Genetic Improvement of Software: A Comprehensive Survey

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

***Abstract*—Genetic improvement (GI) 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 GI back to the origins of computer science itself, our analysis reveals a significant upsurge in activity since 2012. GI 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 GI 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 (GI), survey.**

I. Introduction

GENETIC improvement (GI) uses automated search in order to improve existing software. We present a comprehensive survey of GI, summarizing 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 this survey reveals, evolutionary computing is by far the most widespread computational search technique used in the literature, making GI a field at the intellectual intersection of evolutionary computation, software engineering, optimization, and source code analysis and manipulation.

Manuscript received May 31, 2016; revised November 18, 2016; accepted March 20, 2017. Date of publication April 25, 2017; date of current version May 25, 2018. The work of J. Petke, D. R. White, S. O. Haraldsson, and M. Harman was supported by the EPSRC Project under Grant DAASE EP/J017515/1. The work of W. B. Langdon was supported by the EPSRC Project under Grant GISMOE EP/I033688/1. The work of J. R. Woodward was supported in part by DAASE Project under Grant EP/J017515/1, and in part by the FAIME Project under Grant EP/N002849/1. *(Corresponding author: Justyna Petke.)*

J. Petke, M. Harman, W. B. Langdon, and D. R. White are with University College London, London WC1E 6BT, U.K.

S. O. Haraldsson and J. R. Woodward are with the University of Stirling, Stirling FK94LA, U.K.

This paper has supplementary downloadable material available online at http://ieeexplore.ieee.org provided by the authors.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TEVC.2017.2693219

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 specialization won four "Humies" [1]–[5], awarded for human-competitive results produced by genetic and evolutionary computation [6]. Several papers on GI 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]–[11], demonstrating its influence and reach beyond the research community to the wider developer community and the public at large.

This survey of 3132 distinct titles found, resulted in the identification of 66 core GI papers.[1] However, GI 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 GI 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 GI 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 this survey to define scope and research questions and to help identify primary sources. Nevertheless, although this survey is not an SLR, we did use associated techniques to ensure that we systematically collected potentially relevant publications.

This survey is the first comprehensive analysis of GI research, drawing together its many research strands, results and findings. As this 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.

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

This survey is structured as follows. Section II describes the history of 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 this survey.

II. History of Genetic Improvement

GI 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 (SBSE) (Section II-F).

We provide a brief history of GI, 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 optimization* is due to 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 optimize 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 optimizing 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 optimization 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: 1) program synthesis and 2) 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 minimized 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].

GI is closely related to both synthesis and transformation, yet it differs from each: unlike both program synthesis and program transformation, GI is not always guided by the motivation of correctness-by-construction. Frequently, software testing is used as an oracle for correct system behavior. In this regard, GI 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 U.K. Home Office. Although the idea of evolving programs, particularly Lisp programs, was current amongst Holland's students [32], it was not until his students organized the first Genetic Algorithms conference in Pittsburgh that Cramer [33] published evolved programs in two specially designed languages.

[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: http://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.

In 1988, Koza [34] (also a Ph.D. student of Holland) patented his invention of a GA for program evolution and this was followed by publication in the International Joint Conference on Artificial Intelligence 1989 [35]. The author followed this with 205 publications on genetic programming. (The name "genetic programming" was coined by Goldberg [36], also a Ph.D. student of Holland.) However, it is the series of four books by Koza [29], starting in 1993 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]. By 2016, there were 19 GP books including several intended for students [39]–[41].

Excluding GI, genetic programming research and applications continue to be concentrated on predictive modeling, particularly data mining [42] and financial modeling [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 endeavors [50]–[52]. It led to Draves' [53] electric sheep screen saver. Such distributed evolutionary AI ideas were in the background which led Reynolds [54] to his Boids technique 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 GI'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 GI to the software variant.

E. Testing and Validation

Testing is important for GI, 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 [55] first delineated the role of "program tester" from "program developer." By the 1960s it had been realized 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].

[4]Genetic programming bibliography: http://www.cs.bham.ac.uk/~wbl/biblio/.

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 as follows.

> "*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]–[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 GI 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 GI 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 GI 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 GI, because many of the interventions exploited by GI 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

GI, 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 GI 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 "SBSE" was coined by Harman and Jones [75], in a manifesto for the application of automated search to problems in software engineering.

> "*The thesis underpinning the present paper is that search-based metaheuristic optimization 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 SBSE. 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 GI 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 tradeoffs between potentially competing objectives, particularly where there are many objectives. For example, Lakhotia *et al.* [82] were the first to use multiobjective optimization for test data generation, while Kalboussi *et al.* [83] considered seven objectives when generating test cases. Similarly both Mkaouer *et al.* [84] and Ramírez *et al.* [85] considered the conflict between objectives in software maintenance when reorganizing 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 modeling [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 GI, 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 GI 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 autoparallelization 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 "GI" emerged from a number of previous studies [100]–[102], all of which shared similar goals but with slightly different terminology (such as "evolutionary improvement" and "GI of programs").

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

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. Second, 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 GI 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 GI a timely approach to automated software improvement.

Given the rich heritage on which GI 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 GI, also uses a combination of techniques including those inspired by program synthesis [105] and by genetic programming [2].

III. Survey Methodology

GI 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 3-D 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 GI and other research areas.

Mrazek *et al.* [106], for instance, used cartesian genetic programming to optimize for efficiency and energy consumption. They evaluated approximations of 9-input and 25-input median functions by means of testing, as is typical in GI 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 paper also fits within the field of *program transformation* [110], [111] (see Section V-C for details). Similarly, Orlov and Sipper [102], [112], [113] improved extant software by applying a semantics-preserving crossover operator.

Williams [114] used six evolutionary algorithms to parallelize existing code. In contrast to Walsh and 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 GI 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. Nonstandard approaches to GI 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 tradeoff) with exhaustive and greedy search algorithms. Tan and Roychoudhury [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 Gutiérrez *et al.* [120] performed exhaustive search over small code-level changes for improvement of energy consumption.

TABLE I
Scope of This Survey: Criteria Used to Identify Core Papers on GI (i.e., Conference and Workshop Papers, Journal Articles, and Ph.D. 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.

Given the wide range of topics that fall within the definition of GI, 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 (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* GI paper. These criteria are shown in Table I. We also include position and overview papers on the subject.

In order to provide a thorough overview of core papers on GI, 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: "GI," "software improvement," and "evolutionary improvement." We considered conference and workshop papers, journal articles and Ph.D. 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 GI 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

TABLE II
RESULTS OF PRIMARY SEARCH FOR PAPERS ON GI

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**

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 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

TABLE III
SUMMARY OF SEARCHES CONDUCTED OVER BIBLIOGRAPHIES OF CORE PAPERS ON GI

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

TABLE IV
SUMMARY OF ALL SEARCHES CONDUCTED TO IDENTIFY CORE PAPERS ON GI

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

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 May 3, 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 GI [126], references of which did not contain any additional core publications on GI ("secondary" search step in Table III).

