Genetic Improvement of Software: a Comprehensive Survey

Justyna Petke, Saemundur O. Haraldsson, Mark Harman,
William B. Langdon, David R. White, and John R. Woodward

***Abstract*—Genetic improvement uses automated search to find improved versions of existing software. We present a comprehensive survey of this nascent field of research with a focus on the core papers in the area published between 1995 and 2015. We identified core publications including empirical studies, 96% of which use evolutionary algorithms (genetic programming in particular). Although we can trace the foundations of genetic improvement back to the origins of computer science itself, our analysis reveals a significant upsurge in activity since 2012. Genetic improvement has resulted in dramatic performance improvements for a diverse set of properties such as execution time, energy and memory consumption, as well as results for fixing and extending existing system functionality. Moreover, we present examples of research work that lies on the boundary between genetic improvement and other areas, such as program transformation, approximate computing, and software repair, with the intention of encouraging further exchange of ideas between researchers in these fields.**

***Index Terms*—genetic improvement, survey**

I. Introduction

Genetic improvement (GI) uses automated search in order to improve existing software. We present a comprehensive survey of GI, summarising its scientific origins, technical achievements, publication growth trends, software engineering domain coverage, representations and computational search techniques, and its relationship with other areas of source code analysis and manipulation. As our survey reveals, evolutionary computing is by far the most widespread computational search technique used in the literature, making genetic improvement a field at the intellectual intersection of evolutionary computation, software engineering, optimisation, and source code analysis and manipulation.

Recent work on GI has received notable awards, demonstrating its acceptance and success within the wider software engineering and evolutionary computation communities. For example, work on GI for software repair and specialisation won four 'Humies' [1], [2], [3], [4], [5], awarded for human-competitive results produced by genetic and evolutionary computation [6]. Several papers on genetic improvement also won distinguished paper awards [1], [5] and technical challenges [7]. GI has also been the subject of attention from the broadcast media, as well as popular developer magazines, websites and blogs [8], [9], [10], [11], demonstrating its influence and reach beyond the research community to the wider developer community and the public at large.

J. Petke, M. Harman, W.B. Langdon, and D.R. White are with University College London.
S.O. Haraldsson and J.R. Woodward are with the University of Stirling.

Our survey of 3132 distinct titles found, resulted in the identification of 66 core GI papers[1]. However, genetic improvement research draws on and potentially influences many other areas in software engineering and program analysis. Therefore, we also reviewed the relationship between GI and program synthesis, program transformation, parameter tuning, approximate computing, slicing, partial evaluation and other.

We identified the first distinct publication concerned primarily with genetic improvement from 1995, but we were careful to consider the full history of the field, its influences and origins, both before and since. For our review, we collected and considered all publications on genetic improvement from 1995 to 2015. Since GI is an emerging area that straddles many fields, it is neither sufficiently well-understood nor well-defined to support a Systematic Literature Review (SLR).

Our comprehensive survey provides the foundation for subsequent SLRs, which may use our survey to define scope and research questions and to help identify primary sources. Nevertheless, although our survey is not an SLR, we did use associated techniques to ensure that we systematically collected potentially relevant publications.

Our survey is the first comprehensive analysis of GI research, drawing together its many research strands, results and findings. As the survey reveals, the field of GI has witnessed a rapid recent rise in publications and interest, with more than half (59.70%) of the overall papers appearing in the last three years (2013-2015). This indicates both that the time is ripe for a survey, and that this rapidly growing discipline (and the wider research communities within which it resides) needs such a survey.

The survey is structured as follows: Section II describes the history of genetic improvement (GI); Section III describes the methodology used to gather core papers on GI; Section IV provides details on existing GI research covered in the core papers on the subject; Section V presents related work; Section VI concludes the survey.

II. History of Genetic Improvement

Genetic improvement draws on and develops research in a number of topics including program transformation (Section II-B), program synthesis (Section II-C), genetic programming (Section II-D), software testing (Section II-E) and search based software engineering (Section II-F).

[1]Criteria for classifying GI publications as *core* are presented in Section III in Table I. All 66 papers are presented in the supplemental material.

We provide a brief history of genetic improvement, tracing some of its primary origins and influences (Section II-A), before coming right up to date (Section II-G).

A. Before Electronic Computers

The first mention of *software optimisation* is due to Ada Augusta Lovelace [12], whose 1842 work[2] on the analytical engine is arguably among the most prescient pieces of science writing ever committed to paper. Reading it nearly 200 years later, it is abundantly clear that Ada was well-aware of the need to optimise programs, even though none had actually been executed at the time, and only one program (which she had written herself) had ever been constructed:

> "*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' [12].

Clearly, Ada was aware that programs fell into equivalence classes, and saw the possibility of optimising the choice of a program within an equivalence class. Of course, as is well known, the 'calculating engine' about which she was writing, the Analytical Engine, was not built during her lifetime, and it would be over a century before the first program was executed. Nevertheless, we can trace ideas about program manipulation and optimisation back to her observations, made in 1842.

B. Program Transformation

Two independent, yet interrelated, strands of research that addressed the need for such program manipulation grew up in the 1960s and 1970s: program synthesis and program transformation. Both sought to exploit the equivalence classes noted by Ada, but in different ways. While transformation sought to apply meaning-preserving transformations to refine an existing program, synthesis sought to construct new program code.

Program transformation has origins in early work on compilers, which used transformation to automatically transform computations into canonical minimised forms, for either space or time efficiency. In early pioneering work on Fortran compilation, Sheridan [13] made a clear distinction between general and specific transformations, applicable only to a particular program instance. Sheridan noted that the general transformations allowed an arbitrary program expression to be 'reshuffled into some different order without disturbing the algorithm'.

Throughout the 1960s and 1970s, researchers sought to understand the principles that allowed one syntactic representation to be transformed into another, while preserving semantic correctness, drawing heavily on the foundations laid by Church [14] in the early 1940s. Researchers sought to define the semantics of programming languages, thereby providing a sound mathematical foundation for the theory and practice of software development [15], [16].

A further decade would have to pass before general-purpose declarative language transformation systems started to appear [17], [18]. Increasingly sophisticated systems have been developed for general transformation, such as the Munich CIP system in the 1980s [19], partial evaluation systems, such as Tempo, in the 1990s [20], and the TXL transformation language and system in the 2000s [21].

C. Program Synthesis

By contrast with program transformation, program synthesis sought to construct new program code, such that the resulting program would be correct by construction [22]. One of the earliest implementations, initially constructed[3] in 1961, and used to report the results of experiments with program synthesis was the work on Simon's 'Heuristic compiler' [23].

Both early program synthesis systems and program transformation systems were developed from and inspired by work on the compilers of the day. Program synthesis has remained a topic of continued interest and development, throughout the 1970s [24], 1980s [25], [26] and 1990s [27], to the recent work on spreadsheet macro synthesis by Gulwani et al. [28].

Genetic improvement is closely related to both synthesis and transformation, yet it differs from each: unlike both program synthesis and program transformation, genetic improvement is not always guided by the motivation of correctness-by-construction. Frequently, software testing is used as an oracle for correct system behaviour. In this regard, genetic improvement draws on the rich heritage of genetic programming [29], which is also generally guided by software testing, and does not claim correctness-by-construction.

D. Genetic Programming

The first record of the proposal to evolve programs is probably that of Turing [30]. However, there was a gap of some thirty years before Forsyth [31] demonstrated the evolution of small programs represented as trees to perform classification of crime scene evidence for the UK Home Office. Although the idea of evolving programs, particularly Lisp programs, was current amongst Holland's students [32], it was not until his students organised the first Genetic Algorithms conference in Pittsburgh that Cramer [33] published evolved programs in two specially designed languages.

In 1988 Koza (also a PhD student of Holland) patented his invention of a GA for program evolution [34] and this was followed by publication in the International Joint Conference on Artificial Intelligence IJCAI-89 [35]. Koza followed this with 205 publications on genetic programming. (The name 'genetic programming' was coined by Goldberg, also a PhD student of Holland [36].) However, it is the series of four books by Koza, starting in 1993 [29] and the accompanying videos [37], that really established genetic programming and saw the enormous expansion of number of publications with the Genetic Programming Bibliography[4] passing 10,000 entries [38].

[2]The article was published in 1843 in London as a translation of Menabrae's 1842 paper 'with notes by the translator' and is available online: www.fourmilab.ch/babbage/sketch.html.

[3]Simon's paper [23] was published in 1963, but it was an extended version of an earlier RAND corporation technical report, which appeared in 1961.

[4]Genetic Programming Bibliography: http://www.cs.bham.ac.uk/∼wbl/biblio/

By 2016 there were nineteen GP books including several intended for students [39], [40], [41].

Excluding GI, genetic programming research and applications continue to be concentrated on predictive modelling, particularly data mining [42] and financial modelling [43]. Other active areas include evolving soft sensors (particularly in the chemical industry [44]), design [45] and image processing [46], finance and the chemical industry. Industrial take up includes bioinformatics [47], [48] and the steel industry [49]. GP can also be found in artistic endeavours [50], [51], [52]. It led to Draves' Electric Sheep screen saver [53]. Such distributed evolutionary AI ideas were in the background which led Reynolds to his Boids technique [54] for which he won an Oscar in 1998.

Genetic programming shares with program synthesis its aim of constructing a working program from scratch. Both traditional program synthesis and genetic programming are limited in the size of programs they can generate. GI usually starts from an existing program (rather more like program transformation, although it makes no claim to preserve 'perfect correctness'). Since genetic improvement's starting point can be an existing program of arbitrary size, GI can tackle much larger programs than either program synthesis or genetic programming. Like genetic programming, software testing is often relied upon to guide genetic improvement to the software variant.

E. Testing and Validation

Testing is important for genetic improvement, not only because it can be used as a guide for semantic faithfulness, but also because it is often used to assess the degree to which improvement has been achieved. Software testing research also has a long history, dating back to the 1940s, when Turing first delineated the role of 'program tester' from 'program developer' [55]. By the 1960s it had been realised that automation was essential to manage the scale of the software testing challenge, and the first automated test input generation systems started to appear [56]. Test data generation systems continue to be developed and improved throughout the 1970s [57], 1980s [58], and 1990s [59], [60].

More recently, there has been significant development (including several breakthroughs and dramatic advances) in many areas of testing, such as dynamic symbolic execution [61], search based software testing [62], and mutation testing [63].

Many researchers could be forgiven for believing that testing could *never* be sufficient to ensure faithfulness to the semantics of the original program. After all, it is widely believed that:

> "*The number of different inputs, i.e. the number of different computations for which the assertions claim to hold is so fantastically high, that the demonstration of correctness by sampling is completely out of the question. Program testing can be used to show the presence of bugs, but never to show their absence! Therefore, program correctness should be proved on account of the program text.*" [64]

This highly quotable aphorism of Dijkstra's became an 'article of faith' in an unfortunate battle between testing and verification that has only more recently abated [65], [66], [67]. It is undoubtedly true that testing can never show the absence of all bugs, but it is also highly questionable whether *any* approach to program correctness can now (or could ever) show the absence of *all* bugs. We already have techniques that can prove the absence of bugs *with respect to given assumptions* [68], [69], but testing will always have a role, if only to check whether such assumptions are reasonable.

Work on genetic improvement does not assume that only testing should be used. Indeed, the field is ripe for the incorporation of verification techniques to complement existing test-based approaches. Nevertheless, a great deal of progress has been achieved using testing alone, for both assessing faithfulness to the semantics to be retained, and also for measuring the degree of improvement achieved.

This raises the question as to how genetic improvement could be so successful, yet use a combination of techniques that would appear to be so wrongheaded from the point of view of such illustrious forbearers. The answer may lie in recent empirical results. These *empirical* results confound some of the widely-held long-established assumptions that were based on plausible inferences from the *theoretical* nature of programming and computation.

It seems reasonable to assume that the number of programs possible in a given language is so inconceivably large that genetic improvement could surely not hope to find solutions in the 'genetic material' of the existing program. The test input space is also, in the words of Dijkstra, "so fantastically high" that surely sampling inputs could never be sufficient to capture static truths about computation. Recent empirical results challenge both of these assumptions.

Gabel and Su [70] found that naturally occurring code (as opposed to the space of theoretically-constructable programs) is surprisingly repetitive. A programmer would have to write more than six lines of code in order to create an original code fragment not already located somewhere in sourceforge. This is an important observation, in the context of genetic improvement, because many of the interventions exploited by genetic improvement consisted of fewer than six lines of code; they are patches, fixes and minor modifications.

Barr et al. [71] found that 43% of commits to a large repository of Java projects could be reconstituted from existing code. This suggests that a surprising number of changes made by humans to software systems could already be fabricated by genetic improvement, or similar techniques that reuse existing code as 'mere genetic material' to be manipulated.

These two studies provided empirical evidence that, although the theoretical space of programs is extraordinarily large, the practical space inhabited by human-developed code is far more constrained, making it potentially more amenable to genetic improvement than might be supposed from a purely theoretical standpoint.

