# Using Genetic Improvement & Code Transplants to Specialise a C++ Program to a Problem Class

Justyna Petke[1], Mark Harman[1], William B. Langdon[1], and Westley Weimer[2]

[1] University College London, London, United Kingdom
j.petke@ucl.ac.uk
[2] University of Virginia, Charlottesville, Virginia, United States

**Abstract.** Genetic Improvement (GI) is a form of Genetic Programming that improves an existing program. We use GI to evolve a faster version of a C++ program, a Boolean satisfiability (SAT) solver called MiniSAT, specialising it for a particular problem class, namely Combinatorial Interaction Testing (CIT), using automated code transplantation. Our GI-evolved solver achieves overall 17% improvement, making it comparable with average expert human performance. Additionally, this automatically evolved solver is faster than *any* of the human-improved solvers for the CIT problem.

**Keywords:** genetic improvement, code transplants, code specialisation

## 1 Introduction

Genetic improvement (GI) [14], [17, 18, 19], [23], [28, 29] seeks to automatically improve an existing program using genetic programming. Typically, genetic improvement has focussed on changes using parts of the existing system. We develop the idea of software transplantation [15] and introduce the idea of GI as a means to specialise software.

To investigate and experiment with GI for a particularly challenging problem, we selected the goal of using it to improve the execution performance of the popular Boolean satisfiability (SAT) solver MiniSAT [9]. MiniSAT is a well-known open-source C++ SAT solver. It implements the core technologies of modern SAT solving, including unit propagation, conflict-driven clause learning and watched literals [26].

Improving MiniSAT is challenging because MiniSAT has been iteratively improved over many years by expert human programmers. They have addressed the demand for more efficient SAT solvers and also responded to repeated calls for competition entries to the MiniSAT-hack track of SAT competitions [1]. We use the version of the solver from the first MiniSAT-hack track competition, MiniSAT2-070721[1], as our host system to be improved by GI with transplantation. Furthermore, this competition, in which humans provide modifications to a baseline MiniSAT solver, provides a natural baseline for evaluation and source of evolutionary material, which we call *code bank*.

[1] Solver available at: `http://minisat.se/MiniSat.html`.

This 2009 version of MiniSAT has been repeatedly improved by human programmers, through three iterations of the MiniSAT-hack track of SAT solving competitions, organised biannually. Although GP has been applied to evolve particular SAT heuristics (e.g., [3], [16]), MiniSAT code has never previously been the subject of any *automated* attempt at improvement using genetic programming.

SAT solving has recently been successfully applied to Combinatorial Interaction Testing (CIT) [4], [21], allowing us to experiment with GI for specialisation to that problem domain. CIT is an approach to software testing that produces tests to expose faults that occur when parameters or configurations to a system are combined [22]. CIT systematically considers all combinations of parameter inputs or configuration options to produce a test suite. However, CIT must also minimise the cost of that test suite. The problem of finding such minimal test suites is NP-hard and has attracted considerable attention (e.g., [7, 8], [12], [20], [24]).

SAT solvers have been applied to CIT problems [4], [21], but the solution requires repeated execution of the solver with trial test suite sizes, making solver execution time a paramount concern. We follow the particular formulation of CIT as a SAT problem due to Banbara *et al.* [4], since it has been shown to be efficient.

The **primary contribution of this paper** is the introduction of multi-donor software transplantation and the result of experiments demonstrating that GI can evolve human-competitive versions of a program specialised for a non-trivial problem class. We demonstrate this by improving the 2009 incarnation of MiniSAT. Section 2 introduces our approach to GI. Section 3 presents the set up of our experiments, the results of which are described in Section 4. Section 5 briefly outlines related work and Section 6 concludes.

## 2 Genetic Improvement with Multi-Donor Transplantation and Specialisation

We introduce our approach to GI, which uses multiple author's code for transplantation and specialises the genetically improved software for a specific application domain (in this case CIT). We use a population-based GP. Our work is based on the genetic improvement framework introduced by Langdon and Harman [17] with minor changes. Since we are using a different program, we update the fitness function. We also do not target heavily-used parts of source code, since our program is much smaller than in previous work. Finally, we modify just one C++ file which contains the main solving algorithm. However, unlike Langdon and Harman [17], we use multiple donors and focus on specialising the program to improve it for a specific application domain.

