

Report from Dagstuhl Seminar 18052

Genetic Improvement of Software

Edited by

Justyna Petke¹, Claire Le Goues², Stephanie Forrest³, and
William B. Langdon⁴

1 University College London, GB, j.petke@ucl.ac.uk

2 Carnegie Mellon University, Pittsburgh, US, clegoues@cs.cmu.edu

3 Arizona State University, Tempe, US, stephanie.forrest@asu.edu

4 University College London, GB, w.langdon@cs.ucl.ac.uk

Abstract

We document the program and the immediate outcomes of Dagstuhl Seminar 18052 “Genetic Improvement of Software”. The seminar brought together researchers in Genetic Improvement (GI) and related areas of software engineering to investigate what is achievable with current technology and the current impediments to progress and how GI can affect the software development process. Several talks covered the state-of-the-art and work in progress. Seven emergent topics have been identified ranging from the nature of the GI search space through benchmarking and practical applications. The seminar has already resulted in multiple research paper publications. Four by participants of the seminar will be presented at the GI workshop co-located with the top conference in software engineering - ICSE. Several researchers started new collaborations, results of which we hope to see in the near future.

Seminar January 28–February 2, 2018 – <https://www.dagstuhl.de/18052>

2012 ACM Subject Classification Software and its engineering → Automatic programming,
Software and its engineering → Search-based software engineering

Keywords and phrases genetic improvement GI, search-based software engineering SBSE, soft-
ware optimisation, evolutionary improvement, automated software improvement, automated
program repair, evolutionary computation, genetic programming, GP

Digital Object Identifier 10.4230/DagRep.8.1.158

Edited in cooperation with Nicolas Harrand

1 Executive Summary

Justyna Petke

Stephanie Forrest

William B. Langdon

Claire Le Goues

License  Creative Commons BY 3.0 Unported license

© Justyna Petke, Stephanie Forrest, William B. Langdon, and Claire Le Goues

Genetic improvement (GI) uses automated search to find improved versions of existing software. It can be used for improvement of both functional and non-functional properties of software. Much of the early success came from the field of automated program repair. However, GI has also been successfully used to optimise for efficiency, energy and memory consumption as well as automated transplantation of a piece of functionality from one program to another. These results are impressive especially given that genetic improvement only arose as a separate research area in the last few years. Thus the time was ripe to



Except where otherwise noted, content of this report is licensed
under a Creative Commons BY 3.0 Unported license

Genetic Improvement of Software, *Dagstuhl Reports*, Vol. 8, Issue 01, pp. 158–182

Editors: Justyna Petke, Stephanie Forrest, William B. Langdon, and Claire Le Goues



Dagstuhl Reports
REPORTS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

organise a seminar that would gather researchers from GI and related areas together to summarise the current achievements and identify avenues for further research.

The seminar attracted researchers from various GI-related software engineering areas, ranging from automated software repair through genetic programming and software testing to biological and evolutionary computation. The talks covered the latest research and speculations on future research both in the practical applications of genetic improvement, such as energy consumption optimisation and automated parallelisation, to initial results on much lacking GI theory. In particular, GI theory and indeed software in general were discussed in terms of search landscape analysis. Other talks covered software testing and bug repair. The participants also identified a set of benchmarks and tools for GI. These have been published at the geneticimprovementofsoftware.com website to allow other researchers to compare their new technologies against the state-of-the-art.

The seven breakout groups' topics ranged from re-evaluating the basic components of the GI framework, such as fitness functions and traversing the GI search space, to identifying issues related to adoption of GI in industry. One of the issues has been explanation of the automatically generated changes, which might be a roadblock in applying them in the real-world, especially safety-critical, software.

The seminar has already led to a few publications. For example, four papers accepted to the 4th International Genetic Improvement Workshop (GI-2018)¹, co-located with the International Conference on Software Engineering (ICSE), were written by one or more workshop participants. Indeed most were started in Dagstuhl. Several other collaborations have been established, with plans for visits and further research on topics identified at the seminar. We look forward to results of this work initiated at Dagstuhl.

Introduction

Genetic improvement (GI) uses automated search to find improved versions of existing software [6, 8]. It uses optimisation, machine learning techniques, particularly search based software engineering techniques such as genetic programming [2, 1, 9] to improve existing software. The improved program need not behave identically to the original. For example, automatic bug fixing improves program code by reducing or eliminating buggy behaviour, whilst automatic transplantation adds new functionality derived from elsewhere. In other cases the improved software should behave identically to the old version but is better because, for example: it runs faster, it uses less memory, it uses less energy or it runs on a different type of computer.

GI differs from, for example, formal program translation, in that it primarily verifies the behaviour of the new mutant version by running both the new and the old software on test inputs and comparing their output and performance in order to see if the new software can still do what is wanted of the original program and is now better. Using less constrained search allows not only functional improvements but also each search step is typically far cheaper, allowing GI to scale to substantial programs. Genetic improvement can be used to create large numbers of versions of programs, each tailored to be better for a particular use or for a particular computer, or indeed (e.g. to defeat the authors of computer viruses) simply to be different. Other cases where software need to be changed include porting to new environments (e.g. parallel computing [3] mobile devices) or for code obfuscation to prevent reverse engineering [7].

¹ <http://geneticimprovementofsoftware.com/>

Genetic improvement can be used with multi-objective optimisation to consider improving software along multiple dimensions or to consider trade-offs between several objectives, such as asking GI to evolve programs which trade speed against the quality of answers they give. Of course, it may be possible to find programs which are both faster and give better answers. Mostly Genetic Improvement makes typically small changes or edits (also known as mutations) to the program's source code, but sometimes the mutations are made to assembly code, byte code or binary machine code.

GI arose as a separate field of research only in the last few years. Even though its origins could be traced back to the work by Ryan & Walsh [18] in 1995, it is the work by Arcuri [10] and White [20] that led to the development and wider uptake of the GI techniques. The novelty lay in applying heuristics to search for code mutations that improved existing software. Both Arcuri and White applied genetic programming (GP), with Arcuri using also hill-climbing and random search on a small set of problems. Rather than trying to evolve a program from scratch, as in traditional GP, Arcuri and White took the approach of seeding [5] the initial population with copies of the original program. Next, instead of focusing on evolving a program fulfilling a particular task, as has been done before, Arcuri and White used GP to improve their programs either to fix existing bugs or to improve the non-functional properties of software, in particular, its efficiency and energy consumption. Both Arcuri and White, however, applied their, now known as, GI techniques, to relatively small benchmarks having little resemblance to large scale real-world problems.

The bug fixing approach was taken up by Forrest, Le Goues and Weimer et al. [12, 15, 19] and adapted for large software systems. One of the insights that allowed for this adoption was an observation that full program variants need not be evolved, yet only a sequence of edits, which are then applied to the original program. Validity of the resultant modified software was then evaluated on a set of test cases, assumed to capture desired program behaviour, as in previous work. This strand of research led to the development of first GP-based automated software repair tool called GenProg [15]. Success of this automated bug fixing work led to several best paper awards and two 'Humie' awards (international prizes for human-competitive results produced by genetic and evolutionary computation <http://www.human-competitive.org/>) and inspired work on other automated software repair tools, including Angelix [16], which uses a form of constraint solving to synthesise bug fixes.

