Dynamic Adaptive Search Based Software Engineering*

Mark Harman[1], Edmund Burke[2], John A. Clark[3] and Xin Yao[4]
[1]CREST Centre, University College London, Gower Street, London, WC1E 6BT, UK.
[2]University of Stirling, Stirling, FK9 4LA Scotland, UK.
[3]Department of Computer Science, University of York, Deramore Lane, York, YO10 5GH, UK.
[4]School of Computer Science, The University of Birmingham, Edgbaston, Birmingham B15 2TT, UK.

ABSTRACT

Search Based Software Engineering (SBSE) has proved to be a very effective way of optimising software engineering problems. Nevertheless, its full potential as a means of dynamic adaptivity remains under explored. This paper sets out the agenda for Dynamic Adaptive SBSE, in which the optimisation is embedded into deployed software to create self-optimising adaptive systems. Dynamic Adaptive SBSE will move the research agenda forward to encompass both software development processes and the software products they produce, addressing the long-standing, and as yet largely unsolved, grand challenge of self-adaptive systems.

Categories and Subject Descriptors

D.2 [**Software Engineering**]

General Terms

Search Based Software Engineering (SBSE), Evolution, Automatic Programming, Measurement, Testing

Keywords

SBSE, Search Based Optimization, Self-Adaptive Systems, Autonomic Computing

1. INTRODUCTION

Current software development practices achieve adaptivity at only a glacial pace, largely through enormous human engineering skill and effort. We force highly experienced engineers to waste their time and expertise adapting many tedious implementation details. Often, the resulting software is equally inflexible: users often find themselves relying on their innate human adaptivity to compensate with 'workarounds'. This has to change.

To address the twin goals of adaptivity and automation, we advocate a development of the Search Based Software Engineering (SBSE) agenda that we call 'Dynamic Adaptive Search Based Software Engineering'. We seek greater software engineering automation through the development of hyper heuristics for SBSE. At the same time we seek greater adaptivity through the use of dynamic optimisation; optimisation embedded into the deployed software to re-tune its performance parameters and even to replace large portions of code with automatically re-evolved code.

2. SBSE

Search Based Software Engineering (SBSE) is the name given to a field of research and practice in which computational search (as well as optimisation techniques more usually associated with Operations Research) are used to address problems in Software Engineering [39]. The SBSE approach seeks to optimise software engineering processes and products using generic, robust, flexible, scalable and insight-rich computational search. SBSE provides a mechanism for managed automation of software engineering activities.

SBSE has proved to be a widely applicable and successful approach, with many applications right across the full spectrum of activities in software engineering, from initial requirements, project planning, and cost estimation to regression testing and onward evolution. Few aspects of development and deployment of software systems have remained untouched by the SBSE research agenda.

There is also an increasing interest in search based optimization from the industrial sector, as illustrated by work on testing involving Berner and Mattner and Daimler [49, 64], Ericsson [3], Google [69] and Microsoft [14, 50], and work on requirements analysis and optimisation involving Ericsson [70], Motorola [9] and NASA [20].

The increasing maturity of the field has led to a number of tools for SBSE applications, including AUSTIN (for C language test data generation, [49]), Bunch (for modularisation, [55]), Code-Imp (for automated refactoring, [56]), eTOC (for Java class testing, [63]), EvoSUITE (for Java test data generation, [26]), GenProg (for automated bug patching, [52]), MiLu (for higher order mutation testing, [46]), ReleasePlanner (for Requirements Optimisation, [58]), and SWAT (for PHP server-side test data generation [5]).

*This position paper is written to accompany Mark Harman's keynote talk at the 6^{th} International Symposium on Empirical Software Engineering and Measurement (ESEM 12) in Lund, Sweden. It is joint work with Edmund Burke, John Clark and Xin Yao, funded by the EPSRC programme grant DAASE (EP/J017515/).

ESEM'12, September 19–20, 2012, Lund, Sweden.
ACM 978-1-4503-1056-7/12/09.

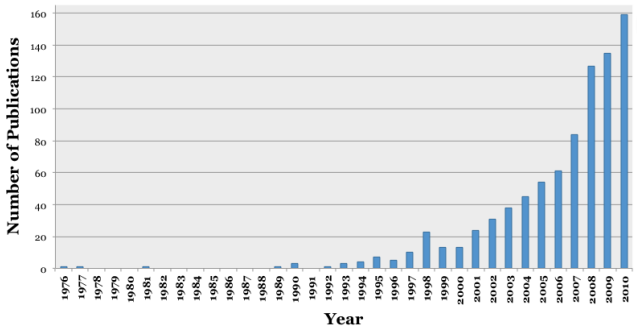

Figure 1: Yearly SBSE publication rates 1976-2010. Source: SBSE Repository [72].

Interest and research activity concerning SBSE has grown rapidly in the past ten years (see Figure 1) and there are now many excellent surveys and reviews on SBSE from which more information can be obtained. Rather than repeating this information, here are some pointers to specific SBSE surveys and reviews on:

- Requirements [71];
- Predictive Modelling [1, 34];
- Non-Functional Properties [2];
- Program Comprehension [32];
- Design [61] and
- Testing [2, 4, 33, 54].

