Achievements, open problems and challenges for search based software testing

Mark Harman, Yue Jia and Yuanyuan Zhang
University College London, CREST Centre, London, UK

***Abstract*—Search Based Software Testing (SBST) formulates testing as an optimisation problem, which can be attacked using computational search techniques from the field of Search Based Software Engineering (SBSE). We present an analysis of the SBST research agenda[1], focusing on the open problems and challenges of testing non-functional properties, in particular a topic we call 'Search Based Energy Testing' (SBET), Multi-objective SBST and SBST for Test Strategy Identification. We conclude with a vision of FIFIVERIFY tools, which would automatically find faults, fix them and verify the fixes. We explain why we think such FIFIVERIFY tools constitute an exciting challenge for the SBSE community that already could be within its reach.**

I. INTRODUCTION

Search Based Software Testing (SBST) is the sub-area of Search Based Software Engineering (SBSE) concerned with software testing [2], [85]. SBSE uses computational search techniques to tackle software engineering problems (testing problems in the case of SBST), typified by large complex search spaces [58]. Test objectives find natural counterparts as the fitness functions used by SBSE to guide automated search, thereby facilitating SBSE formulations of many (and diverse) testing problems. As a result, SBST has proved to be a widely applicable and effective way of generating test data, and optimising the testing process. However, there are many exciting challenges and opportunities that remain open for further research and development, as we will show in this paper.

It is widely believed that approximately half the budget spent on software projects is spent on software testing, and therefore, it is not surprising that perhaps a similar proportion of papers in the software engineering literature are concerned with software testing. We report an updated literature analysis from which we observe that approximately half of all SBSE papers are SBST papers, a figure little changed since the last thorough publication audit (for papers up to 2009), which found 54% of SBSE papers concerned SBST [56]. Many excellent and detailed surveys of the SBST literature can be found elsewhere [2], [4], [55], [85], [126]. Therefore, rather than attempting another survey, we provide an analysis of SBST research trends, focusing on open challenges and areas for future work and development.

[1]This keynote was given by Mark Harman at the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST 2015), but this paper, on which the keynote was based, is the work of all three authors.

II. A BRIEF HISTORY OF SBST

Since the first paper on SBST is also likely to be the first paper on SBSE, the early history of SBST is also the early history of SBSE. SBSE is a sub-area of software engineering with origins stretching back to the 1970s but not formally established as a field of study in its own right until 2001 [51], and which only achieved more widespread acceptance and uptake many years later [38], [43], [100].

The first mention of *software optimisation* (of any kind) is almost certainly due to Ada Augusta Lovelace in 1842. Her English language translation of the article (written in Italian by Menabrae), 'Sketch of the Analytical Engine Invented by Charles Babbage' includes seven entries, labelled 'Note A' to 'Note G' and initialed 'A.A.L'. Her notes constituted an article themselves (and occupied three quarters of the whole document). In these notes we can see perhaps the first recognition of the need for software optimisation and source code analysis and manipulation (a point argued in more detail elsewhere [44]):

> "*In almost every computation a great variety of arrangements for the succession of the processes is possible, and various considerations must influence the selection amongst them for the purposes of a Calculating Engine. One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation.*" Extract from 'Note D'.

The introduction of the idea of software testing is probably due to Turing [115], who suggested the use of manually constructed assertions. In his short paper, we can find the origins of both *software testing* and *software verification*. The first use of *optimisation techniques* in software testing and verification probably dates back to the seminal PhD thesis by James King [67], who used automated symbolic execution to capture path conditions, solved using linear programming. The first formulation of the test input space as a *search space* probably dates back seven years earlier to 1962, when a Cobol test data generation tool was introduced by Sauder [103]. Sauder formulates the test generation problem as one of finding test inputs from a search space, though the search algorithm is *random search*, making this likely to be the first paper on Random Test Data Generation. Sauder's work is also significant because it introduces the idea of constraints to capture path conditions, although these constraints are manually defined and not automatically constructed.

The first paper to use a *meta-heuristic search* technique was probably the work of Boyer, Elspas and Levitt on the SELECT system [16]. The paper is remarkable in many ways. Consider the following paragraph, quoted from the paper:

> "*The limitation of the above algorithms to linear combinations is an unacceptable, and vexing, one. For example, they could not handle an inequality like* $X*Y+10*Z-W \geq 5$ *among its constraints, unless one were prepared to assign to* X *a trial value, and then attempt a solution (assuming the other inequalities are linear). We therefore considered various alternatives that would not be subject to this limitation. The most promising of these alternatives appears to be a conjugate gradient algorithm ('hill climbing' program) that seeks to minimise a potential function constructed from the inequalities.*" [16]

Here we can see, not only the first use of *computational search* (hill climbing) in software engineering, but also a hint at the idea (assignment of concrete values) that was subsequently to become *Dynamic Symbolic Execution (DSE)* [21]. Within this single paragraph we therefore may arguably find the origins of both DSE and SBST (and, by extension, SBSE too).

The SELECT paper is also remarkable in its sober and prescient assessment of the relative merits of testing and verification. Shortly after its publication, these two closely related research communities entered into a protracted and unhelpful 'feud' that generated a great deal more heat than light [29], [31], [35], [60]. Fortunately, we have more recently witnessed an accommodation between the two communities [61], and greater degree of welcome collaboration at their intersection [59]. We really ought to ruefully reflect on the delay in this rapprochement given the 'understanding' already set out by the SELECT paper in 1975. For example, speaking about the complementarity of testing and verification, the authors have this to say:

> "*Even after a mathematical proof of correctness, one cannot be certain that the program will run as intended on a given machine. Testing in the real machine environment on actual data would appear to be a useful complementary technique to formal verification since it is not contingent on [such] assumptions.*" [16]

At about the same time[2] Miller and Spooner [86], were also experimenting with optimisation-based approaches for generating test data (which they refer to as 'test selection' in the sense that they 'select' from the input space, which, in the more recent literature we would refer to as 'test data generation').

[2]The Miller and Spooner paper was published in 1976, but was received by the journal on the 9th of September 1975. The acknowledgements of the 1976 journal paper indicate that it was one of the referees who pointed out the existence of the 1975 conference paper, which the 1976 paper cites. Although the conference was held in April 1975 and the proceedings appeared in the July 1975 issue of ACM SIGPLAN Notices, it is quite likely that Miller and Spooner were already working on their manuscript, which was submitted only a couple of months later.

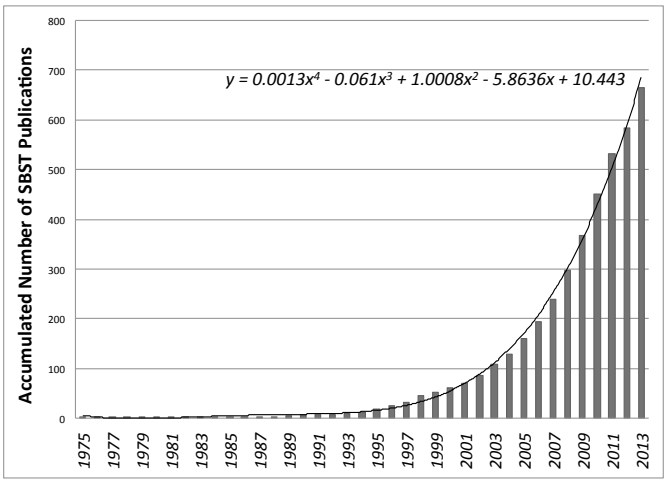

Fig. 1: Cumulative number of Search Based Software Testing papers. As can be seen, the overall trend continues to suggest a polynomial yearly rise in the number of papers, highlighting the breadth of interest and strong health of SBST.

Unlike Boyer et al. [16], Miller and Spooner used concrete execution of the program rather than symbolic execution, making their approach more similar to the techniques that ultimately became SBST, while the work of Boyer et al. followed a closely-related (but different) evolutionary path, which ultimately led to DSE. Current research develops both these techniques, and also hybrids that combine the best features of both [9], [63], [71], [110].

