithmic structure clearer.

The version of the program executed can be the most efficient. However, the version viewed by each programmer can be adapted to suit their individual cognitive style. All versions of the program would, of course, be semantically equivalent, but their structures may be very different. This approach may have wider benefits than merely re-writing programs to tailor them to individual programmer taste. By performing the transformations, one may be able to gain insight into different cognitive styles and, thereby, to identify programmers who can work well together and those who, perhaps, would not work well together.

4.2. Tailored Refactoring

Developing the idea of the previous section a little further, one could imagine a search based refactoring system that seeks to evolve a set of transformation rules (refactoring rules) that capture the cognitive preferences of a particular programmer. Such a set of rules could be evolved by a process of genetic programming. It is unlikely that the genetic program produced would be sufficiently powerful and (more importantly) sufficiently *trusted* that the programmer would simply apply it to code before reading it. However, it is likely that such an approach would yield insight into programmers' cognitive models. It may also capture aspects of the kind of transformation algorithms that would be required in order to improve understandability.

4.3. Balancing Multiple Architectural Requirements

Software architectures have to be constructed to meet a variety of different requirements. Many developers may work on a software system. Each may have their own view concerning choices of architectural constructions and decisions as to which will be best for comprehension and on-going development. One might suppose that it would be possible to adopt an 'architectural transformation' approach to produce a kind of 'architectural pretty printer', following the approach outlined in the previous section. However, what may work at the code level may be simply too complex to achieve at the architectural level.

In such a situation, we may be faced with a familiar engineering compromise; choose the architecture that most suits most engineers. This is a classic example of a multi objective optimization problem, in which the preferences of each engineer denote one of many individual objectives and where the objectives may be in conflict. Balancing such a set of objectives may be difficult when there are many possible candidate solutions.

Even where the search space is small, the decision maker may be guided by biases and hidden (possible erroneous) assumptions. A search based approach can explore the multi dimensional space of candidate solutions to identify 'sweet spots' where good quality solutions can be found. Such a search based solution cannot solve the unsolvable; it will not please all the people all of the time. However, it may allow the decision maker to locate interesting and important areas of the solution space in which, for example, displeasing one engineer a little may result in a solution that pleases many a lot. These sweet spots are very hard to find by hand in multi dimensional search spaces.

4.4. Evolving Visualisations

Different visualisation approaches work for different people [75]. If the components of a visualisation can be represented as atomic entries that can be combined, hierarchically, into ever larger components then it would be possible to consider the evolution of visualisations to actively search for better visualizations. A programmer could even use a visualisation that is evolving as they use it. The construction of a fitness function for such a dynamically evolving visualisation could be achieved in one of two possible ways. As the system is used, the users could click on a 'radio bar' to indicate how helpful they find the visualisation at any given point. This would provide constant feedback to the system, enabling it to guide the visualization towards solutions found more attractive to the engineer.

Alternatively, the engineer may be evaluated for the performance of cognitive tasks by the system as they use it and in this way the visualisation would be attempting to evolve to meet the *needs* of the engineer rather than their stated desires. Of course an interesting experiment, would be to try both approaches and to explore the differences. That is, to what extent do we engineers know what is good for us, cognitively speaking?

This approach is not likely to yield a practical visualisation tool for widespread application. Rather, it would provide a tool for Program Comprehension research, by allowing researchers to explore the space of visualisations and their relationship to the visualisation users' stated desires versus their cognitive needs.

4.5. Co Evolutionary Comprehension

Co evolution is an exciting form of evolutionary optimisation in which two or more populations are evolved in parallel. The fitness function for one population is influenced by the behaviour of the others. Co evolution has been used in SBSE to attempt to evolve program mutants and test cases in parallel [1]. Fitness for the mutants is determined by their ability to avoid being killed, while fitness for the test data is determined by its ability to kill the mutants. It seems likely that co–

evolution will find applications in Program Comprehension research.

Applying the ideas of co-evolution to Program Comprehension may produce several interesting models of optimization. One natural choice would be to attempt to evolve a model of program comprehension (as a set of rules) together with a description of the structure of a system. The fitness for the cognitive model could be the ability to quickly locate important items of information within the structure of the system. The fitness of the system could be determined by the extent to which it can hide information from the cognitive model. In this way, we gain insight into ways in which systems might be constructed to be harmful to comprehension and ways in which the human may adopt strategies to overcome this.