Research on improvement of non-functional software properties has yet to garner the attention and software development effort as the work on automated bug fixing. Langdon et al. [3, 13, 14] published several articles on efficiency improvement and parallelisation using GI. They were able to improve efficiency of large pieces of state-of-the-art software. Moreover, the genetically improved version of a bioinformatics software called BarraCUDA is the first instance of a genetically improved piece of software adapted into development [14, 4].

Petke et al. [17] set themselves a challenge of improving efficiency of a highly-optimised piece of software that has been improved by expert human developers over a period of several years. In particular, a famous Boolean satisfiability (SAT) solver was chosen, called MiniSAT. It implements the core technologies of SAT solving and inspired a MiniSAT-hack track at the annual international SAT solver competitions, where anyone can submit their own version of MiniSAT. Petke et al. showed that further efficiency improvements can be made by using this source of genetic material for the GP process and specializing the solver for a particular downstream application. This work showed the initial potential of what is now called automated software transplantation and was awarded a Silver 'Humie'. Further work on automated software transplantation won an ACM SIGSOFT distinguished paper award and a Gold 'Humie' at this year's Genetic and Evolutionary Computation Conference (GECCO-2017) [11].

Aims of the Seminar

The seminar brought together researchers in this new field of software engineering to investigate what is achievable with current technology and the current impediments to progress (if indeed there are any) of what can be achieved within the field in the future and how GI can affect the software development process.

With the growing popularity of the field, multiple awards and fast progress GI research in the field, it is the right time to gather top the academics in GI and related fields to push the boundaries of what genetic improvement can achieve even further.

This seminar brought researchers working in genetic improvement and related areas, such as automated program repair, software testing and genetic programming, together. It summarized achievements in automated software optimisation. We will use this summary as a basis to investigate how optimisation approaches from the different fields represented at the seminar can be combined to produce a robust industry-ready set of techniques for software improvement.

References

- 1 Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, San Francisco, CA, USA, January 1998.
- 2 John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.
- 3 W. B. Langdon and M. Harman. Evolving a CUDA kernel from an nVidia template. In Pilar Sobrevilla, editor, *2010 IEEE World Congress on Computational Intelligence*, pages 2376–2383, Barcelona, 18–23 July 2010. IEEE.
- 4 W. B. Langdon and Brian Yee Hong Lam. Genetically improved BarraCUDA. *BioData Mining*, 20(28), 2 August 2017.
- 5 W. B. Langdon and J. P. Nordin. Seeding GP populations. In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP’2000*, volume 1802 of *LNCS*, pages 304–315, Edinburgh, 15–16 April 2000. Springer-Verlag.
- 6 William B. Langdon. Genetically improved software. In Amir H. Gandomi, Amir H. Alavi, and Conor Ryan, editors, *Handbook of Genetic Programming Applications*, chapter 8, pages 181–220. Springer, 2015.
- 7 Justyna Petke. Genetic improvement for code obfuscation. In Justyna Petke, David R. White, and Westley Weimer, editors, *Genetic Improvement 2016 Workshop*, pages 1135–1136, Denver, July 20–24 2016. ACM.
- 8 Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. Woodward. Genetic improvement of software: a comprehensive survey. *IEEE Transactions on Evolutionary Computation*. In press.
- 9 Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming*. Published via <http://lulu.com> and freely available at <http://www.gp-field-guide.org.uk>, 2008. (With contributions by J. R. Koza).
- 10 Andrea Arcuri. Automatic software generation and improvement through search based techniques. PhD. Univ. of Birmingham, 2009.
- 11 Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. Automated software transplantation. In *ISSTA*, pages 257–269, 2015.
- 12 Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In *GECCO*, pages 947–954, 2009.
- 13 William B. Langdon and Mark Harman. Optimizing existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*, 19(1):118–135, 2015.

- 14 William B. Langdon, Brian Yee Hong Lam, Justyna Petke, and Mark Harman. Improving CUDA DNA analysis software with genetic programming. In *GECCO*, pages 1063–1070, 2015.
- 15 Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for \$8 each. In *ICSE*, pages 3–13, 2012.
- 16 Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *ICSE*, pages 691–701, 2016.
- 17 Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. Using genetic improvement and code transplants to specialise a C++ program to a problem class. In *EuroGP*, pages 137–149, 2014.
- 18 Paul Walsh and Conor Ryan. Automatic conversion of programs from serial to parallel using genetic programming - the Paragen system. In *ParCo*, pages 415–422, 1995.
- 19 Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *ICSE*, pages 364–374, 2009.
- 20 David R. White. Genetic programming for low-resource systems. PhD. Univ. of York, 2009.

2 Table of Contents

Executive Summary

Justyna Petke, Stephanie Forrest, William B. Langdon, and Claire Le Goues . . . 158

Overview of Talks

Progress in Structural Evolution for Bug Repair in JAVA <i>Wolfgang Banzhaf</i>	165
Automatic Parallelisation of Software Using Genetic Improvement <i>Bobby R. Bruce</i>	165
Assuring Organic Programs <i>Myra B. Cohen</i>	166
Genetic-Improvement of Test suite <i>Benjamin Danglot</i>	166
Software Plasticity <i>Nicolas Harrand</i>	166
DeepBugs: Learning to Find Bugs <i>Michael Pradel</i>	167
Analyzing Neutrality in Program Space <i>Joseph Renzullo</i>	168
Approximate computing <i>Lukas Sekanina</i>	168
An Actionable Performance Profiling for JavaScript <i>Marija Selakovic</i>	168
Repairing crashes in Android apps <i>Shin Hwei Tan</i>	169
BugZoo: A platform for studying historical bugs <i>Christopher Timperley</i>	169
Major Transitions in Information Technology <i>Sergi Valverde</i>	170

Working groups

Pseudo Neutrality <i>Benoit Baudry</i>	170
Genetic Improvement for DevOps <i>Nicolas Harrand</i>	172
Benchmarks and Corpora <i>Myra B. Cohen, William B. Langdon, and Claire Le Goues</i>	173
Energy Breakout Session <i>Markus Wagner</i>	173
Diversity <i>Wolfgang Banzhaf</i>	174

Comprehensibility and Explanation <i>Colin Johnson</i>	175
Fitness Functions for Genetic Improvement <i>Brad Alexander</i>	177
Resources for Genetic Improvement	
Tools, Libraries and Frameworks	179
Benchmarks	180
Following the Seminar, New work and New Connections	
New Work	180
New Connections	181
Participants	182

3 Overview of Talks

3.1 Progress in Structural Evolution for Bug Repair in JAVA

Wolfgang Banzhaf (Michigan State University, US)

License © Creative Commons BY 3.0 Unported license

© Wolfgang Banzhaf

Joint work of Wolfgang Banzhaf, Yuan Yuan (MSU CSE)

Main reference Yuan Yuan, Wolfgang Banzhaf: “ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming”, CoRR, Vol. abs/1712.07804, 2017.