F. Search Based Software Engineering

Work on automatic inference of statically correct assertions [72], has demonstrated that static truth about program computation (in the form of assertions that hold for all

executions), can be inferred from a surprisingly small sample of input-output pairs, in a surprisingly large number of cases. The observation that a small amount of dynamic information can yield static truth has also been found in other software engineering domains [73].

Early automated testing systems [56] formulated test data generation as a search problem within the search space of possible inputs to the program under test. This led to the first application of automated search to software engineering problems [57], [74]. The application of automated search to software engineering problems was taken up only patchily and with arguably less interest than other engineering disciplines. In 2001 the term 'Search Based Software Engineering' was coined by Harman and Jones, in a manifesto for the application of automated search to problems in Software Engineering [75]:

> "*The thesis underpinning the present paper is that search-based metaheuristic optimisation techniques are highly applicable to Software Engineering and that their investigation and application to Software Engineering is long overdue. It is time for Software Engineering to catch up with its more mature counterparts in traditional fields of engineering.* "

There had been notable contributions to the field that came to be known as SBSE, before the term 'SBSE' itself was first coined. However, this [75] was the first paper to advocate a discipline of 'Search Based Software Engineering'. The 'pre-SBSE work on SBSE' attacked problems in software project management [76], [77] software testing [78], [79] and, perhaps most relevant to this survey, novel forms of GP for software engineering problems [80], [81]. We can trace back some of the early ideas associated with genetic improvement to the work by Feldt [81].

Improving software with several objectives in mind is very powerful. Indeed evolutionary computing is well-suited to finding good trade-offs between potentially competing objectives, particularly where there are many objectives. For example, Lakhotia et al. [82] were the first to use multi-objective optimisation for test data generation, while Kalboussi et al. [83] considered seven objectives when generating test cases. Similarly both Mkaouer et al. [84] and Ramirez et al. [85] considered the conflict between objectives in software maintenance when re-organising Java source code.

The years since 2001 have witnessed an upsurge in SBSE activity, with many problems in Software Engineering submitting to solutions grounded in the SBSE approach. There are now surveys on many subareas of SBSE activity, including requirements [86], predictive modelling [87], [88] software project management [89], design [90], testing [62], [91], [92], software product lines [93], and repair [94], and other evidence of its increasing maturity as a discipline within software engineering [95]. However, hitherto, there has been no survey of the area of genetic improvement, which seeks to apply the SBSE approach to software systems' source code itself. This survey seeks to address this gap in the literature.

G. Genetic Improvement and the Way Ahead

Although genetic improvement has a lineage that traces back to program synthesis, genetic programming and program transformation, it is only more recently that it has emerged as an area of research in its own right. This emergence dates back to the early 1990s with the work by Ryan and Walsh [96], [97] on auto-parallelisation and the more recent work of White et al. [98] on energy improvement. It gained impetus with the work on automated repair [94], [99]. The term 'genetic improvement' emerged from a number of previous studies [100], [101], [102], all of which shared similar goals but with slightly different terminology (such as 'evolutionary improvement' and 'genetic improvement of programs').

A natural question arises, why has genetic improvement emerged only relatively *recently* as a separate research area? The key in answering this question lies in the components of genetic improvement, the necessary ingredients for which have only recently come together in sufficiently mature areas of activity that make genetic improvement possible. In particular, powerful test data generation techniques, an abundance of source code publicly available, and importance of non-functional properties have combined to create a technical and scientific environment ripe for the exploitation of genetic improvement.

Over most of the preceding years, software developers have been concerned with program correctness. A lot of work has been devoted to semantics-preserving program transformations that were supposed to serve as building blocks for automatic program synthesis. Secondly, one widely-discussed 'stretch challenge' has concerned the creation of software from scratch, perhaps from some higher level specification. This is so much of a difficult challenge that many influential authors regarded it to be simply unachievable. In 1988 Dijkstra claimed that automated programming was a contradiction:

> "(...) *computing science is — and will always be — concerned with the interplay between mechanized and human symbol manipulation, usually referred to as 'computing' and 'programming' respectively. An immediate benefit of this insight is that it reveals "automatic programming" as a contradiction in terms*." [103]

The more recent trend of genetic improvement research has not sought *entirely* automatic programming, but has considerably pushed back the frontiers of the 'interplay' referred to by Dijkstra, yielding to the machine, a great deal of territory previously occupied by humans. With the abundance of software available for 'genetic reuse', synthesis from scratch seems increasingly suboptimal. Furthermore, the increasing sophistication and power of automated test input generation, the ability to use the original program as an oracle, and the increasing importance of non-functional properties, have all combined to make genetic improvement a timely approach to automated software improvement.

Given the rich heritage on which genetic improvement draws, it is likely that we will see hybrids emerging in future, which draw on aspects of program transformation, program synthesis, genetic programming, and other source code analysis and manipulation [104] techniques. Indeed, recent work on automated program repair, a form of genetic improvement, also uses a combination of techniques including those inspired by program synthesis [105] and by genetic programming [2].

III. Survey Methodology

Genetic improvement draws on other research areas and has only recently emerged as an independent field of research, as presented in Section II. It uses automated search to navigate the three-dimensional search space consisting of the amount of improvement (over the original code), use of existing software (in the input to the improvement framework) and preserved functionality of the original code. Several works that lie on the far ends of this spectrum sit on the boundary between genetic improvement and other research areas.

Mrazek et al. [106], for instance, used cartesian genetic programming to optimise for efficiency and energy consumption. They evaluated approximations of 9-input and 25-input median functions by means of testing, as is typical in genetic improvement work. However, they evolved the functions from scratch by generating the initial population at random.

Kocsis et al. [107], [108] and Burles et al. [109] proposed to use semantics-preserving transformations within their improvement framework in order to retain full functionality of the original code. Therefore, this work also fits within the field of *program transformation* [110], [111] (see Section V-C for details). Similarly, Orlov [102] and Orlov & Sipper [112], [113] improved extant software by applying a semantics-preserving crossover operator.

Williams [114] used six evolutionary algorithms to parallelise existing code. In contrast to Walsh & Ryan [115], who used standard GP trees, they evolved sequences of semantics-preserving transformations. Williams [114] used knowledge from program dataflow and dependency analysis to avoid transformations that break functionality. Where the satisfiability of the derived constraints could not be proven, they took the conservative approach of assuming that the program will not be functionally equivalent to the original if the sequence of transformations were to be performed.

Even if code changes within a genetic improvement framework are restricted to semantics-preserving transformations, the search space of possible software variants is still huge [116]. Therefore, metaheuristics have typically been applied in order to find optimal or near-optimal solutions. Non-standard approaches to genetic improvement involve the use of deterministic search. Sidiroglou-Douskos et al. [117], for instance, explored the combination space of *tunable loops* (whose perforation leads to an acceptable efficiency-accuracy trade-off) with exhaustive and greedy search algorithms. Tan et al. [118] tried applying all their statement-level mutation operators until either all test cases passed or a timeout was reached. Mechtaev et al. [119] systematically derived candidate software repairs from a set of constraints, while Manotas et al. [120] performed exhaustive search over small code-level changes for improvement of energy consumption.

Given the wide range of topics that fall within the definition of genetic improvement, we restrict our detailed analysis to work that is most frequently associated with this new research area. For example, although semantics-preserving methods are extremely interesting (e.g., see work above), the boundary between program transformation and GI is often unclear. We identified four criteria under which we consider a publication to be a *core* genetic improvement paper. These criteria are shown in Table I. We also include position and overview papers on the subject.

TABLE I: **Scope of the Survey:** criteria used to identify core papers on genetic improvement (i.e., conference and workshop papers, journal articles and PhD theses) published by the end of 2015:

1) metaheuristic search is used;
2) non-semantics-preserving software variants can be produced during search;
3) existing software is reused as input to the given improvement framework;
4) modified software is improved over existing software with respect to the given criterion.

In order to provide a thorough overview of core papers on genetic improvement, we devised a rigorous procedure when searching for relevant publications. We searched the Collection of Computer Science Bibliographies [121] and the online libraries of four major publishers in software engineering, that is, ACM (ACM Digital Library [122]), IEEE (IEEE Xplore [123]), Springer (SpringerLink [124]) and Elsevier (ScienceDirect [125]). We used the following exact phrases as keywords: 'genetic improvement', 'software improvement' and 'evolutionary improvement'. We considered conference and workshop papers, journal articles and PhD theses that were published by the end of 2015. We call this step the *primary search*.

Table II presents results of the primary search. It is split into three parts, based on the keywords used. The first column contains the source of the publication found, the second column shows the filters applied, the third column shows the total number of publications found, using the keyword and filters provided, while the last column shows the number of publications on genetic improvement found based on paper title, abstract or keywords. All the searches were conducted independently, hence there was an overlap in the publications found. After removing duplicates, we were left with 54 publications, 40 of which cover work fulfilling all four criteria presented in Table I, based on subsequent manual inspection of their full text.

Subsequently, we inspected bibliographies of selected papers in order to include other publications that we deemed relevant according to the criteria (snowballing). Given that we selected 40 papers in our primary search, we had to look at an estimated number of over 1000 articles appearing in their bibliographies. In order to aid this time-consuming manual process, we devised an automated procedure to partially remove duplicates (i.e., papers we had already considered in previous searches) using pattern matching on paper titles. Table III ('Step 2') shows, for instance, that we found 862 new titles in bibliographies of the 40 publications. We then selected 34 that meet the criteria (based on abstract, title and keywords) and inspected the bibliographies of those 34 papers ('Step 3'). We repeated this procedure for the selected papers until we reached transitive closure over all references of the

TABLE II: Results of primary search for papers on genetic improvement.

Source	Filters	Papers found	Papers on GI
keyword	'genetic improvement'		
ACM Digital Library	Title OR Abstract	28	12
IEEE Xplore	Metadata	12	4
SpringerLink	Full Text Computer Science language: English	69	11
ScienceDirect	Title OR Abstract OR Keywords Computer Science	5	0
Collection Of Computer Science Bilbliographies	Default	165	41
keyword	'evolutionary improvement'		
ACM Digital Library	Title OR Abstract	5	1
IEEE Xplore	Metadata	21	1
SpringerLink	Full Text Computer Science language: English	133	4
ScienceDirect	Title OR Abstract OR Keywords Computer Science	3	0
Collection Of Computer Science Bilbliographies	Default	32	1
keyword	'software improvement'		
ACM Digital Library	Title OR Abstract	45	0
IEEE Xplore	Metadata	83	0
SpringerLink	Full Text Computer Science language: English	421	5
ScienceDirect	Title OR Abstract OR Keywords Computer Science	9	0
Collection Of Computer Science Bilbliographies	Default	100	2
Total		1131	82
Distinct papers on GI found			54
Distinct core papers on GI found			**40**

set of papers covering core GI work (i.e., fulfilling the four criteria shown in Table I), that is, until we did not find any new relevant papers in the bibliographies of selected papers. A summary of this process is shown in Table III.

Given that the primary search was conducted before the end of 2015, we then repeated the primary search step on 3 May 2016, to make sure we include every conference, workshop, journal and thesis publication that was published in 2015. This step revealed one additional core publication on genetic improvement [126], references of which did not contain any additional core publications on genetic improvement ('Secondary' search step in Table III).

TABLE III: Summary of searches conducted over bibliographies of core papers on genetic improvement.

Search step	New titles found	Core papers on GI
Primary	-	40
Step 2	862	34
Step 3	279	27
Step 4	47	8
Step 5	10	0
Secondary	-	1
Step 6	9	0
Total (based on abstract, title or keywords)		110
Distinct core papers on GI found (based on manual inspection of the full text of the 110 selected papers)		66

As a final step for the bibliography search stage, we manually checked, by inspecting the full text of each paper, that there are no duplicates among the selected papers and that each covers genetic improvement work fulfilling the four criteria shown in Table I. After this filtering process, we were left with 66 core papers on genetic improvement. Overall results of all the searches conducted are shown in Table IV.

TABLE IV: Summary of all searches conducted to identify core papers on genetic improvement.

Source	New papers found
Primary search	40
Bibliography search (snowballing)	26
Core papers on GI	**66**

The supplemental material contains several details about the core publications. In particular, for each empirical study that uses a genetic improvement framework, we identify: the improvement criterion, search technique, software representation, characteristics of the fitness function and programming language of the software being modified. In the following section we list several publications on genetic improvement that give an overview of specific GI work.

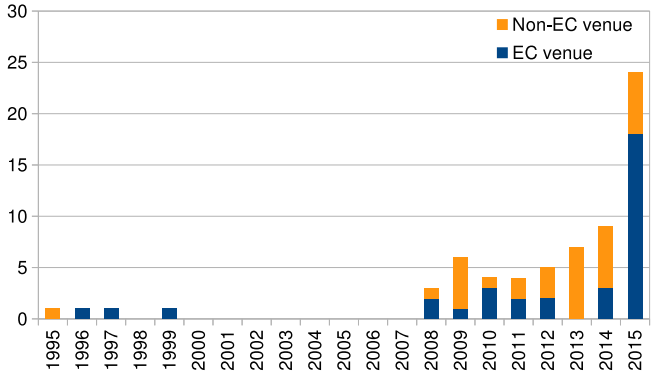
IV. Existing Work on Genetic Improvement

The need for automated software optimisation has long been recognised and resulted in the development of various research areas, such as program transformation and program slicing (see Section II for details). What differentiates genetic improvement from previous approaches, is its *generality* and *adaptability*. It takes advantage of the abundance of source code available by reusing it rather than devising a new optimisation from scratch. Furthermore, genetic improvement opens up a wider search space of software variants by relaxing restrictions on program correctness.