In addition to this topic-specific SBSE literature, there are several more general SBSE surveys [18, 27, 31, 37, 40] and a review covering the relationship between other Artificial Intelligence techniques and SBSE [38]. There is also an SBSE tutorial aimed at those with no prior knowledge of SBSE who seek to adopt and apply search based optimization to software engineering problems of their own [42].

3. HYPER HEURISTIC SBSE

Current work on SBSE has produced significant advances in automated software engineering, particularly in the realms of testing, bug fixing and decision support. SBSE also shows great promise as a technique for handling non-functional properties and noisy, incomplete, and conflicting information concerning fitness.

Current SBSE automates specific problems in isolation, rather than the entire software engineering process. A dramatic increase in the breadth of automation lies within the grasp of the SBSE research and practitioner community. Such a 'holistic' optimisation-centric approach would ensure that SBSE achieves its full potential as a means to embed automated processes throughout the full range of software development and deployment activities.

To illustrate this vision for a more holistic SBSE, suppose we automate large parts of the development process using computational search: requirements engineering, project planning and testing could then become unified into a single automated activity.

To achieve a generic and holistic optimisation that connects diverse engineering activities, we turn to hyper-heuristic search [13] as a methodology for selecting or generating heuristics. That is, while most heuristic methods in the literature operate on a search space of potential solutions to a particular problem, a hyper-heuristic operates on a search space of heuristics. A Hyper Heuristic Search Based Software Engineering would address two important open questions in SBSE:

1. **To reach deeper, we need a holistic SBSE:** Why do we currently need to design special search based algorithms for each problem instance? This is unrealistic: every software engineer cannot be expected to be a computation search algorithm designer too.
2. **To reach wider, we need a generic SBSE:** Why do we currently optimise silos of software engineering activity? This is unrealistic: engineering decision making needs to take account of requirements, designs, test cases and implementation details *simultaneously*.

The Hyper Heuristic SBSE research agenda will raise fundamental questions. For example: how best do we draw the dividing line between adaptive automation for small changes and human intervention to invoke more fundamental adaption and to provide oversight and decision making? While automation is important, it is essential to understand the points at which human oversight, intervention, resumption-of-control and decision making should impinge on automation [35].

In the context of SBSE, this dividing line is the fine line between automated decision taking and automated decision support. Previous work on SBSE has tended to focus on automated decision making for those aspects of the development thought to occur later in the cycle, such as testing. The community has tended to reserve decision support for the early development cycle activities such as requirements analysis, and estimation. However, in a more holistic SBSE, there will be a far more intimate relationship between decision making and decision support, posing new methodological, engineering and pragmatic constraints and concerns.

Our vision of this new Hyper Heuristic Search Based Software Engineering is that it will provide the intellectual and technical tools to address the challenge of deeper, more holistic SBSE that cuts across the traditional software engineering boundaries such as requirements, design, modelling and testing. This vision is unashamedly experimental [67] and empirical [12]. It also aligns well with more agile and adaptive development practices, in which different software engineering activities such as design, re-factoring, testing and requirement elicitation are seen as iterative, integrated and inter-related activities, rather than as separate phases of development.

We also believe that the same Hyper Heuristic Search Based Software Engineering agenda will allow SBSE to reach a wider practitioner audience, by moving us from the bespoke to the generic. Instead of designing bespoke optimisation algorithms for specific instances, we advocate the design of 'reasonably good' hyper heuristic optimisers that have the generality to be applied more readily 'out of the box'. The results obtainable from a carefully crafted, specific, bespoke algorithm will surely out-perform those of a generic hyper heuristic SBSE algorithm.

We do not dispute this. Rather, we seek to surrender a little result quality for a lot of generality, believing that this balance of the meta objectives of quality and applicability will better address the factors that will influence uptake of SBSE. Our motivation is that ease of applicability will often trump quality of results, at least for the initial adopters, without whom there will be no take up. Has it not ever been thus in all technological development?

The hyper heuristic approach will require little tuning and will reduce the need for specific details, thereby significantly reducing the time to deployment and use. The key question will be whether sufficient optimisation power can be maintained so that the increased usability of the approach outweighs the reduction in result quality. This, in itself, is of course a twin-objective, cost-benefit optimisation trade off.

4. DYNAMIC ADAPTIVE SBSE

Self-adaptivity has been a goal of software and systems engineering research for some time, with work on architectures to support adaptive middleware [11, 59], Artificial Immune Systems (AIS) [45] for intrusion detection [47] and fault tolerance [68] and the vision of autonomic systems [29].

This research agenda is far from fully achieved; many authors still seek to address the outstanding grand challenge of self-adaptive systems, with large integrated projects such as the Self Managing Situated Computing project [25] and conferences and workshops, such as the Dagstuhl Seminar on Software Engineering for Self-Adaptive Systems [16].

SBSE has a potential to make a significant contribution to the realisation of this grand challenge. Unlike all other engineering optimization problems it is for *software* that optimisation has the most potential, because of the virtual nature of this extraordinary engineering material [36]. While computational search has been successfully applied to the design of engineering artefacts in civil, mechanical and electronic engineering, the search process cannot directly optimise these materials; the search ranges over a design space, guided by a simulation of a model of reality.