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 GI work fulfilling the four criteria shown in Table I. After this filtering process, we were left with 66 core papers on GI. Overall results of all the searches conducted are shown in Table IV.

The supplementary material contains several details about the core publications. In particular, for each empirical study that uses a GI 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 GI that give an overview of specific GI work.

IV. EXISTING WORK ON GENETIC IMPROVEMENT

The need for automated software optimization has long been recognized and resulted in the development of various research areas, such as program transformation and program slicing (see Section II for details). What differentiates GI 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 optimization from scratch. Furthermore, GI opens up a wider search space of software variants by relaxing restrictions on program correctness.

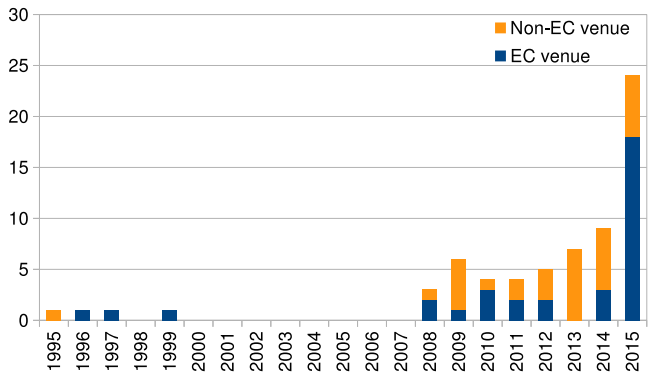

Fig. 1. Number of core papers on GI by year.

The oldest core GI publications concern with software parallelization by Walsh and Ryan [96]. A resurgence of literature in the area can be seen in the late 2000s with Arcuri and Yao's [99] and Arcuri's [127] work on automated software repair and White *et al.*'s [98] and White's [128] work on reduction of energy consumption. 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 GI since 2008, as shown in Fig. 1.

The following sections describe the typical GI process in detail, drawing from the core papers on the subject (listed in the supplementary 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 behavior, or a bug may be fixed while retaining existing nonbuggy 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 and Sipper [113], for instance, used genetic programming with a semantics-preserving crossover to improve existing Java bytecode. Nevertheless, they also encountered 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 tradeoff 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 Fig. 2.

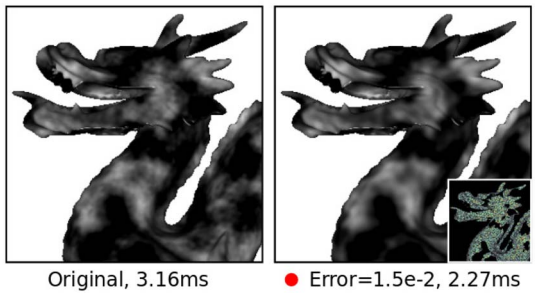

Fig. 2. Selected result of two variants of shader simplification software: original (left) and GI-modified (right). The inset contains a visualization of the per-pixel error.

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

The question remains of how to capture software properties that need to be retained? Several core papers on GI describe work on software transplantation, where a feature is evolved separately from the program to which it is later grafted. Langdon and Harman [130], for instance, evolve, i.e., *grow*, a parallelized version of a part of an existing program, called pknotsRG, used for predicting the minimum binding energy for folding of RNA molecules. They started 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 10 000-fold speedup on certain test instances. Even in this *grow and graft* approach Langdon and 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 GI, software testing was used as a proxy for capturing software properties that needed to be retained (see supplementary 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 prioritization 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] used the following function in their μScalpel tool for automated software transplantation:

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

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 noncrashing and passing test cases respectively. In this case, $\mathit{fitness}(i) = 1$ 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]

$$\begin{aligned}\text{fitness}(C) = W_{\text{PosT}} \times \left|\left\{t \in \text{PosT} \mid P' \text{ passes } t\right\}\right| \\ + W_{\text{NegT}} \times \left|\left\{t \in \text{NegT} \mid P' \text{ passes } t\right\}\right|\end{aligned}$$

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 and Yao [99] opted for a more fine-grained fitness measure for automated software repair. In particular, they used 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 behavior. Arcuri [127] generated the required test cases automatically, showing another advantage of using testing as a means of capturing software functionality.

GI 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 toward fitter individuals. The size of a test suite capturing the desired software behavior is thus potentially infinite. Arcuri and Yao [99] and Arcuri [135] introduced the idea of co-evolving test cases within a GI framework. They used 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 and Tauritz [136] who created CASC, a variant of Arcuri and Yao's [99] framework for C++ programs. Aside from choosing a different target programming language, Wilkerson and Tauritz [136] require the CASC user to provide the fitness function, in contrast to the work of Arcuri and Yao [99].

It is yet unclear how to characterize test suites that would best guide search toward improved software variants. Smith *et al.* [137] conducted an initial investigation in the field of automated software repair. They concluded that "the quality of the patches is proportional to the coverage of the test suite used during repair." They also advocated 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 behavior. GI is a generalist framework and thus allows for other approaches to be explored.

B. Use of Existing Software

The power of GI 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 GI 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 GI 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 GI 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 used 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 paper 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 extracted 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 visualization feature from the CFLOW program into the KATE text editor. Moreover,

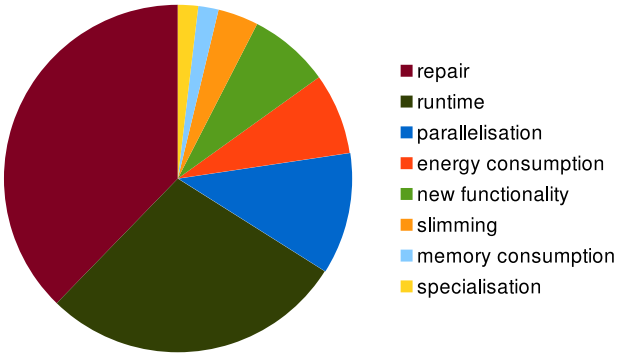

Fig. 3. Software applications of empirical studies in core papers on GI.

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 10 000-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 nonfunctional. References [96], [114], [115], [142], [143], and [144] used evolutionary algorithms to parallelize software. White's [128] Ph.D. thesis focused on energy consumption reduction and both White [128] and Arcuri [127] advocated the use of GI for improvement of nonfunctional properties of software. The importance of this strand of research was re-emphasized in Harman *et al.*'s [101] keynote in 2012. Since then there has been a rapid increase in work on GI and, as shown in Fig. 3, much of this paper has focused on nonfunctional properties.

A major difficulty in nonfunctional software property optimization lies in the measurement of the desired property. For nonfunctional 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 toward desirable software variants.

