

On Search Based Software Evolution

Andrea Arcuri

School of Computer Science, University of Birmingham, Edgbaston, Birmingham B15 2TT, UK.

email: a.arcuri@cs.bham.ac.uk

Abstract

Writing software is a difficult and expensive task. Its automation is hence very valuable. Search algorithms have been successfully used to tackle many software engineering problems. Unfortunately, for some problems the traditional techniques have been of only limited scope, and search algorithms have not been used yet. We hence propose a novel framework that is based on a co-evolution of programs and test cases to tackle these difficult problems. This framework can be used to tackle software engineering tasks such as Automatic Refinement, Fault Correction, Improving Non-functional Criteria and Reverse Engineering. While the programs evolve to accomplish one of these tasks, test cases are co-evolved at the the same time to find new faults in the evolving programs.

1 Introduction

In software engineering there are many tasks that are very expensive, like for example testing the developed software [15]. It is hence important to try to automate these tasks, because it would have a direct impact on software industries.

Re-formulating software engineer as an optimisation problem has led to promising results in the recent years [10, 9]. Many tasks have been addressed by the research community, but some are mainly unexplored. We hence want to feel this gap.

We have designed a novel framework that, with little changes, can be easily applied to automate at least these following software engineering problems:

- *Automatic Refinement*: given as input a formal specification, we want to obtain a correct implementation in an automatic way.
- *Fault Correction*: given as input a program implementation and a set of test cases in which at least one test case is failed, we want to automatically evolve the in-

put program to make it able to pass all the given test cases.

- *Improving Non-functional Criteria*: given as input a program, we want to evolve it to optimise some of its non-functional criteria (e.g., execution time and power consumption) without changing its semantics.
- *Reverse Engineering*: given as input the assembler code or byte-code of a program, we want to automatically derive its source code.

The novel framework we propose is based on co-evolution of programs (evolved for example with Genetic Programming [17]) and test cases (evolved for example with Search Based Software Testing [14]). Programs are rewarded by how many tests they do not fail, whereas the unit tests are rewarded by how many programs they make to fail. This type of co-evolution is similar to what happens in nature between *predators* and *prey*.

Preliminary results confirm that this approach is feasible [4, 2, 5, 3]. However, given the novelty of this approach and the difficulty of these tasks, more research is still required. In particular, we need to analyse whether the proposed approach would *scale* to real-world software.

The paper is organised as follows. Section 2 briefly gives background information of Genetic Programming and Co-evolution. Section 3 describes the novel co-evolutionary framework. The addressed software engineering problems are discussed in section 4. Finally, Section 5 concludes the paper.

2 Background

2.1 Genetic Programming

Genetic Programming (GP) [17] is a paradigm to evolve programs. A genetic program can often be seen as a tree, in which each node is a function whose inputs are the children of that node. A population of programs is held at each generation, where individuals are chosen to fill the next population accordingly to a problem specific fitness function.

Crossover and mutation operators are applied to the programs to generate new offspring.

2.2 Co-evolution

In co-evolutionary algorithms, one or more populations co-evolve influencing each other. There are two types of influences: *cooperative co-evolution* in which the populations work together to accomplish the same task, and *competitive co-evolution* as predators and prey in nature. In our framework we use competitive co-evolution.

Co-evolutionary algorithms are affected by the *Red Queen* effect [16], because the fitness value of an individual depends on the interactions with other individuals. Because other individuals evolve as well, the fitness function is not static. For example, exactly the same individual can obtain different fitness values in different generations. One consequence is that it is difficult to keep trace of whether a population is actually “improving” or not [7].

One of the first applications of competitive co-evolutionary algorithms was the work of Hillis on generating sorting networks [11]. He modelled the task as an optimisation problem, in which the goal is to find a correct sorting network that does as few comparisons of the elements as possible. He used evolutionary techniques to search the space of sorting networks, where the fitness function was based on a finite set of tests (i.e., sequences of elements to sort): the more tests a network was able to correctly pass, the higher fitness value it got. For the first time, Hillis investigated the idea of co-evolving such tests with the networks. The reason for doing so was that random test cases might be too easy, and the networks can learn how to sort a particular set of elements without being able of generalising. The experiments of Hillis showed that shorter networks were found when co-evolution was used.

Ronge and Nordahl used co-evolution of genetic programs and test cases to evolve controllers for a simple “robot-like” simulated vehicle [20]. Similar work has been successively done by Ashlock et al. [6]. In such work, the test cases are instances of the environment in which the robot moves.

Note that the application area of such work was restricted (i.e., sorting networks and robot controllers), whereas we use co-evolution in a more general and complex framework.

In software engineering, a co-evolutionary algorithm has been used in Mutation Testing [1]. The goal is to find test cases that can recognise faulty *mutants* of the tested software, because such a test suite would be good for asserting the reliability of the software. Mutants (generated with a precise set of rules) co-evolve with the test cases, and they are rewarded on how many tests they pass, whereas the test cases are rewarded on how many mutants they identify as semantically different from the original program.

3 Co-evolutionary Framework

In our novel framework, programs co-evolve with test cases. Programs are rewarded by how many tests they do not fail, whereas the unit tests are rewarded by how many programs they make to fail. For example, a program can obtain a value 0 if it passes a test case and 1 otherwise. The fitness function to minimise would hence be the sum of these values given by the execution of the program on all the given test cases. For the fitness function of the test cases, it would be a maximisation problem.

Depending on which software engineering problem is addressed, there are the following components of the framework that needs to be specialised:

- Input of the framework.
- Initialisation of the genetic programs.
- Oracle used for evaluating the expected outputs of the test cases.
- Generation of new test cases.
- Other objectives for the fitness function.