The search space and guidance are very different when we apply computation search to software. We find a new and potent possibility for search based optimisation: we can *directly* optimise the engineering material: the programs themselves. This opens up the possibility for in-situ, on the fly, optimisation to re-balance, re-configure, and even to *redevelop* the deployed software as it operates. This is the goal of Dynamic Adaptive Search Based Software Engineering.

The SBSE community has already developed techniques for tuning the performance of systems by identifying performance affecting parameters and treating them as configuration search spaces [19, 48]. There have also been exciting recent breakthroughs in the use of genetic programming to re-design aspects of systems to fix bugs [8, 65], to migrate to new platforms and languages [51] and to optimise non-functional properties [66].

These results can be thought of as early indications of the potential for Dynamic Adaptive SBSE. The work on parameter tuning shows that we can identify and tune performance parameters. If we can do this off-line, why not perform the tuning *on-line*. That is, compile into the deployed software an optimization algorithm that can identify and tune parameters that affect non-functional requirements. In this way, we would have self-monitoring, self-optimising systems.

By focusing on non-functional requirements we may partly escape the intricacies of requirements capture, with their inherent uncertainties. Functional requirements analysis is known to be plagued by difficulties of knowing exactly what the customer wants [15], something the customer may not even know themselves. Fortunately, the fitness function is often clear and unequivocal when it comes to non-functional requirements. The customer merely needs to state the non-functional requirements that matter (perhaps with acceptable tolerances, thresholds or ranges) and we can seek to optimise for these requirements.

By focusing on non-functional properties, we shall not be in a niche ghetto of 'optimisable software space'. The technological and business winds of change are clearly prevailing in a very non-functional direction. The advent of smart devices as an important computational platform, raises issues of power consumption, memory use and code size. The use of internet-enabled computing, in context aware mobile systems demands attention to bandwidth and response time. Cloud migration brings with it demands on throughput, heat dissipation and other service level properties.

Notice how this migration, from what might be termed the 'discrete' world of functional properties to the more 'continuous' world of non-functional properties, clearly illuminates the age-old debate about the difference between those two recalcitrant siamese twins: computer science and software engineering.

A focus on non-functional requirements such as power consumption, heat dissipation, throughput, response time, memory profile, bandwidth and information leakage will render a kind of Software Engineering more akin to traditional engineering disciplines. The inherent engineering character of software development and deployment will become more compelling than would have ever been thought by those who even ventured to doubt that there was such a thing as software *engineering* or that it could ever share the qualities of traditional engineering disciplines [22, 44, 43]. The rising importance of non-functional properties will also mean that software engineering will become ever more amenable to SBSE-style approaches.

Identification and optimisation of performance sensitive parameters will be one way to achieve Dynamic Adaptive SBSE. However, this will still leave the code largely unchanged. We will be extracting parameters, exposing them at the 'top level' and then searching for sensible settings, as the software executes. While this is likely to have many practical benefits, it merely scratches at the surface of the software.

Perhaps an even more exciting (yet demanding) challenge would be to seek a re-development of part of the software as deployed, while it executes, in situ, to replace the code with a better alternative. This dynamic adaptive SBSE would allow software to be more fundamentally adaptive. With this form of search-based adaptivity, we could hope that software systems would re-develop themselves, over time, to handle changing environments, platforms and contexts, while still seeking to meet the same overall functionality.

To some, such a vision of truly self-modifying code, might seem more of a nightmare than a dream: surely such code would be impossible to understand and to control? How would we ever apply source code analysis to such dynamically adapting code? Would the code not become unreadable?

However, with hindsight it may seem like merely another step on the pathway from assembler code to higher levels of abstraction. We already cede to the compiler a great deal of sovereignty over the code that it produces, seldom interfering with (or even enquiring about) the optimisation choices the compiler makes in producing object code.

With the advent of Dynamic Adaptive SBSE, we will make a further step towards the goal of greater abstraction. What we think of as source code today, may become the object code of tomorrow. In a world where non-functional, performance related requirements are 'optimised in' as the code executes, the programmer can move to a higher plain of abstraction. She will surely want to focus purely on the functional requirements of the system and will be happy to leave the optimisation of non-functional aspects to the SBSE compiler (and the 'on board re-evolver').

We hope to reach the point at which we are able to use Dynamic Adaptive SBSE to simultaneously meet the goals cherished by early pioneers of declarative languages [21] as well as the initial advocates of self-adaptive and autonomic computing [29, 59]: The program would be written in a purely declarative style. Why would anyone wish to code for performance-related details when these can be 'optimised in'? Initial results from new forms of SBSE-inspired genetic programming have indicated that this goal may be within our reach:

Bug fixing: With automated bug fixing, it is already possible to find and fix non trivial bugs [30]. The changes made by an automated patching system, are relatively small changes compared to the overall size of the program. One might think that they would simply be just that: a patch, deployed as a temporary measure to buy time for the more trusted code changers (the humans) to take over. However, there is recent evidence that there may be a longer-term future for such machine-generated patches [28].

Migration: Recent work on code migration using evolutionary improvement [51] showed that it was possible to automatically port the core computation of the unix utility `gzip` from a desk top platform supporting C code to a GPGPU platform supporting CUDA code. The automated re-evolution of the core computation of this utility demonstrates that it is not just patches that can be evolved, but larger pieces of code. It also indicates that it is possible to evolve new code for completely different architectures and languages than those for which the original code was designed. A key insight in this work is that the original program can act as an oracle for the functional requirements of the system to be re-evolved in this way [7].