The oldest core GI publications concern with software parallelisation by Ryan and Walsh [96]. A resurgence of literature in the area can be seen in the late 2000s with Arcuri and Yao's work on automated software repair [99], [127] and White et al.'s work on reduction of energy consumption [98], [128]. Success of these studies has led to a rapid uptake of GI. This trend can be observed in the significant increase of

the number of core publications on genetic improvement since 2008, as shown in Figure 1.

Fig. 1: Number of core papers on genetic improvement by year.



The following sections describe the typical genetic improvement process in detail, drawing from the core papers on the subject (listed in the supplemental material).

A. Preserved Properties

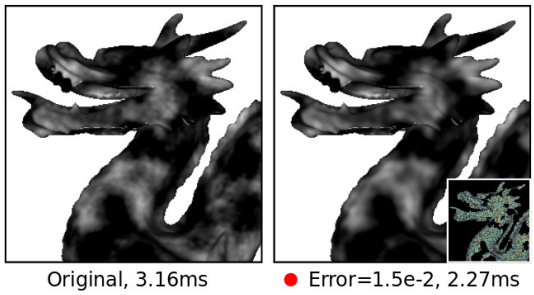
Improvement of software naturally implies that some aspects change (to improve), while others remain unchanged (otherwise it would be entirely different, not merely improved). For instance, a system may become faster through GI, while offering the same behaviour, or a bug may be fixed while retaining existing non-buggy functionality. Therefore, to assess the unchanged functional aspects, we need a way of capturing software functionality that needs to be preserved.

One way of ensuring that the modified software does not break any functionality of the original program, is to use only semantics-preserving transformations. However, this limits the search space of possible program modifications. Furthermore, this approach might still produce incorrect programs. Orlov & Sipper [113], for instance, used genetic programming with a semantics-preserving crossover to improve existing Java bytecode. Nevertheless, they also encounter incorrect individuals during the evolution process:

> *Compatible bytecode crossover prevents verification errors in offspring, in other words, all offspring compile sans error. As with any other evolutionary method, however, it does not prevent production of non-viable offspring – in our case, runtime errors. An exception or a timeout can still occur during an individual's evaluation, and the fitness of the individual should be reset accordingly.*

Relaxing restrictions on functional faithfulness to the original program allows for a trade-off between various software properties. Sitthi-amorn et al. [129], for instance, traded efficiency for accuracy in pixel shaders. They achieved a 67% reduction in runtime by allowing flexibility in image fidelity with respect to the output of the original software. An example is shown in Figure 2.

Fig. 2: Selected result of two variants of shader simplification software: original (left) and GI-modified (right). The inset contains a visualisation of the per-pixel error.



Trading various software properties may prove beneficial especially in resource-constrained environments. As our co-author (WBL) put it: "there's nothing correct about a flat battery".

The question remains of how to capture software properties that need to be retained? Several core papers on genetic improvement describe work on software transplantation, where a feature is evolved separately from the program to which it is later grafted. Langdon & Harman [130], for instance, evolve, i.e. *grow*, a parallelised version of a part of an existing program, called pknotsRG, used for predicting the minimum binding energy for folding of RNA molecules. They start with the existing program and use a combination of manual changes to the host code and GI. The GI *grows* a small piece of new CUDA code. After inserting, i.e. *grafting*, it into the original code, the GI-improved pknotsRG version achieves up to a 10000-fold speedup on certain test instances. Even in this *grow and graft* approach Langdon & Harman [130] needed to capture the properties of the feature evolved that needed to be preserved. Testing was used for this purpose.

In all empirical work that is covered by the core papers on genetic improvement, software testing was used as a proxy for capturing software properties that needed to be retained (see supplemental material). If the set of test cases is all possible test cases, then test equivalence becomes functional equivalence. Of course, such a test set could be conceptually infinite. Relaxing the notion of equivalence, to allow finite test suites to be used as the faithfulness criterion with respect to the original software, has the important technical implication that equivalence becomes computable and tractable. Furthermore, the runtime cost of testing can be reduced by the application of test case selection and prioritisation techniques. Fast et al. [131] and Qi et al. [132] investigated these issues in the context of GI-based automated software repair.

In the simplest case the number of test cases passed serves as a fitness measure. Barr et al. [1] and Marginean et al. [133] use the following function in their μScalpel tool for automated software transplantation:

$$fitness(i) = \begin{cases} 1/3 \times (1 \ + \ |TX_i|/|T| \ + |TP_i \ |T|), \ i \in I_C \\ 0, \ i \notin I_C \end{cases}$$

where i represents the software variant; I_C is the set of compilable programs; test suite T captures the desired functionality

to be transplanted; TX_i and TP_i are the sets of non-crashing and passing test cases respectively. In this case $fitness(i) = 1$ assumes that i preserves all the required functionality.

GenProg [3], a popular tool for automated software repair, also uses a simple weighting scheme in fitness evaluation of modified programs [134]:

$$fitness(C) = W_{PosT} \times |\{t \in PosT \mid P' \ passes \ t\}| + W_{NegT} \times |\{t \in NegT \mid P' \ passes \ t\}|$$

where C stands for the candidate patch, i.e. set of modifications, that produce program P' when applied to the original buggy software; W_{PosT} assigns a weight to the positive test cases, i.e., those that the original program passes; while W_{NegT} is the weight assigned to the number of test cases that fail when run on the original program, but pass when run on P'. Negative tests are weighted twice as heavily as the positive tests. The positive test cases are intended to capture the program functionality that needs to be preserved.

Arcuri & Yao [99] opted for a more fine-grained fitness measure for automated software repair. In particular, they use a *distance function* based on formal software specification. It measures how far the output of the modified software is from the expected result. Arcuri [127] implemented a similar metric within his Java Automatic Fault Fixer (JAFF) tool. He also used the underlying framework of JAFF to improve the efficiency of a triangle classification program. In this case a set of test cases satisfying the branch coverage criterion was exercised to establish whether the modified software preserved the desired behaviour. Arcuri [127] generated the required test cases automatically, showing another advantage of using testing as a means of capturing software functionality.

Genetic improvement typically modifies *existing* software, therefore, the original program serves as an oracle when testing improved software variants. Any software test generation technique can be used to create test inputs. The output of running tests on the original and modified software can then be compared to guide search towards fitter individuals. The size of a test suite capturing the desired software behaviour is thus potentially infinite. Arcuri & Yao [99], [135] introduced the idea of co-evolving test cases within a genetic improvement framework. They use a genetic algorithm, generating unit tests that pass when run on the original and fail on the modified programs. The same approach was later adapted by Wilkerson & Tauritz [136] who created CASC, a variant of Arcuri & Yao's [99] framework for C++ programs. Aside from choosing a different target programming language, Wilkerson & Tauritz [136] require the CASC user to provide the fitness function, in contrast to the work of Arcuri & Yao.

It is yet unclear how to characterise test suites that would best guide search towards improved software variants. Smith et al. [137] conducted an initial investigation in the field of automated software repair. They conclude that "the quality of the patches is proportional to the coverage of the test suite used during repair". They also advocate that further research is needed to fully understand the characteristics of an appropriate test suite for automated software repair. Fast et al. [131] proposed to use dynamic program invariants, i.e. predicates, with testing to evaluate candidate programs for automated software repair. Their proposed approach leads to more precise fitness values than the traditional weighted sum approach. Similar studies have not been conducted in the context of improvement of other software properties.

Hitherto, in the literature, testing and semantics-preserving transformations have been applied to capture program behaviour. Genetic improvement is a generalist framework and thus allows for other approaches to be explored.

B. Use of existing software

The power of genetic improvement lies in its applicability to a plethora of real-world software systems. Typically, GI does not start from scratch. All work covered by the core papers starts from an existing system [138].

In the evolutionary computation field existing software reuse corresponds to 'genetic transfer'.

1) Source of Genetic Material for Genetic Improvement: The idea of using existing code is central to the genetic improvement process. The plastic surgery hypothesis [71] assumes that the content of new code can often be assembled out of fragments of code that already exist. Barr et al. [71] investigated this hypothesis, showing that changes are 43% graftable from the exact version of the software being changed.

Within the selected core papers on genetic improvement one can find three options for the choice of code for software reuse: the program being improved, a different program written in the same language and a piece of code generated from scratch. A currently unexplored option is an import from a different programming language than the software to be improved.

2) Code Transplants: Another area of genetic improvement when it comes to its software reuse component arises from the work on software transplantation. Harman et al. [139] set out a vision for automated software transplantation in their keynote, where they presented an overview of practices and ideas from GI and GP that are applicable for reverse engineering.

Petke et al. [4] were the first to use the concept of code transplants [139] in the GI context. In particular, they use multiple software variants of the same program, namely MiniSAT, a Boolean satisfiability (SAT) solver. Code for software reuse was taken from the MiniSAT-hack track competition specifically designed to encourage SAT practitioners to submit their *manually-modified* versions of the solver. GI-improved variants achieved up to 17% speedup.

Barr et al. [1] took this work further and programmatically set up a scaffolding mechanism in which genetic programming can transform a software feature from one system to be transplanted into another system. They automatically extract a feature from the *donor* program (source) and transplant it into the *host* program (target). Several experiments were conducted demonstrating feasibility of the approach, including a real-world example where a particular video codec was transplanted into the popular VLC media player.

Marginean et al. [133] applied the same transplantation technique to transfer a call graph visualisation feature from the CFLOW program into the KATE text editor. Moreover, Sidiroglou-Douskos et al. [140] developed a systematic transplantation approach for automated software repair, reusing

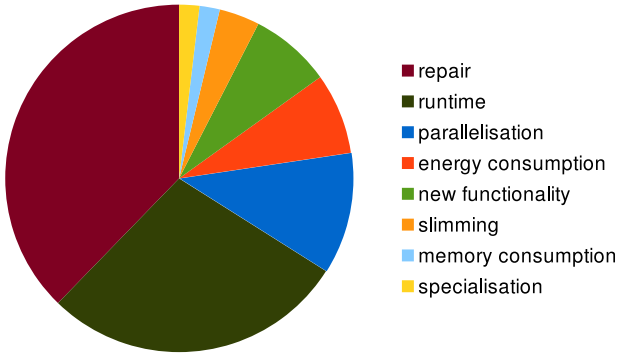
fixes available in open source projects. This approach has yet to be tried with metaheuristic search.

In the absence of the functionality of interest in existing software, a few researchers used genetic programming to evolve the desired feature from scratch. Harman et al. [7] evolved a language translation feature and transplanted it into a popular instant messaging system. Jia et al. [141] grew a citation service and grafted it into a web development framework; the evolved service is available online (see [141]). Langdon and Harman [130] evolved an enhanced parallel feature which gave up to 10000-fold speedup in the original software when run with the evolved feature.

C. Criteria for improvement

Criteria for software improvement can be divided into functional and non-functional. Ryan et al. [96], [115], [142], [143] and Williams & Williams [114], [144] used evolutionary algorithms to parallelise software. White's 2009 PhD thesis [128] focused on energy consumption reduction and both White and Arcuri [127] advocated the use of GI for improvement of non-functional properties of software. The importance of this strand of research was re-emphasized in Harman et al.'s keynote in 2012 [101]. Since then there has been a rapid increase in work on GI and, as shown in Figure 3, much of this work has focused on non-functional properties.

Fig. 3: Software applications of empirical studies in core papers on genetic improvement.



A major difficulty in non-functional software property optimisation lies in the measurement of the desired property. For non-functional properties such as energy consumption, precise measurements might simply be infeasible. However, these are not needed for the GI approach; we only need relative not absolute accuracy. GI only requires a *fitness function* that will guide search towards desirable software variants.

1) Testing as a fitness measure: Most of the genetic improvement work covered in core GI papers relies on testing to evaluate the fitness of candidate software variants. The number of test cases passed has been the prevalent measure in optimisation of functional properties of software systems. In particular, Shulte et al. [145], Arcuri et al. [99], [146], Wilkerson & Tauritz [136], Forrest et al. [2] and Le Goues et al. [3] equated successful software improvement with passing test suites in their work on automated program repair.

Ryan et al. [96], [115], [142], [143] and Williams & Williams [114], [144] used the number of test cases passed in their work on software parallelisation.

A lot of the core GI literature is concerned with automated bug fixing (see Figure 3), improving the correctness of programs as measured by testing. The largest body of work revolves around the tool called GenProg [147], either directly contributing to its development [2], [3], [5], [94], [134], [148], [149], [150] or using it for comparison [151], [152], [153]. Other examples of program repair with GI include Arcuri et al.'s pioneering work on small programs [99], [154], [155] which predates GenProg, as well as Schulte et al.'s [145], [156], [157], [158] work where the fixing was done post-compilation on the assembly code. Wilkerson et al. [159] compared his multi-objective approach to program repair with Arcuri's work. Ackling et al. [160] repaired software written in the Python language with same principles as GenProg.