It appears that SBST research lay dormant for at approximately a decade until the work of Korel [68], which introduced a practical test data generation approach, the Alternating Variable Method (AVM), based on hill climbing. The first use of genetic algorithms for software engineering problems is usually attributed also to the field of SBST, with the work of Xanthakis et al. [122], who introduced a genetic algorithm to develop whole test suites. Subsequent theoretical and empirical results tend to suggest that AVM outperforms genetic algorithms (in 'non-royal road' test data generation problems), at least for imperative programs in the C language [57]. Since the late 1990s, with a greater overall software engineering focus on SBSE, there has been an explosion in SBST publications as the analysis below indicates.

Analysis of Trends in SBST: Figure 1 shows the growth in papers published on SBST. The data is taken from the SBSE repository [130]. The aim of the repository is to contain every SBSE paper, underpinned by regular and careful human-based update. Although no repository can guarantee 100% precision and recall, the SBSE repository has proved sufficiently usable that it has formed the basis of several other detailed analyses of the literature [27], [38], and is widely used by the SBSE community as a first source of information on related work. We found a close fit to a quartic function, indicating strong polynomial growth. If the trend continues, there will be more than 1,700 SBST papers before the end of this decade.

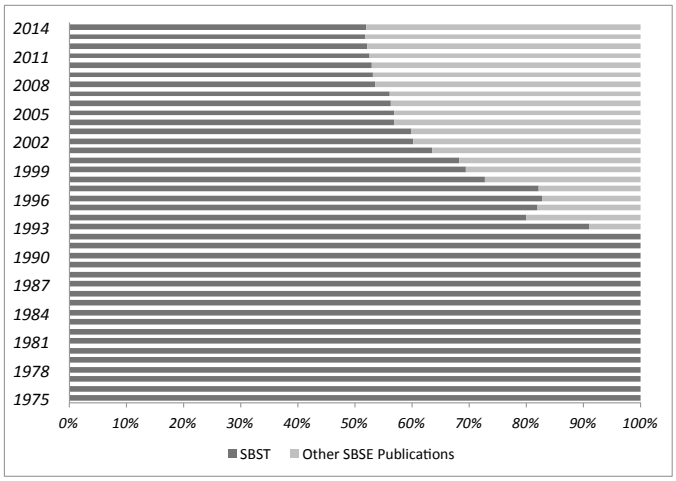

Fig. 2: The changing ratio of SBSE papers that are SBST papers. Initially, SBST dominated SBSE. Over the years, this ratio has decreased, stabilising at around 50%. This represents the growth in non-testing related areas of SBSE rather than any decline in the number of papers on SBST (as can be seen by comparing this figure with Figure 1).

The data for the most recent years is usually an under estimate of the number of papers published, because there is an inherent latency involved, depressing the assessment of the number of papers. Notwithstanding these issues, Figure 1 clearly shows, the number of SBST papers published is rising polynomially, indicating continued health, growth and interest in this approach to software testing problems.

In order to understand the relative contribution of software testing to the overall body of literature on SBSE, we also report, in Figure 2, the relative proportion of papers in the overall field of SBSE that concerns software testing problems. As can be seen from this figure, the proportion of papers on SBST is reducing, although their number is increasing. These two figures, taken together, thus reflect growing interest in SBSE, rather than decreasing interest in SBST.

Perhaps it is a sign of SBSE's growing maturity, that the proportion of SBSE papers on testing problems appears to be stabilising at approximately the value (roughly half) witnessed throughout software engineering more generally. We believe that the proportional SBST decline is therefore more a stabilisation to a reasonable proportion. This proportion, perhaps not coincidentally, is also widely regarded as the estimate of the amount of effort/budget allocated by practising software engineers to software testing.

SBST's Industrial Applications and Tools: SBST is now sufficiently mature that it has transitioned from laboratory study to industrial application, for example at Daimler [117], Ericsson [3] and Microsoft [111]. There are also publicly available SBSE tools for automated program repair [76], and tools for SBST for popular languages, such as AUSTIN [69], an open source SBST system for the C language, and EvoSuite [36], an open source SBST system for Java.

EvoSuite has proved to be particularly effective as a tool for testing Java programs. It is provided as a plug-in to Eclipse that works 'out-of-the-box' (the user simply needs to click 'run EvoSuite'). A great deal of engineering effort has been directed towards the usability of the tool for practical software testing. For example, most computational search algorithms are 'anytime' algorithms; they can be stopped at any time and yield the best result found so far. EvoSuite exploits this by ensuring that all executions complete within reasonable time.

For regression testing, the selection and prioritisation algorithms are easy to implement. For such regression testing tools the fitness function need not be a part of the tool itself, as it is for test data generation. Instead, the search based regression test optimisation tool simply relies on recorded information concerning the properties of interest of the test suite. This makes these algorithms easy to deploy in a real world setting, provided data is available. Adoption effort is more normally found to be that associated with data collection rather than tool deployment in our experience.

Breadth of SBST Applications to Testing Problems: SBST for structural coverage is the most well studied and well understood paradigm within SBST. This was true when last surveyed in 2009 [55] and it remains the case among the 718 papers published on SBST to the present day in the analysis we present in this paper.

The structural code coverage achieved is not always as high as we might hope [70], with the result that we may need to rely on non-adequate test suites and all that this entails [39] using currently available tools. However, the principles are relatively well understood and progress continues with regular newly published incremental advances on the state-of-the-art.

The breadth and diversity of other testing paradigms, domains and applications attacked using SBST is a compelling testament to its general and widespread applicability. For any desirable properties of good test data that are captured as adequacy criteria, these criteria naturally reformulate as fitness functions. As has also been known, since at least 1962 [103], a system's input space makes a very natural *search* space, in which we can automate the process of searching for test inputs that meet these test adequacy criteria.

Here is a long (yet partial) list of just some of the testing problems with citations to a few example papers (of many) that adopt an SBST approach to find suitable test data: functional testing [118], safety testing [11], [32], security testing [41], robustness testing [104], integration testing [18], [26], service-based testing [24], temporal testing [19], [113], [119], exception testing [114], Combinatorial Interaction Testing (CIT) [20], [25], [95], (and Software Product Line (SPL) testing [48]), state [77] and state-based-model testing [30], [78] (including popular modelling notations such as MATLAB Simulink [90], [129]), and mutation based test [37], [49] and mutant [65], [92] generation.

The State of the Art: SBST has made many achievements, and demonstrated its wide applicability and increasing uptake. Nevertheless, there are pressing open problems and challenges that need more attention and to which we now turn.

Specifically:

1) We need to extend SBST to test non-functional properties, a topic that remains relatively under-explored, compared to structural testing (as revealed in Section III below). In particular, we need more work on Search Based Energy Testing (SBET).
2) We need Search Based Test Strategy Identification (SBTSI). Regression test process optimisation is well developed and understood, but techniques for finding *test generation* strategies remain under-developed.
3) We need more work on multi-objective test data generation techniques (MoSBaT). Previous work on search based test data generation has tended to focus solely on a single objective optimisation (such as branch coverage), with comparatively little work on multi-objective test data generation. Unfortunately, real-world testing problems are messy, constrained and are unlikely to be captured by a single objective.

In the remainder of this paper, we present a roadmap of future work in these three areas of Search Based Energy Testing (SBET), Search Based Test Strategy Identification (SBTSI) and Multi-objective Search Based Testing (MoSBaT). We wish to conclude on a positive note, highlighting the exciting opportunities that arise because of the extraordinary progress in SBST in particular, and SBSE in general.

We therefore close the paper with an outline of 'FiFiVerify tools'; tools that use SBSE and verification to automatically find faults, fix them and verify the fixes. Such FiFiVerify tools would be a fitting development and realisation of testing and verification complementarity, which was expressed so eloquently by Boyer, Elspas and Levitt in their 1975 SELECT paper (discussed earlier in this section).

III. Search Based Energy Testing (SBET)

An excellent survey of the state-of-the-art in search based software testing for non-functional system-level properties was presented by Afzal, Torkar and Feldt [2]. We used the SBSE repository [130] to extend to 2014, the quantitative analysis of publications contained in the paper by Afzal et al.

The results are presented in Figure 3. As can be seen from this figure, there remains activity in this area. However, given the overall growth in papers on search based software testing, revealed by Figure 1, it is surprising (and perhaps disappointing) that more work is not focused on these properties.