Trading Functional and non-functional requirements: Previous work on searching for alternative balances between functional and non-functional requirements has also been promising. White et al. [66] showed how different versions of a pseudo random number generator could be evolved with a range of power-consumption characteristics. Crucially, in this work, functionality was sacrificed for non-functional properties. This may seem a curious approach to adopt after several decades of emphasis on correctness of functional requirements. However, for battery-powered platforms, power consumption is king:

$$correctness + flat\_battery = useless$$

A user might sacrifice the 'sacred cow' of complete functional correctness (did we ever attain that anyway?) should it come into stark conflict with longer battery life.

Indeed, do we not already do so when we switch off features of our smart phones to enable longer lifetime to the next re-charge?

The road to achieve our vision is not without challenges. There are fundamental obstacles to be overcome in computational search itself. It is still unclear whether it is even theoretically possible to evolve and adapt software from just a declarative description of functional requirements. There is a need to understand what is and is not possible using the SBSE approach and how efficient and effective such an approach is in evolving software dynamically.

5. EXPERIMENTAL VS. EMPIRICAL

The essence of science and engineering and their considerable achievements rest upon the careful construction of experiments, from which (often painstaking) observations are made. Experimentation is the foundation stone on which rests much of science, widely believed to be the principle driver behind the growth of scientific knowledge. Experimentation is the scientific credo enshrined in the Popperian view of science [60].

There has been much debate about the role of experimentation in computer science and software engineering too, with many arguing the case for experimental approaches [10, 57, 62]. However, there is a subtle distinction between purely experimental and empirical research in software engineering. This distiction is less important (and thus under-emphasied) in other science and engineering disciplines.

A scientific experiment is normally taken to mean the careful observation of one or more dependent attributes, under carefully controlled circumstances. The control of circumstances is crucial; one often uses the phrase 'under laboratory conditions' for such experiments.

By contrast, the term 'empirical' is typically used to define *any* statement about the world that is related to observation or experience. It is helpful to distinguish pure experimentation from the more general class of empirical investigation. Of course, a scientific experiment is an act of observation and experience; the experience of the scientist making those 'careful observations'. Therefore, any experimental approach is inherently empirical. Nevertheless, the controlled scientific experiment enjoys a special place in the scientific discovery process, because it is a way to determine and *measure* the effect of one quantity on another.

There is a long history of empirical observation dating back to the Babylonian astronomers, who provided data charting the movements of the heavenly bodies, from which present day astronomy continues to profit. As such, the concept of empirical observation considerably predates the scientific method of experimentation. Indeed, these ancient empirical stargazers were not only forerunners of present day astronomy, they were also astrologers, concerned as much with magic and mysticism as there were with reason and scientific experimentation [24].

Arbitrary empirical observations on their own, can provide no more than case studies in the observation of real world phenomena. While real world empirical observations have an important place in the testing of engineering artefacts in situ and in their final operation context, the first duty of scientist and engineer lies within the realm of pure experimentation, under laboratory conditions, where laboratory control serves as a mechanism for removing selection bias, confounding effect and miss observation.

6. SYNTHETIC DATA IN SOFTWARE ENGINEERING

In software engineering, pure experimentation often makes use of synthetically generated problem instances. For example, to understand the effect of a requirements analysis problem by generating instances of hypothetical requirements or the effects of a data mining approach by construction of a large number of different kinds of data set.

Curiously, in stark contrast to similar experimental work in longer-established scientific and engineering disciplines, pure experimentation is often frowned upon by computer scientists and software engineers. However, under properly controlled laboratory conditions it remains the primary way in which scientists can investigate the effects of the independent variables on the dependent variables — a principle widely accepted in all fields of science and engineering.

Therefore, it is important not to overlook the value of purely experimental studies. While laboratory conditions are not the same as real world conditions, they can be controlled. In empirical software engineering we need both laboratory controlled data and data based on real world empirical experimentation, not one or the other.

However, caution is needed. The empirical software engineering researcher might fall into the trap of using synthetic data as a *surrogate* for real data rather than as an *augmentation*; seeking to answer research questions that really should be answered using real data. When using purely experimental research for appropriate questions the experiment must be carefully designed.

Nevertheless, this does not mean that synthetic data has no role to play in empirical software engineering. For example, a data mining researcher might use synthetic data to investigate whether their algorithm could reveal interesting surprises about system behaviour. This is a question clearly best answered by real data: a 'surprise' found in synthetic data cannot really be a genuine surprise (by definition). However, synthetic data could be used to tests the scalability of the data mining algorithm.

Naturally, similar issues arise as with pure experimentation in other sciences and engineering; field trials are always required to augment laboratory testing. Fortunately, a 'best of both worlds' is also sometimes possible. Repositories may be large enough that one can find sufficiently many examples to cover a wide range of possibilities in the fine granularity required for experimental research questions. However, there are questions that can only be answered with experiments on synthetic data. For example, when exploring behaviour with corrupted, noisy and atypical cases, it may be not only necessary, but desirable to use synthetic examples.