**Program Representation**: We modify the code (in this case MiniSAT) at the level of lines of source code. A specialised BNF grammar is used to create a template containing all the lines from which new individuals are composed. Such a template is created automatically and ensures that classes, types, functions and data structures are retained. For instance, opening and closing brackets in C++

programs are ensured to stay in the same place, but the lines between them can be modified. Moreover, initialisation lines are also left untouched. An extract of a template for MiniSAT is shown in Figure 1. The genome used in our GP is a list of mutations (see below). Header files and comments are not included in our representation.

```
<Solver_156> ::= "{\n"
<Solver_157> ::= "Clause* c = Clause_new(ps, false);\n"
<_Solver_158>::= "clauses.push(c);"
<_Solver_159>::= "attachClause(*c);"
<Solver_160> ::= "}\n"
```

**Fig. 1.** Lines 156–160 from the `Solver.C` MiniSAT file represented in our specialised BNF grammar. Lines marked with `_Solver` can be modified.

**Code Transplants**: We propose to evolve one program by transplanting and modifying lines of code from other programs [15]. Thus our GP has access to both the *host* program being evolved, as well as the *donor* program(s). We call all the lines of code to which GP has access the *code bank*. The donor code statements are then available for mutations of the *host* instance, but need not be used in the same order. For example, our search may combine the first half of an optimisation from one version of MiniSAT with the second half of an optimisation from another and then specialise the resulting code to CIT problems. This re-use and improvement of existing developer expertise is critical to the success of our technique.

**Mutation Operator**: A new version of a program (i.e. a new *individual*) is created by making multiple changes to the original program. Each such *mutation* is either a DELETE, REPLACE or COPY operation. The changes are made at the level of lines of source code, with a special case for conditional statements. A DELETE operation simply deletes a line of code, a REPLACE operation replaces a line of code with another line of code from the code bank and COPY operation inserts a line of code from the code bank into the program. In the case of conditional statements, we focus on and modify their predicate expressions[2]. For instance, the second part of a FOR loop (e.g., `i<0`) can only be replaced by the second part of another FOR loop (e.g., `i<10`) and any IF condition can be replaced with any other IF condition. Examples of the three mutation types are shown in Figure 2.

```
<_Solver_159>                          # Delete line 159
<for3_Solver_533><for3_Solver_772>     # Replace the 3rd part of the 'for'
                                       # loop (i.e., loop variable increment)
                                       # in line 533 with the 3rd part of
                                       # the 'for' loop in line 772
<_Solver_806>+<_Solver_949>            # Add line 949 in front of line 806
```

**Fig. 2.** Examples of the three types of mutations allowed.

[2] In the case of a DELETE operation we replace the predicate expression with '0' to prevent compilation errors.

**Crossover Operator**: We choose to represent each individual as a list of mutations with respect to the original, which we call *edit list*. This representation allows our technique to apply to programs of significant size [13], since we do not keep the whole of each version of the program in memory — just a list of changes. When creating individuals for the next generation, a *crossover* operation simply concatenates two individuals from the current population by appending one list to another. The first parent is chosen based on its fitness value while the other is chosen uniformly among those individuals that compiled, as in previous work [17].

**Fitness Function**: We evaluate the fitness of an individual in terms of a combination of functional properties (those related to software correctness) and non-functional properties (those related to performance, quality of service, etc.) by observing its performance on SAT instances. Before the GP starts, the training set of SAT instances is divided into five groups by difficulty, which we meassure in required solving time. In each generation one test case is sampled uniformly from each group (or 'bin' following other terminology [17]) and all individuals are run on the selected test cases. This sampling helps to avoid overfitting. To evaluate an individual, the corresponding list of changes is applied to the original and the resulting source code is compiled, producing a new SAT solver that can then be executed (individuals that fail to compile are never selected).

To guide the GP search toward a more efficient version of the program, our fitness function takes into account both solution quality and program speed. For internal fitness calculations, efficiency is measured in terms of lines of code executed based on simple counter-based program instrumentation. The use of line counts (instead of CPU or wall-clock times) avoids environmental bias and provides a deterministic fitness signal. For the final presentation of our empirical results, timing measurements in seconds are also presented (see Section 4).