Lack of work on non-functional properties is surprising because of the increasing importance of non-functional properties. It is disappointing because search based software testing techniques have the significant advantage that they can, theoretically, be applied to *any* testing problem for which the adequacy criterion can be captured as a fitness function. In principle, testing for execution time, quality of service, and energy consumption, should be no more difficult than testing for branch coverage; we simply require a different fitness function. Of course, the *measurements* that inform fitness may come with their own sets of challenges, peculiar to each non-functional property of interest.

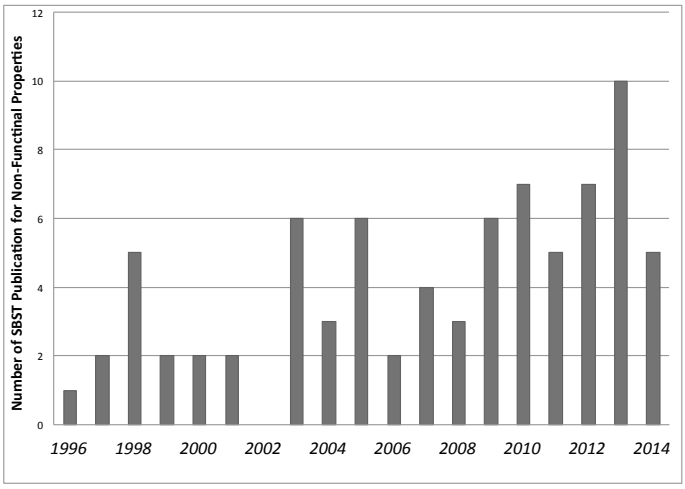

Fig. 3: SBST for non-functional properties. Data from 1996 to 2007 comes from the SLR by Afzal et al. [2]. Data from 2008 to 2014 was computed using the SBSE repository [130].

Analysis of all Work on Non-Functional SBST: In total, since the review by Afzal et al. (i.e., since 1st. January 2008), there have been 44 SBST papers on non-functional properties (9% of the 464 in total on SBST over the same period). This compares to 35 papers, (16% of the 221 published) over the period of the study by Afzal et al. Although the number of papers is steadily rising, this could be simply due to overall SBST growth; the *proportion* appears to be falling, a troubling finding when we consider the importance of non-functional properties. The proportion of SBST papers concerning non-functional properties ought to be closer to 50% than 10%, if research activity is to adequately reflect importance.

Analyzing sub-topic distribution between the two periods, we compared the results reported by Afzal et al., with those we obtained, by extending their analysis. Afzal et al. identified 5 categories. We observed activity in all 5 of these, and new activity in a further 6. We thus conclude that SBST has been used to test at least 11 different non-functional attributes, with overall research output in the ratios given by Figure 4.

The Startling Lack of SBET Work: There is work on SBSE for improving energy consumption. For example, Li et al. [80] formulate energy optimisation as the problem of finding the mobile device screen colour choices that minimise energy consumption, while maintaining colour contrast. Monotas et al. [83] also define a search space for energy optimisation choices. They currently use an exhaustive search, but plan to extend to full SBSE for scalability to larger search spaces. Both approaches are similar, in spirit, to Genetic Improvement [53], since they search the space of program improvements. However, we could find only a single paper that has been published on Search Based Energy Testing (SBET) [15]. It is possible that our search has failed to find all papers. However, we remain confident that the overall trends we report are reasonably accurate and can be fairly confident about the finding that SBET is under-developed in the literature.

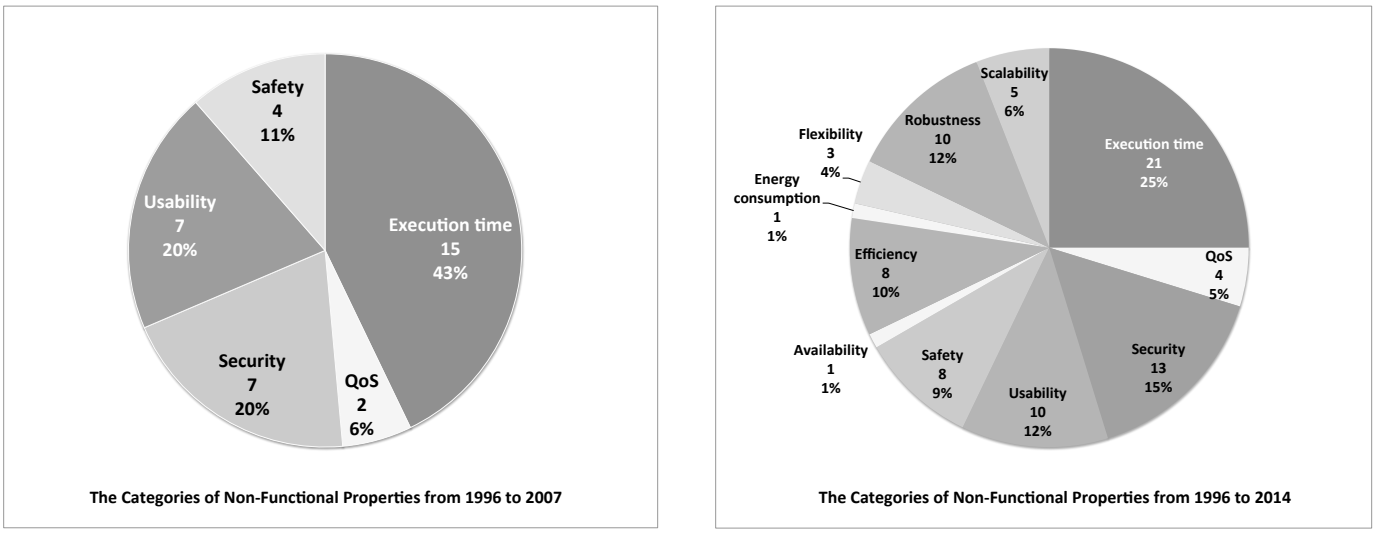

Fig. 4: The pie chart on the left shows the relative distribution of papers published over the five categories identified by Afzal et al. [2] in their systematic literature review (1996-2007), while the one on the right shows the 11 categories we identified in 2014 (which extends the analysis of Afzal et al. to include all papers from 1996-2014). As can be seen, a number of different non-functional properties that have been attacked using SBST has grown, yet there is very little work on Search Based Energy Testing (SBET), which is particularly surprising, given the importance of this problem.

Energy optimisation has been a topic of interest for at least 20 years [112], and is gaining considerable recent interest because of its implications for the environment, and due to the dramatic increase in battery-powered computing. In order to make progress on search based software testing of non-functional properties, we need to *measure* the non-functional properties of concern with sufficiently computationally efficient fitness functions. This need for efficient fitness computation may mandate the use of surrogates or approximations to the true measurement [47]. In this section, we focus on Search Based Energy Testing (SBET), for which we believe immediate progress can be made and for which there are already potential measurement approaches [42], [94], and possible surrogates [88].

The problem of inadequate battery life is routinely bemoaned by many mobile device users [33] and the space occupied by the battery is becoming the predominant driver of device size. This clearly affects smart general purpose mobile devices, such as phones, notepads and laptops, for which the battery may occupy as much as 90% of the available space. However, it is also important for medical devices such as pacemakers, where the battery can typically occupy at least 50% of the device [91].

Estimates for the carbon footprint of computational energy consumption vary, but all accounts agree that the proportion of energy consumed by computation is rising and that it denotes a nontrivial fraction of global energy demand. Claims that a smartphone could consume more energy per year than a medium-sized refrigerator are deemed to be exaggerated by, perhaps, a factor of four [116], so there may be some degree of hyperbole at work.

Nevertheless, the total energy consumed by computation is undoubtedly rising. One study, conducted in 2009 and again, by the same authors, in 2011 [108], estimated the proportion of global electricity consumption due to information and communications technology rose from 3% to 6% between the two years at which the assessments were reported. Testing and optimising energy consumption is therefore an ecological imperative as well as pressing user need [82].