1) Testing as Fitness Measure: Most of the GI 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 optimization of functional properties of software systems. In particular, Shulte *et al.* [145], Arcuri and Yao [99] and Arcuri [146], Wilkerson and 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.

References [96], [114], [115], [142], [143], and [144] used the number of test cases passed in their work on software parallelization.

A lot of the core GI literature is concerned with automated bug fixing (see Fig. 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]–[150] or using it for comparison [151]–[153]. Other examples of program repair with GI include Arcuri *et al.*'s [99], [154], [155] pioneering work on small programs which predates GenProg, as well as Schulte *et al.*'s [145], [156]–[158] work where the fixing was done post-compilation on the assembly code. Wilkerson *et al.* [159] compared his multiobjective 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 behavior [156]. Schulte *et al.* [145], [156], [157] used sandboxing in their work and ran tests on a virtual machine to ensure no damage is done by the altered software.

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 nonfunctional 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 optimization has been studied before [161], the core papers present only a single example of memory usage optimization 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 nonfunctional 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]–[165]. Langdon *et al.* [130], [166]–[170] reported wall clock speedups.

GI can also be applied per system and hardware by specializing 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 optimized to a certain extent so software must also be adapted.

An issue with energy optimization 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 optimization [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 generalizes to any nonfunctional (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) Multiobjective Improvement: Optimization of nonfunctional properties might sometimes mean degradation of other software properties. These can be either functional or nonfunctional. 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 multiobjective algorithms. Wu *et al.* [162] applied this approach to optimize both runtime and memory consumption. However, multiobjective GI of several nonfunctional properties is still largely unexplored.

D. Search

The power of GI 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 supplementary 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 GI covered by the core papers on the subject. Given that most of this paper uses evolutionary algorithms, genetic programming in particular, they inherit similar search operators.

Landsborough *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 minimize the final repair. Le Goues *et al.* [3] extended this approach and used fault localization to bias the repair search. Arcuri and Yao [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 GI, populations of entire programs were stored. However, as GI 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 [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, nonfunctional properties can be measured when it is converted into source code [98].

Prevalent work on GI focuses on C and C++ software, including Petke *et al.* [163], Bruce *et al.* [174], and Langdon and Harman's [180], [181] work. They used 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. Python evolutionary debugger [160] 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 [127] work. 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.* [165], [182] used a syntax tree representation 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 reported 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, and 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 the 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. Burles *et al.* [185], who created Gen-O-Fix, suggested 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 GI. Balzer replaced formal specifications with natural language [189]. By using natural language, he emphasized the importance of the user-in-the-loop. More recently, Gulwani *et al.* [28], [190], [191], explored and evaluated example-based program synthesis. They synthesized 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 synthesize 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 GI.

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 GI and the automated design of algorithms is that GI 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 synthesize 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 this survey by Monperrus [195], which also includes software repair applications using GI, 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 Pei *et al.* [197], who caught violated preconditions and executed different code to correct the erroneous state. Related work was performed by Dallmeier *et al.* [198]. This paper and the work of He and Gupta [199] assume the availability of pre- and post-conditions, and the presence of a single error. They relied on test-based localization 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 localization 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 behavior, a heuristic often employed is to minimize the syntactical or semantic change to the program. By applying recent advances in SMT solvers, Mechtaev *et al.* [119] synthesized minimal patches (in terms of their semantics) from a possible patch-space based on fault localization. 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 behavior 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 synchronization 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 optimize 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 [203] employed Tarantula for fault localization 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 optimize 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 optimizations is the SEEDS framework [120].

Much work optimizes 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 specialize.

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 and Williams [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 nonfunctional properties using GI can lead to the exploration of a Pareto front in objective space, trading

off some functionality in return for nonfunctional 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 optimizing 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, Hoffmann *et al.* [212] provided "dynamic dials" to allow a user to tune the performance tradeoffs, 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 tradeoffs 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 GI, 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 Specialization

Program slicing [215] reduces a program to a minimal form that retains a desired subset of its original behavior. It can thus be used both to optimize 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 behavior. 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 specialization*; optimizing the program for an expected range of inputs. An area of specialization 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 optimize a program by specializing it with respect to some known inputs. Both partial evaluation and program slicing aim to simplify software. However, the output behavior 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 specialized 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 color and power consumption in order to minimize energy usage of webpages; they used the simulated annealing metaheuristic to explore the space of color transforms. In the same vein, Linares-Vásquez *et al.* [219] optimized the color scheme of Android apps, incorporating color theory to reduce the aesthetic impact of their transformations.

VI. Conclusion

We provide an overview of research work in GI. 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 optimization 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. GI 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 familiarize 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 GI techniques.

References

[1] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, "Automated software transplantation," in *Proc. ACM Int. Symp. Softw. Test. Anal. (ISSTA)*, Baltimore, MD, USA, 2015, pp. 257–269.

[2] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Proc. ACM Genet. Evol. Comput. Conf. (GECCO)*, Montreal, QC, Canada, 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 *Proc. Int. Conf. Softw. Eng. (ICSE)*, Zürich, Switzerland, 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 *Proc. Eur. Conf. Genet. Program. (EuroGP)*, Granada, Spain, 2014, pp. 137–149.

[5] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proc. IEEE Int. Conf. Softw. Eng. (ICSE)*, Vancouver, BC, Canada, 2009, pp. 364–374.

[6] *The 'Humies' Awards Held at the Annual Genetic and Evolutionary Computation Conference (GECCO)*. Accessed on May 10, 2017. [Online]. Available: 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 *Proc. Int. Symp. Search Based Softw. Eng. (SSBSE)*, Fortaleza, Brazil, 2014, pp. 247–252.

[8] *BBC Click Interview With Prof. Mark Harman*. Accessed on Aug. 4, 2015. [Online]. Available: http://www.bbc.co.uk/programmes/p02y78pp

[9] J. Temperton. *Code 'Transplant' Could Revolutionise Programming*. Accessed on Jul. 30, 2015. [Online]. Available: http://www.wired.co.uk/article/code-organ-transplant-software-myscalpel

[10] J. R. Woodward, J. Petke, and W. B. Langdon. *How Computers are Learning to Make Human Software Work More Efficiently*. Accessed on Jul. 1, 2015. [Online]. Available: http://cacm.acm.org/news/189000-how-computers-are-learning-to-make-human-software-work-more-efficiently

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

[12] A. A. Lovelace. (1843). *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*. [Online]. Available: https://www.fourmilab.ch/babbage/sketch.html

[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, NJ, USA: Princeton Univ. Press, 1941.

[15] J. McCarthy, "Towards a mathematical theory of computation," in *Proc. Int. Found. Inform. Process. Congr. (IFIP)*, vol. 62. Amsterdam, The Netherlands, 1962, pp. 21–28.

[16] J. E. Stoy, *Denotational Semantics: The Scott—Strachey Approach to Programming Language Theory*. Cambridge, MA, USA: MIT Press, 1985.