URL <http://arxiv.org/abs/1712.07804>

Here we argue that (virtually) any structure can be made evolvable if one chooses the right elements of the structure and the proper rules of their combination, and provides sufficient guidance for the randomness of the mutation and crossover operators. We exemplify that argument by proposing a JAVA bug repair system that was inspired by GenProg, but further developed and adapted to JAVA. Results show the efficacy of evolutionary search over random search, multi-objective optimisation over single objective optimisation and “knowledge-enhanced” (smart) operators over others. A new set of bugs from the Defects4J benchmark suit can be successfully repaired, including multi-location bugs.

3.2 Automatic Parallelisation of Software Using Genetic Improvement

Bobby R. Bruce (University College London, GB)

License © Creative Commons BY 3.0 Unported license

© Bobby R. Bruce

Joint work of Bobby R. Bruce, Justyna Petke

While the use of hardware accelerators, like GPUs, can significantly improve software performance, developers often lack the expertise or time to properly translate source code to do so. We highlight two approaches to automatically offload computationally intensive tasks to a system’s GPU by generating and inserting OpenACC directives; one using grammar-based genetic programming, and another using a bespoke four stage process. We find that the grammar-based genetic programming approach is capable of reducing execution time by 2.60% on average, across the applications studied, while the bespoke four-stage approach reduces execution time by 2.44%.

However, our investigation shows a handwritten OpenACC implementation is capable of reducing execution time by 65.68%, suggesting our techniques could be improved upon. Comparing the differences, we find our techniques do not handle data to and from the GPU in a sensible manner and that, if they did, considerably execution time savings are possible. We therefore advise future researchers to focus on the automation of transferring data between main and GPU memory; a problem search-based software engineering is capable of solving.

3.3 Assuring Organic Programs

Myra B. Cohen (University of Nebraska, Lincoln, US)

License  Creative Commons BY 3.0 Unported license
 © Myra B. Cohen

Joint work of Myra, B. Cohen, Justin Firestone, Massimiliano Pierobon
Main reference Myra B. Cohen, Justin Firestone, Massimiliano Pierobon: "The Assurance Timeline: Building Assurance Cases for Synthetic Biology", in Proc. of the Computer Safety, Reliability, and Security - SAFECOMP 2016 Workshops, ASSURE, DECSoS, SASSUR, and TIPS, Trondheim, Norway, September 20, 2016, Proceedings, Lecture Notes in Computer Science, Vol. 9923, pp. 75–86, Springer, 2016.

URL https://doi.org/10.1007/978-3-319-45480-1_7

Recent research in genetic improvement and self-adaptation have created a class of programs that we call organic, since they follow an evolution cycle similar to that of living organisms. Traditional testing techniques assume that program modifications are planned, systematic and well understood. However, this may not be true for organic programs. I discuss the use of an assurance case to argue about the dependability and safety of an organic program using an exemplar from synthetic biology (which are living organic programs). I present an orthogonal dimension to an assurance case, the assurance timeline, which aims to reason about the dynamic, evolving aspects of these systems.

3.4 Genetic-Improvement of Test suite

Benjamin Danglot (INRIA Lille, FR)

License  Creative Commons BY 3.0 Unported license
 © Benjamin Danglot

In the literature there is a rather clear segregation between tests manually written by developers and automatically generated ones. DSpot explores a third solution: automatically improving existing test cases written by developers. DSpot takes as input developer-written tests and synthesizes an improved version. Those improvements are given to the developer as a pull-request than can be directly integrated into their code-base. DSpot uses mutation operators on the code of each test, it produces assertions and selects them according to a given test criterion such as coverage. In 26/40 cases, DSpot has been able to create a better version of a test class. We proposed pull requests to real developers and 7 of them have been added permanently to their test suite.

3.5 Software Plasticity

Nicolas Harrand (KTH Royal Institute of Technology, Stockholm, SE)

License  Creative Commons BY 3.0 Unported license
 © Nicolas Harrand

Joint work of Benoit Baudry, Nicolas Harrand

Approximate computing, automatic diversification and genetic improvement are techniques that all rely on *speculative transformations*: transformations that aim at producing variants of a program that are functionally similar to the original, yet execute slightly differently. The intuition of all the techniques cited above potential enhancements lie in these acceptable behavioural differences (enhanced performance, security, reliability, etc.).

The design of the speculative transformations that can yield these improvements remains a critical challenge. These transformations must target regions of programs that can tolerate changes in the execution flow, while maintaining the correctness of the program. We call them *plastic code regions*. We contribute with fundamental new knowledge about these regions in object-oriented programs, as well as with new kinds of speculative transformations that directly exploit this new knowledge.

Our empirical inquiry of plastic code regions starts from a random exploration of three classical speculative transformations: add, replace and delete statements. We synthesize 24 583 variants from 6 real-world Java programs, and focus our analysis on the 5305 that are similar, modulo test suite, to the original. Our key insights about plastic regions are as follows: developers naturally write code that supports fine-grain behavioural changes; statement deletion is a surprisingly effective; high-level design decisions, such as the choice of a data structure, are natural points that can evolve while keeping functionality. Based on these new findings, we design targeted speculative transformations and show that they are very effective at producing variants that are both similar (modulo tests) and different from the original.

3.6 DeepBugs: Learning to Find Bugs

Michael Pradel (TU Darmstadt, DE)

License  Creative Commons BY 3.0 Unported license
© Michael Pradel

Joint work of Michael Pradel, Koushik Sen

Automated bug detection, e.g., through pattern-based static analysis, is an increasingly popular technique to find programming errors and other code quality issues. Traditionally, bug detectors are program analyses that are manually written and carefully tuned by an analysis expert. Unfortunately, the huge amount of possible bug patterns makes it difficult to cover more than a small fraction of all bugs. I present a new approach toward creating bug detectors. The basic idea is to replace manually writing a program analysis with training a machine learning model that distinguishes buggy from non-buggy code. To address the challenge that effective learning requires both positive and negative training examples, we use simple code transformations that create likely incorrect code from existing code examples. We present a general framework, called DeepBugs, that extracts positive training examples from a code corpus, leverages simple program transformations to create negative training examples, trains a model to distinguish these two, and then uses the trained model for identifying programming mistakes in previously unseen code. As a proof of concept, we create four bug detectors for JavaScript that find a diverse set of programming mistakes, e.g., accidentally swapped function arguments, incorrect assignments, and incorrect binary operations. To find bugs, the trained models use information that is usually discarded by program analyses, such as identifier names of variables and functions. Applying the approach to a corpus of 150,000 JavaScript files shows that learned bug detectors have a high accuracy, are very efficient, and reveal 132 programming mistakes in real-world code.