**Selection**: The GP process is run for a fixed number of generations (in our case 20) with a fixed population size (in our case 100). In the initial population each individual consists of a single mutation applied to the original program. After the fitness of each of the individuals is calculated, the fittest half of the population is chosen, filtered to include only those individuals that exceed a threshold fitness value. We focus on exploiting high-quality solutions, and thus our fitness threshold is set to select those individuals that either (1) return the correct answer in all cases, or (2) return the correct answer in all but one case and take no more than twice as long as the original solver.

Next, a set of offspring individuals is created using crossover on those selected from the current population. Also a new mutation is added to each of the parent individuals selected to create offspring. Both crossover and mutation are applied with 50% probability. If mutation is chosen, one of the three operations (i.e. REPLACE, COPY and DELETE) is selected with equal probability. If mutation and crossover do not create a sufficient number of individuals for the next generation, new individuals are created consisting of one mutation (in practice, this occurs 38% of the time). Finally, the fitness of the newly-created individ-

uals is calculated, as described previously, and the process continues until the generation limit is reached.

**Filtering**: We have observed that many program optimisations are independent and synergistic. As a result, we propose a final step that combines all mutations from the fittest individuals evolved and retains all synergistic edits. This post-processing step is simplified by our edit list representation and helps to ensure that our final output benefits from more of the search space exploration conducted by the GP. Exploring all subsets of edits is infeasible. Our prototype implementation uses a greedy algorithm. Each mutation from the best individuals from all of our experiments is considered separately. We apply each operation to the original program and evaluate its fitness. Next, we order the mutations by their fitness value[3] and iteratively consider these adding only those edits that do not decrease fitness. Other efficient techniques, such as constructing a 1-minimal subset of edits [30], are possible.

## 3 Experimental Setup

The main hypothesis investigated in this paper is:

> *Genetic improvement with transplantation finds faster CIT-specialised MiniSAT versions than any developed by expert human programmers.*

**Host & Donor Programs**: We evolve MiniSAT2-070721, in particular the C++ file containing its main solving algorithm. This version was used as a reference solver in the first MiniSAT-hack competition, organised in 2009. Unless otherwise noted, we use MiniSAT and MiniSAT2-070721 interchangeably. The main solver algorithm involves 478 of the 2419 lines in MiniSAT. For our experiments we use two donor programs, which altogether provide 104 new lines of source code. The first donor is the winner of the MiniSAT-hack competition from 2009, called "MiniSAT 09z". We refer to this solver as MiniSAT-best09. The second donor program is the "MiniSat2hack" solver, the best performing solver from the competition when run on our CIT-specific benchmarks. Thus we refer to this solver as MiniSAT-bestCIT. We also added all the donor code to MiniSAT and ran this hybrid solver for comparison. We refer to this solver as MiniSAT-best09+bestCIT.

**Test Cases**: Real-world SAT instances from the combinatorial interaction testing area can take hours or even days to run. Thus we evaluate MiniSAT performance on a set of synthetic CIT benchmarks. Using the encoding of Banbara *et al.* [4], we translated 130 CIT benchmarks into SAT instances[4] We kept the number of values for each of the parameters the same in every instance. This allows us to verify observed results against public catalogues of best known results [8]. We use one-third of these CIT benchmarks in the training set (which is divided

[3] Note that since each individual is represented by a list of edits (or mutations) and at the filtering stage we consider one mutation in turn, we use the word 'mutation' and 'individual' interchangeably.

[4] Available from Justyna Petke j.petke@ucl.ac.uk.

into five groups, as discussed in Section 2)[5]) and the rest in the verification set. We use execution time to define instance difficulty and divide the training set into five groups based on that measure. The largest instances contain over 1 million SAT clauses and MiniSAT is able to produce an answer for each of these within 30 seconds.

**Code Transplants**: In our experiments the seeds of high-level human optimisations targeting a generic benchmark set serve as donor code and are selected and recombined with novel changes to produce a specialised host SAT solver.[6] We conduct three sets of experiments, varying the code bank while holding the rest of the GI process constant. The donor code is selected in turn from:

1. MiniSAT-best09;
2. MiniSAT-bestCIT;
3. MiniSAT-best09 and MiniSAT-bestCIT.

We compare our evolved solver with both the host and donor programs in each of the experiments. We call our evolved solver MiniSAT-gp. Finally, we refer to the solver that results from our postprocessing filtering step (see Section 2) as MINISAT-gp-combined.