[17] J. Darlington and R. M. Burstall, "A system which automatically improves programs," *Acta Inf.*, vol. 6, no. 1, pp. 41–60, 1976.

[18] S. L. Gerhart, "Correctness-preserving program transformations," in *Proc. Principles Program. Lang. (POPL)*, Palo Alto, CA, USA, 1975, pp. 54–66.

[19] H. Partsch, "The CIP transformation system," in *Program Transformations and Programming Environments*. Heidelberg, Germany: Springer, 1984, pp. 305–322.

[20] C. Consel *et al.*, "Tempo: Specializing systems applications and beyond," *ACM Comput. Surveys*, vol. 30, no. 3, 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, nos. 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* (Complex Adaptive Systems). Cambridge, MA, USA: 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] N. L. Cramer, "A representation for the adaptive generation of simple sequential programs," in *Proc. Int. Conf. Genet. Algorithms (ICGA)*, Pittsburgh, PA, USA, 1985, pp. 183–187.

[33] N. L. Cramer, "A representation for the adaptive generation of simple sequential programs," in *Proc. Int. Conf. Genet. Algorithms (ICGA)*. Pittsburgh, PA, USA, 1985, pp. 183–187.

[34] J. R. Koza, "Non-linear genetic algorithms for solving problems," U.S. Patent 4 935 877, 1990.

[35] J. R. Koza, "Hierarchical genetic algorithms operating on populations of computer programs," in *Proc. Int. Joint Conf. Artif. Intell. (IJCAI)*, Detroit, MI, USA, 1989, pp. 768–774.

[36] D. E. Goldberg, "Computer-aided gas pipeline operation using genetic algorithms," Ph.D. dissertation, Dept. Comput. Sci., Univ. Michigan, Ann Arbor, MI, USA, 1983. [Online]. Available: http://www.worldcat.org/title/computer-aided-gas-pipeline-operation-using-genetic-algorithms-and-rule-learning/oclc/35825281?referer=di&ht=edition

[37] J. R. Koza and J. P. Rice, *Genetic Programming: The Movie*. Cambridge, MA, USA: MIT Press, 1992.

[38] T. Hu, W. Banzhaf, and J. H. Moore, "The effects of recombination on phenotypic exploration and robustness in evolution," *Artif. 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*. San Francisco, CA, USA: Morgan Kaufmann, 1998.

[40] R. Poli, W. B. Langdon, and N. F. McPhee, *A Field Guide to Genetic Programming*. Raleigh, NC, USA: Lulu Press, 2008, [Online]. Available: https://en.wikipedia.org/wiki/Lulu%28company%29

[41] D. Rivero, M. Gestal, and J. R. Rabunal, *Genetic Programming: Key Concepts and Examples, A Brief Tutorial on Genetic Programming*. Saarbrücken, Germany: Lambert Acad., 2011.

[42] A. A. Freitas, *Data Mining and Knowledge Discovery With Evolutionary Algorithms*. Heidelberg, Germany: Springer, 2002.

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

[44] A. K. Kordon, *Applying Computational Intelligence—How to Create Value*. Heidelberg, Germany: Springer, 2010.

[45] J. R. Koza, "Human-competitive machine invention by means of genetic programming," *Artif. Intell. Eng. Design Anal. Manuf.*, 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 *Proc. Genet. Evol. Comput. Conf. (GECCO)*, Seattle, WA, USA, 2004, pp. 1114–1125.

[47] J. Taylor *et al.*, "Genetic algorithm decoding for the interpretation of infra-red spectra in analytical biotechnology," in *Proc. Eur. Workshop Genet. Program. (EuroGP)*, Paris, France, 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. Kovačič and B. Šarler, "Application of the genetic programming for increasing the soft annealing productivity in steel industry," *Mater. Manuf. Processes*, vol. 24, no. 3, pp. 369–374, 2009.

[50] W. B. Langdon, "Global distributed evolution of L-systems fractals," in *Proc. Eur. Conf. Genet. Program. (EuroGP)*, Coimbra, Portugal, 2004, pp. 349–358.

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

[52] Y. Jia, *Picassevo*, Android App, London, U.K., 2016.

[53] S. Draves, "The electric sheep screen-saver: A case study in aesthetic evolution," in *Proc. Appl. Evol. Comput. (EvoWorkshops)*, Lausanne, Switzerland, 2005, pp. 458–467.

[54] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," in *Proc. ACM Int. Conf. Comput. Graph. Interact. Techn. (SIGGRAPH)*, Anaheim, CA, USA, 1987, pp. 25–34.

[55] A. M. Turing, "Checking a large routine," in *Proc. Rep. Conf. High Speed Automat. Calculating Mach.*, 1949, pp. 67–69.

[56] R. L. Sauder, "A general test data generator for COBOL," in *Proc. AFIPS Spring Joint Comput. 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 *Proc. Int. Conf. Rel. Softw.*, Los Angeles, CA, USA, 1975, pp. 234–245.

[58] H. Inamura, H. Nakano, and Y. Nakanishi, "Trial-and-error method for automated test data generation and its evaluation," *Syst. Comput. 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. Softw. Eng.*, vol. 17, no. 9, pp. 900–910, Sep. 1991.

[60] B. Korel, "Automated software test data generation," *IEEE Trans. Softw. Eng.*, vol. 16, no. 8, pp. 870–879, Aug. 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 *Proc. Int. Conf. Softw. Test. (ICST)*, Graz, Austria, 2015, pp. 1–12.

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

[64] E. W. Dijkstra. (1969). *Structured Programming*. [Online]. Available: http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD268.PDF

[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 *Proc. IEEE Int. Conf. Softw. Eng. (ICSE)*, Berlin, Germany, 1996, pp. 233–234.

[66] M.-C. Gaudel, "Testing can be formal, too," in *Proc. Int. Joint Conf. Theory Pract. Softw. Develop. (TAPSOFT)*, Aarhus, Denmark, 1995, pp. 82–96.

[67] R. M. Hierons *et al.*, "Using formal specifications to support testing," *ACM Comput. Surveys*, vol. 41, no. 2, pp. 1–76, 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. 1–66, 2011.

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

[70] M. Gabel and Z. Su, "A study of the uniqueness of source code," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, Santa Fe, NM, USA, 2010, pp. 147–156.

[71] E. T. Barr, Y. Brun, P. T. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proc. SIGSOFT ACM Int. Symp. Found. Softw. Eng. (FSE)*, Hong Kong, 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. Softw. Eng.*, vol. 27, no. 2, pp. 99–123, Feb. 2001.

[73] D. Binkley *et al.*, "ORBS and the limits of static slicing," in *Proc. Int. Working Conf. Source Code Anal. Manipulation (SCAM)*, Bremen, Germany, 2015, pp. 1–10.