Of course, one can play this kind of game in reverse. The fitness function for the human could be strategies that expend a lot of effort in order to find useful information, while the fitness of the system could be the speed at which even such a naïve cognitive model can find information with ease. In this formulation of the 'co evolution game', the approach may yield insight into programming structures that make certain forms of information very readily available.

4.6. Information Theoretic Fitness Functions

Rudi Lutz [50] introduced the idea of using a measure of information content as a fitness function in software engineering. This idea has a clear resonance with work on Program Comprehension. One might speculate that by measuring the information content of a representation of code, one would be achieving some measure of understandability. If this be the case, then by optimising a representation for information content, one may thereby reduce the cognitive effort required for comprehension.

Many Program Comprehension applications have either an implicit or an explicit notion of information content. One of the primary goals of comprehension research is to explore the interface between the human, information from the real world and its software abstractions with which the human has to reason. The work of Lutz provides a hint of what may be possible when this information content can be measured and used as a basis for fitness evaluation and optimization.

4.7. Linguistic Evolution

Many hours are spent, often late into the night, arguing over which syntactic formulation is the best in order to represent a certain programming idea. The subject can evoke considerable passion among programmers. Using evolution, it would be possible to evolve the grammar of a language to express certain programming constructs in a manner considered suitable for comprehension. For example, guidelines for good program comprehension could be coded as constraints in the search. Another possibility, would be the formulation of a fitness function that takes account for programmer performance at comprehension based tasks. A further possibility is to explore Shannon style information–theoretic formulations of fitness that capture the information content of the syntactic productions, as discussed in the previous section.

4.8. Revealing Hidden Assumptions

Humans have hidden assumptions about what makes some representation of code more easy to understand than another. These assumptions may be difficult to identify. They may even be unknown to the human who possess them. Furthermore, the assumptions upon which a human rests their belief about which forms of code presentation suit them may be misplaced.

It is not uncommon for humans to state their assumptions only to break them. For instance, when looking for a suitable house, a client might tell an estate agent that the house *must* have 3 bedrooms and that this is an absolute requirement that cannot be broken. Upon speculatively showing the client a house with only two rooms the estate agent is surprised that the client is very interested. "What about the three bedroom requirement?" asks the agent. The client responds that the existing study can be converted to a bedroom and that this is more than adequate for their requirements. The 'three bedroom' requirement was a misplaced assumption. The 'prepared to do without a study' observation was an implicit assumption.

With software, an engineer may, initially, believe that high coupling is 'always bad'. Using a re-modularization tool, the same engineer may be surprised to find that library modules become artificially refactored, by migration into clustered modules together with the code that most uses them. Mancoridis et al. [62] call such highly coupled modules 'omnipresent'. They are removed from consideration by the Bunch tool to prevent this form of artificial refactoring.

The software engineer starts with the assumption that high coupling is always bad, but subsequent search based optimization reveals a further degree of subtlety; 'high coupling is OK for a library module'. In this way search can be used to reveal hidden assumptions. The engineer formulates a fitness function. The search algorithm is guided solely by the fitness function and therefore considers solutions that a human would consider to be 'ridiculous'. By locating such solutions the search process reveal the hidden assumptions.

In the past, search based approaches to engineering design have proved remarkably good at revealing assumptions and confounding intuition, occasionally yielding results that are superior to those found by the human. For example, evolutionary algorithms have led to patented designs for digital filters [70] and the discovery of patented antenna designs [49]. There is every reason to hope that these search based approaches may be equally good at revealing insights into hidden assumptions that explain aspects of Program Comprehension.

4.9. GP Models of Human Comprehension

Genetic programming has proved to be good at capturing models of behaviour that can be expressed computationally. It is tempting to model human comprehension behaviour as a computational process, in which the human considers code in a certain order, or responds to stimuli from the program development environment in a certain well defined manner. For such a model of comprehension behaviour it is possible to use genetic programming to attempt to capture aspects of behaviour. Different engineers comprehension styles will be reflected by differently evolved genetic programs, each of which may yield insights into the engineer's comprehension model.

5. Summary

This paper has presented the Search Based Software Engineering approach and shown how it has been applied to problems closely related to Program Comprehension at the source code and design levels of abstraction. The paper has also explored some of the possible ways in which SBSE techniques may be applied to Program Comprehension in the future. It is hoped that the set of challenges in Section 4 will serve to stimulate further research in this area.