## 4 Results

To evaluate the efficacy of our technique, we evolve improved and specialised versions of MiniSAT and compare them to human-improved SAT solvers in terms of both runtime cost and solution quality. While internal fitness calculations are measured in terms of lines of code executed, all final results are presented in terms of CPU time data based on runs on a 1.6GHZ Lenovo 3000 N200 laptop with an Intel Core 2 Duo processor and 2GB of RAM. The GP was run with a population size of 100 and 20 generations.

In all experiments the compilation rate (using MiniSAT's provided Makefile) was high, between 79% and 81%. This high compilation rate results from our use of a specialised BNF grammar for edits, preventing most syntax errors. Runtime data reported in Table 1 is an average of 20 runs of each solver. The number of lines of code executed in each of the runs stayed the same, while time variation was less than 3%.

### 4.1 Transplanting from MiniSAT-best09

When the code bank included lines from the original as well as the overall best version of the solver from the MiniSAT-hack competition, GP produced a mutated version of MiniSAT that was, on average, over 5% faster than the original solver on CIT instances (see Table 1). None of the new code from MiniSAT-best09 was selected by GP in the improved individual. We observe that the best

[5] The first two groups contain the fastest running instances, while those that require the longest time are in group five. Additionally, the second and fourth group contain unsatisfiable instances only, while the first and third only satisfiable ones.

[6] Adding a donor statement X to the code bank is equivalent, in terms of the search space explored, to adding IF (0) X to the input program in a preprocessing step.

**Table 1.** Normalized runtime comparison of MiniSAT versions, based on averages over 20 runs. The first four solvers are human written, the last four were evolved by our technique. The "Donor" column indicates the source of the donor code available in the code bank. "Lines" indicates lines of code executed, "Time" indicates CPU time executed (lower is better, all measurements normalized to original MiniSAT).

| Solver | Donor | Lines | Time |
|---|---|---|---|
| MiniSAT (original) | — | 1.00 | 1.00 |
| MiniSAT-best09 | — | 1.46 | 1.76 |
| MiniSAT-bestCIT | — | 0.72 | 0.87 |
| MiniSAT-best09+bestCIT | — | 1.26 | 1.63 |
| MiniSAT-gp | best09 | 0.93 | 0.95 |
| MiniSAT-gp | bestCIT | 0.72 | 0.87 |
| MiniSAT-gp | best09+bestCIT | 0.94 | 0.96 |
| MiniSAT-gp-combined | best09+bestCIT | **0.54** | **0.83** |

solver from the competition — which evaluated on a general, non-CIT benchmark suite — was not the most efficient one on the instances from the CIT domain. In fact, the original MiniSAT without modifications was even faster than the winner of the 2009 competition in this domain. It is thus not surprising that using donor code from MiniSAT-best09 did not admit the evolution of efficient solvers specialised to the CIT domain.

Evolution achieved runtime improvement by switching off IF and FOR loop conditions. Also, execution times of certain FOR loops were decreased using REPLACE operations. Table 2 shows all the mutations made in the fastest evolved version of MiniSAT.

**Table 2.** Mutations in the genetically improved solver (with best09 as donor).

| mutation | mutated code | changes |
|---|---|---|
| DELETE | IF statement condition | 5 |
| DELETE | line of code | 8 |
| REPLACE | FOR loop condition | 7 |
| REPLACE | IF statement condition | 2 |
| COPY | line of code | 1 |
| *total* | | 23 |

Among the evolved changes, an addition operation on a variable used solely for statistical purposes was deleted, as were three assertions. The evolved changes specialised MiniSAT to the CIT instances tested, but did not retain functionality for instances from other domains. For example, one deletion removed a memory optimisation function, potentially increasing solver's chance of running into an out-of-memory error for larger instances.

### 4.2 Transplanting from MiniSAT-bestCIT

In the next experiment the GP code bank contained source code both from the original MiniSAT solver as well as MiniSAT-bestCIT. The evolved version of

MiniSAT is, on average, 13% faster than the original solver (see Table 1). Given that it usually takes hours or even days to run a SAT solver on real-world CIT instances, such a performance improvement could have a noticeable impact.

The human-written MiniSAT-bestCIT solver also provides similar runtime results — in fact, the performance of our evolved version and the human-written version are not different in a statistically significant sense. The similarities can be explained by the changes made by the GP process, shown in Table 3.