It is a challenge to measure the amount of energy consumed by the execution of a software system in a reliable and accurate manner. However, if we can find suitable metrics that can measure energy consumption and that can be reformulated as fitness functions, according to the standard SBSE mantra 'metrics are fitness functions too' [46], then we can use these to search for worst-case and best-case energy consumption, and to find anomalies, 'energy bugs' and 'hotspots' [10].

This agenda would constitute a nascent subfield of SBST called 'Search Based Energy Testing' (SBET). In the remainder of this section we outline some of the issues and outline potential solutions to problems in energy measurement for SBET.

Efficiency: We shall require that we can measure energy consumption quickly, because the overall search based approach will need to consider many different test cases in order to search for worst case or anomalous case energy consumption.

Granularity: The measurement of energy consumption can be fine grained (assessing the individual contribution of each line of code to energy consumed), mid-grained (focussing on energy consumed by a block of code or a method/procedure) or coarse-grained (simply reporting energy consumed by the program execution over a period of time).

Fine grained approaches such as eLens [42] and Eprof [94], would be needed to profile for sensitivity analysis. Energy sensitivity information would be useful for SBSE applications such as genetic improvement. Such techniques have been used for optimising energy usage [83], [120] and for which sensitivity analysis is helpful [73]. For SBST, however, the primary need for measurement will be to capture the energy consumed by a test execution, which can be coarse-grained. This is important because coarse-grained energy measurement is likely to come with fewer technical challenges, compared to fine grained measurement.

Hawthorne effect: We have to be careful for potential 'Hawthorne'-like effects, in which the property we seek to measure is affected by the measurement process. In particular, any non-functional property we measure by instrumenting the code will likely be influenced by instrumentation code itself, thereby reducing the measurements' reliability. If the measurements' influence on the non-functional property is minimal or constant, then we might choose to either ignore it or factor it out. However, since many non-functional properties will be interesting, precisely because their effect is *context-sensitive*, we should not assume that the effect of instrumentation will be constant, and it may not be minimal.

One possible solution would be to create two versions of the system under test: one with normal instrumentation, and one with duplicated instrumentation. We can measure the non-functional property of interest for both, subtracting one from the other to determine the amount of non-functional property due purely to instrumentation. This doubles the total amount of computation required, but it potentially provides a context-sensitive and more accurate way to factor out the instrumentation influences.

Specificity: It is natural to design tools for search based software testing that are generally applicable, but non-functional properties such as energy, are inherently device and platform specific. There will be a tension between the applicability of an approach and the degree of information that it can return. By being specific, we may not merely test the energy consumed, but may additionally give detailed assessments of where this energy is consumed. Such a detailed and specific assessment might highlight ways to reduce energy consumption. For example, the Running Average Power Limit (RAPL) approach [28], has been developed by Intel to distinguish between the energy consumed in CPU, the dynamic random access memory, and the so-called 'CPU uncore' (such as caches and on-chip graphics processing units). This specificity, so closely coupled to the hardware it assesses, gives more insights as to the causes of energy consumption, but the insights it yields are naturally pertinent only to specific devices.

Specialised Hardware Requirements: Measuring the amount of energy consumed using specialised hardware, can lead to more accurate assessment of energy consumption, but requires specialised equipment [107]. Hardware-based energy measurement has been used for thread management [97] and to assess the energy implications of code obfuscation on the Android platform [102].

Hardware based approaches typically consist of several phases. For example, the SEEP approach [62] uses symbolic execution to capture paths, which are subsequently executed with concrete values to give platform-specific energy consumption for basic blocks.

For SBST, the number of executions required by test generation may make the use of hardware-based approaches prohibitive when no such API is provided. By contrast, for test case management, such as regression testing, there is a fixed pool of test cases, each of which needs to be assessed for the non-functional property of interest only once, prior to a subsequent optimisation phase. Once this is known, the optimisation problem consists of either prioritising, selecting or minimising the test suite according to the non-functional properties of interest [45], [126]. Therefore, for test management applications, such as regression test optimisation, it may be acceptable to build a specialised hardware test rig. The rig measures, once and for all, but with a greater degree of human effort, the non-functional properties of each test case. Hardware-based approaches, even those without a software API, may be applicable to test suite optimisation. Indeed, the LEAP node approach [107] has recently been used for just such a test suite optimisation [79].

IV. Search Based Test Strategy Identification (SBTSI)

Most forms of test data generation have been concerned with finding specific inputs or sets of inputs (test suites) that have desirable properties. Other SBSE formulations, as yet underexplored, have more of the character of Test Strategy Identification (TSI) problems, as we outline in this section.

Genetic Programming for SBSTI: Genetic programming is increasingly finding applications in SBSE [54], [73], [76], [121]. The primary difference between genetic programming and other forms of evolutionary computation is that the search space is a space over programs expressed in some programming language. The programming language can be as general or as specific as the application demands. Suppose we formulate simple testing strategies in a formal language. Could we then use genetic programming to search test strategies for those well adapted to a particular testing problem?

The idea of searching for testing strategies [98] rather than searching for test cases is appealing, because it may help us to raise our abstraction level; finding strategies for finding test cases rather than finding test cases themselves. It also may yield insight, which may ultimately prove to be more valuable than test suites. In the remainder of this section, we give one example of such insight, outlining how test strategy identification can be used to cluster programs and the faults they may contain.

Using SBSTI to cluster programs: Suppose we search for test strategies for a particular suite of programs that achieve high mutation score. Given a particular set of mutants and a particular set of programs, a particular strategy will emerge that is adapted to the set of programs concerned.

The difficulty in finding a suitable strategy will be partly governed by the degree to which the programs have some commonality, and the degree to which effective mutant killing submits to some particular strategy.

The difficulty of finding a solution can be measured quite naturally in terms of the fitness achieved for a given budget of computational search effort. One very desirable outcome is obviously the test strategy itself, if we can find a good one. However, even when TSI fails to identify good strategies, strategy identification *difficulty* can be used as a fitness function to help us to identify fault categories, and the programs which may contain them:

We can cluster programs with respect to a given set of faults. The cluster identification approach will, itself, be a multi-objective search problem: minimise the number of clusters, while simultaneously maximising the fitness achieved by TSI within each cluster. Programs residing in a given cluster exhibit related fault behaviour; there is a single unifying strategy for testing them in order to reveal these faults.

One possible formulation would be: Given a set of programs P, find the largest subset S for which TSI achieves a mutation adequacy (mutation score) above α on a set of mutants M. The fitness function could be the size of the subset S (including more programs is better, because TSI is more widely applicable). This formulation seeks the most general strategy for achieving at least α.

There is a great degree of choice available in the particular formulation we might adopt. For instance, we might fix the subset of programs, S, and search for a strategy that achieves the highest mutation adequacy on a given set of mutants, M. This formulation seeks the best possible strategy for finding a particular class of faults (captured by M) on a given set of programs, S.

A co-evolutionary approach to SBSTI: Suppose we vary the sets of faults considered (varying M). We might formulate this problem as a co-evolutionary search that seeks to partition the set of programs of interest, on the one hand, while simultaneously partitioning the set of mutants on the other. Both competitive [6] and cooperative [101] co-evolution have been used in the SBSE literature, but only competitive co-evolution has been used in SBST, and co-evolution, in general, remains relatively under-explored in the SBSE literature [56].

One possible co-evolutionary formulation would be to evolve the subset S and the set of mutants M. A co-operative formulation would use set size as the fitness for S and M, such that there is a strategy that achieves 100% mutation adequacy with respect to M on all of the programs S. This more co-operative approach tries to find sets of faults and programs which 'co-operate' in the sense that the faults can easily be found with a particular strategy on a large set of programs.

A competitive formulation might define the fitness of S to be the size of the largest such set for which a strategy exists that kills all mutants in M, while the fitness of M is the size of the largest such set that avoids being killed by all programs in S.

Assignment problems: Assignment problems are increasingly interesting in software engineering. They can often be formulated as systems that recommend engineers for particular tasks, such as debugging and testing [5], [14]. These recommender systems have an inherent optimisation flavour [99]: In general, we seek an assignment of solution techniques to problem instances that maximises the quality of solutions found.