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

[75] M. Harman and B. F. Jones, "Search-based software engineering," *Inf. Softw. Technol.*, vol. 43, no. 14, pp. 833–839, 2001.

[76] C. E. Chang, C. Chao, S.-Y. Hsieh, and I. Alsalqan, "SPMNet: A formal methodology for software management," in *Proc. Int. Comput. Softw. Appl. Conf. (COMPSAC)*, Taipei, Taiwan, 1994, p. 57.

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

[78] S. Xanthakis *et al.*, "Application of genetic algorithms to software testing," in *Proc. Int. Conf. Softw. Eng. Appl.*, 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 *Proc. IFIP Int. Workshop Depend. Comput. Appl. (DCIA)*, 1998, pp. 169–180.

[80] R. Feldt, "Genetic programming as an explorative tool in early software development phases," in *Proc. Int. Workshop Soft Comput. Appl. Softw. Eng. (SCASE)*, 1999, pp. 11–20.

[81] R. Feldt, "Generating diverse software versions with genetic programming: An experimental study," *IEE Proc. Softw.*, vol. 145, no. 6, pp. 228–236, Dec. 1998.

[82] K. Lakhotia, M. Harman, and P. McMinn, "A multi-objective approach to search-based test data generation," in *Proc. ACM Genet. Evol. Comput. Conf. (GECCO)*, London, U.K., 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 *Proc. Int. Symp. Search Based Softw. Eng. (SSBSE)*, vol. 8084. St. Petersburg, Russia, 2013, pp. 245–250.

[84] W. Mkaouer *et al.*, "Many-objective software remodularization using NSGA-III," in *Proc. ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 3, pp. 1–45, 2015.

[85] A. Ramírez, J. R. Romero, and S. Ventura, "A comparative study of many-objective evolutionary algorithms for the discovery of software architectures," *Empir. Softw. 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 *Proc. Int. Working Conf. Requirements Eng. Found. Softw. Qual. (REFSQ)*, vol. 5025. Montpellier, France, 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. 11984–11997, 2011.

[88] M. Harman, "The relationship between search based software engineering and predictive modeling," in *Proc. Int. Conf. Predictive Models Softw. Eng. (PROMISE)*, Timişoara, Romania, 2010, p. 1.

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

[90] O. Räihä, "A survey on search-based software design," *Comput. Sci. Rev.*, 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," *Inf. Softw. Technol.*, vol. 51, no. 6, pp. 957–976, 2009.

[92] P. McMinn, "Search-based software testing: Past, present and future," in *Proc. Int. Conf. Softw. Test. Workshops (ICST)*, Berlin, Germany, 2011, pp. 153–163.

[93] M. Harman *et al.*, "Search based software engineering for software product line engineering: A survey and directions for future work," in *Proc. ACM Int. Softw. Product Line Conf. (SPLC)*, Florence, Italy, 2014, pp. 5–18.

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

[95] F. G. de Freitas and J. T. de Souza, "Ten years of search based software engineering: A bibliometric analysis," in *Proc. Int. Symp. Search Based Softw. Eng. (SSBSE)*, vol. 6956. Szeged, Hungary, 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 *Proc. Parallel Comput. Conf. (PARCO)*, vol. 11. Ghent, Belgium, 1995, pp. 415–422.

[97] C. Ryan, "Reducing premature convergence in evolutionary algorithms," Ph.D. dissertation, Dept. Comput. Sci., Univ. College Cork, Cork, Ireland, 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 *Proc. Genet. Evol. Comput. Conf. (GECCO)*, Atlanta, GA, USA, 2008, pp. 1775–1782.

[99] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *Proc. IEEE Congr. Evol. Comput.*, Hong Kong, 2008, pp. 162–168.

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

[101] M. Harman *et al.*, "The GISMOE challenge: Constructing the Pareto program surface using genetic programming to find better programs (keynote paper)," in *Proc. ACM Int. Conf. Autom. Softw. Eng. (ASE)*, Essen, Germany, 2012, pp. 1–14.

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

[103] E. W. Dijkstra. (1988). *On the Cruelty of Really Teaching Computing Science*. [Online]. Available: http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1036.PDF

[104] M. Harman, "Why source code analysis and manipulation will always be important," in *Proc. IEEE Int. Working Conf. Source Code Anal. Manipulation (SCAM)*, Timişoara, Romania, 2010, pp. 7–19.

[105] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program repair via semantic analysis," in *Proc. IEEE Int. Conf. Softw. Eng. (ICSE)*, San Francisco, CA, USA, 2013, pp. 772–781.

[106] V. Mrazek, Z. Vasícek, and L. Sekanina, "Evolutionary approximation of software for embedded systems: Median function," in *Proc. Genet. Evol. Comput. Conf. (GECCO Companion)*, Madrid, Spain, 2015, pp. 795–801.

[107] Z. A. Kocsis *et al.*, "Repairing and optimizing hadoop hashCode implementations," in *Proc. Int. Symp. Search Based Softw. Eng. (SSBSE)*, vol. 8636. Fortaleza, Brazil, 2014, pp. 259–264.

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

[109] N. Burles *et al.*, "Object-oriented genetic improvement for improved energy consumption in Google Guava," in *Proc. Int. Symp. Search Based Softw. Eng. (SSBSE)*, vol. 9275. Bergamo, Italy, 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*. Heidelberg, Germany: Springer, 1990.

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

[113] M. Orlov and M. Sipper, "Genetic programming in the wild: Evolving unrestricted bytecode," in *Proc. ACM Genet. Evol. Comput. Conf. (GECCO)*, Montreal, QC, Canada, 2009, pp. 1043–1050.

[114] K. P. Williams, "Evolutionary algorithms for automatic parallelization," Ph.D. dissertation, Dept. Comput. Sci., Univ. at Reading, Reading, U.K., 1998.

[115] P. Walsh and C. Ryan, "Paragen: A novel technique for the autoparallelisation of sequential programs using genetic programming," in *Proc. Genet. Evol. Comput. Conf. (GECCO)*, Stanford, CA, USA, 1996, pp. 406–409.

[116] D. Fatiregun, M. Harman, and R. M. Hierons, "Search based transformations," in *Proc. Genet. Evol. Comput. Conf. (GECCO)*, vol. 2724. Chicago, IL, USA, 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 *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, Szeged, Hungary, 2011, pp. 124–134.

[118] S. H. Tan and A. Roychoudhury, "Relifix: Automated repair of software regressions," in *Proc. IEEE Int. Conf. Softw. Eng. (ICSE)*, Florence, Italy, 2015, pp. 471–482.

[119] S. Mechtaev, J. Yi, and A. Roychoudhury, "DirectFix: Looking for simple program repairs," in *Proc. IEEE Int. Conf. Softw. Eng. (ICSE)*, Florence, Italy, 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 *Proc. ACM Int. Conf. Softw. Eng. (ICSE)*, Hyderabad, India, 2014, pp. 503–514.