6. Acknowledgements

This paper has drawn on the author's work within the Search Based Software Engineering (SBSE) community. The discussions within this community have helped to form the ideas presented in this paper. Harman's work is currently funded by the EPSRC project, SEBASE (2006-2011), for which the other principal investigators are John Clark (University of York) and Xin Yao (University of Birmingham) and industrialists from DaimlerChrysler Berlin, Motorola and IBM. He is also supported by the EU Specific Targeted Research Project: EvoTest (2006-2009). His work on Program Comprehension is supported by EPSRC project, ConTRACTS (2005-2008). This paper draws on these projects and from other keynotes and tutorials on SBSE prepared in collaboration with Joachim Wegener [30, 39, 40, 41].

References

[1] K. Adamopoulos, M. Harman, and R. M. Hierons. Mutation testing using genetic algorithms: A co-evolution approach. In *Genetic and Evolutionary Computation Conference (GECCO 2004), LNCS 3103*, pages 1338–1349, Seattle, Washington, USA, June 2004. Springer.

[2] J. Aguilar-Ruiz, I. Ramos, J. C. Riquelme, and M. Toro. An evolutionary approach to estimating software development projects. *Information and Software Technology*, 43(14):875–882, Dec. 2001.

[3] G. Antoniol, M. D. Penta, and M. Harman. Search-based techniques applied to optimization of project planning for a massive maintenance project. In 21^{st} *IEEE International Conference on Software Maintenance*, pages 240–249, Los Alamitos, California, USA, 2005. IEEE Computer Society Press.

[4] V. Babovic. Mining sediment transport data with genetic programming. In *Proceedings of the First International Conference on New Information Technologies for Decision Making in Civil Engineering*, pages 875–886, Montreal, Canada, 11-13 Oct. 1998.

[5] A. Bagnall, V. Rayward-Smith, and I. Whittley. The next release problem. *Information and Software Technology*, 43(14):883–890, Dec. 2001.

[6] A. Baresel, D. W. Binkley, M. Harman, and B. Korel. Evolutionary testing in the presence of loop–assigned flags: A testability transformation approach. In *International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 108–118, Omni Parker House Hotel, Boston, Massachusetts, July 2004. Appears in Software Engineering Notes, Volume 29, Number 4.

[7] A. Baresel, H. Sthamer, and M. Schmidt. Fitness function design to improve evolutionary structural testing. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1329–1336, San Francisco, CA 94104, USA, 9-13 July 2002. Morgan Kaufmann Publishers.

[8] T. V. Belle and D. H. Ackley. Code factoring and the evolution of evolvability. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1383–1390, San Francisco, CA 94104, USA, 9-13 July 2002. Morgan Kaufmann Publishers.

[9] F. H. Bennett III, M. A. Keane, D. Andre, and J. R. Koza. Automatic synthesis of the topology and sizing for analog electrical circuits using genetic programming. In K. Miettinen, M. M. Mäkelä, P. Neittaanmäki, and J. Periaux, editors, *Evolutionary Algorithms in Engineering and Computer Science*, pages 199–229, Jyväskylä, Finland, 30 May - 3 June 1999. John Wiley & Sons.

[10] P. J. Bentley and J. P. Wakefield. Generic representation of solid geometry for genetic search. *Microcomputers in Civil Engineering*, 11(3):153–161, 1996.

[11] T. J. Biggerstaff, B. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *15th International Conference on Software Engineering*, Los Alamitos, California, USA, May 1993. IEEE Computer Society Press.

[12] R. R. Birge. Protein-based optical computing and memories. *Computer*, 25(11):56–67, Nov. 1992.

[13] L. Bottaci. Instrumenting programs with flag variables for test data search by genetic algorithms. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1337–1342, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[14] S. Bouktif, G. Antoniol, E. Merlo, and M. Neteler. A novel approach to optimize clone refactoring activity. In *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 2, pages 1885–1892, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[15] S. Bouktif, H. Sahraoui, and G. Antoniol. Simulated annealing for improving software quality prediction. In *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 2, pages 1893–1900, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[16] L. C. Briand, Y. Labiche, and M. Shousha. Stress testing real-time systems with genetic algorithms. In *Genetic and Evolutionary Computation Conference, GECCO 2005, Proceedings, Washington DC, USA, June 25-29, 2005*, pages 1021–1028. ACM, 2005.

[17] Cameron and R. D. An abstract pretty printer. *IEEE Software*, 5(6):61–67, 1988.