In order for SBSE to be a viable approach, we need a representation, fitness function and a search space that is sufficiently large to make enumeration infeasible [58]. Assignment problems typically come with some form of representation, r that captures the mapping between solution and problem instances. There is guaranteed to be some method, a, for assessing solution quality, otherwise no intelligent assignment can be performed. It is reasonable to believe that the search space will be too large to be feasibly innumerable, since assignment problem search spaces grow exponentially. The open research problem is to find appropriate reformulations, that use a computational search technique, guided by fitness function defined in terms of a, to search the space r.

When using SBSE to attack assignment problems in software testing, we need not restrict ourselves merely to the assignment of engineers. Since we have an array of different testing techniques, and a bafflingly complicated set of possible programs and test problems to which they might be applied, there is an important assignment problem for researchers and practitioners has remained under-explored: How do we find the best assignment of test techniques to testing problems and particular programs? This is a problem for which hyper-heuristics has recently been successful [64].

V. Multi-objective Search Based Testing (MoSBaT)

For problems concerned with test suite selection and prioritisation, multi-objective approaches are increasingly prevalent [8], [13], [17], [87], [105], [106], [125]. However, for test data generation problems, the large majority of existing approaches are single objective. Relatively few attack multi-objective test case generation [7], [34], [74], [84], [124], despite it having been proposed sometime ago [52]. This is unrealistic because practising software testers are unlikely to be concerned only with a single test objective [45]. Therefore, we believe that more work is required on multi-objective search based test data generation.

Perhaps one of the reasons why multi-objective techniques have not received the attention they deserve, lies in the under development of the field of SBST for non-functional properties (discussed in Section III). Certainly, many of the additional objectives that practising testers may seek to achieve are likely to concern non-functional properties. For example, a tester may be interested in achieving higher coverage, but while also targeting unusually long execution times, security properties, or energy consumption (or all of these). Since the community seems sluggish in its uptake of non-functional properties, this may have had a concomitant effect on applications of multi-objective techniques.

Fortunately, search based techniques are readily available for multi-objective optimisation. Since many different test adequacy criteria have been captured as fitness functions, all that remains is to consider how to combine these in multi-objective frameworks, methods and tools.

Multi-objective Understanding: Such multi-objective test data generation may not be confined merely to the revelation of faults; It may so be used at a more strategic level, to understand, investigate and highlight problems at the level of policy formulation. For example, there is a well-known tension between usability and security [1], two non-functional properties that we might also seek to measure and test.

In order to investigate this phenomenon and its practical ramifications for a particular security policy, we can capture user behaviour in a simple language that defines the strategies that a user might take to increase usability. Suppose we can measure usability properties. We can now search for user strategies that maximise usability (using a similar approach to SBTSI), thereby investigating the limitations and shortcomings of security policies that sacrifice usability for security.

Furthermore, this approach could be extended to help identify potential security policies. We can formulate the trade-off between usability and security using a multi-objective approach. If we have a language for defining security policies as well as a language for defining likely user behaviours, then we can co-evolve a security policy and user behaviour using co-evolution. In this co-evolutionary framework, the fitness of a security policy is defined by the security level achieved with respect to the population of user behaviours, while the fitness of the user behaviour strategy is defined by its ability to maximise usability with respect to the security policies. Using variations on this theme, we may be able to find security policies that are well adapted to particular user behaviours, thereby balancing usability and security.

The Path from Automated Testing to Automated Improvement: In this discussion we have moved relatively seamlessly from seeking to search for test cases, to using testing to discover improved systems. This is one of the principles that underlies the recent upsurge in work on Genetic Improvement (GI) [6], [50], [53], [54], [72], [73], [75], [96], [109], [121]; If we can search for test cases that expose suboptimal system behaviour, can we not also search for versions of the system that improve this behaviour? We believe that there is a symbiotic relationship between SBST and GI: SBST can generate test cases to help guide GI [53], but it also suggest intellectual routes through which we can make the technical and practical journey from automating testing to automating improvement.

VI. Find, Fix, Verify (FiFiVerify)

We are tantalisingly within sight of exciting future testing tools that we would like to outline in this section; tools that will find, fix and verify the systems to which they are applied. Such near-future software engineering tools will take a program that may contain bugs (from some identified bug class, such as memory faults) and return an improved program. The improved program has all bugs for the specified class fixed and is verified as being free from this class of faults. It may also come with a regression test suite that gives the engineer some degree of confidence that the improved system has not regressed and/or a proof that the improved version is 'no less correct' than the original.

We name this type of hypothesised tool a 'FiFiVerify tool' (short for 'Find, Fix and Verify'). Though any FiFiVerify tool would be giant leap forward from current testing and debugging technology, we believe that such tools are already *within the grasp* of the verification and testing community. In the remainder of this section, we outline the case that the techniques and algorithms required to build a FiFiVerify tool, are already available and reported in the literature.

Verification: Verification techniques are sufficiently mature that they can verify non-trivial systems free from memory faults, scaling to complete verification (with respect to a given property) of device drivers (thousands of Lines of Code) [123] and partial verification of much larger systems [22], [23]. Where there remain faults, we can use fault localisation [66], [127] to highlight likely 'suspicious' statements on which we can target automated repair [76].

A New Application for Fault Localisation: Fault localisation has known theoretical limits [128]. There has also been recent discussion of whether it offers real benefits to human programmers [93]. However, the practical concerns are pertinent only for applications to human debuggers; fault localisation definitely offers benefits to *automated* repair techniques [89]. We believe that automated repair may prove to be a much more profitable use-case for automated fault localisation, and we hope for more work on fault localisation specifically tailored to automated repair (and, more generally, genetic improvement).

Find and Fix: Combining this work on test generation, localisation and repair will allow us to find and fix bugs automatically. This will allow us to find and fix bugs (a FiFi tool).

FiFi *and* Verify: We can then alternate between find-and-fix and verification until the verification system is able to prove freedom from the class of faults of interest. This is a rather naive outline of a FiFiVerify tool. A more sophisticated approach would seek a more intimate combination of these technologies, so that testing can inform verification and vice versa, making each more efficient and effective. However, a simple iterative sequential composition would provide a proof of concept FiFiVerify tool.

FiFi and Verify Absence of Regression Faults: Finally, as outlined in Section II we have test data generation techniques that can achieve reasonable coverage, possibly augmented (or, where feasible, replaced by) verification [40], [81]. These can be used to help find the bugs, to guide the repair process and they could be used to provide a regression test suite. Since there is no oracle problem for regression testing [12], the regression testing also can be entirely automated.

VII. Conclusion

In this paper we have reviewed work on Search Based Software Testing, its origins, trends in publication and open problems. We showed that the area continues to grow, with a polynomial increase in publications, but there are causes for concern. We presented evidence that the range of different non-functional properties being attacked using SBST is rising, but the proportion of papers on this topic is falling, which is troubling, given the increasing importance of non-functional properties to testers. Specifically, we highlighted the lack of work on Search Based Energy Testing (SBET), outlining energy measurement techniques that might be reused as fitness functions and some of the issues involved.

We also argue the case for multi-objective software testing, since we believe that most testers will have more than one objective in mind when they search for a test suite. Although multi-objective techniques have penetrated the regression testing problem space, they have yet to make a significant impact in the area of software test data generation. We give some examples of open problems and possible opportunities for multi-objective test data generation.

We conclude with an upbeat assessment of the exciting possible SBSE tools that may appear in the near future, posing the FiFiVerify tool challenge. To qualify as a FiFiVerify tool, the tool must automatically find faults in a given class, fix them and verify that the faults had been fixed. We believe that rudimentary FiFiVerify tools are already within the current capabilities of the research community.

Acknowledgment

Mark Harman is partly supported by the EPSRC grants EP/J017515/1 (DAASE) and EP/I033688/1 (GISMO). Yuanyuan Zhang and Yue Jia are fully supported by the DAASE grant. The authors would like to thank Daniel Kroening, Bill Langdon, Phil McMinn, Peter O'Hearn, Matheus Paixão and Justyna Petke for many discussions on testing, verification and SBSE, that influenced some of the ideas presented here. We also thank Gordon Fraser, Darko Marinov and Franz Wotawa for the invitation to give this keynote at ICST 2015.