Apart from the obvious risk of creating new bugs there is also the danger of introducing malevolent behaviour [156]. Schulte et al. used sandboxing in their work and ran tests on a virtual machine to ensure no damage is done by the altered software [145], [156], [157].

Most GI bug fixing work assumes that the faulty program contains its own potential fixes [148] and assumes freedom from typographical errors and incorrect variable names [160].

2) Other fitness functions: Most fitness functions that do not count passing and (or) failing test cases measure some non-functional property of software. These are dependent on hardware and some of them can be handled in part by compilers, such as memory usage. Although memory optimisation has been studied before [161], the core papers present only a single example of memory usage optimisation with GI. Wu et al. [162] explored how variables and constants in the source code can be exposed as parameters that can be tuned for the purpose of improving memory efficiency and reducing execution time.

The most frequently improved non-functional property is execution time. It, however, varies between systems and hardware. The number of lines or instructions executed per input has been considered as a system-independent proxy for execution time [163], [164], [165]. Langdon et al. [130], [166], [167], [168], [169], [170] report wall clock speedups.

Genetic improvement can also be applied per system and hardware by specialising to program classes [4] or input distributions [100], [128]. Recent advances in the mobile device market have seen greater computational power with increased drain on batteries. Hardware can only be optimised to a certain extent so software must also be adapted.

An issue with energy optimisation is how the usage is measured [171]. GI is, however, well-equipped to deal with noisy measures such as energy consumption [172]. A number of core papers on GI show promising results for energy optimisation [109], [128], [173], [174] using various methods to approximate its consumption. White [128] used simulation and a linear model to assess the energy consumption while Bruce et al. [174] used the Intel Power Gadget API to approximate the usage. For both methods the software is run in isolation to reduce noise in energy readings due to other processes. Although Schulte et al. [173] opted to use hardware

counters and a model of energy usage to approximate energy consumption. Similarly, Burles et al. [109] used an energy model that is specifically made for Java bytecode to reduce energy consumption of a function in Google's Guava library.

Given the difficulty of providing a correct energy measure for fitness evaluation, Harman and Petke [175] proposed to use GI to evolve the fitness function itself for a subsequent GI process. This idea generalises to any non-functional (or functional) property of software. Moreover, Johnson and Woodward [176] proposed to measure the fitness gain, rather than a total fitness measure, in terms of the accumulated information at each executed step of the program. GI was also proposed for product line engineering [177].

3) Multi-objective improvement: Optimisation of non-functional properties might sometimes mean degradation of other software properties. These can be either functional or non-functional. One might, for example, significantly reduce runtime by deleting certain software functionality, or reduce memory consumption at the expense of increased execution time. Given the many conflicting improvement criteria, Arcuri [127], White et al. [100], [128] and Harman et al. [101] suggested the application of multi-objective algorithms. Wu et al. [162] applied this approach to optimise both runtime and memory consumption. However, multi-objective genetic improvement of several non-functional properties is still largely unexplored.

D. Search

The power of genetic improvement lies in automatically evaluating multiple software versions in order to find ones that satisfy the improvement criteria and preserve the desired properties. Consider bug fixing: human programmers spend a large amount of their time on this activity. Within the same amount of time a GI framework can evaluate thousands of candidate software programs. In order to explore the huge search space an efficient search algorithm needs to be used.

Currently the state-of-the art heuristic approach for software improvement is genetic programming, as is shown in the supplemental material. However, we anticipate more research in the future on the use of other search heuristics in GI.

1) Search operators: A variety of search operators has been used in work on genetic improvement covered by the core papers on the subject. Given that most of this work uses evolutionary algorithms, genetic programming in particular, they inherit similar search operators.

Lansborough et al. [178] used dynamic tracing and genetic algorithms to remove unused features of programs. They applied deletion operations on the binary of the program to be slimmed. Schulte et al. [173] maintained a steady state population, by selecting, modifying and then replacing binary code using an evolutionary algorithm.

Forrest et al. [2] focused on software repair. They operated on the Abstract Syntax Tree (AST) level by deleting, swapping and inserting a statement of code. They also applied a tree-structured differencing to minimise the final repair. Le Goues et al. [3] extended this approach and used fault localisation to bias the repair search. Arcuri et al. [99] operated on the same level of granularity, but additionally co-evolved test cases to improve their ability to produce valid bug repairs.

Wu et al. [162] used mutation testing to change logical, numerical, arithmetical, incremental, relational and bitwise operators in the original program. They represented program changes with a linear chromosome where each gene (itself an integer) represents one of the program statements or parameters that can be modified by GI. Jia et. al. [179] also proposed higher-order mutation-based GI framework to increase the search granularity of GI.

2) Representation of Programs used for Genetic Improvement: There are a number of options for representing modifications to programs. These include ASTs, bytecode, and the source code itself (e.g., treated as a text file).

In some of the early papers on genetic improvement, populations of entire programs were stored. However, as genetic improvement targets large programs, memory becomes an issue. Therefore, most papers on GI currently evolve a population of edits (also called repairs or patches) that are applied to a single master copy of the original program. Representing just the changes that need to be made to a program avoids storing redundant copies of unmodified code [3], [94]. In addition, a number of methods require access to the source code (and operate on the source code itself), while other methods operate directly on low-level code (bytecode or binaries).

Arcuri et al. [154] converted programs into a syntax tree that was then evolved using ECJ, a GP system also used by White et al. [100]. While the original code is translated into a GP representation in ECJ, non-functional properties can be measured when it is converted into source code [98].

Prevalent work on genetic improvement focuses on C and C++ software, including Petke et al. [163], Bruce et al. [174] and Langdon et al.'s [180], [181] work. They use a BNF grammar representation of the code.

With the increasing number of parallel processors available, there is a need to translate code for parallel processing. This is a difficult task when done manually, so a natural question is: can GI achieve this automatically? Early work [96], [115], [142] described *Paragen*, which is designed to be language independent. Programs were represented as tree structures where each line of the original is a terminal [115].

ASTs are one natural approach to representing programs for the purposes of genetic programming. pyEDB [160] (python evolutionary debugger) used ASTs and was applied to Python applications. However, its underlying algorithm is language independent. Python was chosen as the modules required are part of the standard library, including AST compilation and modification, and tracing of execution paths. The representation is a bit string that is translated into a list of edits that point to locations in the AST of the program [160].

Le Goues et al. [3] described *GenProg*, which automatically repairs bugs, following Arcuri's work [127]. GenProg has generated a great deal of interest and uptake of GI and related techniques. An important contribution of GenProg is the choice of representation. In previous work each individual was represented by the entire AST. Software variants in GenProg, in contrast, are represented as patches, a sequence of edit operations for the AST. This promotes scalability, since

the population contains edit sequences, thereby occupying dramatically less space than full ASTs for large programs to which only a few changes are required.

To compute each modified program's fitness, its AST is output as source code, compiled, and executed for each test case. This is done in a sandbox environment [131]. Forrest et al. [2] and Weimer et al. [5] represented C programs as ASTs, but instead of targeting the whole syntax-tree with possible modifications, they selected only the nodes on the execution path as determined by negative test cases. By contrast Cody-Kenny et al. used a syntax tree representation [165], [182] to implement GI for Java programs.

GI has been applied directly to binaries as well as high level source code. Schulte et al. [156] states that 'binary GI' has the following benefits:

1) The technique is potentially applicable to any programming language that compiles to assembly code.
2) Intricate repairs at the statement-level can be performed. These include for example, changing type declarations, comparison operators, and assignments to variables.
3) The complete assembly code language typically consists of a small set of instructions.

Schulte et al. [157] repaired defects in ARM, x86 assembly as well as ELF binaries, and achieved improvements of 86% in memory consumption and 95% in disk requirements. They report a 62% decrease in time to repair the binaries, compared to similar source-level repair techniques. They showed that their technique can also be applied to different languages (Java, C, Haskell) [156] and they repaired two security vulnerabilities [145]. Their approach does not require access to source code.

Landsborough et al. [178] removed unnecessary binary files from programs such as the Unix *echo* utility. Results when using a genetic approach were reported to be better than those obtained merely by using a trace-based approach alone. This allows 'slimmer' versions of the software to exist, with the functionality desired by the user.

GI can either operate offline or online. Online GI modifies software as it executes, whereas offline GI does not, instead improving software for re-deployment. We have described offline GI in previous sections. Online approaches include ECSELR and Gen-O-Fix [183]. ECSELR [184] embeds adaptation inside the target software system enabling the system to transform itself via evolution in a self-contained manner. The software system benefits autonomously, avoiding the problems involved in engineering and maintaining such properties. Swan et al. [185], who created Gen-O-Fix, suggest that developer-specified variation points should be used to define the scope of improvement. Hybrid online-offline approaches were proposed that seek to develop a set of modifications offline, based on data collected during previous online monitoring of execution. These 'dreaming devices' can then be subsequently applied for the next online execution [186].

V. Related Work

We also give an overview of work that we found during our searches, that is either related to or can be considered as work on GI. In our selection of core papers on GI we used the criteria presented in Table I. The work in this section does not conform to one or more of these expectations.

The strength of GI lies in its general applicability, and other approaches explicitly sacrifice this generality in order to obtain guarantees of correctness or higher success rates within a limited range of application. A subset of previous work also operates at a higher level of abstraction, for example, at the architectural rather than code level. As well as restricting generality in terms of the range of transformations that may be applied, some research also makes assumptions that limit its applicability to specific subdomains. For example, assuming the availability of formal contracts written in a language such as Eiffel [187], or the provision of handwritten imperative structural integrity constraints [188], might limit applicability.

The field may benefit from the further incorporation of some of these approaches within a metaheuristic framework. Similarly, related work often employs metaheuristics, and methods developed in GI may improve the applicability and effectiveness of these methods.

A. Program Synthesis

Program synthesis [22] is the automated construction of a new program from a specification. Early work in this area aimed to provide formal guarantees of correctness, but more recent work often relies on a test suite to assess faithfulness to desired semantics, in the same vein as genetic improvement. Balzer replaced formal specifications with natural language [189]. By using natural language, he emphasised the importance of the user-in-the-loop. More recently, Gulwani et al. [28], [190], [191], explored and evaluated example-based program synthesis. They synthesised relatively small-but-useful Microsoft Excel spreadsheet functions, for example, learning useful string processing functions and table transformations that can be inserted into spreadsheets to improve them.

Traditionally, work on program synthesis did not attempt to reuse existing code, preferring to synthesise new functionality from scratch. However, the term 'synthesis' has recently been used to refer to the addition of new functionality into existing software, as achieved by Gulwani et al. [28], allowing for a certain level of code reuse in a similar manner to code transplantation and recent advances in genetic improvement.

In addition to program construction and extension, synthesis also includes the duplication of functionality useful for *n-version programming* [192], where multiple programs are derived from the same formal specification. A topic closely related to program synthesis is the *Automated Design of Algorithms* [193], which uses computational search to discover and improve algorithms for particular problems. The key difference between genetic improvement and the automated design of algorithms is that genetic improvement is applied in-situ or directly to the source code while automated design of algorithms works ex-situ, i.e., evolves an algorithm.

Within the field of genetic programming, some work has crossed into the field of formal synthesis. Katz and Peled [194] applied GP to synthesise mutual exclusion algorithms, verified using model checking, as well as using testing over parametric problems when model-checking fails to scale. They also considered searching for test cases and software repair.

B. Software repair

An overview of all related software repair work, can be found in the survey by Monperrus et al. [195], which also includes software repair applications using genetic improvement, and the survey of Le Goues et al. [94]. Typically, non-GI approaches to repair make assumptions about available specifications [187], they limit the types of bug under consideration [196], or restrict the transformations that may be applied [105]. The advantages of such restrictions are that they make it possible to exhaustively explore potential repairs, or to formally verify the correctness of the repair (at least with respect to a set of supplied test cases).

The most formal approaches rely on the availability of specifications, typically by assuming contracts or invariants specified in the implementation. For example, Wei et al. [187] fixed bugs in Eiffel code by using their specified contracts. First, they applied random testing to a large amount of code and observed failing predicates. They then repaired the program by restoring the relevant invariant through the application of template-based transformations. A similar approach to catching violated contracts is exemplified by Yu et al. [197], who catch violated preconditions and execute different code to correct the erroneous state. Related work was performed by Dallmeir et al. [198]. This work and the work of Gupta [199] assume the availability of pre- and post- conditions, and the presence of a single error. They relied on test-based localisation to narrow down the space of possible changes.

When specifications are unavailable, information can be derived from the program and its test cases. The SemFix tool [105] uses fault localisation and a Satisfiability Modulo Theories (SMT) solver [200]. It derives a partial specification via symbolic execution and generates constraints that a single-line fix must satisfy, while limiting potential repairs to those captured by a set of templates inferred from human studies.

