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, 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.

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 Menabrea), ‘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.

1This 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.
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 $X^2 + Y + 10Z - W \geq 5$ 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 time² 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’).

²The 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.

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 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.
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.

---

**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).

---

**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.
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.

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 484 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. Monotis 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.
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 SBTSI 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 $\alpha$ 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 $\alpha$.

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 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 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 that 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 FiF 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.
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 da Assunção, 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.


REFERENCES

[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, 36th Symposium on