[18] G. Canfora, M. D. Penta, R. Esposito, and M. L. Villani. An approach for qoS-aware service composition based on genetic algorithms. In H.-G. Beyer and U.-M. O'Reilly, editors, *Genetic and Evolutionary Computation Conference, GECCO 2005, Proceedings, Washington DC, USA, June 25-29, 2005*, pages 1069–1075. ACM, 2005.

[19] J. Clark, J. J. Dolado, M. Harman, R. M. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *IEE Proceedings — Software*, 150(3):161–175, 2003.

[20] M. Cohen, S. B. Kooi, and W. Srisa-an. Clustering the heap in multi-threaded applications for improved garbage collection. In *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 2, pages 1901–1908, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[21] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM Sigplan 1999 Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99)*, volume 34.7 of *ACM Sigplan Notices*, pages 1–9, NY, May 5 1999. ACM Press.

[22] O. Cordón, F. Herrera, and L. Sánchez. Evolutionary learning processes for data analysis in electrical engineering applications. In D. Quagliarella, J. Périaux, C. Poloni, and G. Winter, editors, *Genetic Algorithms and Evolution Strategy in Engineering and Computer Science*, pages 205–224. John Wiley and Sons, Chichester, 1998.

[23] J. J. Dolado. A validation of the component-based method for software size estimation. *IEEE Transactions on Software Engineering*, 26(10):1006–1021, 2000.

[24] J. J. Dolado. On the problem of the software cost function. *Information and Software Technology*, 43(1):61–72, Jan. 2001.

[25] D. Fatiregun, M. Harman, and R. Hierons. Evolving transformation sequences using genetic algorithms. In 4^{th} *International Workshop on Source Code Analysis and Manipulation (SCAM 04)*, pages 65–74, Los Alamitos, California, USA, Sept. 2004. IEEE Computer Society Press.

[26] D. Fatiregun, M. Harman, and R. Hierons. Search-based amorphous slicing. In 12^{th} *International Working Conference on Reverse Engineering (WCRE 05)*, pages 3–12, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, Nov. 2005.

[27] C.-W. Feng, L. Liu, and S. A. Burns. Using Genetic Algorithms to Solve Construction Time-Cost Trade-Off Problems. *Journal of Computing in Civil Engineering*, 10(3):184–189, 1999.

[28] P. Funes, E. Bonabeau, J. Herve, and Y. Morieux. Interactive multi-participant task allocation. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, pages 1699–1705, Portland, Oregon, 20-23 June 2004. IEEE Press.

[29] N. Gold, M. Harman, Z. Li, and K. Mahdavi. A search based approach to overlapping concept boundaries. In 22^{nd} *International Conference on Software Maintenance (ICSM 06)*, Philadelphia, Pennsylvania, USA, Sept. 2006. To appear.

[30] M. Harman. The current state and future of search based software engineering. In L. Briand and A. Wolf, editors, *Future of Software Engineering 2007*, Los Alamitos, California, USA, 2007. IEEE Computer Society Press. To appear.

[31] M. Harman and J. Clark. Metrics are fitness functions too. In 10^{th} *International Software Metrics Symposium (METRICS 2004)*, pages 58–69, Los Alamitos, California, USA, Sept. 2004. IEEE Computer Society Press.

[32] M. Harman, R. Hierons, and M. Proctor. A new representation and crossover operator for search-based optimization of software modularization. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1351–1358, San Francisco, CA 94104, USA, 9-13 July 2002. Morgan Kaufmann Publishers.

[33] M. Harman, L. Hu, R. M. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, Jan. 2004.

[34] M. Harman and B. F. Jones. Search based software engineering. *Information and Software Technology*, 43(14):833–839, Dec. 2001.

[35] M. Harman, K. Lakhotia, and P. McMinn. A multi-objective approach to search-based test data generation. In *GECCO 2007: Proceedings of the 9^{th} annual conference on Genetic and evolutionary computation*, London, UK, July 2007. ACM Press. To appear.

[36] M. Harman, K. Steinhöfel, and A. Skaliotis. Search based approaches to component selection and prioritization for the next release problem. In 22^{nd} *International Conference on Software Maintenance (ICSM 06)*, Philadelphia, Pennsylvania, USA, Sept. 2006. To appear.

[37] M. Harman, S. Swift, and K. Mahdavi. An empirical study of the robustness of two module clustering fitness functions. In *Genetic and Evolutionary Computation Conference (GECCO 2005)*, pages 1029–1036, Washington DC, USA, June 2005. Association for Computer Machinery.