3.7 Analyzing Neutrality in Program Space

Joseph Renzullo (Arizona State University, Tempe, US)

License  Creative Commons BY 3.0 Unported license

© Joseph Renzullo

URL <https://docs.google.com/presentation/d/18-0b4Mdnvum28IRCCLUb966Gb2VDHUf0D88DhlmakGI/edit?usp=sharing>

I present evidence of interaction between multiple edits (both positive and negative epistasis) in the region near the original program. There are a few cases where repairs were found which were attributed to multiple independent patches working in combination (previous results have shown that these often minimise to one patch) here we show evidence that this is not always the case.

Additionally, I raise questions about how methods in biology (particularly borrowing from theoretical biology) may be used to characterise (and hopefully exploit) the topology of neutral space.

3.8 Approximate computing

Lukas Sekanina (Brno University of Technology, CZ)

License  Creative Commons BY 3.0 Unported license

© Lukas Sekanina

Joint work of Lukas Sekanina, Zdenek Vasicek, Vojtech Mrazek

Main reference Vojtech Mrazek, Syed Shakib Sarwar, Lukás Sekanina, Zdenek Vasícek, Kaushik Roy: “Design of power-efficient approximate multipliers for approximate artificial neural networks”, in Proc. of the 35th International Conference on Computer-Aided Design, ICCAD 2016, Austin, TX, USA, November 7-10, 2016, p. 81, ACM, 2016.

URL <http://dx.doi.org/10.1145/2966986.2967021>

A new design paradigm—approximate computing—was established to investigate how computer systems can be made better (e.g. more energy efficient, faster, and less complex) by relaxing the requirement that they are exactly correct. We provide a brief introduction to approximate computing and indicates how evolutionary computation, in general, and genetic improvement, in particular, can be employed to provide requested approximations. An important case study is presented in the area of evolutionary approximation of multipliers (which are key to performance) for deep neural networks.

3.9 An Actionable Performance Profiling for JavaScript

Marija Selakovic (TU Darmstadt, DE)

License  Creative Commons BY 3.0 Unported license

© Marija Selakovic

Many programs suffer from performance problems, but unfortunately, finding and fixing such problems is a cumbersome and time-consuming process. My work focuses on JavaScript, for which little is known about performance issues and how developers address them. To address these questions, I present the main findings from the empirical study of ≈ 100 reproduced performance-related issues from popular JavaScript projects. To help developers find and fix recurrent performance issues I present two profiling approaches. The first approach focuses on detecting finding the optimal order of checks in logical expressions and switch statements

and proposing beneficial changes to the developers. Optimizing the order of evaluations reduces the execution time of individual functions by between 2.5% and 59%, and leads to statistically significant application-level performance improvements that range between 2.5% and 6.5%. The second approach helps developers find and fix performance problems related to API usages. The technique focuses on finding conditionally-equivalent but performance-wise different APIs. Our evaluation with 939 APIs from 8 popular JavaScript libraries shows the prevalence of conditionally equivalent APIs. In particular, out of 217 API pairs that are equivalent for a subset of all inputs, our technique derives an equivalence condition for 149 pairs. Furthermore, it finds that 147 API pairs have different performance, enabling developers to exploit conditional equivalences to speed up their code.

3.10 Repairing crashes in Android apps

Shin Hwei Tan (National University of Singapore, SG)

License  Creative Commons BY 3.0 Unported license
 Shin Hwei Tan

Joint work of Shin Hwei Tan, Zhen Dong, Xiang Gao, Abhik Roychoudhury

Main reference Shin Hwei Tan, Zhen Dong, Xiang Gao, Abhik Roychoudhury: “Repairing Crashes in Android Apps”, in Proc. of the ACM/IEEE Int'l Conf. on Software Engineering (ICSE).., To Appear, 2018.

Android apps are omnipresent, and frequently suffer from crashes. This leads to poor user experience and loss of revenue. Past work has focused on automated test generation to detect crashes in Android apps. However automated repair of crashes has not been studied. We propose the first approach to automatically repair Android apps, specifically we propose a technique for fixing crashes in Android apps. Unlike most test-based repair approaches, we do not need a test-suite; instead a single failing test is meticulously analyzed for crash locations and reasons behind these crashes. Unlike most test-based repair approaches, we do not need a test-suite; instead a single failing test is meticulously analyzed for crash locations and reasons behind these crashes. Our approach hinges on a careful empirical study which seeks to establish common root-causes for crashes in Android apps, and then distils the remedy of these root-causes in the form of eight generic transformation operators. These operators are applied using a search-based repair framework embodied in our repair tool *Droix*. We also prepare a benchmark *DroixBench* capturing reproducible crashes in Android apps. Our evaluation of *Droix* on *DroixBench* reveals that the automatically produced patches are often syntactically identical to the human patch, and on some rare occasion even better than the human patch (in terms of avoiding regressions). These results confirm our intuition that our proposed transformations form a sufficient set of operators to patch crashes in Android.

3.11 BugZoo: A platform for studying historical bugs

Christopher Timperley (Carnegie Mellon University, Pittsburgh, US)

License  Creative Commons BY 3.0 Unported license
 Christopher Timperley
URL <https://github.com/squaresLab/BugZoo>

I introduce BugZoo to the genetic improvement community: BugZoo is an open-source platform for studying historical software bugs that helps researchers to conduct high-quality reproducible experiments. BugZoo can be used to conduct experiments in a diversity of

fields including but not limited to software testing, program repair, genetic improvement, fault localisation, and program analysis. By providing a rich API, a decentralised means of distribution bugs, and a controlled execution environment, BugZoo makes it faster and easier to perform research.

3.12 Major Transitions in Information Technology

Sergi Valverde (UPF, Barcelona, ES)

License  Creative Commons BY 3.0 Unported license

© Sergi Valverde

Main reference Sergi Valverde: “Major Transitions in Information Technology”, Phil. Trans. R. Soc. B, 371: 20150450, 2016.

URL <http://dx.doi.org/10.1098/rstb.2015.0450>

When looking at the history of technology, we can see that all inventions are not of equal importance. Only a few technologies have the potential to start a new branching series (specifically, by increasing diversity), have a lasting impact in human life and ultimately became turning points. Technological transitions correspond to times and places in the past when a large number of novel artefact forms or behaviours appeared together or in rapid succession. Why does that happen? Is technological change continuous and gradual or does it occur in sudden leaps and bounds? The evolution of information technology (IT) allows for a quantitative and theoretical approach to technological transitions. The value of information systems experiences sudden changes (i) when we learn how to use this technology, (ii) when we accumulate a large amount of information, and (iii) when communities of practice create and exchange free information. The coexistence between gradual improvements and discontinuous technological change is a consequence of the asymmetric relationship between complexity and hardware and software. Using a cultural evolution approach, we suggest that sudden changes in the organization of ITs depend on the high costs of maintaining and transmitting reliable information.