**Table 3.** Mutations occurring in the genetically improved solver.

| mutation | mutated code | changes |
|---|---|---|
| DELETE | line of code | 1 |
| REPLACE | IF statement condition | 1 |
| *total* | | 2 |

By replacing the IF statement condition, the GP enabled a function that contained 95% of the 'new' human-written lines from MiniSAT-bestCIT. The other one-line deletion simply removed an assertion.

### 4.3 Transplanting from MiniSAT-best09 and MiniSAT-bestCIT

Finally, we allowed evolution to inject code from both MiniSAT-best09 and MiniSAT-bestCIT. Runtime results are presented in Table 1: the best evolved program achieved 4% runtime improvement over the original solver.

Table 4 shows the set of changes produced by genetic improvement. Lines involved in about half of the mutations were never executed in the fastest genetically modified program. Thus, GP essentially removed dead code. Moreover, five assertions were removed as well as three updates to statistical variables. In four cases parts of code were replaced with semantically-equivalent (but not necessarily equally expensive) computations.

**Table 4.** Mutations occurring in the genetically improved solver.

| mutation | mutated code | changes |
|---|---|---|
| DELETE | IF statement condition | 10 |
| DELETE | line of code | 30 |
| DELETE | FOR loop condition | 10 |
| REPLACE | FOR loop condition | 10 |
| REPLACE | IF statement condition | 4 |
| REPLACE | line of code | 6 |
| COPY | line of code | 5 |
| *total* | | 75 |

### 4.4 Combining Results

In the previous experiment the GP identified a 'good change': a one-line modification that allowed 95% of the donor code to be executed. Even though the GP process produced individuals containing such a change, other mutations within all such individuals caused slower runtime or compilation errors. Our approach

based on filtering (see Section 2) holds out the promise of combining the best parts of all variants discovered.

We started with the individual with the best runtime performance, and iteratively added mutations from the next performant individual. Only changes that do not reduce performance or correctness are retained. The resulting 'combined' solver performs 17% faster than the original MiniSAT and outperforms all other human-written solvers considered by at least 4% (this difference is statistically significant, see Table 1).

In total, this version involved 56 evolved mutations. Eight among these were one-line assertion removals. Details of all the mutations selected are presented in Table 5.

**Table 5.** Mutations occurring in the combination of the fastest genetically improved solvers.

| mutation | mutated code | number of changes |
|---|---|---|
| DELETE | IF statement condition | 9 |
| DELETE | line of code | 22 |
| DELETE | FOR loop condition | 6 |
| REPLACE | FOR loop condition | 8 |
| REPLACE | IF statement condition | 3 |
| REPLACE | line of code | 4 |
| COPY | line of code | 4 |
| *total* | | 56 |

By combining the synergistic optimisations found in the three best evolved individuals, our approach produced the fastest specialised SAT solver for CIT among all solvers developed by expert human programmers that were entered into the 2009 MiniSAT-hack competition. On the 130 benchmark instances this automatically-constructed solver performed better in 128 instances (in terms of lines of code executed). In the other two cases it was only slightly worse.

However, since small benchmarks were chosen for the training set, the evolved individual might not scale to larger problems. Manual inspection suggests that optimisations relevant to large instances may not be retained, but a systematic evaluation on separate instances is left to future work. However, we note that the evolved individual retained required functionality on the two-thirds of the instances that were held out for verification, even though it was only exposed to the other third for testing.

## 5 Summary of Related Work

Genetic improvement has been successfully used to automate the process of bug fixing [19]. GI has also been used to improve non-functional properties of relatively small laboratory programs [23], [27,28,29], as well as larger real world systems [17]. It has also been used to automatically migrate a system from one platform to another [18].

In this previous work on genetic improvement, GP was concerned with a single program; the program to be improved. Code is extracted, perhaps modified

and then reinserted back into the program at a different location. The focus of the present paper on transplantation from multiple programs therefore denotes an important departure from this previous literature. As a result of multiple transplantation, GP is no longer concerned with a single program to be improved, but multiple donor programs, from which code can be extracted to help guide genetic improvement.