In order to preserve existing correct behaviour, a heuristic often employed is to minimise the syntactical or semantic change to the program. By applying recent advances in SMT solvers, Mechtaev et al. [119] synthesised minimal patches (in terms of their semantics) from a possible patch-space based on fault localisation. They argued that GI tools such as GenProg are better suited to bugs that require multiple changes to a program. Constraint solving can also be applied in some scenarios; for example, Samimi et al. [201] demonstrated the use of string-based constraint solving to repair PHP code that generates HTML. The repairs are validated using a test suite.

Concurrency bugs are a popular target for automated repair, because the correct or desired behaviour is usually straightforward to infer, and the possible transformations can be restricted. Jin et al. [196], [202] repaired atomicity violations and other concurrency bugs by inserting suitable synchronisation primitives. They relied on testing to ensure the repairs are faithful to the desired semantics.

One common alternative to metaheuristic search over a large space of arbitrary transformations is to restrict the transformations to a relatively small number specified by the instantiation of a set of templates. The templates are usually human-designed, or may be extracted from human-written transformations. For example, Kim et. al. [151] manually examined 65,536 human-written patches to identify common templates, such as a change to a method call or branch condition. They used the templates to eliminate bugs in other software.

Kocsis et al. [107] exploited available contracts for the Java equals and hash methods to probe existing Hadoop code for violations. Simple program transformations were used to repair violations, before a metaheuristic was used to further optimise their repairs, in order to improve the quality of hash functions.

In mutation testing, mutants are used to measure how effective test suites are at detecting faulty programs. Debroy and Wong [203] and Schulte et al. [204] suggested that the same operators can be used to repair faulty programs. Debroy and Wong employed Tarantula for fault localisation to reduce the set of possible locations for mutation. They also considered only programs with a single fault at a time.

C. Program Transformation

Program transformation traditionally seeks to improve programs automatically [110], [111] in order to optimise non-functional criteria through the deterministic application of semantics-preserving transformations, although some recent work relies on test suites as opposed to semantics-preserving transformations. The general application of search to selecting transformation sequences has previously been proposed [116].

Code refactoring [205] is one form of transformation. A hybrid of automatic and manual transformations is demonstrated by Meng et al. [206]. Their LASE tool identifies code edits that need to be repeated elsewhere in a program, while taking into account the context of the code and matching 'edit scripts' expressed as an AST using clone detection.

Traditional program transformation can also be achieved using metaheuristic search [143], [207], [208]. Usually these search-based transformations are restricted to semantics-preserving operations. Kocsis and Swan [108] applied point mutation to select between alternative algebraic data types that offer varying asymptotic complexity. An approach that focuses on the individual software engineer's role in performing similar optimisations is the SEEDS framework [120].

Much work optimises software by replacing heuristic components. An elegant example is the Templar tool [209], which employs a generative hyper-heuristic to improve energy efficiency based on a simple power model, enabling the user to specify a 'variation point' for the search process to specialise.

D. Parameter Tuning

Existing software may offer a set of parameters that can be used to tune the performance of a program. A well-known example is the extensive array of options offered by modern compilers, a target suggested by Williams et al. [144]. *Automated parameter tuning* [210] can be regarded as a subset of automated design of algorithms work where search is used to select the best set of parameters for a given problem.

E. Approximate Computing

Improvement of non-functional properties using genetic improvement can lead to the exploration of a Pareto front in objective space, trading off some functionality in return for non-functional gains. This pushes GI onto the frontier of Approximate Computing [211].

Sidiroglou-Douskos et al. [117] introduced the notion of *loop perforation*, which omits some iterations of a loop to gain speed at the cost of accuracy. First, they eliminated perforations that are fatal to the program, before optimising multiple remaining perforations via a greedy algorithm. They applied a well-defined set of transformations to the loop, much like template-based methods discussed above.

Similarly, Hoffman et al. [212] provided 'dynamic dials' to allow a user to tune the performance trade-offs, by transforming static configuration parameters into dynamic code using 'influence tracing'. They added a 'heartbeat' to the program to provide feedback on its performance, and adjust the trade-offs based on this feedback.

F. Data Structure Repair

Erroneous programs can result in invalid data structures, corruption that may be detected by the violation of structural integrity constraints, such as those provided by a programmer in a repOK function. By detecting when violations occur, the Juzi tool [188], [213] uses symbolic execution of the repOK function to suggest repairs to the data structure and restore the invariant in a repair that is sound but not complete. It performs a systematic search through the variables provided.

G. Studies of Existing Code

In order to pursue research goals in genetic improvement, it is useful to help both researchers and developers to understand the search space. Several papers examine open source projects in detail to investigate the assumptions of tools like GenProg.

One assumption of GI software repair tools is that the material required to fix the fault lies within the existing code. Martinez et al. [214] examined the code of open source projects and examined whether code changes, i.e., commits, contain material previously seen in the source code repository of the project. They repeated this investigation at a line and token level, and found extensive redundancy at a token level: up to 52% of commits are composed entirely of tokens written by human programmers in the project. However, they do not correlate this with bug-fixes, and redundancy at line level is less common. Related work can be found in Schulte et al. [204]. Similarly, Barr et al. [71] examined the 'graftability' of code commits, examining Apache projects at line level. They considered the parent revision of the software, code that was later removed, and also code available from other projects.

H. Slicing, Partial Evaluation, and Specialisation

Program slicing [215] reduces a program to a minimal form that retains a desired subset of its original behaviour. It can thus be used both to optimise the size of existing software as well as to extract a desired functionality. Traditional program slicing required program semantics to be preserved. More recently, *observation-based slicing* (ORBS) has been proposed [216]: ORBS relaxes the functionality-preservation criterion by using a test suite as a proxy for desired program behaviour. In this sense observation-based slicing is a subset of GI, where the improvement criterion is the reduction in size of the program, and the only allowed operation is code deletion.

Closely related to slicing is the idea of *program specialisation*; optimising the program for an expected range of inputs. An area of specialisation that has received much attention in manual software development is the selection of application-specific memory managers, typically in embedded or performance-critical systems. Risco-Martín et al. [161] profiled C++ programs and simulated the impact of memory manager configuration on allocation and fragmentation.

Partial evaluation [217] seeks to optimise a program by specialising it with respect to some known inputs. Both partial evaluation and program slicing aim to simplify software. However, the output behaviour of the input program can be different from its slice. In partial evaluation, on the other hand, the transformed program must return the same answer as the original, given the same inputs. Partial evaluation can also be regarded as a subset of GI, where the typical criteria for improvement is software efficiency.

Programs can further be specialised for the *environment* in which they execute. For example, the predominance of mobile apps in software development has led to renewed focus on energy consumption, and in mobile phones and tablets energy consumption is dominated by display screens. Li et al. [218] exploited the correlation between display colour and power consumption in order to minimise energy usage of webpages; they used the simulated annealing metaheuristic to explore the space of colour transforms. In the same vein, Linares-Vàsquez et al. [219] optimised the colour scheme of Android apps, incorporating colour theory to reduce the aesthetic impact of their transformations.

VI. Summary

We provide an overview of research work in genetic improvement. With the ever growing amount and size of software being developed, the need for automated techniques for software improvement is paramount. Because of the abundance of code available optimisation approaches need not start from scratch. Furthermore, metaheuristics, such as evolutionary algorithms, have long been shown to be successful at exploring large search spaces such as the space of possible software variants. Genetic improvement combines these insights to improve software through the application of search. We provide a thorough literature review of papers published between 1995 and 2015 to familiarise the reader with the key results and concepts used in this new research area. We hope that this survey will lead to further uptake of genetic improvement techniques.

References

[1] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, "Automated software transplantation," in *Inter. Symp. on Software Testing and Analysis ISSTA*. ACM, 2015, pp. 257–269.

[2] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Genetic and Evolutionary Computation Conf. GECCO*. ACM, 2009, pp. 947–954.

[3] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Inter. Conf. on Soft. Eng. ICSE*. IEEE Computer Society, 2012, pp. 3–13.

[4] J. Petke, M. Harman, W. B. Langdon, and W. Weimer, "Using genetic improvement and code transplants to specialise a C++ program to a problem class," in *European Conf. on Genetic Programming EuroGP*, ser. LNCS, vol. 8599. Springer, 2014, pp. 137–149.

[5] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Inter. Conf. on Soft. Eng. ICSE*. IEEE, 2009, pp. 364–374.

[6] "The 'Humies' awards. Held at the annual Genetic and Evolutionary Computation Conf. (GECCO)," http://www.human-competitive.org/.

[7] M. Harman, Y. Jia, and W. B. Langdon, "Babel Pidgin: SBSE can grow and graft entirely new functionality into a real world system," in *Inter. Symp. on Search Based Soft. Eng. SSBSE*, ser. LNCS, vol. 8636. Springer, 2014, pp. 247–252.

[8] "BBC Click Interview with Prof. Mark Harman," http://www.bbc.co.uk/programmes/p02y78pp, Online: 4 Aug 2015.

[9] J. Temperton, "Code 'transplant' could revolutionise programming," http://www.wired.co.uk/article/code-organ-transplant-software-myscalpel, Online: 30 Jul 2015.

[10] J. R. Woodward, J. Petke, and W. B. Langdon, "How computers are learning to make human software work more efficiently," http://cacm.acm.org/news/189000-how-computers-are-learning-to-make-human-software-work-more-efficiently, Online: 1 Jul 2015.

[11] W. B. Langdon and J. Petke, "Genetic improvement," http://blog.ieeesoftware.org/2016/02/genetic-improvement.html, Online: 3 Feb 2016.

[12] A. A. Lovelace, "Sketch of the analytical engine invented by Charles Babbage by L. F. Menabrea of Turin, officer of the military engineers, with notes by the translator," https://www.fourmilab.ch/babbage/sketch.html, 1843.

[13] P. B. Sheridan, "The arithmetic translator-compiler of the IBM FORTRAN automatic coding system," *Commun. ACM*, vol. 2, no. 2, pp. 9–21, 1959.

[14] A. Church, *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

[15] J. McCarthy, "Towards a mathematical theory of computation," in *Inter. Found. of Inform. Processing Congress IFIP*, vol. 62, 1962, pp. 21–28.

[16] J. E. Stoy, *Denotational semantics: The Scott–Strachey approach to programming language theory*. MIT Press, 1985.

[17] J. Darlington and R. M. Burstall, "A system which automatically improves programs," *Acta Inf.*, vol. 6, pp. 41–60, 1976.

[18] S. L. Gerhart, "Correctness-preserving program transformations," in *Principles of Prog. Lang. POPL*, 1975, pp. 54–66.

[19] H. Partsch, *The CIP Transformation System*. Springer, 1984.

[20] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, and E. Volanschi, "Tempo: Specializing systems applications and beyond," *ACM Comput. Surv.*, vol. 30, no. 3es, p. 19, 1998.

[21] J. R. Cordy, "The TXL source transformation language," *Sci. Comput. Program.*, vol. 61, no. 3, pp. 190–210, 2006.

[22] Z. Manna and R. J. Waldinger, "Toward automatic program synthesis," *Commun. ACM*, vol. 14, no. 3, pp. 151–165, 1971.

[23] H. A. Simon, "Experiments with a heuristic compiler," *J. ACM*, vol. 10, no. 4, pp. 493–506, 1963.

[24] Z. Manna and R. J. Waldinger, "Knowledge and reasoning in program synthesis," *Artif. Intell.*, vol. 6, no. 2, pp. 175–208, 1975.

[25] W. Bibel, "Syntax-directed, semantics-supported program synthesis," *Artif. Intell.*, vol. 14, no. 3, pp. 243–261, 1980.

[26] J. Traugott, "Deductive synthesis of sorting programs," *J. Symb. Comput.*, vol. 7, no. 6, pp. 533–572, 1989.

[27] C. Paulin-Mohring and B. Werner, "Synthesis of ML programs in the system Coq," *J. Symb. Comput.*, vol. 15, no. 5/6, pp. 607–640, 1993.

[28] S. Gulwani, W. R. Harris, and R. Singh, "Spreadsheet data manipulation using examples," *Commun. ACM*, vol. 55, no. 8, pp. 97–105, 2012.

[29] J. R. Koza, *Genetic programming - on the programming of computers by means of natural selection*, ser. Complex adaptive systems. MIT Press, 1993.

[30] A. M. Turing, "Computing machinery and intelligence," *Mind*, vol. 49, pp. 433–460, 1950.

[31] R. Forsyth, "BEAGLE A Darwinian approach to pattern recognition," *Kybernetes*, vol. 10, no. 3, pp. 159–166, 1981.

[32] T. H. Westerdale, Personal communication.

[33] N. L. Cramer, "A representation for the adaptive generation of simple sequential programs," in *Inter. Conf. on Genetic Algorithms ICGA*. Lawrence Erlbaum Associates, 1985, pp. 183–187.

[34] J. R. Koza, "Non-linear genetic algorithms for solving problems," United States Patent 4935877, 1990.

[35] ——, "Hierarchical genetic algorithms operating on populations of computer programs," in *Inter. Joint Conf. on Artificial Intelligence IJCAI*. Morgan Kaufmann, 1989, pp. 768–774.

[36] D. E. Goldberg, "Computer-aided gas pipeline operation using genetic algorithms," Ph.D. dissertation, University of Michigan, 1983.

[37] J. R. Koza and J. P. Rice, "Genetic programming: The movie," 1992.