[121] T. C. O. C. S. Bibliographies. *Online Collection of Bibliographies of Scientific Literature in Computer Science From Various Sources*. Accessed on May 3, 2016. [Online]. Available: http://www.sciencedirect.com/

[122] ACM. *Digital Library*. Accessed on May 3, 2016. [Online]. Available: http://dl.acm.org/

[123] I. Xplore. *Digital Library*. Accessed on May 3, 2016. [Online]. Available: http://ieeexplore.ieee.org/Xplore/home.jsp

[124] SpringerLink. *Online Search Platform*. Accessed on May 3, 2016. [Online]. Available: http://link.springer.com/

[125] ScienceDirect. *Elsevier's Online Search Platform*. Accessed on May 3, 2016. [Online]. Available: http://www.sciencedirect.com/

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

[127] A. Arcuri, "Automatic software generation and improvement through search based techniques," Ph.D. dissertation, School Comput. Sci., Univ. at Birmingham, Birmingham, U.K., 2009.

[128] D. R. White, "Genetic programming for low-resource systems," Ph.D. dissertation, Dept. Comput. Sci., Univ. at York, York, U.K., 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. 1–12, 2011.

[130] W. B. Langdon and M. Harman, "Grow and graft a better CUDA pknotsRG for RNA pseudoknot free energy calculation," in *Proc. ACM Genet. Evol. Comput. Conf. (GECCO Companion)*, Madrid, Spain, 2015, pp. 805–810.

[131] E. Fast, C. Le Goues, S. Forrest, and W. Weimer, "Designing better fitness functions for automated program repair," in *Proc. ACM Genet. Evol. Comput. Conf. (GECCO)*, Portland, OR, USA, 2010, pp. 965–972.

[132] Y. Qi, X. Mao, and Y. Lei, "Efficient automated program repair through fault-recorded testing prioritization," in *Proc. IEEE Int. Conf. Softw. Maint. Evol. (ICSM)*, Eindhoven, The Netherlands, 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 *Proc. Int. Symp. Search Based Softw. Eng. (SSBSE)*, vol. 9275. Bergamo, Italy, 2015, pp. 262–268.

[134] C. Le Goues, "Automatic program repair using genetic programming," Ph.D. dissertation, Faculty School Eng. Appl. Sci., Univ. Virginia, Charlottesville, VA, USA, 2013.

[135] A. Arcuri, "On search based software evolution," in *Proc. Int. Symp. Search Based Softw. Eng. (SSBSE)*, Windsor, U.K., 2009, pp. 39–42.

[136] J. L. Wilkerson and D. R. Tauritz, "Coevolutionary automated software correction," in *Proc. ACM Genet. Evol. Comput. Conf. (GECCO)*, Portland, Oregon, USA, 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 *Proc. ACM ESEC/SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, Bergamo, Italy, 2015, pp. 532–543.

[138] D. R. White and J. Singer, "Rethinking genetic improvement programming," in *Proc. ACM Genet. Evol. Comput. Conf. (GECCO Companion)*, Madrid, Spain, 2015, pp. 845–846.

[139] M. Harman, W. B. Langdon, and W. Weimer, "Genetic programming for reverse engineering," in *Proc. IEEE Working Conf. Reverse Eng. (WCRE)*, Koblenz, Germany, 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 *Proc. ACM Program. Lang. Design Implement. (PLDI)*, Portland, OR, USA, 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 *Proc. Int. Symp. Search Based Softw. Eng. (SSBSE)*, vol. 9275. Bergamo, Italy, 2015, pp. 269–275.

[142] C. Ryan and L. Ivan, "Automatic parallelization of arbitrary programs," in *Proc. Eur. Conf. Genet. Program. (EuroGP)*, vol. 1598. Gothenburg, Sweden, 1999, pp. 244–254.

[143] C. Ryan and P. Walsh, "The evolution of provable parallel programs," in *Proc. Genet. Evol. Comput. Conf. (GECCO)*, 1997, pp. 295–302.

[144] K. P. Williams and S. A. Williams, "Genetic compilers: A new technique for automatic parallelisation," in *Proc. Eur. School Parallel Program. Environ. (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 *Proc. ACM Genet. Evol. Comput. Conf. (GECCO Companion)*, Madrid, Spain, 2015, pp. 847–854.

[146] A. Arcuri, "On the automation of fixing software bugs," in *Proc. ACM Int. Conf. Softw. Eng. (ICSE Companion)*, Leipzig, Germany, 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. Softw. Eng.*, vol. 38, no. 1, pp. 54–72, Jan./Feb. 2012.

[148] C. Le Goues, W. Weimer, and S. Forrest, "Representations and operators for improving evolutionary software repair," in *Proc. ACM Genet. Evol. Comput. Conf. (GECCO)*, Philadelphia, PA, USA, 2012, pp. 959–966.

[149] T. Nguyen, W. Weimer, C. Le Goues, and S. Forrest, "Using execution paths to evolve software patches," in *Proc. IEEE Int. Conf. Softw. Test. (ICST) Workshops*, 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 *Proc. IEEE Int. Conf. Autom. Softw. Eng. (ASE)*, Silicon Valley, CA, USA, 2013, pp. 356–366.

[151] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proc. Int. Conf. Softw. Eng. (ICSE)*, San Francisco, CA, USA, 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 *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, Baltimore, MD, USA, 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 *Proc. Int. Conf. Softw. Eng. (ICSE)*, Hyderabad, India, 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 *Proc. Int. Conf. Simulat. Evol. Learn. (SEAL)*, vol. 5361. 2008, pp. 61–70.

[156] E. M. Schulte, S. Forrest, and W. Weimer, "Automated program repair through the evolution of assembly code," in *Proc. Int. Conf. Autom. Softw. Eng. (ASE)*, Antwerp, Belgium, 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 *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, Houston, TX, USA, 2013, pp. 317–328.

[158] E. M. Schulte, "Neutral networks of real-world programs and their application to automated software evolution," Ph.D. dissertation, Dept. Comput. Sci., Univ. New Mexico, Albuquerque, NM, USA, 2014.

[159] J. L. Wilkerson, D. R. Tauritz, and J. M. Bridges, "Multi-objective coevolutionary automated software correction," in *Proc. Genet. Evol. Comput. Conf. (GECCO)*, Philadelphia, PA, USA, 2012, pp. 1229–1236.

[160] T. Ackling, B. Alexander, and I. Grunert, "Evolving patches for software repair," in *Proc. Genet. Evol. Comput. Conf. (GECCO)*, Dublin, Ireland, 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," *J. Syst. Softw.*, vol. 91, pp. 109–123, May 2014.

[162] F. Wu, W. Weimer, M. Harman, Y. Jia, and J. Krinke, "Deep parameter optimisation," in *Proc. Genet. Evol. Comput. Conf. (GECCO)*, Madrid, Spain, 2015, pp. 1375–1382.