References

[1] Anne Adams and Martina Angela Sasse. Users are not the enemy. *Communications of the ACM*, 42(12):40–46, December 1999.

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

[3] Wasif Afzal, Richard Torkar, Robert Feldt, and Greger 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 September 2010.

[4] Shaukat Ali, Lionel C. Briand, Hadi Hemmati, and Rajwinder Kaur 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] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *28th International Conference on Software Engineering (ICSE '06)*, pages 361–370, New York, NY, USA, 2006. ACM.

[6] Andrea Arcuri, David Robert White, John A. Clark, and Xin 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.

[7] Nesa Asoudeh and Yvan Labiche. A multi-objective genetic algorithm for generating test suites from extended finite state machines. In *Proceedings of the 5th International Symposium on Search Based Software Engineering (SSBSE '13)*, volume 8084, St. Petersburg, Russia, 24-26 August 2013. Springer.

[8] Wesley Klewerton Guez Assuno, Thelma Elita Colanzi, Silvia Regina Vergilio, and Aurora Pozo. A multi-objective optimization approach for the integration and test order problem. *Information Sciences*, 267:119–139, May 2014.

[9] Arthur Baars, Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, Paolo Tonella, and Tanja Vos. Symbolic search-based testing. In 26^{th} *IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 53 – 62, Lawrence, Kansas, USA, 6th - 10th November 2011.

[10] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE'14, pages 588–598, New York, NY, USA, 2014. ACM.

[11] André Baresel, Harmen Sthamer, and Joachim Wegener. Applying evolutionary testing to search for critical defects. In *Proceedings of the 2004 Conference on Genetic and Evolutionary Computation (GECCO '04)*, volume 3103 of *Lecture Notes in Computer Science*, pages 1427–1428, Seattle, Washington, USA, 26-30 June 2004. Springer Berlin / Heidelberg.

[12] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 2015. To appear.

[13] Iain Bate and Usman Khan. WCET analysis of modern processors using multi-criteria optimisation. *Empirical Software Engineering*, 16(1):5–28, February 2011.

[14] Olga Baysal, Michael W. Godfrey, and Robin Cohen. A bug you like: A framework for automated assignment of bugs. In *International conference on program comprehension (ICPC '09)*, pages 297–298. IEEE Computer Society, 2009.

[15] Karel P. Bergmann and Jörg Denzinger. Testing of precision agricultural networks for adversary-induced problems. In *Proceeding of The 15th Annual Conference on Genetic and Evolutionary Computation (GECCO '13)*, pages 1421–1428, Amsterdam, The Netherlands, 6-10 July 2013. ACM.

[16] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT – a formal system for testing and debugging programs by symbolic execution. In *International Conference on Reliable Software*, pages 234–245, New York, NY, USA, 1975. ACM.

[17] Lionel Briand, Yvan Labiche, and Kathy Chen. A multi-objective genetic algorithm to rank state-based test cases. In *Proceedings of the 5th International Symposium on Search Based Software Engineering (SSBSE '13)*, volume 8084, pages 66–80, St. Petersburg, Russia, 24-26 August 2013. Springer.

[18] Lionel C. Briand, Jie Feng, and Yvan Labiche. Using genetic algorithms and coupling measures to devise optimal integration test orders. In *Software Engineering and Knowledge Engineering (SEKE 02)*, pages 43–50, 2002.

[19] Lionel C. Briand, Yvan Labiche, and Marwa Shousha. Using genetic algorithms for early schedulability analysis and stress testing in real-time systems. *Genetic Programming and Evolvable Machines*, 7(2):145–170, June 2006.

[20] Renée C. Bryce and Charles Joseph Colbourn. One-test-at-a-time heuristic search for interaction test suites. In *Proceedings of the 9th annual Conference on Genetic and Evolutionary Computation (GECCO '07)*, pages 1082–1089, London, England, 7-11 July 2007. ACM.

[21] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56(2):82–90, February 2013.

[22] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In Zhong Shao and Benjamin C. Pierce, editors, 36^{th} *Symposium on*

Principles of Programming Languages (POPL 2009), pages 289–300, Savannah, GA, USA, 2009. ACM.

[23] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *Journal of the ACM*, 58(6):26:1–26:66, 2011.

[24] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. An approach for QoS–aware service composition based on genetic algorithms. In Hans-Georg Beyer and Una-May O'Reilly, editors, *Genetic and Evolutionary Computation Conference, GECCO 2005, Proceedings, Washington DC, USA, June 25-29, 2005*, pages 1069–1075. ACM, 2005.

[25] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA '07)*, pages 129–139, London, United Kingdom, 9-12 July 2007. ACM.

[26] Thelma Elita Colanzi, Wesley Klewerton Guez Assuno, Silvia Regina Vergilio, and Aurora Trinidad Ramirez Pozo. Integration test of classes and aspects with a multi-evolutionary and coupling-based approach. In 3^{rd} *International Symposium on Search Based Software Engineering (SSBSE '11)*, Szeged, Hungary, 10-12 September 2011. Springer.

[27] Thelma Elita Colanzi, Silvia Regina Vergilio, Wesley Klewerton Guez Assuncao, and Aurora Pozo. Search based software engineering: Review and analysis of the field in Brazil. *Journal of Systems and Software*, 86(4):970–984, April 2013.

[28] H. David, E. Gorbatov, Ulf R. Hanebutte, R. Khanna, and C. Le. Rapl: Memory power estimation and capping. In *Proceedings of the 2010 ACM/IEEE International Symposium on Low-Power Electronics and Design*, ISLPED'10, pages 189–194, Aug 2010.

[29] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, May 1979. An earlier version appeared in *ACM Symposium on Principles of Programming Languages (POPL) , Los Angeles, California* , 1977 pp. 206–214.

[30] Karnig Agop Derderian. *Automated Test Sequence Generation for Finite State Machines using Genetic Algorithms*. PhD thesis, School of Information Systems, Computing and Mathematics, Brunel University, 2006.

[31] Edsger 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.

[32] Richard M. Everson and Jonathan E. Fieldsend. Multiobjective optimization of safety related systems: An application to short-term conflict alert. *IEEE Transactions on Evolutionary Computation*, 10(2):187–198, April 2006.

[33] Denzil Ferreira, Anind K. Dey, and Vassilis Kostakos. Understanding human-smartphone concerns: A study of battery life. In *9th International Conference on Pervasive Computing (Pervasive'11)*, pages 19–33, Berlin, Heidelberg, 2011. Springer-Verlag.

[34] Javier Ferrer, Francisco Chicano, and Enrique Alba. Evolutionary algorithms for the multi-objective test data generation problem. *Software: Practice and Experience*, 42(11):1331–1362, November 2012.

[35] James H. Fetzer. Program verification: the very idea. *Communications of the ACM*, 31(9):1048–1063, September 1988.

[36] Gordon Fraser and Andrea 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.

[37] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. In *International Symposium on Software Testing and Analysis (ISSTA 2010)*, pages 147–158, Trento, Italy, 2010. ACM.

[38] Fabrício G. Freitas and Jerffeson 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.

[39] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Comparing non-adequate test suites using coverage criteria. In Mauro Pezzè and Mark Harman, editors, *International Symposium on Software Testing and Analysis (ISSTA 2013)*, pages 302–313, Lugano, Switzerland, 2013. ACM.

[40] Benny Godlin and Ofer Strichman. Regression verification: proving the equivalence of similar programs. *Software Testing, Verification and Reliability*, 23(3):241–258, 2013.

[41] Concettina Del Grosso, Giuliano Antoniol, Massimiliano Di Penta, Philippe Galinier, and Ettore Merlo. Improving network applications security: a new heuristic to generate stress testing data. In *Cconference on Genetic and evolutionary computation (GECCO 2005)*, volume 1, pages 1037–1043, Washington DC, USA, 25-29 June 2005. Association for Computer Machinery.

[42] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. Estimating mobile application energy consumption using program analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE'13, pages 92–101, Piscataway, NJ, USA, 2013. IEEE Press.

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

[44] Mark Harman. Why source code analysis and manipulation will always be important (keynote). In 10^{th} *IEEE International Working Conference on Source Code Analysis and Manipulation*, Timisoara, Romania, 2010.