6.1 The Role of Synthetic data in SBSE

Synthetic data can be useful as a means of experimenting with algorithms based on computational search. Such experiments cannot fully answer whether some SBSE approach will be useful in practice; evidence for this must ultimately accrue from empirical investigations using real world systems. The generation of synthetic data sets also requires care. For instance, the data must be reasonable and represent characteristics that may be found in the real world data sets that the techniques may encounter. Nevertheless, there are a number of experimental SBSE research questions that can be addressed using purely experimental analysis on synthetically generated data sets:

Scalability: How well does the algorithm scale with characteristics of the data? Scalability concerns resource consumption (typically space and time) of the search based algorithm as the characteristics of the input data vary. Scalability is a paramount concern in almost all software engineering applications. The input data variation is not necessarily merely a matter of the sheer size of the data set (though this is often important). The performance of some optimisation algorithms may also depend on other characteristics, such as density of dependence relations, correlations between elements and other non-size-based data characteristics. A purely experimental approach allows precise, fine-grained variation of data characteristics to explore the relationship between empirical algorithmic scalability and theoretical complexity bounds.

While general empirical and theoretical algorithm performance may be known for arbitrary problems, the specific software engineering problem in hand may exhibit peculiar scalability trends. Scalability is influenced by choices of representation and fitness function as well as the choice of search algorithm. An empirical scalability study can also determine the size and data characteristics at which an 'intelligent' search outperforms a purely random search, as has been done for requirements engineering problems [73].

Robustness: How resilient are the results on the presence of bias, noise, incompleteness and incorrectness in the inputs? Software engineering problems are often characterised by noisy, incomplete and even inconsistent data. SBSE has been argued to be well-suited to this paradigm [31]; search algorithms are naturally robust in the presence of incomplete and noisy data, and cope well with competing and conflicting objectives. However, the degree to which the choice of algorithms, fitness and representation cope with forms of bias, noise and incompleteness is often best assessed in laboratory conditions, where precise control can be exerted over the degree of challenge with which the algorithm is presented.

Algorithmic Performance Comparison: How do a set of search based algorithms compare for a problem over a wide range of data sets. There has been much recent progress in theoretical analysis of SBSE problems [6, 17, 41, 53]. Nevertheless, there remain many SBSE problems for which the only way to determine the best choice of search algorithm remains entirely empirical. In these situations one would certainly like to know how each algorithm performs on real world problems. These real world results can often be complemented by a more thorough purely experimental study, in which the factors that affect the choice can be explored in more detail. A study that exploits the full control afford by an experimental design unfettered by the availability of suitable real world data sets. Such a purely experimental approach necessitates the separate research problem of instance generation, a problem that has been considered in comparative studies of SBSE algorithms for requirements engineering [23].

Non-Functional Properties: How does the approach behave with respect to non-functional properties? Such properties of the search algorithm, such as its power consumption, response to change and communication bandwidth have not traditionally been the subject of intense investigation. However, in order to realise the Dynamic Adaptive SBSE agenda outlined in this paper, it will be necessary to compile (or otherwise embed) the search based computation into the deployed software to achieve search based adaptivity.

In this new paradigm of Dynamic Adaptive SBSE, non-functional characteristics of the search algorithms will be inherited by the software it is used to create. The complex interplay between several non-functional properties and the many problem characteristics that potentially influence them will mean that a full and thorough empirical evaluation will require a large and diverse body of data sets. Once again, the best way to ensure controllability of experimental method may be to create synthetic problem instances.

Adaptability: How well does a proposed SBSE approach cope with changes in the context or environment? The Dynamic Adaptive SBSE agenda will require algorithms and approaches that retain strong performance and result quality in the presence of changes in context and operating environment. Controlling for the operating environment of an approach is something that, almost inherently, calls for some form of laboratory experimentation, rather than a 'real world' evaluation; achieving laboratory control of experimental variables in a production deployed environment is unlikely to be realistic. Of course, results from such laboratory experiments should be augmented with field trials, but a field trial may not enable the researcher to report results for a wide variety of challenging contexts, which a purely experimental study can.

In many of the situations above, a purely experimental study alone will be insufficient and should be augmented with real world studies. Where real world data is abundant (for example when studying open source code as the subject of the empirical study), it may even be possible to find scale and variety in the available real world data sets sufficient to support a detailed experimental evaluation. However, in many situations, it is the very nature of the research questions asked that prohibits the use of real world data. For example, when attempting to assess scalability or robustness beyond what could reasonably be *currently* expected, the researcher must, to some extent, generate the experimental data set in order for it to be demanding.

7. THE DAASE PROJECT

The research agenda briefly outlined in this paper forms the focus of the DAASE project (DAASE: Dynamic Adaptive Automated Software Engineering).

DAASE is a major research initiative running from June 2012 to May 2018, funded by £6.8m from the Engineering and Physical Sciences Research Council (the EPSRC). DAASE also has matching support from University College London and the Universities of Birmingham, Stirling and York, which will complement the 22 EPSRC-funded post doctoral researchers recruited to DAASE with 26 fully funded PhD studentships and 6 permanent faculty positions (assistant and associate professors).