The idea of code transplantation using GP was proposed by Harman et al. [15], but it has not hitherto been implemented, nor has it previously been demonstrated to be useful in practice. We are the first to use GP to implement and evaluate transplantation for genetic improvement.

The goal of improvement adopted by the present paper also differs from that of previous work on genetic improvement, which focused on improving non-functional properties, such as execution time [17], [23] and power consumption [28]. It has also been used to migrate code [18] and to improve functional properties (by fixing bugs) [2], [10], [13], [19]. In all of these scenarios, the full functionality of the original program is to be retained; part of the fitness function specifically checks for the absence of regression faults.

Instead we aim to *specialise* a program using genetic improvement; the full functionality of the original program therefore need not be retained. In this way, this specialisation-oriented genetic improvement is reminiscent of partial evaluation [5, 6], [11], which also seeks to achieve automated program specialisation. However, whereas partial evaluation uses meaning-preserving transformation to 'hard wire' parameter choices into code (thereby specialising it to those parameter choices), we use genetic programming to search for transplants that specialise a program to a class of inputs.

In preliminary experiments with MiniSAT [25] a varied set of instances from SAT competitions were used. However, this approach led to only very modest runtime improvements (up to 2%). We have significantly improved on this preliminary work using multi-donor transplantation to achieve a human-competitive 17% improvement.

## 6 Conclusions

We evolved a specialised version of a C++ program using genetic improvement with transplants. Previously, genetic programming has successfully been applied to improve software behaviour of various systems leading to significant speed-ups. We investigated whether this could be achieved on a well-known software system that has been engineered by many expert human programmers. We specialised this program for a particular hard problem class and used a novel idea of code transplantation.

For our experiments we chose MiniSAT, a very popular Boolean satisfiability (SAT) solver that has been thoroughly studied. The MiniSAT-hack track of SAT competitions is specifically designed to encourage humans to make minor changes to MiniSAT code that could lead to significant runtime improvements, and hence lead to new insights into SAT solving technology. Thus this competition provides a natural source of genetic material for code transplants, as

well as a natural baseline for assessing human-competitive results. We evaluated how our automated approach applies to a particular application domain, namely Combinatorial Interaction Testing.

Our fastest evolved MiniSAT version achieved 13% runtime improvement over the original solver, similar to the best version of MiniSAT for CIT. By combining the synergistic optimisations from our individuals we achieved a 17% runtime improvement. For the CIT domain our evolved solver outperforms *all* of the human-written solvers entered into that competition.

## References


1. MiniSAT-hack track of SAT competition (2009), `http://www.satcompetition.org/2009/`, In 2009 this was part of the 12th International Conference on Theory and Applications of Satisfiability Testing
2. Arcuri, A., Yao, X.: A Novel Co-evolutionary Approach to Automatic Software Bug Fixing. In: Proceedings of the IEEE Congress on Evolutionary Computation (CEC '08). pp. 162–168. IEEE Computer Society, Hong Kong, China (1-6 June 2008)
3. Bader-El-Den, M.B., Poli, R.: Generating SAT local-search heuristics using a GP hyper-heuristic framework. In: Monmarché, N., Talbi, E.G., Collet, P., Schoenauer, M., Lutton, E. (eds.) Artificial Evolution. Lecture Notes in Computer Science, vol. 4926, pp. 37–49. Springer (2007)
4. Banbara, M., Matsunaka, H., Tamura, N., Inoue, K.: Generating combinatorial test cases by efficient SAT encodings suitable for CDCL SAT solvers. In: 17th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Yogyakarta, India. pp. 112–126 (2010)
5. Beckman, L., Haraldson, A., Oskarsson, O., , Sandewall, E.: A partial evaluator, and its use as a programming tool. Artificial Intelligence 7(4), 319–357 (1976)
6. Binkley, D., Danicic, S., Harman, M., Howroyd, J., Ouarbya, L.: A formal relationship between program slicing and partial evaluation. Formal Aspects of Computing 18(2), 103–119 (2006)
7. Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C.: The AETG system: an approach to testing based on combinatorial design. IEEE Transactions on Software Engineering 23(7), 437–444 (1997)
8. Colbourn, C.: Covering Array Tables. `http://www.public.asu.edu/~ccolbou/src/tabby/catable.html` (2013)
9. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Theory and applications of satisfiability testing. pp. 502–518. Springer (2004)
10. Fry, Z.P., Landau, B., Weimer, W.: A human study of patch maintainability. In: International Symposium on Software Testing and Analysis (ISSTA'12). Minneapolis, Minnesota, USA (July 2012), to appear
11. Futamura, Y.: Partial evaluation of computation process – an approach to a compiler-compiler. Systems, Computers, Controls 2(5), 721–728 (Aug 1971)
12. Garvin, B.J., Cohen, M.B., Dwyer, M.B.: Evaluating improvements to a meta-heuristic search for constrained interaction testing. Empirical Software Engineering 16(1), 61–102 (2011)
13. Goues, C.L., Dewey-Vogt, M., Forrest, S., Weimer, W.: A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In: International Conference on Software Engineering (ICSE 2012). Zurich, Switzerland (2012)