4 Working groups

4.1 Pseudo Neutrality

Benoit Baudry (KTH Royal Institute of Technology, Stockholm, SE)

License  Creative Commons BY 3.0 Unported license
© Benoit Baudry

The synthesis of pseudo-neutral program variants is a key challenge for genetic improvement. Given an original program that one wishes to improve, pseudo-neutral variants are those programs that are functionally similar to the original, yet exhibit some differences in their behaviour. More precisely, given a program P that passes all the tests in TS , we wish to generate variants of P that are synthesised by transformation τ and that are such that

- $\tau(P)$ passes all tests in TS , i.e., P and $\tau(P)$ are equivalent modulo TS
- $\{traces\}_P \neq \{traces\}_{\tau(P)}$ i.e., P and $\tau(P)$ are semantically different

This definition is summarised in Figure 1

Key insight: pseudo-neutral program variants exist!

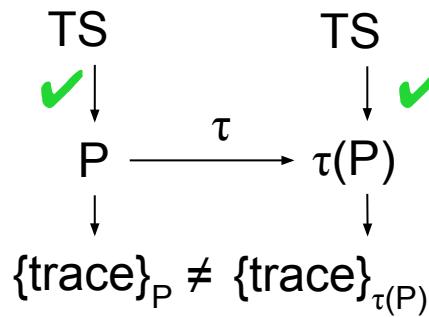


Figure 1 Pseudo-neutral program variants.

We have strong empirical evidence of their existence in large quantities and in many languages (Java, C and assembly code) [1, 2]. Our results also demonstrate that the existence of these variants is independent of the strength of the test suite TS that used to assess the functional similarity between variants.

It is important to note that these program variants are not equivalent mutants in the sense of mutation testing. Indeed, as illustrated in Figure 1, the execution traces vary between the original and the transformed program. This means that the behaviour are not equivalent and that the transformed program is not equivalent to the original.

4.1.1 Challenges

The variants and the original must have different traces for the same input. Yet, there are many ways to capture the traces (function calls, system calls, states, etc.). The question about what is the most appropriate or relevant level to capture traces is still open.

The synthesis of variants relies on transformations on the code (syntactic changes), yet the goal is to produce semantic variations. One challenge here is to know how to predict the semantic impact of a syntactic change. A similar questions is to know what makes software prone to the existence of these pseudo mutational robust variants.

Can we design an experiment to explore whether the following biological phenomenon holds in software: adding levels of complexity enhances robustness and evolvability in a multilevel genotype-phenotype map.

References

- 1 Benoit Baudry, Simon Allier, and Martin Monperrus. Tailored source code transformations to synthesize computationally diverse program variants. In *Proc. of the Int. Symp. on Software Testing and Analysis (ISSTA)*, pages 149–159, CA, USA, 2014.
- 2 Eric Schulte, Zachary Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. Software mutational robustness. *Genetic Programming and Evolvable Machines*, 15(3):281–312, September 2014.

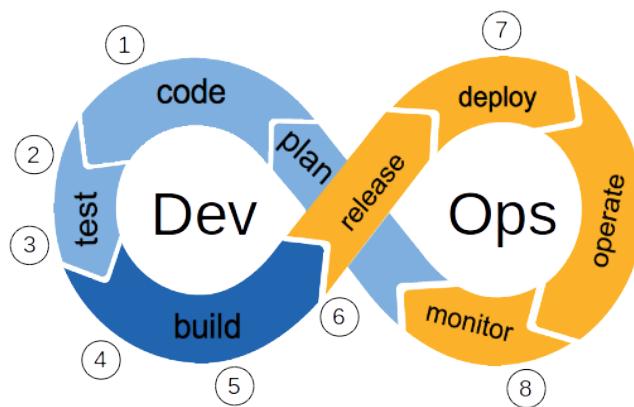


Figure 2 DevOps Software life cycle.

4.2 Genetic Improvement for DevOps

Nicolas Harrand (KTH Royal Institute of Technology, Stockholm, SE)

License © Nicolas Harrand

We discussed the opportunities to deploy genetic improvement offered by DevOps [2]. DevOps aims to close the loop between development and operation of software. To achieve this goal, it relies heavily on automation of the software construction process. In this inclination toward automation and the resulting problems, lie many opportunity to integrate Genetic Improvement into the software construction pipeline. Among them, we identified the following:

1. Fixing merge conflicts
2. Genetically improve the test suite
3. Use GI to fix flaky tests
4. Integrate bug repair into Continuous Integration (CI) tools.
5. Automatic fixes in dependency conflicts
6. Container minimization
7. Deployment of a diverse population of software
8. Use of monitoring feedback for further improvements

The discussion resulted in the publication of a workshop paper listing and detailing these different opportunities [1].

References

- 1 Benoit Baudry, Nicolas Harrand, Eric Schulte, Chris Timperley, Shin Hwei Tan, Marija Selakovic, Emamurho Ugherughe. A spoonful of DevOps helps the GI go down. In *Proceedings of Workshop on Genetic Improvement (GI 2018)*, pp. 35–37, Gothenburg, Sweden, June 2018. ACM.
- 2 Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. Devops. *IEEE Software*, 33(3):94–100, 2016.

4.3 Benchmarks and Corpora

Myra B. Cohen (University of Nebraska, Lincoln, US), William B. Langdon (University College London, GB), and Claire Le Goues (Carnegie Mellon University, Pittsburgh, US)

License  Creative Commons BY 3.0 Unported license
© Myra B. Cohen, William B. Langdon, and Claire Le Goues

This working group discussed the need for benchmarks and a corpus of programs that have been created through genetic improvement. The group came up with two dimensions of this problem, (1) **benchmarking** and (2) **building a corpus**:

1. **Benchmarking**, i.e. providing programs and example bugs/functionality, etc. for others to evaluate their GI techniques. Some benchmarks already exist.
Potential issue: There is a risk that people will overfit their techniques to these benchmarks.
Types of Artefacts that we should collect:
 - Failing and passing test cases (or other witnesses of desired/undesired behaviour)
 - Program with bugs and set of properties of interest
 - Patches: This would include a patch and undo approach (always return to base model to re-patch)
2. **Corpus**, i.e. providing the artefacts from GI throughout a program's history which includes the program, the patches, the new programs, etc.
Some people may not want to re-run the programs and build their own artefacts. This provides programs and other GI artefacts for them to study.
Artefacts:
 - Base program
 - History of the evolved program and patches
 - This assumes each patch builds on another
 - Includes patches for bugs, optimisation and transplantation
 - Can be used to mine information, understand the artefacts

4.4 Energy Breakout Session

Markus Wagner (University of Adelaide, AU)

License  Creative Commons BY 3.0 Unported license
© Markus Wagner

The minimisation of energy consumption is an important challenge in many domains. We focussed on two domains: electric circuits and mobile phones.

Researchers who develop new circuits can often rely on good models. Some are lightweight and provide estimates based on switching activity and gate size analysis, and they are often good enough to reliably drive tournament selection. For the final functional validation, either an actual implementation is used, or SAT solvers. Vasicek and Sekanina [1] applied simple and so cheap area and delay estimation techniques during the evolutionary approximation of digital circuits. Parameters of best-evolved circuits were then verified by means of a professional circuit design tool. The quality of estimated values was sufficient for their purposes. Mrazek et al. [2] applied these estimation techniques in evolutionary design/improvement of specialised multipliers for deep neural networks.