The DAASE project is keen to collaborate with leading researchers and research groups. We are also interested in collaboration with industrial parters and other organisations interested in joining the existing DAASE industrial partners which include Berner & Mattner, British Telecom, Ericsson, GCHQ, Honda, IBM and Microsoft. We have a programme for short and longer term visiting scholars (at all levels from PhD student to full professor) and arrangements for staff exchanges and internships with other organisations.

For more information, contact Lena Hierl, the DAASE Administrative Manager (crest-admin@ucl.ac.uk) or Mark Harman, the DAASE project director.

8. CONCLUSION

Dynamic adaptive search based software engineering is a development of the SBSE research agenda in which we seek to embed into the deployed code the optimisation techniques developed over the past decade of SBSE research. In so-doing we seek to address the goals espoused by advocates of self-adaptive and autonomic computing, not merely to fix faults and cope with anomalies, but as a routine and natural means of on-line adaptivity to meet new challenges, environments and platforms. The approach may be particularly effective in the emerging world of more continuous non-functional properties.

We also look towards a Hyper-Heuristic future for SBSE, in which hyper heuristic search is used to improve the applicability and generality of SBSE techniques at the expense of some loss in quality of results. We argue that this may prove to be an important step in the wider practitioner uptake.

Both real world empirical studies and purely experimental studies (using laboratory-controlled synthetic examples) will be required to evaluate the practical aspects of Dynamic Adaptive SBSE. Theoretical analysis of problem characteristics, algorithm choices and solution space properties will also be needed to provide a sound scientific underpinning for this optimisation-based approach to dynamic adaptivity.

Acknowledgements: We would like to thank those whose ideas influenced this work (with apologies to those whom we may have failed to list here specifically): Enrique Alba, Nadia Alshahwan, Andrea Arcuri, Peter Bentley, Lionel Briand, Javier Dolado, Robert Feldt, Stephanie Forrest, Carlo Ghezzi, Rob Hierons, Mike Holcombe, Yue Jia, Bryan Jones, Kiran Lakhotia, Bill Langdon, Claire Le Goues, Spiros Mancoridis, Phil McMinn, Tim Menzies, Riccardo Poli, Marc Roper, Martin Shepperd, Paolo Tonella, Shin Yoo, Wes Weimer, Joachim Wegener, David White, Andreas Zeller & Yuanyuan Zhang. Thanks also to Lena Hierl for proof reading.

9. REFERENCES

[1] W. Afzal and R. Torkar. On the application of genetic programming for software engineering predictive modeling: A systematic review. *Expert Systems Applications*, 38(9):11984–11997, 2011.

[2] W. Afzal, R. Torkar, and R. Feldt. A systematic review of search-based testing for non-functional system properties. *Information and Software Technology*, 51(6):957–976, 2009.

[3] W. Afzal, R. Torkar, R. Feldt, and G. Wikstrand. Search-based prediction of fault-slip-through in large software projects. In *Second International Symposium on Search Based Software Engineering (SSBSE 2010)*, pages 79–88, Benevento, Italy, 7-9 Sept. 2010.

[4] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test-case generation. *IEEE Transactions on Software Engineering*, pages 742–762, 2010.

[5] N. Alshahwan and M. Harman. Automated web application testing using search based software engineering. In 26^{th} *IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 3 – 12, Lawrence, Kansas, USA, 6th - 10th November 2011.

[6] A. Arcuri. It does matter how you normalise the branch distance in search based software testing. In *International Conference on Software testing (ICST 2010)*, pages 205–214, Paris, France, 2010. IEEE Computer Society.

[7] A. Arcuri, D. R. White, J. A. Clark, and X. Yao. Multi-objective improvement of software using co-evolution and smart seeding. In 7^{th} *International Conference on*

Simulated Evolution and Learning (SEAL 2008), pages 61–70, Melbourne, Australia, December 2008. Springer.

[8] A. Arcuri and X. Yao. A Novel Co-evolutionary Approach to Automatic Software Bug Fixing. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC '08)*, pages 162–168, Hongkong, China, 1-6 June 2008. IEEE Computer Society.

[9] P. Baker, 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)*, pages 176–185, Philadelphia, Pennsylvania, USA, Sept. 2006.

[10] V. R. Basili, R. W. Selby, and D. H. Hutchens. Experimentation in software engineering. *IEEE Transactions on Software Engineering*, 12(7):733–743, July 1986.

[11] J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger. A survey of self-management in dynamic software architecture specifications. In D. Garlan, J. Kramer, and A. L. Wolf, editors, *Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managed Systems (WOSS 2004)*, pages 28–33, California, USA, October 31 - November 1 2004. ACM.

[12] L. Briand. Embracing the engineering side of software engineering. *IEEE Software*, 2012. To appear.

[13] E. K. Burke, B. McCollum, A. Meisels, S. Petrovic, and R. Qu. A Graph-Based Hyper-Heuristic for Timetabling Problems. *European Journal of Operational Research*, 176(1):177–192, 2007.

[14] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: preliminary assessment. In 33^{rd} *International Conference on Software Engineering (ICSE'11)*, pages 1066–1071, New York, NY, USA, 2011. ACM.

[15] B. Cheng and J. Atlee. From state of the art to the future of requirements engineering. In L. Briand and A. Wolf, editors, *Future of Software Engineering 2007*, Los Alamitos, California, USA, 2007. IEEE Computer Society Press. This volume.