[163] J. Petke, W. B. Langdon, and M. Harman, "Applying genetic improvement to MiniSAT," in *Proc. Int. Symp. Search Based Softw. Eng. (SSBSEP)*, vol. 8084. St. Petersburg, Russia, 2013, pp. 257–262.

[164] W. B. Langdon and M. Harman, "Genetically improved CUDA C++ software," in *Proc. Eur. Conf. Genet. Program. (EuroGP)*, vol. 8599. Granada, Spain, 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 *Proc. Int. Symp. Search Based Softw. Eng. (SSBSE)*, vol. 8084. St. Petersburg, Russia, 2013, pp. 306–311.

[166] W. B. Langdon, "Genetic improvement of programs," in *Proc. SYNASC*, Timişoara, Romania, 2014, pp. 14–19.

[167] W. B. Langdon, "Genetic improvement of software for multiple objectives," in *Proc. Int. Symp. Search Based Softw. Eng. (SSBSE)*, vol. 9275. Bergamo, Italy, 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 *Proc. Genet. Evol. Comput. Conf. (GECCO)*, Madrid, Spain, 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 *Proc. Genet. Evol. Comput. Conf. (GECCO)*, Madrid, Spain, 2015, pp. 1063–1070.

[170] W. B. Langdon, "Performance of genetic programming optimised Bowtie2 on genome comparison and analytic testing (GCAT) benchmarks," *BioData Min.*, vol. 8, no. 1, 2015, Art. no. 1.

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

[172] B. R. Bruce, "Energy optimisation via genetic improvement: A SBSE technique for a new era in software development," in *Proc. Genet. Evol. Comput. Conf. (GECCO Companion)*, Madrid, Spain, 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 *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, Salt Lake City, UT, USA, 2014, pp. 639–652.

[174] B. R. Bruce, J. Petke, and M. Harman, "Reducing energy consumption using genetic improvement," in *Proc. Genet. Evol. Comput. Conf. (GECCO)*, Madrid, Spain, 2015, pp. 1327–1334.

[175] M. Harman and J. Petke, "GI4GI: Improving genetic improvement fitness functions," in *Proc. Genet. Evol. Comput. Conf. (GECCO Companion)*, Madrid, Spain, 2015, pp. 793–794.

[176] C. G. Johnson and J. R. Woodward, "Fitness as task-relevant information accumulation," in *Proc. Genet. Evol. Comput. Conf. (GECCO Companion)*, Madrid, Spain, 2015, pp. 855–856.

[177] R. E. Lopez-Herrejon *et al.*, "Genetic improvement for software product lines: An overview and a roadmap," in *Proc. Genet. Evol. Comput. Conf. (GECCO Companion)*, Madrid, Spain, 2015, pp. 823–830.

[178] J. Landsborough, S. Harding, and S. Fugate, "Removing the kitchen sink from software," in *Proc. Genet. Evol. Comput. Conf. (GECCO Companion)*, Madrid, Spain, 2015, pp. 833–838.

[179] Y. Jia, F. Wu, M. Harman, and J. Krinke, "Genetic improvement using higher order mutation," in *Proc. Genet. Evol. Comput. Conf. (GECCO Companion)*, Madrid, Spain, 2015, pp. 803–804.

[180] W. B. Langdon and M. Harman, "Evolving a CUDA kernel from an nVidia template," in *Proc. IEEE Congr. Evol. Comput.*, Barcelona, Spain, 2010, pp. 1–8.

[181] W. B. Langdon and M. Harman, "Optimizing existing software with genetic programming," *IEEE Trans. Evol. Comput.*, vol. 19, no. 1, pp. 118–135, Feb. 2015.

[182] B. Cody-Kenny, E. G. López, and S. Barrett, "locoGP: Improving performance by genetic programming Java source code," in *Proc. Genet. Evol. Comput. Conf. (GECCO Companion)*, Madrid, Spain, 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," Dept. Comput. Sci. Math., Univ. at Stirling, Stirling, U.K., Tech. Rep. CSM-195, 2014.

[184] K. Yeboah-Antwi and B. Baudry, "Embedding adaptivity in software systems using the ECSELR framework," in *Proc. Genet. Evol. Comput. Conf. (GECCO Companion)*, Madrid, Spain, 2015, pp. 839–844.

[185] N. Burles *et al.*, "Embedded dynamic improvement," in *Proc. Genet. Evol. Comput. Conf. (GECCO Companion)*, Madrid, Spain, 2015, pp. 831–832.

[186] M. Harman *et al.*, "Genetic improvement for adaptive software engineering (keynote)," in *Proc. Int. Symp. Softw. Eng. Adapt. Self-Manag. Syst. (SEAMS)*, Hyderabad, India, 2014, pp. 1–4.

[187] Y. Wei *et al.*, "Automated fixing of programs with contracts," in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, Trento, Italy, 2010, pp. 61–72.

[188] B. Elkarablieh and S. Khurshid, "Juzi: A tool for repairing complex data structures," in *Proc. Int. Conf. Softw. Eng. (ICSE)*, Leipzig, Germany, 2008, pp. 855–858.

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

[190] S. Gulwani, "Synthesis from examples: Interaction models and algorithms," in *Proc. Int. Symp. Symbolic Numeric Algorithms Sci. Comput. (SYNASC)*, Timişoara, Romania, 2012, pp. 8–14.

[191] W. R. Harris and S. Gulwani, "Spreadsheet table transformations from examples," in *Proc. Program. Lang. Design Implement. (PLDI)*, San Jose, CA, USA, 2011, pp. 317–328.

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

[193] S. O. Haraldsson and J. R. Woodward, "Automated design of algorithms and genetic improvement: Contrast and commonalities," in *Proc. Genet. Evol. Comput. Conf. (GECCO Companion)*, Vancouver, BC, Canada, 2014, pp. 1373–1380.

[194] G. Katz and D. A. Peled, "Synthesizing, correcting and improving code, using model checking-based genetic programming," in *Proc. Haifa Verification Conf.*, vol. 8244. 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 *Proc. Int. Conf. Softw. Eng. (ICSE)*, Hyderabad, India, 2014, pp. 234–242.

[196] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," in *Proc. Program. Lang. Design Implement. (PLDI)*, San Jose, CA, USA, 2011, pp. 389–400.

[197] Y. Pei, Y. Wei, C. A. Furia, M. Nordio, and B. Meyer, "Code-based automated program fixing," in *Proc. Int. Conf. Autom. Softw. Eng. (ASE)*, Lawrence, KS, USA, 2011, pp. 392–395.

[198] V. Dallmeier, A. Zeller, and B. Meyer, "Generating fixes from object behavior anomalies," in *Proc. Int. Conf. Autom. Softw. Eng. (ASE)*, Auckland, New Zealand, 2009, pp. 550–554.