When dealing with mobile phones, there are software engineering issues to solve before getting to the optimisation problem. In addition, complex interactions on the actual phone make it difficult to consistently see the benefit of a change. Possible optimisation approaches range from working on the hardware (voltage schedules, frequency adjustments) to code changes. The group discussed various targets: screen, communication, GPS, and code. It was not clear to the group if hardware-in-the-loop is necessary for the evolution, although the group identified cases where the creation of sufficiently precise models is out of question. When it comes to in-vivo optimisation, then the processes need to deal with large amounts of noise from various resources. This is immensely prevalent when attempting to optimise communication. Also, the use of external power meters is becoming increasingly difficult as the phone's communication with the battery is hard to mimic. Bokhari et al. [3,4] characterised noise and challenges, and performed multi-objective configuration optimisation on Android 6 devices.

In-vivo optimisation is interesting when the target device's exact configuration is now known. Recently, Yoo et al. [5,6] demonstrated that this is possible for performance optimisation.

References

- 1 Vasicek Zdenek and Sekanina Lukas. Circuit Approximation Using Single and Multi-Objective Cartesian GP. In: EuroGP. Springer, 2015, pp. 217–229.
- 2 Mrazek Vojtech, Sarwar Syed Shakib, Sekanina Lukas, Vasicek Zdenek and Roy Kaushik. Design of Power-Efficient Approximate Multipliers for Approximate Artificial Neural Networks. In: Proceedings of the IEEE/ACM International Conference on Computer-Aided Design. Austin, USA, 2016, pp. 811–817.
- 3 Mahmoud A. Bokhari, Bobby R. Bruce, Brad Alexander, and Markus Wagner. 2017. Deep parameter optimisation on Android smartphones for energy minimisation: a tale of woe and a proof-of-concept. In GECCO-2017. pp. 1501–1508.
- 4 Mahmoud A. Bokhari, Yuanzhong Xia, Bo Zhou, Brad Alexander, Markus Wagner. Validation of Internal Meters of Mobile Android Devices. 2017. <https://arxiv.org/abs/1701.07095>
- 5 Jeongju Sohn and Seongmin Lee and Shin Yoo. Deep Parameter Optimisation of GPGPU Work Group Size for OpenCV. In: SSBSE 2016, LNCS 9962, 211–217. Springer.
- 6 Shin Yoo. Amortised Optimisation of Non-functional Properties in Production Environments. In: SSBSE 2015, LNCS 9275, pp 31–46. Springer.

4.5 Diversity

Wolfgang Banzhaf (Michigan State University, US)

License  Creative Commons BY 3.0 Unported license
© Wolfgang Banzhaf

With the prevalence of neutrality in computer code, we agreed that the more important issue is how to create diversity in a population. As we start evolution by a working program that needs to be improved, this is an issue, since we come from a situation where there is a solution, but one which we want to further improve on. So, how do we create diversity, since we are not allowed to just randomly create programs?

There was discussion about the fact that the neutral networks are actually quite intricately connected. So would diversity actually be so important?

As far as neutrality is concerned, many mentioned the issue of really very flat fitness landscapes. Where would there be a signal for improvements? Two measures where emphasised for avoiding getting stuck on the plain:

1. Random subset selection
2. Co-evolutionary approaches

4.6 Comprehensibility and Explanation

Colin Johnson (University of Kent, UK)

License  Creative Commons BY 3.0 Unported license
© Colin Johnson

An important issue in applying genetic improvement in practical software development is convincing developers to take up the improvements suggested by GI systems. This can be tackled in a number of different ways. For example, running the modified programs on test data can be used both to check whether test cases are still satisfied post-improvement, and to measure improvements to non-functional properties. Another approach is to apply static analysis and verification techniques to GI-modified programs to examine properties. A third approach, which we focus on here, is that of making modified code comprehensible to human programmers, and for the GI system to provide human-comprehensible explanations and annotations for developers.

4.6.1 Human Readability

One way to make improvements convincing is to make changes so that a human programmer can easily read the results from the suggested improvement. This could in part be achieved by keeping code changes small and focused (perhaps only altering code regions specified by the developer), and avoiding side-effects of genetic operations such as code bloat. A related, almost opposite, issue, is avoiding the GI system making excessively “clever” convoluted improvements that might use unusual language or API features or use language in a non-idiomatic way. One approach to this would use some notion of robustness, i.e. measuring whether syntactically-similar programs have similar behaviour. Another might use an approach inspired by economics or ecology, giving the GI system a fixed budget of changes to use. A final approach might be capturing, measuring, and optimising for the notion of idiomatic code, the kind of code that humans use.

4.6.2 Explainability

Another approach to making improvements convincing is for the GI system to generate an explanation for the improvement alongside the improvement itself. At a simple level, this explanation could consist of giving some examples that exemplify the improvement; for example, in a fault-fixing system, examples of test cases that are now satisfied that weren’t before the fix. A harder challenge is to provide a higher-level explanation of the improvements made, particularly having the system explain the overall effect of many small changes to the code. This might come from some analysis of the improvement process, or by some post-improvement comparative analysis of the improved code against the original code.

4.6.3 Comprehensibility as Improvement

Alternatively, human-comprehensibility might be the aim of a GI process. A GI process might aim to refactor code to bring it into a common style, for example in the use of consistent names, common code idioms, exception handling. Another related area, which had already been explored somewhat in the literature, is optimising code against measures of code complexity, for example cohesion and coupling in Object-Oriented systems. A related idea would be to use GI to refactor code to use common design patterns. Another issue is about changing the granularity of code: breaking a single expression into sub-components to allow more fine grained change/tuning; or, abstracting away from detailed code into a higher-level framework, replacing detailed code with a macro or API call.

4.6.4 Trade-offs

We can consider how we might trade off comprehensibility against other properties, particularly non-functional properties. Perhaps, given a sufficiently reliable GI system, we could see a system that allowed rapid automated refactoring of code: for example, a human-readable piece of code being transformed into an energy-efficient one for deployment, then changed back into a human-comprehensible one for a developer to make improvements, then into a more evolvable structure for the computer to make different kinds of improvements, ... Finally, there is the difficult issue of the effect of GI and other code-transformation methods on developer's mental models of code. However readable the code is in isolation, there is still the issue of how much a set of changes breaks a specific developer's understanding of how their specific piece of code works.

References

- 1 Raymond P.L. Buse and Westley R. Weimer. Automatically documenting program changes. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 33–42, New York, NY, USA, 2010. ACM.
- 2 Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Commun. ACM*, 59(5):122–131, April 2016.
- 3 Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press.
- 4 Katsuhisa Maruyama and Ken-ichi Shima. Automatic method refactoring using weighted dependence graphs. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 236–245, New York, NY, USA, 1999. ACM.
- 5 Mark O'Keeffe and Mel O Cinneide. Search-based software maintenance. In *Proceedings of the Conference on Software Maintenance and Reengineering*, CSMR '06, pages 249–260, Washington, DC, USA, 2006. IEEE Computer Society.
- 6 Ali Ouni, Marouane Kessentini, Mel O Cinneide, Houari Sahraoui, Kalyanmoy Deb, and Katsuro Inoue. More: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells. *Journal of Software: Evolution and Process*, 29(5):e1843.
- 7 Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the “naturalness” of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 428–439, New York, NY, USA, 2016. ACM.
- 8 Tushar Sharma. Identifying extract-method refactoring candidates automatically. In *Proceedings of the Fifth Workshop on Refactoring Tools*, WRT '12, pages 50–53, New York, NY, USA, 2012. ACM.