[45] Mark Harman. Making the case for MORTO: Multi objective regression test optimization. In 1^{st} *International Workshop on Regression Testing (Regression 2011)*, Berlin, Germany, March 2011.

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

[47] Mark Harman, John Clark, and Mel Ó Cinnédie. Dynamic adaptive search based software engineering needs fast approximate metrics (keynote paper). In 4^{th} *International Workshop on Emerging Trends in Software Metrics (WeTSOM 2013)*, San Francisco, USA, May 2013.

[48] Mark Harman, Yue Jia, Jens Krinke, Bill Langdon, Justyna Petke, and Yuanyuan Zhang. Search based software engineering for software product line engineering: a survey and directions for future work (keynote paper). In 18^{th} *International Software Product Line Conference (SPLC 14)*, pages 5–18, Florence, Italy, September 2014.

[49] Mark Harman, Yue Jia, and William B. Langdon. Strong higher order mutation-based test data generation. In 8^{th} *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '11)*, pages 212–222, New York, NY, USA, September 5th - 9th 2011. ACM.

[50] Mark Harman, Yue Jia, William B. Langdon, Justyna Petke, Iman Hemati Moghadam, Shin Yoo, and Fan Wu. Genetic improvement for adaptive software engineering (keynote). In 9^{th} *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2014)*, pages 1–4, New York, NY, USA, 2014. ACM.

[51] Mark Harman and Bryan F. Jones. Search based software engineering. *Information and Software Technology*, 43(14):833–839, December 2001.

[52] Mark Harman, Kiran Lakhotia, and Phil McMinn. A multi-objective approach to search-based test data generation. In 9^{th} *annual conference on Genetic and evolutionary computation (GECCO 2007)*, pages 1098 – 1105, London, UK, July 2007. ACM Press.

[53] Mark Harman, William B. Langdon, Yue Jia, David R. White, Andrea Arcuri, and John A. Clark. The GISMOE challenge: Constructing the pareto program surface using genetic programming to find better programs (keynote paper). In 27^{th} *IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*, pages 1–14, Essen, Germany, September 2012.

[54] Mark Harman, William B. Langdon, and Westley Weimer. Genetic programming for reverse engineering (keynote paper). In Rocco Oliveto and Romain Robbes, editors, 20^{th} *Working Conference on Reverse Engineering (WCRE 2013)*, Koblenz, Germany, 14-17 October 2013. IEEE.

[55] Mark Harman, Afshin Mansouri, and Yuanyuan Zhang. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. Technical Report TR-09-03, Department of Computer Science, King's College London, April 2009.

[56] Mark Harman, Afshin Mansouri, and Yuanyuan Zhang. Search based software engineering: Trends, techniques and applications. *ACM Computing Surveys*, 45(1):11:1–11:61, November 2012.

[57] Mark Harman and Phil 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.

[58] Mark Harman, Phil McMinn, Jerffeson Teixeira de Souza, and Shin Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In Bertrand Meyer and Martin Nordio, editors, *Empirical software engineering and verification: LASER 2009-2010*, pages 1–59. Springer, 2012. LNCS 7007.

[59] Rob Hierons, Kirill Bogdanov, Jonathan Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Luettgen, Tony Simons, Sergiy Vilkomir, Martin Woodward, and Hussein Zedan. Using formal methods to support testing. *ACM Computing Surveys*, 41(2), February 2009. Article 9.

[60] Charles Anthony Richard Hoare. The engineering of software: A startling contradiction. In David Gries, editor, *Programming Methodology, A Collection of Articles by Members of IFIP WG2.3*. Springer-Verlag, New York, NY, 1978.

[61] Charles Anthony Richard 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, March 1996.

[62] Timo Hönig, Christopher Eibel, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat. Seep: Exploiting symbolic execution for energy-aware programming. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, HotPower'11, pages 4:1–4:5, New York, NY, USA, 2011. ACM.

[63] Kobi Inkumsah and Tao Xie. Evacon: a framework for integrating evolutionary and concolic testing for object-oriented programs. In R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer 0002, editors, *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, pages 425–428. ACM, 2007.

[64] Yue Jia, Myra B. Cohen, Mark Harman, and Justyna Petke. Learning combinatorial interaction testing strategies using hyperheuristic search. Technical Report RN/13/17, University College London, 2013.

[65] Yue Jia and Mark Harman. Constructing subtle faults using higher order mutation testing. In *8th International Working Conference on Source Code Analysis and Manipulation (SCAM'08)*, pages 249–258, Beijing, China, 2008. IEEE Computer Society.

[66] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *24th International Conference on Software Engineering (ICSE '02)*, pages 467–477, New York, NY, USA, 2002. ACM.

[67] James Cornelius King. *A Program Verifier*. PhD thesis, Carnegie Mellon University, 1969.

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

[69] Kiran Lakhotia, Mark Harman, and Hamilton Gross. AUSTIN: An open source tool for search based software testing of C programs. *Journal of Information and Software Technology*, 55(1):112–125, January 2013.

[70] Kiran Lakhotia, Phil McMinn, and Mark Harman. Automated test data generation for coverage: Haven't we solved this problem yet? In 4^{th} *Testing Academia and Industry Conference — Practice And Research Techniques (TAIC PART'09)*, pages 95–104, Windsor, UK, 4th–6th September 2009.

[71] Kiran Lakhotia, Nikolai Tillmann, Mark Harman, and Jonathan 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.

[72] William B. Langdon and Mark Harman. Genetically improved CUDA C++ software. In 17^{th} *European Conference on Genetic Programming (EuroGP)*, Granada, Spain, April 2014. To Appear.

[73] William B. Langdon and Mark Harman. Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation (TEVC)*, 2014. To appear.

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

[75] Claire Le Goues, Stephanie Forrest, and Westley Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.

[76] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[77] Raluca Lefticaru and Florentin Ipate. Automatic state-based test generation using genetic algorithms. In *Proceedings of the 9th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC '07)*, pages 188–195, Timisoara, Romania, 26-29 September 2007. IEEE Computer Society.

[78] Per Kristian Lehre and Xin Yao. Runtime analysis of (1+1) EA on computing unique input output sequences. In *Proceedings of 2007 IEEE Congress on Evolutionary Computation (CEC '07)*, pages 1882–1889, Singapore, 25-28 September 2007. IEEE.

[79] Ding Li, Yuchen Jin, Cagri Sahin, James Clause, and William G. J. Halfond. Integrated energy-directed test suite optimization. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA'14, pages 339–350, New York, NY, USA, 2014. ACM.

[80] Ding Li, Angelica Huyen Tran, and William G. J. Halfond. Making web applications more energy efficient for OLED smartphones. In *36th International Conference on Software Engineering (ICSE 2014)*, pages 527–538, New York, NY, USA, 2014. ACM.

[81] Francesco Logozzo and Thomas Ball. Modular and verified automatic program repair. In *27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 133–146, 2012.

[82] Irene Manotas, Christian Bird, Lori Pollock, and James Clause. An empirical study of practitioners' perspectives on green software engineering. Technical Report 2014/003, University of Delaware, 2014.

[83] Irene Manotas, Lori Pollock, and James Clause. SEEDS: A software engineer's energy-optimization decision support framework. In *36th International Conference on Software Engineering*, pages 503–514, New York, NY, USA, 2014. ACM.

[84] Bogdan Marculescu, Robert Feldt, and Richard Torkar. Objective re-weighting to guide an interactive search based software testing system. In *Machine Learning and Applications (ICMLA)*, pages 102–107. IEEE, 2013.

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

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

[87] Siavash Mirarab, Soroush Akhlaghi, and Ladan Tahvildari. Size-constrained regression test case selection using multi-criteria optimization. *IEEE Transactions on Software Engineering*, 38(4):936–956, July-Aug 2012.

[88] I. M. Murwantara, B. Bordbar, and Leandro L. Minku. Measuring energy consumption for web service product configuration. In *16th International Conference on Information Integration and Web-based Applications & Services (iiWAS'14)*, December 2014. To appear.