[38] M. Harman and L. Tratt. Pareto optimal search-based refactoring at the design level. In *GECCO 2007: Proceedings of the 9^{th} annual conference on Genetic and evolutionary computation*, London, UK, July 2007. ACM Press. To appear.

[39] M. Harman and J. Wegener. Evolutionary testing: Tutorial. In *Genetic and Evolutionary Computation (GECCO)*, Chicago, July 2003.

[40] M. Harman and J. Wegener. Getting results with search–based software engineering: Tutorial. In 26^{th} *IEEE International Conference and Software Engineering (ICSE 2004)*, pages 728–729, Los Alamitos, California, USA, 2004. IEEE Computer Society Press.

[41] M. Harman and J. Wegener. Search based testing. In 6^{th} *Metaheuristics International Conference (MIC 2005)*, Vienna, Austria, Aug. 2005. To appear.

[42] W. M. Jenkins. The genetic algorithm-or can we improve design by breeding. In *IEE Colloquium on Artificial Intelligence in Civil Engineering*, pages 1/1–4, London, UK, Jan., 16 1992. IEE.

[43] T. M. Khoshgoftaar, L. Yi, and N. Seliya. A multiobjective module-order model for software quality enhancement. *IEEE Transactions on Evolutionary Computation*, 8(6):593– 608, December 2004.

[44] C. Kirsopp, M. Shepperd, and J. Hart. Search heuristics, case-based reasoning and software project effort prediction. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1367–1374, San Francisco, CA 94104, USA, 9-13 July 2002. Morgan Kaufmann Publishers.

[45] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.

[46] J. E. Labossiere and N. Turrkan. On the optimization of the tensor polynomial failure theory with a genetic algorithm. *Transactions of the Canadian Society for Mechanical Engineering*, 16(3-4):251–265, 1992.

[47] S. Letovsky. Cognitive processes in program comprehension. *The Journal of Systems and Software*, 7(4):325–339, Dec. 1987.

[48] Z. Li, M. Harman, and R. Hierons. Meta-heuristic search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*. To appear.

[49] D. S. Linden. Innovative antenna design using genetic algorithms. In D. W. Corne and P. J. Bentley, editors, *Creative Evolutionary Systems*, chapter 20. Elsevier, Amsterdam, The Netherland, 2002.

[50] R. Lutz. Evolving good hierarchical decompositions of complex systems. *Journal of Systems Architecture*, 47:613–634, 2001.

[51] K. Mahdavi, M. Harman, and R. M. Hierons. A multiple hill climbing approach to software module clustering. In *IEEE International Conference on Software Maintenance*, pages 315–324, Los Alamitos, California, USA, Sept. 2003. IEEE Computer Society Press.

[52] J. I. Maletic and A. Marcus. Supporting program comprehension using semantic and structural information. In 23^{rd} *International Confernece on Software Engineering (ICSE 2001)*, pages 103–112, Los Alamitos, California, USA, May 2001. IEEE Computer Society Press.

[53] S. Mancoridis, B. S. Mitchell, Y.-F. Chen, and E. R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings; IEEE International Conference on Software Maintenance*, pages 50–59. IEEE Computer Society Press, 1999.

[54] S. Mancoridis, B. S. Mitchell, C. Rorres, Y.-F. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *International Workshop on Program Comprehension (IWPC'98)*, pages 45–53, Los Alamitos, California, USA, 1998. IEEE Computer Society Press.

[55] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev. Static techniques for concept location in object-oriented code. In 13^{th} *IEEE International Workshop on Program Comprehension (IWPC'05)*, pages 33–42. IEEE Computer Society Press, 2005.

[56] W. M. McKeeman. Algorithm 268: ALGOL 60 reference language editor. *Communications of the ACM*, 8(11):667–668, Nov. 1965.

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

[58] P. McMinn, M. Harman, D. Binkley, and P. Tonella. The species per path approach to search-based test data generation. In *International Symposium on Software Testing and Analysis (ISSTA 06)*, pages 13–24, Portland, Maine, USA., 2006.

[59] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.

[60] W. Miller and D. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, 2(3):223–226, 1976.

[61] B. S. Mitchell and S. Mancoridis. Using heuristic search techniques to extract design abstractions from source code. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1375–1382, San Francisco, CA 94104, USA, 9-13 July 2002. Morgan Kaufmann Publishers.

[62] B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.