4.7 Fitness Functions for Genetic Improvement

Brad Alexander (*University of Adelaide, AU*)

License  Creative Commons BY 3.0 Unported license
© Brad Alexander

4.7.1 Test suite selection

This topic examines the potential to make use of improved test suites and other data to provide for faster and better-informed search. The motivations of this topic are observed problems in terms flat fitness landscapes for some applications such as defect repair. More generally, there is also the problem that full evaluation all tests for each fitness evaluation impacts on the speed of search.

4.7.1.1 Reducing evaluation times

One approach (Langdon) to improving the speed of each evaluation is to select a representative subset of tests for each fitness evaluation [1]. For the GI objective of reducing execution time the representative subset might be three tests, one short-lived, one of intermediate length, and one long-running. Run the short lived one first and, if that fails, don't necessarily bother with the others.

Questions arising from this approach include by how much this improves the speed of search? Some of the trade-offs have already been studied in: [3]. This showed that, when coupled with a more informed fitness landscape, being selective in the tests run can greatly speed search in the domain of defect repair (more on this below). Other suggestions included using a steady state GA to minimise the number of fitness evaluations [8].

More broadly, some proposals for choosing representative test samples included (Joseph) Eigentest weighting for the most representative sample of tests. (Celso) The literature on test selection in search-based software engineering (SBSE) is quite strong [6]. There is still work to be done on productive strategies to use for particular GI objectives.

Another approach to minimizing test suite evaluations time is to only apply tests that exercise the code that is changed by a patch. This approach is used in industry. Is it used in GI?

4.7.1.2 Boosting approaches

For the objective of defect repair (Joseph) one approach is to favor subsets of tests that are most likely to fail at the current stage of search. This helps shape the search toward overcoming challenging cases first. This boosting approach has been long-used in Genetic Programming [7]. This approach has also been applied in the improvement of search spaces in automated program repair [9].

There was also some discussion of the potential benefits of applying subsets of tests to individuals in terms of preserving useful genetic material. That is, individuals that might, when applied to all tests, achieve low fitness could still have useful materials that contributes to solutions through its progeny. It has been observed (Joseph) that such individuals, if allowed to persist can act as repositories for useful material – if the sub-set of tests two which they are exposed allow them to survive. This approach mirrors Lee Spector's Lexicase testing approach – randomly select a test case to select parents – only if there is a tie do we select a second case and so on (see: [5] for recent study).

4.7.1.3 Test feature selection

Questions arising from this approach include by how much this improves the speed of search? Is this approach sensitive to the selection of test features. In this context, a test feature is some intrinsic or manifest property of the test. Examples of test features might be: the length of time that it takes to run a test; the current likelihood that a given test will fail relative to the population of variants; and the coverage spectrum of a particular test.

4.7.1.4 Specialised domains

Improving tests involving GUI interactions, (e.g. for mobile devices) is a concern for GI objectives such as energy optimisation. The Monkey test generator generates shallow traversals of GUI interfaces due to its unguided nature. In contrast Sapienz does well in traversing through interfaces of mobile devices but can sometimes aggressively generate states that app programmers might consider infeasible. GUI ripping (e.g. [2]) can provide some help in this regard.

4.7.2 Landscape Improvement

For some GI objectives such as defect repair the landscape can be very flat and uninformative. Some work such as [3] automatically derived program invariants and was able to leverage these to speed up search. More recent work in program invariants [2] has been used for fault localisation – perhaps this can be leveraged both for better localisation of repair locations but also for giving a more informed fitness response to programs. Evosuite [4] continues to be developed as a way to reverse engineer assertions that serve as oracles from which to generate tests (the approach is contingent on the assumption that the version of the program used to generate tests is correct). The extent to which tests based on invariants can be synthesised from programs with bugs is an interesting question.

Another approach is the use of smart operators that are more likely to preserve semantics. Elements of this approach, combined with the use of genotypes that allow for separate evolution of the source, destination, and operation in a defect repair setting have been recently applied with some success in AJAR[10].

References

- 1 William B. Langdon and Mark Harman. Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*, 19(1):118–135, February 2015. doi:10.1109/TEVC.2013.2281544.
- 2 Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 258–261, New York, NY, USA, 2012. ACM.
- 3 Ethan Fast, Claire Le Goues, Stephanie Forrest, and Westley Weimer. Designing better fitness functions for automated program repair. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO ’10, pages 965–972, 2010. ACM.
- 4 Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE ’11, pages 416–419, New York, NY, USA, 2011. ACM.
- 5 Ting Hu and Karoliina Oksanen. Lexicase selection promotes effective search and behavioural diversity of solutions in linear genetic programming. CEC-2017

- 6 Yuanyuan Zhang, Mark Harman, Yue Jia. Achievements, open problems and challenges for search based software testing.
- 7 Gregory Parisi, Denis Robilliard, and Cyril Fonlupt. Applying boosting techniques to genetic programming. In Pierre Collet, Cyril Fonlupt, Jin-Kao Hao, Evelyne Lutton, and Marc Schoenauer, editors, *Artificial Evolution*, pages 267–278, 2002. Springer Berlin Heidelberg.
- 8 Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. Post-compiler software optimization for reducing energy. *SIGARCH Comput. Archit. News*, 42(1):639–652, February 2014.
- 9 Stephanie Forrest, Westley Weimer, Zachary P. Fry. Leveraging program equivalence for adaptive program repair: Models and first results. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 356–366, 2013. doi:10.1109/ASE.2013.6693094.
- 10 Yuan Yuan and Wolfgang Banzhaf. ARJA: Automated repair of java programs via multi-objective genetic programming. *arXiv:1712.07804*.

5 Resources for Genetic Improvement

The seminar was also the occasion for participants to share useful resources such as state-of-the-art tools and frameworks etc. In particular, since the evaluation of new techniques in a replicable and time efficient fashion may prove challenging, the exchange of benchmarks is a valuable output of the seminar.