14. Harman, M., Langdon, W.B., Jia, Y., White, D.R., Arcuri, A., Clark, J.A.: The GISMOE challenge: Constructing the Pareto program surface using genetic programming to find better programs (keynote paper). In: $27^{th}$ IEEE/ACM International Conference on Automated Software Engineering (ASE 2012). Essen, Germany (September 2012)
15. Harman, M., Langdon, W.B., Weimer, W.: Genetic programming for reverse engineering. In: Oliveto, R., Robbes, R. (eds.) 20th Working Conference on Reverse Engineering (WCRE 2013). IEEE, Koblenz, Germany (14-17 October 2013)
16. Kibria, R.H., Li, Y.: Optimizing the initialization of dynamic decision heuristics in DPLL SAT solvers using genetic programming. In: Collet, P., Tomassini, M., Ebner, M., Gustafson, S., Ekárt, A. (eds.) EuroGP. Lecture Notes in Computer Science, vol. 3905, pp. 331–340. Springer (2006)
17. Langdon, W.B., Harman, M.: Optimising existing software with genetic programming. IEEE Transactions on Evolutionary Computation To appear
18. Langdon, W.B., Harman, M.: Evolving a CUDA kernel from an nVidia template. In: IEEE Congress on Evolutionary Computation. pp. 1–8. IEEE (2010)
19. Le Goues, C., Forrest, S., Weimer, W.: Current challenges in automatic software repair. Software Quality Journal 21(3), 421–443 (2013)
20. Lei, Y., Kacker, R., Kuhn, D.R., Okun, V., Lawrence, J.: IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing. Softw. Test., Verif. Reliab. 18(3), 125–148 (2008)
21. Nanba, T., Tsuchiya, T., Kikuno, T.: Constructing test sets for pairwise testing: A SAT-based approach. In: ICNC. pp. 271–274. IEEE Computer Society (2011)
22. Nie, C., Leung, H.: A survey of combinatorial testing. ACM Computing Surveys 43(2), 11:1 – 11:29 (2011)
23. Orlov, M., Sipper, M.: Flight of the FINCH through the Java wilderness. IEEE Transactions Evolutionary Computation 15(2), 166–182 (2011)
24. Petke, J., Cohen, M.B., Harman, M., Yoo, S.: Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In: European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13. pp. 26–36. ACM, Saint Petersburg, Russian Federation (August 2013)
25. Petke, J., Langdon, W.B., Harman, M.: Applying genetic improvement to MiniSAT. In: Proceedings of the 5th International Symposium on Search Based Software Engineering (SSBSE '13). vol. 8084, pp. 257–262. Springer, St. Petersburg, Russia (24-26 August 2013)
26. Silva, J.P.M., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 131–153. IOS Press (2009)
27. Sitthi-amorn, P., Modly, N., Weimer, W., Lawrence, J.: Genetic programming for shader simplification. ACM Trans. Graph. 30(6), 152 (2011)
28. White, D.R., Clark, J., Jacob, J., Poulding, S.: Searching for resource-efficient programs: Low-power pseudorandom number generators. In: 2008 Genetic and Evolutionary Computation Conference (GECCO 2008). pp. 1775–1782. ACM Press, Atlanta, USA (Jul 2008)
29. White, D.R., Arcuri, A., Clark, J.A.: Evolutionary improvement of programs. IEEE Transactions on Evolutionary Computation 15(4), 515–538 (2011)
30. Zeller, A.: Yesterday, my program worked. Today, it does not. Why? In: Foundations of Software Engineering. pp. 253–267 (1999)