[38] T. Hu, W. Banzhaf, and J. H. Moore, "The effects of recombination on phenotypic exploration and robustness in evolution," *Artificial Life*, vol. 20, no. 4, pp. 457–470, 2014.

[39] W. Banzhaf, F. D. Francone, R. E. Keller, and P. Nordin, *Genetic Programming: An Introduction: On the Automatic Evolution of Computer Programs and Its Applications*. Morgan Kaufmann, 1998.

[40] R. Poli, W. B. Langdon, and N. F. McPhee, *A Field Guide to Genetic Programming*. Published via lulu.com and freely available at http://www.gp-field-guide.org.uk, 2008.

[41] D. Rivero, M. Gestal, and J. R. Rabunal, *Genetic Programming: Key concepts and examples, A brief tutorial on Genetic Programming*. LAP Lambert Academic Publishing, 2011.

[42] A. Freitas, *Data Mining and Knowledge Discovery with Evolutionary Algorithms*. Springer, 2002.

[43] E. P. K. Tsang, J. Li, and J. M. Butler, "EDDIE beats the bookies," *Softw., Pract. Exper.*, vol. 28, no. 10, pp. 1033–1043, 1998.

[44] A. K. Kordon, *Applying Computational Intelligence - How to Create Value*. Springer, 2010.

[45] J. R. Koza, "Human-competitive machine invention by means of genetic programming," *AI EDAM*, vol. 22, no. 3, pp. 185–193, 2008.

[46] B. T. Lam and V. Ciesielski, "Discovery of human-competitive image texture feature extraction programs using genetic programming," in *Genetic and Evolutionary Computation Conf. GECCO (2)*, ser. LNCS, vol. 3103. Springer, 2004, pp. 1114–1125.

[47] J. Taylor, J. J. Rowland, R. J. Gilbert, A. Jones, M. K. Winson, and D. B. Kell, "Genetic algorithm decoding for the interpretation of infrared spectra in analytical biotechnology," in *European Workshop on Genetic Programming, EuroGP*, 1998, pp. 21–25.

[48] W. B. Langdon and A. P. Harrison, "GP on SPMD parallel graphics hardware for mega bioinformatics data mining," *Soft Comput.*, vol. 12, no. 12, pp. 1169–1183, 2008.

[49] M. Kovacic and B. Sarler, "Application of the genetic programming for increasing the soft annealing productivity in steel industry," *Materials and Manufacturing Processes*, vol. 24, no. 3, pp. 369–374, 2009.

[50] W. B. Langdon, "Global distributed evolution of L-systems fractals," in *European Conf. on Genetic Programming EuroGP*, ser. LNCS, vol. 3003. Springer, 2004, pp. 349–358.

[51] S. R. DiPaola and L. Gabora, "Incorporating characteristics of human creativity into an evolutionary art algorithm," *Genetic Programming and Evolvable Machines*, vol. 10, no. 2, pp. 97–110, 2009.

[52] Y. Jia, "Picassevo," Android App, 2016.

[53] S. Draves, "The electric sheep screen-saver: A case study in aesthetic evolution," in *Applications of Evolutionary Computing: EvoWorkshops*, 2005, pp. 458–467.

[54] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," in *Inter. Conf. on Computer Graphics and Interactive Techniques SIGGRAPH*. ACM, 1987, pp. 25–34.

[55] A. M. Turing, "Checking a large routine," in *Report of a Conf. on High Speed Automatic Calculating Machines*, 1949, pp. 67–69.

[56] R. L. Sauder, "A general test data generator for COBOL," in *AFIPS Spring Joint Computer Conf.*, 1962, pp. 317–323.

[57] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT – a formal system for testing and debugging programs by symbolic execution," in *Inter. Conf. on Reliable Software*, 1975, pp. 234–245.

[58] H. Inamura, H. Nakano, and Y. Nakanishi, "Trial-and-error method for automated test data generation and its evaluation," *Systems and Computers in Japan*, vol. 20, no. 2, pp. 78–92, 1989.

[59] R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Trans. Soft. Eng*, vol. 17, no. 9, pp. 900–910, 1991.

[60] B. Korel, "Automated software test data generation," *IEEE Trans. Soft. Eng*, vol. 16, no. 8, pp. 870–879, 1990.

[61] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[62] M. Harman, Y. Jia, and Y. Zhang, "Achievements, open problems and challenges for search based software testing," in *International Conference in Software Testing ICST*, 2015, pp. 1–12.

[63] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Soft. Eng*, vol. 37, no. 5, pp. 649–678, 2011.

[64] E. W. Dijkstra, "Structured programming," http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD268.PDF, 1969, circulated privately.

[65] C. A. R. Hoare, "The role of formal techniques: Past, current and future or how did software get so reliable without proof? (extended abstract)," in *Inter. Conf. on Soft. Eng. ICSE*. IEEE Computer Society, 1996, pp. 233–234.

[66] M. Gaudel, "Testing can be formal, too," in *Inter. Joint Conf. on Theory and Practice of Software Development TAPSOFT*, ser. LNCS, vol. 915. Springer, 1995, pp. 82–96.

[67] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. J. Krause, G. Lüttgen, A. J. H. Simons, S. A. Vilkomir, M. R. Woodward, and H. Zedan, "Using formal specifications to support testing," *ACM Comput. Surv.*, vol. 41, no. 2, pp. 9:1–9:76, 2009.

[68] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang, "Compositional shape analysis by means of bi-abduction," *J. ACM*, vol. 58, no. 6, pp. 26:1–26:66, 2011.

[69] B. Godlin and O. Strichman, "Regression verification: Proving the equivalence of similar programs," *Softw. Test., Verif. Reliab.*, vol. 23, no. 3, pp. 241–258, 2013.

[70] M. Gabel and Z. Su, "A study of the uniqueness of source code," in *SIGSOFT Inter. Symp. on the Found. of Soft. Eng. FSE*. ACM, 2010, pp. 147–156.

[71] E. T. Barr, Y. Brun, P. T. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *SIGSOFT Inter. Symp. on the Found. of Soft. Eng. FSE*. ACM, 2014, pp. 306–317.

[72] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Trans. Soft. Eng*, vol. 27, no. 2, pp. 99–123, 2001.

[73] D. Binkley, N. E. Gold, M. Harman, S. S. Islam, J. Krinke, and S. Yoo, "ORBS and the limits of static slicing," in *Inter. Working Conf. on Source Code Analysis and Manipulation SCAM*. IEEE Computer Society, 2015, pp. 1–10.

[74] W. Miller and D. L. Spooner, "Automatic generation of floating-point test data," *IEEE Trans. Soft. Eng*, vol. 2, no. 3, pp. 223–226, 1976.

[75] M. Harman and B. F. Jones, "Search-based software engineering," *Information & Software Technology*, vol. 43, no. 14, pp. 833–839, 2001.

[76] C. K. Chang, C. Chao, S. Hsieh, and Y. Alsalqan, "SPMNet: A formal methodology for software management," in *Inter. Computer Software and Applications Conf. COMPSAC*. IEEE, 1994.

[77] J. J. Dolado, "A validation of the component-based method for software size estimation," *IEEE Trans. Soft. Eng*, vol. 26, no. 10, pp. 1006–1021, 2000.

[78] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios, "Application of genetic algorithms to software testing," in *Inter. Conf. on Soft. Eng. and Applications*, 1992, pp. 625–636.

[79] N. Tracey, J. A. Clark, and K. Mander, "The way forward for unifying dynamic test-case generation: The optimisation-based approach," in *IFIP Inter. Workshop on Dependable Computing and Its Applications DCIA*, 1998, pp. 169–180.

[80] R. Feldt, "Genetic programming as an explorative tool in early software development phases," in *Inter. Workshop on Soft Computing Applied to Soft. Eng. SCASE*, 1999, pp. 11–20.

[81] ——, "Generating diverse software versions with genetic programming: An experimental study," *IEE Proceedings - Software*, vol. 145, no. 6, pp. 228–236, 1998.

[82] K. Lakhotia, M. Harman, and P. McMinn, "A multi-objective approach to search-based test data generation," in *Genetic and Evolutionary Computation Conf. GECCO*. ACM, 2007, pp. 1098–1105.

[83] S. Kalboussi, S. Bechikh, M. Kessentini, and L. B. Said, "Preference-based many-objective evolutionary testing generates harder test cases for autonomous agents," in *Inter. Symp. on Search Based Soft. Eng. SSBSE*, ser. LNCS, vol. 8084. Springer, 2013, pp. 245–250.

[84] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, and A. Ouni, "Many-objective software remodularization using NSGA-III," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 3, pp. 17:1–17:45, 2015.

[85] A. Ramírez, J. R. Romero, and S. Ventura, "A comparative study of many-objective evolutionary algorithms for the discovery of software architectures," *Empirical Soft. Eng.*, vol. 21, no. 6, pp. 2546–2600, 2016.

[86] Y. Zhang, A. Finkelstein, and M. Harman, "Search based requirements optimisation: Existing work and challenges," in *Inter. Working Conf. on Requirements Eng.: Found. for Software Quality REFSQ*, ser. LNCS, vol. 5025. Springer, 2008, pp. 88–94.

[87] W. Afzal and R. Torkar, "On the application of genetic programming for software engineering predictive modeling: A systematic review," *Expert Syst. Appl.*, vol. 38, no. 9, pp. 11 984–11 997, 2011.

[88] M. Harman, "The relationship between search based software engineering and predictive modeling," in *Inter. Conf. on Predictive Models in Soft. Eng. PROMISE*, 2010, p. 1.

[89] F. Ferrucci, M. Harman, and F. Sarro, "Search-based software project management," in *Software Project Management in a Changing World*. Springer, 2014, pp. 373–399.

[90] O. Räihä, "A survey on search-based software design," *Computer Science Review*, vol. 4, no. 4, pp. 203–249, 2010.

[91] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Information & Software Technology*, vol. 51, no. 6, pp. 957–976, 2009.

[92] P. McMinn, "Search-based software testing: Past, present and future," in *International Conference in Software Testing ICST Workshops*. IEEE Computer Society, 2011, pp. 153–163.

[93] M. Harman, Y. Jia, J. Krinke, W. B. Langdon, J. Petke, and Y. Zhang, "Search based software engineering for software product line engineering: A survey and directions for future work," in *Inter. Software Product Line Conf. SPLC*. ACM, 2014, pp. 5–18.

[94] C. Le Goues, S. Forrest, and W. Weimer, "Current challenges in automatic software repair," *Software Quality Journal*, vol. 21, no. 3, pp. 421–443, 2013.

[95] F. G. de Freitas and J. Teixeira de Souza, "Ten years of search based software engineering: A bibliometric analysis," in *Inter. Symp. on Search Based Soft. Eng. SSBSE*, ser. LNCS, vol. 6956. Springer, 2011, pp. 18–32.

[96] P. Walsh and C. Ryan, "Automatic conversion of programs from serial to parallel using genetic programming - the Paragen system," in *Parallel Computing Conf. PARCO*, ser. Advances in Parallel Computing, vol. 11. Elsevier, 1995, pp. 415–422.

[97] C. Ryan, "Reducing premature convergence in evolutionary algorithms," Ph.D. dissertation, University College Cork, 1996.

[98] D. R. White, J. A. Clark, J. Jacob, and S. M. Poulding, "Searching for resource-efficient programs: Low-power pseudorandom number generators," in *Genetic and Evolutionary Computation Conf. GECCO*, 2008, pp. 1775–1782.

[99] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *IEEE Congress on Evolutionary Computation*. IEEE, 2008, pp. 162–168.

[100] D. R. White, A. Arcuri, and J. A. Clark, "Evolutionary improvement of programs," *IEEE Trans. Evolutionary Computation*, vol. 15, no. 4, pp. 515–538, 2011.

[101] M. Harman, W. B. Langdon, Y. Jia, D. R. White, A. Arcuri, and J. A. Clark, "The GISMOE challenge: Constructing the pareto program surface using genetic programming to find better programs (keynote paper)," in *Inter. Conf. on Automated Soft. Eng. ASE*. ACM, 2012, pp. 1–14.

[102] M. Orlov and M. Sipper, "Flight of the FINCH through the java wilderness," *IEEE Trans. Evolutionary Computation*, vol. 15, no. 2, pp. 166–182, 2011.

[103] E. W. Dijkstra, "On the cruelty of really teaching computing science," http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1036.PDF, 1988, circulated privately.

[104] M. Harman, "Why source code analysis and manipulation will always be important," in *Inter. Working Conf. on Source Code Analysis and Manipulation SCAM*. IEEE Computer Society, 2010, pp. 7–19.

[105] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: program repair via semantic analysis," in *Inter. Conf. on Soft. Eng. ICSE*. IEEE Computer Society, 2013, pp. 772–781.

[106] V. Mrazek, Z. Vasícek, and L. Sekanina, "Evolutionary approximation of software for embedded systems: Median function," in *Genetic and Evolutionary Computation Conf. GECCO (Companion)*. ACM, 2015, pp. 795–801.