5.1 Tools, Libraries and Frameworks

1. DSpot: a tool for Genetic Improvement of test suites
<https://github.com/STAMP-project/dspot>
2. PyGGI: Python General Framework for GI
<https://github.com/coinse/pyggi>
<https://coinse.github.io/pyggi/> (API documentation)
3. GIN: GI in no Time
<https://github.com/gintool/gin>
4. GenProg:
<https://squareslab.github.io/genprog-code/> GitHub io page,
<https://github.com/squaresLab/genprog-code> GitHub source
5. Software Engineering Library (support for C/C++ source w/ CLANG, ASM, ELF, future: Coq, Java):
<https://github.com/GrammaTech/sel> Github Source
<https://grammatech.github.io/sel/Manual>
<https://grammatech.github.io/sel/Usage.html> Installation and easy examples to start
<https://github.com/GrammaTech/clang-mutate> C/C++ manipulation tooling
6. ARJA:
<https://github.com/yyxhdy/arja>
7. Reproduce and repair failing builds:
<https://github.com/Spirals-Team/librepair/tree/master/repairnator>
8. MuScalpel: automated software transplantation.
<http://crest.cs.ucl.ac.uk/autotransplantation/downloads/muScalpel.zip>

9. History of programming languages
<https://github.com/svalver/Proglang>
10. Agent-based model for the cultural diffusion of programming languages (code)
http://modelingcommons.org/browse/one_model/4611
11. JavaScript parser:
<http://esprima.org/>
<https://github.com/estools/escodegen>
<https://github.com/estools/estraverse>
12. Astor4Android: program repair for Android App
<https://github.com/kayquesousa/astor4android>

5.2 Benchmarks

1. BugZoo (Docker containers for ManyBugs): <https://github.com/squaresLab/BugZoo>
2. CodeFlaws
<https://github.com/codeflaws/codeflaws>
3. Parsec:
<http://parsec.cs.princeton.edu/>
4. SPEC INT:
<https://www.spec.org/benchmarks.html>
5. Microsoft Version Control repos with Bug Info related to commits:
<http://msr.uwaterloo.ca/msr2009/challenge/msrchallengedata.html>
6. DBGBENCH : evaluation of automated fault localization, diagnosis, and repair techniques w.r.t. the judgement of human experts
<https://github.com/rjust/defects4j> a collection of reproducible bugs
<https://droix2017.github.io> a set of reproducible crashes in Android apps
7. ARJA Benchmark of seed bugs
<https://github.com/yyxhdy/SeededBugs>

6 Following the Seminar, New work and New Connections

6.1 New Work

References

- 1 Afsoon Afzal, Jeremy Lacomis, Claire Le Goues, and Christopher S. Timperley. A Turing test for genetic improvement. In Justyna Petke, Kathryn Stolee, William B. Langdon, and Westley Weimer, editors, *GI-2018, ICSE workshops proceedings*, pages 17–18, Gothenburg, Sweden, 2 June 2018. ACM.
- 2 Gabin An, Jinhan Kim, and Shin Yoo. Comparing line and AST granularity level for program repair using PyGGI. In Justyna Petke, Kathryn Stolee, William B. Langdon, and Westley Weimer, editors, *GI-2018, ICSE workshops proceedings*, pages 19–26, Gothenburg, Sweden, 2 June 2018. ACM.
- 3 Benoit Baudry, Nicolas Harrand, Eric Schulte, Marija Selakovic, Shin Hwei Tan, Christopher Timperley, and Emamurho Ugherughe. A spoonful of DevOps helps the GI go down. In Justyna Petke, Kathryn Stolee, William B. Langdon, and Westley Weimer, editors, *GI-2018, ICSE workshops proceedings*, pages 35–37 Gothenburg, Sweden, 2 June 2018. ACM.

- 4 Joseph Renzullo, Stephanie Forrest, Westley Weimer, and Melanie Moses. Neutrality and epistasis in program space. In Justyna Petke, Kathryn Stolee, William B. Langdon, and Westley Weimer, editors, *GI-2018, ICSE workshops proceedings*, pages 1–8, Gothenburg, Sweden, 2 June 2018. ACM.

6.2 New Connections

The followings outcomes were reported by the participants:

1. Eric Schulte, Benoit Baudry, Stephanie Forrest and Nicolas Harrand plan to collaborate on the following topics:
 - The “older but wiser” hypothesis
 - Mapping the “envelope” where executions of neutral variants diverge from one another and identify quiescent points where they converge.
 - Investigating the hypothesis: Systems with more interpretive steps between the “source code” and execution are more robust than those with fewer steps?
2. Eric Schulte, Claire Le Goues and Chris Timperley plan to work at CMU on experimental framework merging (BugZoo)
3. Prof. Banzhaf and Prof. Langdon are planning an experimental evaluation of long term evolution in continuous domains.
4. Dr. Markus Wagner plans to visit Prof. Krawiec Krzysztof during his sabbatical in 2019.
5. Based on a discussion between Prof. Sekanina, Dr. Vasicek and Prof. Krawiec, new research directions have been identified in the area of genetic programming using formal verification methods. Possible ways of collaboration on this topic are under discussion.
6. Dr. Leonardo Trujillo and Dr. John Woodward plan to work on the following question: What are the similarities and differences of Decision Forest representations and algorithms and Geometric Semantic Genetic Programming. Both of these approaches have attracted considerable attention over the past few years.
These two approaches have significant similarities in the types of models they construct, but also some differences.
They believe these similarities are more than superficial and ask what can these two areas learn from one another.
7. Dr. Claire Le Goues and Prof. Stephanie Forrest will collaborate on round 2 of an idea they tried out several years ago, but now have new ideas for. They will (sometime in the indefinite future) write a paper about improving fitness functions for automated program repair, to include information beyond test suite success.
8. Profs. Colin Johnson and Krzysztof Krawiec talked about the possibilities of using machine learning to measure program quality in genetic improvement and program synthesis, and about the role of program comprehension (and its measurement) in that process. And hope to make progress together in this area.

Participants

- Brad Alexander
University of Adelaide, AU
- Wolfgang Banzhaf
Michigan State University, US
- Benoit Baudry
KTH Royal Institute of Technology – Stockholm, SE
- Bobby R. Bruce
University College London, GB
- Celso G. Camilo-Junior
Federal University of Goiás, BR
- Myra B. Cohen
University of Nebraska – Lincoln, US
- Benjamin Danglot
INRIA Lille, FR
- Stephanie Forrest
Arizona State University – Tempe, US
- Nicolas Harrand
KTH Royal Institute of Technology – Stockholm, SE
- Colin G. Johnson
University of Kent – Canterbury, GB
- Krzysztof Krawiec
Poznan University of Technology, PL
- William B. Langdon
University College London, GB
- Claire Le Goues
Carnegie Mellon University – Pittsburgh, US
- Alexandru Marginean
University College London, GB
- Michael Pradel
TU Darmstadt, DE
- Joseph Renzullo
Arizona State University – Tempe, US
- Eric Schulte
GrammaTech Inc. – Ithaca, US
- Lukas Sekanina
Brno University of Technology, CZ
- Marija Selakovic
TU Darmstadt, DE
- Shin Hwei Tan
National University of Singapore, SG
- Christopher Timperley
Carnegie Mellon University – Pittsburgh, US
- Leonardo Trujillo
Instituto Tecnológico de Tijuana, MX
- Emamurho Ugherughe
SAP SE – Berlin, DE
- Sergi Valverde
UPF – Barcelona, ES
- Zdenek Vasicek
Brno University of Technology, CZ
- Markus Wagner
University of Adelaide, AU
- John R. Woodward
Queen Mary University of London, GB
- Shin Yoo
KAIST – Daejeon, KR