[16] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, editors. *Software Engineering for Self-Adaptive Systems (Dagstuhl Seminar)*, volume 08031 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2008.

[17] J. F. Chicano, J. Ferrer, and E. Alba. Elementary landscape decomposition of the test suite minimization problem. In M. B. Cohen and M. Ó. Cinnéide, editors, 3^{rd} *International Symposium on Search Based Software Engineering (SSBSE 2011)*, volume 6956 of *Lecture Notes in Computer Science*, pages 48–63, Szeged, Hungary, 2011. Springer.

[18] 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.

[19] A. Corazza, S. D. Martino, F. Ferrucci, C. Gravino, F. Sarro, and E. Mendes. How effective is tabu search to configure support vector regression for effort estimation? In 6^{th} *International Conference on Predictive Models in Software Engineering (PROMISE '10)*, Timisoara, Romania, 12-13 September 2010. IEEE.

[20] S. L. Cornford, M. S. Feather, J. R. Dunphy, J. Salcedo, and T. Menzies. Optimizing Spacecraft Design - Optimization Engine Development: Progress and Plans. In *Proceedings of the IEEE Aerospace Conference*, pages 3681–3690, Big Sky, Montana, March 2003.

[21] J. Darlington and R. M. Burstall. A system which automatically improves programs. *Acta Informatica*, 6:41–60, 1976.

[22] E. W. Dijkstra. On a political pamphlet from the middle ages (A response to the paper 'social processes and proofs of theorems and programs' by DeMillo, Lipton, and Perlis). *ACM SIGSOFT, Software Engineering Notes*, 3(2):14–17, 1978.

[23] J. J. Durillo, Y. Zhang, E. Alba, M. Harman, and A. J. Nebro. A study of the bi-objective next release problem. *Empirical Software Engineering*, 16(1):29–60, 2011.

[24] P. Fara. *Science: A 4000-year history*. Oxford University Press, 2009.

[25] A. Filieri, C. Ghezzi, A. Leva, and M. Maggio. Self-adaptive software meets control theory: A preliminary approach supporting reliability requirements. In P. Alexander, C. S. Pasareanu, and J. G. Hosking, editors, *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 283–292, Lawrence, KS, USA, November 2011. IEEE.

[26] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In 8^{th} *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '11)*, pages 416–419. ACM, September 5th - 9th 2011.

[27] F. G. Freitas and J. T. Souza. Ten years of search based software engineering: A bibliometric analysis. In 3^{rd} *International Symposium on Search based Software Engineering (SSBSE 2011)*, pages 18–32, 10th - 12th September 2011.

[28] Z. P. Fry, B. Landau, and W. Weimer. A human study of patch maintainability. In *International Symposium on Software Testing and Analysis (ISSTA'12)*, Minneapolis, Minnesota, USA, July 2012. To appear.

[29] A. G. Ganek. Autonomic computing: Implementing the vision. In *Active Middleware Services*, pages 2–3. IEEE Computer Society, 2003.

[30] C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering (ICSE 2012)*, Zurich, Switzerland, 2012. to appear.

[31] M. Harman. The current state and future of search based software engineering. In L. Briand and A. Wolf, editors, *Future of Software Engineering 2007*, pages 342–357, Los Alamitos, California, USA, 2007. IEEE Computer Society Press.

[32] M. Harman. Search based software engineering for program comprehension. In 15^{th} *International Conference on Program Comprehension (ICPC 07)*, pages 3–13, Banff, Canada, 2007. IEEE Computer Society Press.

[33] M. Harman. Open problems in testability transformation. In *1st International Workshop on Search Based Testing (SBT 2008)*, Lillehammer, Norway, 2008.

[34] M. Harman. The relationship between search based software engineering and predictive modeling. In 6^{th} *International Conference on Predictive Models in Software Engineering*, Timisoara, Romania, 2010.

[35] M. Harman. Why source code analysis and manipulation will always be important. In 10^{th} *IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 7–19, Timisoara, Romania, 2010.

[36] M. Harman. Why the virtual nature of software makes it ideal for search based optimization. In 13^{th} *International Conference on Fundamental Approaches to Software Engineering (FASE 2010)*, pages 1–12, Paphos, Cyprus, March 2010.

[37] M. Harman. Software engineering meets evolutionary computation. *IEEE Computer*, 44(10):31–39, Oct. 2011.

[38] M. Harman. The role of artificial intelligence in software engineering. In 1^{st} *International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE 2012)*, Zurich, Switzerland, 2012.

[39] M. Harman and B. F. Jones. Search based software

engineering. *Information and Software Technology*, 43(14):833–839, Dec. 2001.

[40] M. Harman, A. Mansouri, and Y. Zhang. Search based software engineering trends, techniques and applications. *ACM Computing Surveys*, 2012. To appear.

[41] M. Harman and P. McMinn. A theoretical and empirical study of search based testing: Local, global and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247, 2010.

[42] M. Harman, P. McMinn, J. Souza, and S. Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In B. Meyer and M. Nordio, editors, *Empirical software engineering and verification: LASER 2009-2010*, pages 1–59. Springer, 2012. LNCS 7007.

[43] C. A. R. Hoare. The engineering of software: A startling contradiction. In D. Gries, editor, *Programming Methodology, A Collection of Articles by Members of IFIP WG2.3*. Springer-Verlag, New York, NY, 1978.