[89] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: program repair via semantic analysis. In Betty H. C. Cheng and Klaus Pohl, editors, *35th International Conference on Software Engineering (ICSE 2013)*, pages 772–781, San Francisco, USA, May 18-26 2013. IEEE.

[90] Jungsup Oh, Mark Harman, and Shin Yoo. Transition coverage testing for Simulink/Stateflow models using messy genetic algorithms. In *Genetic Algorithms and Evolutionary COmputation Conference (GECCO 2011)*, pages 1851–1858, Dublin, Ireland, 2011.

[91] Rosemary O'Keeffe. The health benefits of stress. *The Boolean*, 2014:125–129, June 2014.

[92] Elmahdi Omar, Sudipto Ghosh, and Darrell Whitley. Comparing search techniques for finding subtle higher order mutants. In *Conference on Genetic and Evolutionary Computation (GECCO 2014)*, pages 1271–1278. ACM, 2014.

[93] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *International Symposium on Software Testing and Analysis (ISSTA '11)*, pages 199–209, New York, NY, USA, 2011. ACM.

[94] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys'12, pages 29–42, New York, NY, USA, 2012. ACM.

[95] Justyna Petke, Myra B. Cohen, Mark Harman, and Shin Yoo. Efficiency and early fault detection with lower and higher strength combinatorial

interaction testing. In *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13*, pages 26–36, Saint Petersburg, Russian Federation, August 2013. ACM.

[96] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. Using genetic improvement & code transplants to specialise a C++ program to a problem class. In 17^{th} *European Conference on Genetic Programming (EuroGP)*, Granada, Spain, April 2014. To Appear.

[97] Gustavo Pinto, Fernando Castor, and Yu David Liu. Understanding energy behaviors of thread management constructs. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA'14, pages 345–360, New York, NY, USA, 2014. ACM.

[98] Simon Poulding, Robert Alexander, John A. Clark, and Mark J. Hadley. The optimisation of stochastic grammars to enable cost-effective probabilistic structural testing. In *15th Annual Conference on Genetic and Evolutionary Computation (GECCO '13)*, pages 1477–1484, New York, NY, USA, 2013. ACM.

[99] Md. Mainur Rahman, Guenther Ruhe, and Thomas Zimmermann. Optimized assignment of developers for fixing bugs an initial evaluation for eclipse projects. In *3rd International Symposium on Empirical Software Engineering and Measurement (ESEM '09)*, pages 439–442, Washington, DC, USA, 2009. IEEE Computer Society.

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

[101] Jian Ren, Mark Harman, and Massimiliano Di Penta. Cooperative co-evolutionary optimization on software project staff assignments and job scheduling. In 3^{rd} *International Symposium on Search based Software Engineering (SSBSE 2011)*, pages 127–141, 10th - 12th September 2011. LNCS Volume 6956.

[102] Cagri Sahin, Philip Tornquist, Ryan McKenna, Zachary Pearson, and James Clause. How does code obfuscation impact energy usage? In *Proceedings of the 30th International Conference on Software Maintenance and Evolution*, ICSME'14, 2014.

[103] Richard L. Sauder. A general test data generator for COBOL. In *AFIPS Spring Joint Computer Conference*, pages 317–323, New York, NY, USA, 1962. ACM.

[104] Alan C. Schultz, John J. Grefenstette, and Kenneth A. De Jong. Test and evaluation by genetic algorithms. *IEEE Expert (also IEEE Intelligent Systems and Their Applications)*, 8(5):9–14, 1993.

[105] Jeffery Shelburg, Marouane Kessentini, and Daniel R. Tauritz. Regression testing for model transformations: A multi-objective approach. In *Proceedings of the 5th International Symposium on Search Based Software Engineering (SSBSE '13)*, volume 8084, pages 209–223, St. Petersburg, Russia, 24-26 August 2013. Springer.

[106] August Shi, ALex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. Balancing trade-offs in test-suite reduction. In *22nd International Symposium on Foundations of Software Engineering (FSE 2014)*, pages 246–256, New York, NY, USA, 2014. ACM.

[107] Digvijay Singh, Peter A. H. Peterson, Peter L. Reiher, and William J. Kaiser. The atom leap platform for energy–efficient embedded computing: Architecture, operation, and system implementation. Technical report, University of California, Los Angeles, December 2010.

[108] Pavel Somavat and Vinod Namboodiri. Energy consumption of personal computing including portable communication devices. *Journal of Green Engineering*, 1(4):447–475, 2011.

[109] Jerry Swan, Michael G. Epitropakis, and John R. Woodward. Gen-o-fix: An embeddable framework for dynamic adaptive genetic improvement programming. Technical Report CSM-195, Computing Science and Mathematics, University of Stirling, Stirling FK9 4LA, Scotland, January 2014.

[110] Nikolai Tillmann, Jonathan de Halleux, and Tao Xie. Transferring an automated test generation tool to practice: From Pex to Fakes and Code Digger. In *29th ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 385–396, 2014.

[111] Nikolai Tillmann, Jonathan de Halleux, and Tao Xie. Transferring an automated test generation tool to practice: From Pex to Fakes and Code Digger. In *ASE*, 2014.

[112] Vivek Tiwari, Sharad Malik, and Andrew Wolfe. Power analysis of embedded software: A first step towards software power minimization. In *Proceedings of the 1994 IEEE/ACM International Conference on Computer-aided Design*, ICCAD'94, pages 384–390, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[113] Marouane Tlili, Stefan Wappler, and Harmen Sthamer. Improving evolutionary real-time testing. In *Proceedings of the 8th annual Conference on Genetic and Evolutionary Computation (GECCO '06)*, pages 1917–1924, Seattle, Washington, USA, 8-12 July 2006. ACM.

[114] Nigel Tracey, John Clark, John McDermid, and Keith Mander. A search-based automated test-data generation framework for safety-critical systems. In Peter Henderson, editor, *Systems engineering for business process change: new directions*, pages 174–213. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[115] Alan M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, Cambridge, England, June 1949. University Mathematical Laboratory.

[116] Bryan Walsh. The surprisingly large energy footprint of the digital economy [UPDATE]. *Time Magazine*, August 14th 2013.

[117] Joachim Wegener and Oliver 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.

[118] Joachim Wegener and Oliver Bühler. Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system. In *Proceedings of the 2004 Conference on Genetic and Evolutionary Computation (GECCO '04)*, volume 3103 of *Lecture Notes in Computer Science*, pages 1400–1412, Seattle, Washington, USA, 26-30 June 2004. Springer Berlin / Heidelberg.

[119] Joachim Wegener and Frank Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, 21(3):241–268, November 2001.

[120] David R. White, John Clark, Jeremy Jacob, and Simon 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.

[121] David Robert White, Andrea Arcuri, and John A. Clark. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation (TEVC)*, 15(4):515–538, 2011.

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

[123] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In Aarti Gupta and Sharad Malik, editors, 20^{th} *International Conference on Computer Aided Verification (CAV 2008)*, volume 5123 of *Lecture Notes in Computer Science*, pages 385–398, Princeton, NJ, USA, 2008. Springer.

[124] Thaise Yano, Eliane Martins, and Fabiano Luis de Sousa. A multi-objective evolutionary algorithm to obtain test cases with variable lengths. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation (GECCO '11)*, pages 1875–1882, Dublin, Ireland, 12-16 July 2011. ACM.

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

[126] Shin Yoo and Mark Harman. Regression testing minimisation, selection and prioritisation: A survey. *Journal of Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[127] Shin Yoo, Mark Harman, and David Clark. Fault localization prioritization: Comparing information theoretic and coverage based approaches. *ACM Transactions on Software Engineering and Methodology*, 22(3 (Article 19)), July 2013.

[128] Shin Yoo, Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, and Mark Harman. No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist. Technical Report RN/14/14, University College London, 2014.

[129] Yuan Zhan and John A. Clark. A search-based framework for automatic testing of MATLAB/simulink models. *Journal of Systems and Software*, 81(2):262–285, February 2008.

[130] Yuanyuan Zhang, Mark Harman, and Afshin Mansouri. The SBSE repository: A repository and analysis of authors and research articles on search based software engineering. crest-web.cs.ucl.ac.uk/resources/sbse_repository/.