[63] Y. Miyamoto, Y. Miyatake, S. Kurosaka, and Y. Mori. A parameter turning for dynamic simulation of power plants using genetic algorithms. *Electrical Engineering Japan*, 115(1):104–113, 1995.

[64] Neal and L. Rubin. Cognition-sensitive design and user modeling for syntax-directed editors. In *Proceedings of ACM CHI+GI'87 Conference on Human Factors in Computing Systems and Graphics Interface*, Adaptive Interfaces, pages 99–102, 1987.

[65] A. Nisbet. GAPS: A compiler framework for genetic algorithm (GA) optimised parallelisation. In P. M. A. Sloot, M. Bubak, and L. O. Hertzberger, editors, *High-Performance Computing and Networking, International Conference and Exhibition, HPCN Europe 1998, Amsterdam, The Netherlands, April 21-23, 1998, Proceedings*, volume LNCS 1401, pages 987–989. Springer, 1998.

[66] M. O'Keeffe and M. O'Cinneide. Search-based software maintenance. In *Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 249–260, Mar. 2006.

[67] R. Poli, S. Cagnoni, and G. Valli. Genetic design of optimum linear and nonlinear QRS detectors. *IEEE Transactions on Biomedical Engineering*, 42(11):1137–41, Nov. 1995.

[68] C. Ryan. *Automatic re-engineering of software using genetic programming*. Kluwer Academic Publishers, 2000.

[69] E. Sanchez, H. Miyano, and J. P. Brachet. Optimization of fuzzy queries with genetic algorithms. applications to a data base of patents in biomedical engineering. *Proc. Sixth International Fuzzy Systems Association World Congress (IFSA'95)*, 2:293–296, 1995. Sao Paulo.

[70] T. Schnier, X. Yao, and P. Liu. Digital filter design using multiple pareto fronts. *Soft Computing*, 8(5):332–343, April 2004.

[71] O. Seng, M. Bauer, M. Biehl, and G. Pache. Search-based improvement of subsystem decompositions. In H.-G. Beyer and U.-M. O'Reilly, editors, *Genetic and Evolutionary Computation Conference, GECCO 2005, Proceedings, Washington DC, USA, June 25-29, 2005*, pages 1045–1051. ACM, 2005.

[72] O. Seng, J. Stammel, and D. Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 2, pages 1909–1916, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[73] O. Seng, J. Stammel, and D. Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In M. Keijzer, M. Cattolico, D. Arnold, V. Babovic, C. Blum, P. Bosman, M. V. Butz, C. Coello Coello, D. Dasgupta, S. G. Ficici, J. Foster, A. Hernandez-Aguirre, G. Hornby, H. Lipson, P. McMinn, J. Moore, G. Raidl, F. Rothlauf, C. Ryan, and D. Thierens, editors, *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 2, pages 1909–1916, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[74] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423 and 623–656, July and October 1948.

[75] M. Tory and T. Möller. Human factors in visualization research. *IEEE Transactions on Visualization and Computer Graphics*, 10(1):72–84, 2004.

[76] A. Walenstein. Theory-based analysis of cognitive support in software comprehension tools. In 10^{th} *IEEE International Workshop on Program Comprenhesion (IWPC'02)*, pages 75–84. IEEE Computer Society Press, 2002.

[77] S. Wappler and J. Wegener. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 2, pages 1925–1932, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[78] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology Special Issue on Software Engineering using Metaheuristic Innovative Algorithms*, 43(14):841–854, 2001.

[79] R. Weiss and J. T. F. Knight. Engineered communications for microbial robotics. In *Proceedings 6th DIMACS Workshop on DNA Based Computers, held at the University of Leiden, Leiden, The Netherlands, 13 - 17 June 2000*, DIMACS: Series in Discrete Mathematics and Theoretical Computer Science., pages 5–19. Leiden center for natural computing, 2000.

[80] K. P. Williams. *Evolutionary Algorithms for Automatic Parallelization*. PhD thesis, University of Reading, UK, Department of Computer Science, Sept. 1998.

[81] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios. Application of genetic algorithms to software testing (Application des algorithmes génétiques au test des logiciels). In 5^{th} *International Conference on Software Engineering and its Applications*, pages 625–636, Toulouse, France, 1992.

[82] Y. Zhang, M. Harman, and A. Mansouri. The multi-objective next release problem. In *GECCO 2007: Proceedings of the* 9^{th} *annual conference on Genetic and evolutionary computation*, London, UK, July 2007. ACM Press. To appear.