[44] C. A. R. Hoare. How did software get so reliable without proof? In *FME '96: Industrial Benefit and Advances in Formal Methods: Third International Symposium of Formal Methods Europe*, number 1051 in LNCS, pages 1–17. Springer-Verlag, Mar. 1996.

[45] S. A. Hofmeyr and S. Forrest. Immunity by design: An artificial immune system. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '99)*, 2:1289–1296, 1999.

[46] Y. Jia and M. Harman. Milu: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In 3^{rd} *Testing Academia and Industry Conference - Practice and Research Techniques (TAIC PART'08)*, pages 94–98, Windsor, UK, August 2008.

[47] J. Kim, P. J. Bentley, U. Aickelin, J. Greensmith, G. Tedesco, and J. Twycross. Immune system approaches to intrusion detection - A review. *Natural Computing: An international journal*, 6, Dec. 2007.

[48] K. Krogmann, M. Kuperberg, and R. Reussner. Using genetic search for reverse engineering of parametric behaviour models for performance prediction. *IEEE Transactions on Software Engineering*, 36(6):865–877, November-December 2010.

[49] K. Lakhotia, M. Harman, and H. Gross. AUSTIN: A tool for search based software testing for the C language and its evaluation on deployed automotive systems. In 2^{nd} *International Symposium on Search Based Software Engineering (SSBSE 2010)*, pages 101 – 110, Benevento, Italy, September 2010.

[50] K. Lakhotia, N. Tillmann, M. Harman, and J. de Halleux. FloPSy — Search-based floating point constraint solving for symbolic execution. In 22^{nd} *IFIP International Conference on Testing Software and Systems (ICTSS 2010)*, pages 142–157, Natal, Brazil, November 2010. LNCS Volume 6435.

[51] W. B. Langdon and M. Harman. Evolving a CUDA kernel from an nVidia template. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2010.

[52] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[53] P. K. Lehre and X. Yao. Runtime analysis of search heuristics on software engineering problems. *Frontiers of Computer Science in China*, 3(1):64–72, 2009.

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

[55] 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.

[56] I. H. Moghadam and Mel Ó Cinnéide. Code-Imp: A tool for automated search-based refactoring. In *Proceeding of the 4th workshop on Refactoring Tools (WRT '11)*, pages 41–44, Honolulu, HI, USA, 2011.

[57] A. Newell and H. A. Simon. Computer science as empirical inquiry: symbols and search. *Communications of the ACM*, 19:113–126, 1976.

[58] A. Ngo-The and G. Ruhe. A systematic approach for solving the wicked problem of software release planning. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 12(1):95–108, August 2008.

[59] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14:54–62, May 1999.

[60] K. R. Popper. *Conjectures and Refutations: The Growth of Scientific Knowledge*. Routledge, 2003.

[61] O. Räihä. A survey on search–based software design. *Computer Science Review*, 4(4):203–249, 2010.

[62] W. F. Tichy. Should computer scientists experiment more? *IEEE Computer*, 31(5):32–40, May 1998.

[63] P. Tonella. Evolutionary testing of classes. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '04)*, pages 119–128, Boston, Massachusetts, USA, 11-14 July 2004. ACM.

[64] J. Wegener and O. Bühler. Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system. In *Genetic and Evolutionary Computation Conference (GECCO 2004)*, pages 1400–1412, Seattle, Washington, USA, June 2004. LNCS 3103.

[65] W. Weimer, T. V. Nguyen, C. L. Goues, and S. Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering (ICSE 2009)*, pages 364–374, Vancouver, Canada, 2009.

[66] D. R. White, J. Clark, J. Jacob, and S. Poulding. Searching for resource-efficient programs: Low-power pseudorandom number generators. In *2008 Genetic and Evolutionary Computation Conference (GECCO 2008)*, pages 1775–1782, Atlanta, USA, July 2008. ACM Press.

[67] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Kluwer Academic Publishers, 2000.

[68] S. Xanthakis, C. Karapoulios, R. Pajot, and A. Rozz. Immune system and fault-tolerant computing. *Artificial Evolution (Lecture Notes in Computer Science)*, 1063:181–197, 1996.

[69] S. Yoo, R. Nilsson, and M. Harman. Faster fault finding at Google using multi objective regression test optimisation. In 8^{th} *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '11)*, Szeged, Hungary, September 5th - 9th 2011. Industry Track.

[70] Y. Zhang, E. Alba, J. J. Durillo, S. Eldh, and M. Harman. Today/future importance analysis. In *ACM Genetic and Evolutionary Computation COnference (GECCO 2010)*, pages 1357–1364, Portland Oregon, USA, 7th–11th July 2010.

[71] Y. Zhang, A. Finkelstein, and M. Harman. Search based requirements optimisation: Existing work and challenges. In *International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ'08)*, volume 5025, pages 88–94, Montpellier, France, 2008. Springer LNCS.

[72] Y. Zhang, M. Harman, and A. Mansouri. The SBSE repository: A repository and analysis of authors and research articles on search based software engineering. crestweb.cs.ucl.ac.uk/resources/sbse_repository/.

[73] 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*, pages 1129 – 1137, London, UK, July 2007. ACM Press.