[107] Z. A. Kocsis, G. Neumann, J. Swan, M. G. Epitropakis, A. E. I. Brownlee, S. O. Haraldsson, and E. Bowles, "Repairing and optimizing hadoop hashcode implementations," in *Inter. Symp. on Search Based Soft. Eng. SSBSE*, ser. LNCS, vol. 8636. Springer, 2014, pp. 259–264.

[108] Z. A. Kocsis and J. Swan, "Asymptotic genetic improvement programming via type functors and catamorphisms," in *Semantic GP Workshop, PPSN*, 2014.

[109] N. Burles, E. Bowles, A. E. I. Brownlee, Z. A. Kocsis, J. Swan, and N. Veerapen, "Object-oriented genetic improvement for improved energy consumption in Google Guava," in *Inter. Symp. on Search Based Soft. Eng. SSBSE*, ser. LNCS, vol. 9275. Springer, 2015, pp. 255–261.

[110] R. M. Burstall and J. Darlington, "A transformation system for developing recursive programs," *J. ACM*, vol. 24, no. 1, pp. 44–67, 1977.

[111] H. A. Partsch, *The Specification and Transformation of Programs: A Formal Approach to Software Development*. Springer, 1990.

[112] M. Orlov and M. Sipper, "Evolutionary software improvement for instruction set meta-evolution," in *Workshop and Summer School on Evolutionary Computing WSSEC*, 2008, pp. 60–63.

[113] ——, "Genetic programming in the wild: Evolving unrestricted bytecode," in *Genetic and Evolutionary Computation Conf. GECCO*. ACM, 2009, pp. 1043–1050.

[114] K. P. Williams, "Evolutionary algorithms for automatic parallelization," Ph.D. dissertation, University of Reading, UK, 1998.

[115] P. Walsh and C. Ryan, "Paragen: A novel technique for the autoparallelisation of sequential programs using genetic programming," in *Genetic and Evolutionary Computation Conf. GECCO*, 1996.

[116] D. Fatiregun, M. Harman, and R. M. Hierons, "Search based transformations," in *Genetic and Evolutionary Computation Conf. GECCO*, ser. LNCS, vol. 2724. Springer, 2003, pp. 2511–2512.

[117] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. C. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *SIGSOFT Inter. Symp. on the Found. of Soft. Eng. FSE*. ACM, 2011, pp. 124–134.

[118] S. H. Tan and A. Roychoudhury, "relifix: Automated repair of software regressions," in *Inter. Conf. on Soft. Eng. ICSE (1)*. IEEE Computer Society, 2015, pp. 471–482.

[119] S. Mechtaev, J. Yi, and A. Roychoudhury, "DirectFix: Looking for simple program repairs," in *Inter. Conf. on Soft. Eng. ICSE (1)*. IEEE Computer Society, 2015, pp. 448–458.

[120] I. L. M. Gutiérrez, L. L. Pollock, and J. Clause, "SEEDS: a software engineer's energy-optimization decision support framework," in *Inter. Conf. on Soft. Eng. ICSE*. ACM, 2014, pp. 503–514.

[121] T. C. O. C. S. Bibliographies, "Online collection of bibliographies of scientific literature in computer science from various sources." http://www.sciencedirect.com/.

[122] ACM, "Digital library," http://dl.acm.org/.

[123] I. Xplore, "Digital library," http://ieeexplore.ieee.org/Xplore/home.jsp.

[124] SpringerLink, "Online search platform," http://link.springer.com/.

[125] ScienceDirect, "Elsevier's online search platform," http://www.sciencedirect.com/.

[126] W. B. Langdon and J. Petke, "Software is not fragile," in *Complex Systems Digital Campus CS-DC*. Springer, 2017, pp. 203–211.

[127] A. Arcuri, "Automatic software generation and improvement through search based techniques," Ph.D. dissertation, University of Birmingham, UK, 2009.

[128] D. R. White, "Genetic programming for low-resource systems," Ph.D. dissertation, University of York, UK, 2009.

[129] P. Sitthi-amorn, N. Modly, W. Weimer, and J. Lawrence, "Genetic programming for shader simplification," *ACM Trans. Graph.*, vol. 30, no. 6, pp. 152:1–152:12, 2011.

[130] W. B. Langdon and M. Harman, "Grow and graft a better CUDA pknotsrg for RNA pseudoknot free energy calculation," in *Genetic and Evolutionary Computation Conf. GECCO (Companion)*. ACM, 2015, pp. 805–810.

[131] E. Fast, C. Le Goues, S. Forrest, and W. Weimer, "Designing better fitness functions for automated program repair," in *Genetic and Evolutionary Computation Conf. GECCO*. ACM, 2010, pp. 965–972.

[132] Y. Qi, X. Mao, and Y. Lei, "Efficient automated program repair through fault-recorded testing prioritization," in *Inter. Conf. on Soft. Maintenance and Evolution ICSM*. IEEE Computer Society, 2013, pp. 180–189.

[133] A. Marginean, E. T. Barr, M. Harman, and Y. Jia, "Automated transplantation of call graph and layout features into Kate," in *Inter. Symp. on Search Based Soft. Eng. SSBSE*, ser. LNCS, vol. 9275. Springer, 2015, pp. 262–268.

[134] C. Le Goues, "Automatic program repair using genetic programming," Ph.D. dissertation, Faculty of the School of Engineering and Applied Science, University of Virginia, 2013.

[135] A. Arcuri, "On search based software evolution," in *Inter. Symp. on Search Based Soft. Eng. SSBSE*, 2009, pp. 39–42.

[136] J. L. Wilkerson and D. R. Tauritz, "Coevolutionary automated software correction," in *Genetic and Evolutionary Computation Conf. GECCO*. ACM, 2010, pp. 1391–1392.

[137] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? Overfitting in automated program repair," in *ESEC/SIGSOFT Inter. Symp. on the Found. of Soft. Eng. FSE*. ACM, 2015, pp. 532–543.

[138] D. R. White and J. Singer, "Rethinking genetic improvement programming," in *Genetic and Evolutionary Computation Conf. GECCO (Companion)*. ACM, 2015, pp. 845–846.

[139] M. Harman, W. B. Langdon, and W. Weimer, "Genetic programming for reverse engineering," in *Working Conf. on Reverese Eng. WCRE*. IEEE Computer Society, 2013, pp. 1–10.

[140] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *Programming Language Design and Implementation PLDI*. ACM, 2015, pp. 43–54.

[141] Y. Jia, M. Harman, W. B. Langdon, and A. Marginean, "Grow and serve: Growing Django citation services using SBSE," in *Inter. Symp. on Search Based Soft. Eng. SSBSE*, ser. LNCS, vol. 9275. Springer, 2015, pp. 269–275.

[142] C. Ryan and L. Ivan, "Automatic parallelization of arbitrary programs," in *European Conf. on Genetic Programming EuroGP*, ser. LNCS, vol. 1598. Springer, 1999, pp. 244–254.

[143] C. Ryan and P. Walsh, "The evolution of provable parallel programs," in *Genetic and Evolutionary Computation Conf. GECCO*, 1997, pp. 295–302.

[144] K. P. Williams and S. A. Williams, "Genetic compilers: A new technique for automatic parallelisation," in *European School of Parallel Programming Environments ESPPE*, 1996, pp. 27–30.

[145] E. M. Schulte, W. Weimer, and S. Forrest, "Repairing COTS router firmware without access to source code or test suites: A case study in evolutionary software repair," in *Genetic and Evolutionary Computation Conf. GECCO (Companion)*. ACM, 2015, pp. 847–854.

[146] A. Arcuri, "On the automation of fixing software bugs," in *Inter. Conf. on Soft. Eng. ICSE Companion*. ACM, 2008, pp. 1003–1006.

[147] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Trans. Soft. Eng*, vol. 38, no. 1, pp. 54–72, 2012.

[148] C. Le Goues, W. Weimer, and S. Forrest, "Representations and operators for improving evolutionary software repair," in *Genetic and Evolutionary Computation Conf. GECCO*. ACM, 2012, pp. 959–966.

[149] T. Nguyen, W. Weimer, C. Le Goues, and S. Forrest, "Using execution paths to evolve software patches," in *International Conference in Software Testing ICST Workshops*. IEEE Computer Society, 2009, pp. 152–153.

[150] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *Inter. Conf. on Automated Soft. Eng. ASE*. IEEE, 2013, pp. 356–366.

[151] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Inter. Conf. on Soft. Eng. ICSE*. IEEE Computer Society, 2013, pp. 802–811.

[152] Z. Qi, F. Long, S. Achour, and M. C. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Inter. Symp. on Software Testing and Analysis ISSTA*. ACM, 2015, pp. 24–36.

[153] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Inter. Conf. on Soft. Eng. ICSE*. ACM, 2014, pp. 254–265.

[154] A. Arcuri, "Evolutionary repair of faulty software," *Appl. Soft Comput.*, vol. 11, no. 4, pp. 3494–3514, 2011.

[155] A. Arcuri, D. R. White, J. A. Clark, and X. Yao, "Multi-objective improvement of software using co-evolution and smart seeding," in *Inter. Conf. on Simulated Evolution and Learning SEAL*, ser. LNCS, vol. 5361. Springer, 2008, pp. 61–70.

[156] E. M. Schulte, S. Forrest, and W. Weimer, "Automated program repair through the evolution of assembly code," in *Inter. Conf. on Automated Soft. Eng. ASE*. ACM, 2010, pp. 313–316.

[157] E. M. Schulte, J. DiLorenzo, W. Weimer, and S. Forrest, "Automated repair of binary and assembly programs for cooperating embedded devices," in *Inter. Conf. on Architectural Support for Prog. Lang. and Operating Systems ASPLOS*. ACM, 2013, pp. 317–328.

[158] E. M. Schulte, "Neutral networks of real-world programs and their application to automated software evolution," Ph.D. dissertation, University of New Mexico, 2014.

[159] J. L. Wilkerson, D. R. Tauritz, and J. M. Bridges, "Multi-objective coevolutionary automated software correction," in *Genetic and Evolutionary Computation Conf. GECCO*. ACM, 2012, pp. 1229–1236.

[160] T. Ackling, B. Alexander, and I. Grunert, "Evolving patches for software repair," in *Genetic and Evolutionary Computation Conf. GECCO*. ACM, 2011, pp. 1427–1434.

[161] J. L. Risco-Martín, J. M. Colmenar, J. I. Hidalgo, J. Lanchares, and J. Díaz, "A methodology to automatically optimize dynamic memory managers applying grammatical evolution," *Journal of Systems and Software*, vol. 91, pp. 109–123, 2014.

[162] F. Wu, W. Weimer, M. Harman, Y. Jia, and J. Krinke, "Deep parameter optimisation," in *Genetic and Evolutionary Computation Conf. GECCO*. ACM, 2015, pp. 1375–1382.

[163] J. Petke, W. B. Langdon, and M. Harman, "Applying genetic improvement to MiniSAT," in *Inter. Symp. on Search Based Soft. Eng. SSBSE*, ser. LNCS, vol. 8084. Springer, 2013, pp. 257–262.

[164] W. B. Langdon and M. Harman, "Genetically improved CUDA C++ software," in *European Conf. on Genetic Programming EuroGP*, ser. LNCS, vol. 8599. Springer, 2014, pp. 87–99.

[165] B. Cody-Kenny and S. Barrett, "The emergence of useful bias in self-focusing genetic programming for software optimisation," in *Inter. Symp. on Search Based Soft. Eng. SSBSE*, ser. LNCS, vol. 8084. Springer, 2013, pp. 306–311.

[166] W. B. Langdon, "Genetic improvement of programs," in *SYNASC*. IEEE Computer Society, 2014, pp. 14–19.

[167] ——, "Genetic improvement of software for multiple objectives," in *Inter. Symp. on Search Based Soft. Eng. SSBSE*, ser. LNCS, vol. 9275. Springer, 2015, pp. 12–28.

[168] W. B. Langdon, M. Modat, J. Petke, and M. Harman, "Improving 3D medical image registration CUDA software with genetic programming," in *Genetic and Evolutionary Computation Conf. GECCO*. ACM, 2014, pp. 951–958.

[169] W. B. Langdon, B. Y. H. Lam, J. Petke, and M. Harman, "Improving CUDA DNA analysis software with genetic programming," in *Genetic and Evolutionary Computation Conf. GECCO*. ACM, 2015, pp. 1063–1070.

[170] W. B. Langdon, "Performance of genetic programming optimised Bowtie2 on genome comparison and analytic testing (GCAT) benchmarks," *BioData Mining*, vol. 8, p. 1, 2015.

[171] S. O. Haraldsson and J. R. Woodward, "Genetic improvement of energy usage is only as reliable as the measurements are accurate," in *Genetic and Evolutionary Computation Conf. GECCO (Companion)*. ACM, 2015, pp. 821–822.

[172] B. R. Bruce, "Energy optimisation via genetic improvement: A SBSE technique for a new era in software development," in *Genetic and Evolutionary Computation Conf. GECCO (Companion)*. ACM, 2015, pp. 819–820.

[173] E. M. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer, "Post-compiler software optimization for reducing energy," in *Inter. Conf. on Architectural Support for Prog. Lang. and Operating Systems ASPLOS*. ACM, 2014, pp. 639–652.