[199] H. He and N. Gupta, "Automated debugging using path-based weakest preconditions," in *Proc. Int. Conf. Fundamental Approaches Softw. Eng. (FASE)*, vol. 2984. 2004, pp. 267–280.

[200] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability* (Frontiers in Artificial Intelligence and Applications), vol. 185. Amsterdam, The Netherlands: IOS Press, 2009, pp. 825–885.

[201] H. Samimi *et al.*, "Automated repair of HTML generation errors in PHP applications using string constraint solving," in *Proc. Int. Conf. Softw. Eng. (ICSE)*, Zürich, Switzerland, 2012, pp. 277–287.

[202] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu, "Automated concurrency-bug fixing," in *Proc. Symp. Oper. Syst. Design Implement. (OSDI)*, 2012, pp. 221–236.

[203] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *Proc. Int. Conf. Softw. Test. (ICST)*, Paris, France, 2010, pp. 65–74.
[204] E. M. Schulte, Z. P. Fry, E. Fast, W. Weimer, and S. Forrest, "Software mutational robustness," *Genet. Program. Evolvable Mach.*, vol. 15, no. 3, pp. 281–312, 2014.
[205] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Trans. Softw. Eng.*, vol. 30, no. 2, pp. 126–139, Feb. 2004.
[206] N. Meng, M. Kim, and K. S. McKinley, "LASE: Locating and applying systematic edits by learning from examples," in *Proc. Int. Conf. Softw. Eng. (ICSE)*, San Francisco, CA, USA, 2013, pp. 502–511.
[207] D. Fatiregun, M. Harman, and R. M. Hierons, "Search-based amorphous slicing," in *Proc. Working Conf. Reverse Eng. (WCRE)*, Pittsburgh, PA, USA, 2005, pp. 3–12.
[208] D. Fatiregun, M. Harman, and R. M. Hierons, "Evolving transformation sequences using genetic algorithms," in *Proc. Int. Working Conf. Source Code Anal. Manipulation (SCAM)*, Chicago, IL, USA, 2004, pp. 65–74.
[209] J. Swan and N. Burles, "Templar—A framework for template-method hyper-heuristics," in *Proc. Eur. Conf. Genet. Program. (EuroGP)*, vol. 9025. Copenhagen, Denmark, 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 *Proc. Design Auto. Conf. (DAC)*, San Francisco, CA, USA, 2012, pp. 913–917.
[212] H. Hoffmann *et al.*, "Dynamic knobs for responsive power-aware computing," in *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, Newport Beach, CA, USA, 2011, pp. 199–212.
[213] S. Khurshid, I. García, and Y. L. Suen, "Repairing structurally complex data," in *Proc. Int. SPIN Symp. Model Checking Softw. (SPIN)*, vol. 3639. San Francisco, CA, USA, 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 *Proc. Int. Conf. Softw. Eng. (ICSE Companion)*, Hyderabad, India, 2014, pp. 492–495.
[215] M. Weiser, "Program slicing," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 4, pp. 352–357, Jul. 1984.
[216] D. Binkley *et al.*, "ORBS: Language-independent program slicing," in *Proc. SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, Hong Kong, 2014, pp. 109–120.
[217] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation* (Prentice Hall International Series in Computer Science). New York, NY, USA: Prentice-Hall, 1993.
[218] D. Li, A. H. Tran, and W. G. J. Halfond, "Making Web applications more energy efficient for OLED smartphones," in *Proc. Int. Conf. Softw. Eng. (ICSE)*, Hyderabad, India, 2014, pp. 527–538.
[219] M. L. Vásquez *et al.*, "Optimizing energy consumption of GUIs in Android apps: A multi-objective approach," in *Proc. ESEC/SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, Bergamo, Italy, 2015, pp. 143–154.

Justyna Petke received the Doctoral degree in computer science from the University of Oxford, Oxford, U.K., with a focus on constraint solving.

She is a Senior Research Associate with the Centre for Research on Evolution, Search and Testing, University College London, London, U.K. She has published articles on the applications of genetic improvement.

Dr. Petke was a recipient of the ACM SIGSOFT Distinguished Paper Award at ISSTA and two Humie's at GECCO 2014 and 2016, for human-competitive results.

Saemundur O. Haraldsson received the M.Sc. degree in industrial engineering from the University of Iceland, Reykjavik, Iceland. He is currently pursuing the Ph.D. degree with the Department of Computing Science and Mathematics, University of Stirling, Stirling, U.K.

He is the primary developer of the first live software system that autonomously uses genetic improvement to fix bugs within itself. His current research interests include genetic improvement and automatic program repair in particular.

Mark Harman (M'11) received the Ph.D. degree in computer science from the Polytechnic of North London, London, U.K., in 1992.

He is currently an Engineering Manager with Facebook, London, U.K, and a part-time Professor of Software Engineering with the Department of Computer Science, University College London (UCL), London, U.K., where he directed the CREST Centre, from 2006 to 2017 and was a Head of Software Systems Engineering from 2012 to 2017. He is the Co-Founder (and was the Co-Director) of Appredict, an app store analytics company, spun out from UCL's UCLappA Group, and the Chief Scientific Advisor with Majicke, London, an automated test data generation start-up. In 2017, he and the other two co-founders of Majicke (Y. Jia and K. Mao) moved to Facebook, in order to develop their research and technology as part of Facebook. 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 1600 authors spread over more than 40 countries. His SBSE and testing work has been used by many organizations, including Daimler, Ericsson, Google, Huawei, Microsoft, and Visa.

William B. Langdon received the Ph.D. degree in genetic programming with University College London (UCL), London, U.K., supported by National Grid plc.

He is a Professorial Research Fellow with UCL. He researched on distributed real time databases for control and monitoring of power stations with Central Electricity Research Laboratories, London. He then joined Logica, London, to research on distributed control of gas pipelines and later on computer and telecommunications networks. He was with the University of Birmingham, Birmingham, U.K., CWI, Amsterdam, The Netherlands, Essex University, Colchester, U.K., and King's College London, London.

David R. White received the Ph.D. degree in computer science from the University of York, York, U.K.

He is a Researcher with the Department of Computer Science, University College London (UCL), London, U.K., where he is part of the EPSRC DAASE project in automated and adaptive software engineering. He was a SICSA Research Fellow with the University of Glasgow, Glasgow, U.K., where he led the Raspberry Pi Cloud project, and later researched on the EPSRC AnyScale project. He published some of the seminal papers on genetic improvement. His current research interests include program synthesis through heuristic search and the optimization of nonfunctional properties of embedded systems.

John R. Woodward received the B.Sc. degree in theoretical physics, M.S. degree in cognitive science, and Ph.D. degree in computer science from the University of Birmingham, Birmingham, U.K.

He is currently with the School of Computer Science and Mathematics, University of Stirling, Stirling, U.K, where 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), Meyrin, Switzerland, 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 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.