[174] B. R. Bruce, J. Petke, and M. Harman, "Reducing energy consumption using genetic improvement," in *Genetic and Evolutionary Computation Conf. GECCO*. ACM, 2015, pp. 1327–1334.

[175] M. Harman and J. Petke, "GI4GI: improving genetic improvement fitness functions," in *Genetic and Evolutionary Computation Conf. GECCO (Companion)*. ACM, 2015, pp. 793–794.

[176] C. G. Johnson and J. R. Woodward, "Fitness as task-relevant information accumulation," in *Genetic and Evolutionary Computation Conf. GECCO (Companion)*. ACM, 2015, pp. 855–856.

[177] R. E. Lopez-Herrejon, L. Linsbauer, W. K. G. Assunção, S. Fischer, S. R. Vergilio, and A. Egyed, "Genetic improvement for software product lines: An overview and a roadmap," in *Genetic and Evolutionary Computation Conf. GECCO (Companion)*. ACM, 2015, pp. 823–830.

[178] J. Landsborough, S. Harding, and S. Fugate, "Removing the kitchen sink from software," in *Genetic and Evolutionary Computation Conf. GECCO (Companion)*. ACM, 2015, pp. 833–838.

[179] Y. Jia, F. Wu, M. Harman, and J. Krinke, "Genetic improvement using higher order mutation," in *Genetic and Evolutionary Computation Conf. GECCO (Companion)*. ACM, 2015, pp. 803–804.

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

[181] ——, "Optimizing existing software with genetic programming," *IEEE Trans. Evolutionary Computation*, vol. 19, no. 1, pp. 118–135, 2015.

[182] B. Cody-Kenny, E. G. López, and S. Barrett, "locoGP: Improving performance by genetic programming Java source code," in *Genetic and Evolutionary Computation Conf. GECCO (Companion)*. ACM, 2015, pp. 811–818.

[183] J. Swan, M. G. Epitropakis, and J. R. Woodward, "Gen-O-Fix: an embeddable framework for dynamic adaptive genetic improvement programming," Computing Science and Mathematics, University of Stirling, Tech. Rep., 2014.

[184] K. Yeboah-Antwi and B. Baudry, "Embedding adaptivity in software systems using the ECSELR framework," in *Genetic and Evolutionary Computation Conf. GECCO (Companion)*. ACM, 2015, pp. 839–844.

[185] N. Burles, J. Swan, E. Bowles, A. E. I. Brownlee, Z. A. Kocsis, and N. Veerapen, "Embedded dynamic improvement," in *Genetic and Evolutionary Computation Conf. GECCO (Companion)*. ACM, 2015, pp. 831–832.

[186] M. Harman, Y. Jia, W. B. Langdon, J. Petke, I. H. Moghadam, S. Yoo, and F. Wu, "Genetic improvement for adaptive software engineering (keynote)," in *Inter. Symp. on Soft. Eng. for Adaptive and Self-Managing Systems SEAMS*. ACM, 2014, pp. 1–4.

[187] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *Inter. Symp. on Software Testing and Analysis ISSTA*. ACM, 2010, pp. 61–72.

[188] B. Elkarablieh and S. Khurshid, "Juzi: A tool for repairing complex data structures," in *Inter. Conf. on Soft. Eng. ICSE*. ACM, 2008, pp. 855–858.

[189] R. Balzer, "A 15 year perspective on automatic programming," *IEEE Trans. Soft. Eng*, vol. 11, no. 11, pp. 1257–1268, 1985.

[190] S. Gulwani, "Synthesis from examples: Interaction models and algorithms," in *Inter. Symp. on Symbolic and Numeric Algorithms for Scientific Computing SYNASC*. IEEE Computer Society, 2012, pp. 8–14.

[191] W. R. Harris and S. Gulwani, "Spreadsheet table transformations from examples," in *Programming Language Design and Implementation PLDI*. ACM, 2011, pp. 317–328.

[192] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," in *Inter. Symp. on Fault-Tolerant Computing FTCS*, 1978, pp. 3–9.

[193] S. O. Haraldsson and J. R. Woodward, "Automated design of algorithms and genetic improvement: Contrast and commonalities," in *Genetic and Evolutionary Computation Conf. GECCO (Companion)*. ACM, 2014, pp. 1373–1380.

[194] G. Katz and D. A. Peled, "Synthesizing, correcting and improving code, using model checking-based genetic programming," in *Haifa Verification Conf.*, ser. LNCS, vol. 8244. Springer, 2013, pp. 246–261.

[195] M. Monperrus, "A critical review of "automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair," in *Inter. Conf. on Soft. Eng. ICSE*. ACM, 2014, pp. 234–242.

[196] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," in *Programming Language Design and Implementation PLDI*. ACM, 2011, pp. 389–400.

[197] Y. Pei, Y. Wei, C. A. Furia, M. Nordio, and B. Meyer, "Code-based automated program fixing," in *Inter. Conf. on Automated Soft. Eng. ASE*. IEEE Computer Society, 2011, pp. 392–395.

[198] V. Dallmeier, A. Zeller, and B. Meyer, "Generating fixes from object behavior anomalies," in *Inter. Conf. on Automated Soft. Eng. ASE*. IEEE Computer Society, 2009, pp. 550–554.

[199] H. He and N. Gupta, "Automated debugging using path-based weakest preconditions," in *Inter. Conf. on Fundamental Approaches to Soft. Eng. FASE*, ser. LNCS, vol. 2984. Springer, 2004, pp. 267–280.

[200] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*, ser. Frontiers in Artif. Intell. and Applications. IOS Press, 2009, vol. 185, pp. 825–885.

[201] H. Samimi, M. Schäfer, S. Artzi, T. D. Millstein, F. Tip, and L. J. Hendren, "Automated repair of HTML generation errors in PHP applications using string constraint solving," in *Inter. Conf. on Soft. Eng. ICSE*. IEEE Computer Society, 2012, pp. 277–287.

[202] G. Jin, W. Zhang, and D. Deng, "Automated concurrency-bug fixing," in *Symp. on Operating Systems Design and Implementation OSDI*. USENIX Association, 2012, pp. 221–236.

[203] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *International Conference in Software Testing ICST*. IEEE Computer Society, 2010, pp. 65–74.

[204] E. M. Schulte, Z. P. Fry, E. Fast, W. Weimer, and S. Forrest, "Software mutational robustness," *Genetic Programming and Evolvable Machines*, vol. 15, no. 3, pp. 281–312, 2014.

[205] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Trans. Soft. Eng*, vol. 30, no. 2, pp. 126–139, 2004.
[206] N. Meng, M. Kim, and K. S. McKinley, "LASE: locating and applying systematic edits by learning from examples," in *Inter. Conf. on Soft. Eng. ICSE*. IEEE Computer Society, 2013, pp. 502–511.
[207] D. Fatiregun, M. Harman, and R. M. Hierons, "Search-based amorphous slicing," in *Working Conf. on Reverese Eng. WCRE*. IEEE Computer Society, 2005, pp. 3–12.
[208] ——, "Evolving transformation sequences using genetic algorithms," in *Inter. Working Conf. on Source Code Analysis and Manipulation SCAM*. IEEE Computer Society, 2004, pp. 66–75.
[209] J. Swan and N. Burles, "Templar - A framework for template-method hyper-heuristics," in *European Conf. on Genetic Programming EuroGP*, ser. LNCS, vol. 9025. Springer, 2015, pp. 205–216.
[210] H. H. Hoos, "Programming by optimization," *Commun. ACM*, vol. 55, no. 2, pp. 70–80, 2012.
[211] C. M. Kirsch and H. Payer, "Incorrect systems: It's not the problem, it's the solution," in *Design Auto. Conf. DAC*. ACM, 2012, pp. 913–917.
[212] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. C. Rinard, "Dynamic knobs for responsive power-aware computing," in *Inter. Conf. on Architectural Support for Prog. Lang. and Operating Systems ASPLOS*. ACM, 2011, pp. 199–212.
[213] S. Khurshid, I. García, and Y. L. Suen, "Repairing structurally complex data," in *Inter. SPIN Symp. on Model Checking of Software SPIN*, ser. LNCS, vol. 3639. Springer, 2005, pp. 123–138.
[214] M. Martinez, W. Weimer, and M. Monperrus, "Do the fix ingredients already exist? An empirical inquiry into the redundancy assumptions of program repair approaches," in *Inter. Conf. on Soft. Eng. ICSE Companion*. ACM, 2014, pp. 492–495.
[215] M. Weiser, "Program slicing," *IEEE Trans. Soft. Eng*, vol. 10, no. 4, pp. 352–357, 1984.
[216] D. Binkley, N. Gold, M. Harman, S. S. Islam, J. Krinke, and S. Yoo, "ORBS: language-independent program slicing," in *SIGSOFT Inter. Symp. on the Found. of Soft. Eng. FSE*. ACM, 2014, pp. 109–120.
[217] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial evaluation and automatic program generation*, ser. Prentice Hall international series in computer science. Prentice Hall, 1993.
[218] D. Li, A. H. Tran, and W. G. J. Halfond, "Making web applications more energy efficient for OLED smartphones," in *Inter. Conf. on Soft. Eng. ICSE*. ACM, 2014, pp. 527–538.
[219] M. L. Vásquez, G. Bavota, C. E. Bernal-Cárdenas, R. Oliveto, M. D. Penta, and D. Poshyvanyk, "Optimizing energy consumption of GUIs in Android apps: A multi-objective approach," in *ESEC/SIGSOFT Inter. Symp. on the Found. of Soft. Eng. FSE*. ACM, 2015, pp. 143–154.

Justyna Petke is a Senior Research Associate at the Centre for Research on Evolution, Search and Testing (CREST) in University College London. She has published articles on the applications of genetic improvement. Her work on GI won an ACM SIGSOFT distinguished paper award at ISSTA and two Humie's at GECCO 2014 and 2016 (awarded for human-competitive results). She is supported by the Dynamic Adaptive Automated Software Engineering grant from the UK Engineering and Physical Sciences Research Council (EPSRC). She also has a doctorate in Computer Science from the University of Oxford in the area of constraint solving.

Saemundur O. Haraldsson is a PhD candidate in the Department of Computing Science and Mathematics at the University of Stirling. He holds an MSc in industrial engineering from the University of Iceland. He is the primary developer of the first live software system that autonomously uses Genetic Improvement to fix bugs within itself. His research interests are focused on Genetic Improvement, automatic program repair in particular.

Mark Harman is currently an engineering manager at Facebook and a part time professor of Software Engineering in the Department of Computer Science at University College London, where he directed the CREST centre for ten years (2006-2017) and was Head of Software Systems Engineering (2012-2017). He is widely known for work on source code analysis, software testing, app store analysis and Search Based Software Engineering (SBSE), a field he co-founded and which has grown rapidly to include over 1,600 authors spread over more than 40 countries. His SBSE and testing work has been used by many organisations including Daimler, Ericsson, Google, Huawei, Microsoft and Visa. Prof. Harman is a co-founder (and was co-director) of Appredict, an app store analytics company, spun out from UCL's UCLappA group, and is the chief scientific advisor to Majicke, an automated test data generation start up. In February 2017, he and the other two co-founders of Majicke (Yue Jia and Ke Mao) moved to Facebook, London, in order to develop their research and technology as part of Facebook.

William B. Langdon is a professorial research fellow in UCL. He worked on distributed real time databases for control and monitoring of power stations at the Central Electricity Research Laboratories. He then joined Logica to work on distributed control of gas pipelines and later on computer and telecommunications networks. After returning to academia to gain a PhD in genetic programming at UCL (sponsored by National Grid plc.), he worked at the University of Birmingham, the CWI, UCL, Essex University, King's College London and now for a third time at University College London.

David R. White is a researcher in the Department of Computer Science at UCL. He has a PhD in Computer Science from the University of York, where he published some of the seminal papers on Genetic Improvement. He subsequently worked as a SICSA Research Fellow at the University of Glasgow, where he led the Raspberry Pi Cloud project, and later worked on the EPSRC AnyScale project. At UCL, he is part of the EPSRC DAASE project in automated and adaptive software engineering. His research interests include program synthesis through heuristic search and the optimisation of non-functional properties of embedded systems.

John R. Woodward holds degrees in theoretical physics, cognitive science, and computer science, all from the University of Birmingham, UK. Currently, he is with the School of Computer Science and Mathematics at the University of Stirling, He is a member of the Computational Heuristics, Operational Research and Decision Support research group. He was with the European Organization for Nuclear Research (CERN), where he conducted research into particle physics, the Royal Air Force as an Environmental Noise Scientist, and Electronic Data Systems as a Systems Engineer. He is funded by the DAASE project (EP/J017515/1 Dynamic Adaptive Automated Software Engineering) and (EP/N002849/1 FAIME: A Feature based Framework to Automatically Integrate and Improve Metaheuristics via Examples). He is an investigator on an EPSRC grant (EP/N029577/1 TRANSIT: Towards a Robust Airport Decision Support) with three other universities. The aim is to take existing routing and scheduling software and adapt it for specific airport layouts. Partners include Manchester Airport, Air France KLM, Rolls-Royce, and BAE Systems.