These specific points will be discussed in the next section.

4 Software Engineering Problems

4.1 Automatic Refinement

Since the 1970s the goal of generating programs in an automatic way has been sought [12]. A user would just define what he expects from the program (i.e., the requirements), and it should be automatically generated by the computer without the help of any programmer.

Unfortunately, this task is much harder than expected [19]. Transformation methods are usually employed to address this problem. The requirements need to be written in a formal specification, and sequences of transformations are used to transform these high-level constructs into low-level implementations. Unfortunately, this process can rarely be automated completely, because the gap between the high-level specification and the target implementation language might be too wide.

We analysed the application of our framework to this problem [4]. The presence of a gap does not preclude the application of our framework. The specifications we used were written in first order logic. Other formal specification languages could be used (e.g., Z [21]), but we would need to implement a fitness function for them.

The input to the framework is a formal specification of the program we seek. The genetic programs are initialised

at random, and the formal specification is used as an oracle. The formal specification can also be used to guide the evolution of more challenging test cases.

4.2 Fault Correction

A lot of research has been done on software testing. However, if the presence of a fault is discovered, it is still duty of the programmers to fix the software. We hence seek to also automate this expensive task. In this research field, the very few proposed techniques (e.g., [23]) have many limitations. Only very restricted types of modifications are allowed, that because these techniques are mainly doing an exhaustive search. The types of modifications they consider do not guarantee that between any two programs there is a sequence of transformations to obtain the second program from the first. So given any faulty program, it could be impossible to fix it. On the other hand, our approach can potentially fix any code level fault. This because we are searching in the entire space of possible programs.

The input of the framework would be a faulty program and a set of test cases in which at least one is failed. The test cases need to be provided (e.g., by a software tester). A formal specification can be given as input as well although it is not essential. The genetic programs are initialised with heuristics based on the input program (e.g., they can simply be copies of it). New test cases can be evaluated against a formal specification, otherwise no new test case can be generated. In that case, the programs are executed only on a subset of the given test cases. The co-evolution will be hence used to choose which subset of test cases to use at each generation. More details can be found in our preliminary work [2, 5].

We can consider fault correction as a special case of automatic refinement, in which the faulty input program is a solution structurally close to a global optimum. The input faulty program is hence heuristically exploited to help the search for a correct refinement. Because programmers do not write code at random [8], we would expect that fault correction would be much easier than automatic refinement. We can hence speculate that it could scale to real-world software, that because software can be often fixed with only few code changes [18].

4.3 Improving Non-functional Criteria

Optimising non-functional properties of software is an important part of the implementation process. One such property is execution time, and compilers target a reduction in execution time using a variety of optimisation techniques. Compiler optimisation is not always able to produce semantically equivalent alternatives that improve execution times,

even if such alternatives are known to exist. Often, this is due to the local nature of such optimisations.

The input to the framework is a program we want to optimise. That program is also used as an oracle for the test cases. The co-evolved test cases are used to test the preservation of the original semantics. The initialisation of the genetic programs is not trivial. On one hand, doing that at random will make the evolution of correct programs very difficult to achieve. On the other hand, using copies might constrain the search in a particular sub-optimal area of the search space. The non-functional criteria needs to be evaluated by executing the programs on a separated set of test cases that does not co-evolve. Because there is more than one objective to optimise, multi-objective algorithms can be used. More details can be found in our preliminary work [3].

4.4 Reverse Engineering

One application of Reverse Engineering [22] is that, given as input the assembler or byte-code of a program, we want to obtain the original source code. We can use our framework to tackle this problem.

The execution of the input assembler can be used as an oracle for the test cases. Initialising the genetic programs at random is possible, but likely it will make the search too difficult. Therefore, smart seeding strategies that exploit the assembler code should be designed.

We have not carried out yet any experiment with our framework on this problem. However, GP often generate code that is difficult to understand by humans. So the readability needs to be carefully taken in account in the search, but that is a common problem in GP that is not specific to only our application. At any rate, there are cases of obfuscating techniques that make difficult to obtain even a compilable source code (e.g.,[13]). In those cases, our technique would be useful even if the evolved code would be difficult to read.

5 Conclusions and Future Work

In this paper we have described the application of a novel co-evolutionary framework to solve some software engineering problems. Co-evolutionary algorithms are not novel, but our contribution is to show how to apply co-evolution of software and test cases in software engineering.

Each software engineering problem has its own specific properties, and care needs to be spent to adapt the framework to solve them. In other words, different types of optimisations of the framework can be done, and each of them is specific to the addressed problem. However, the conceptual co-evolutionary framework would still be the same.

In this paper we briefly discussed the application of our framework to four different software engineering problems: Automatic Refinement, Fault Correction, Improving Non-functional Criteria and Reverse Engineering. Other software engineering problems might be addressed with our framework as well.

We have already obtained preliminary results in some of these tasks, and more specific details can be found in our previous papers [4, 2, 5, 3]. These software engineering tasks are very important, and we believe that our work will help the community to setup the bases from which more research can be built on.

For future work, we want to study in more details some of these problems, like for example Fault Correction. We in fact believe there is large space left for improvements and for designing more sophisticated and tailored algorithms to improve the performance. That would be a compulsory step if we want our technique to scale to real-world software. Other important point to investigate is how to improve the co-evolution in general (e.g., reward diversity in the test cases based on code coverage criteria).

6 Acknowledgements

The author is grateful to Xin Yao, David White and Rami Bahsoon for insightful discussions. This work is supported by EPSRC grant EP/D052785/1.

References

- [1] K. Adamopoulos, M. Harman, and R. M. Hierons. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1338–1349, 2004.
- [2] A. Arcuri. On the automation of fixing software bugs. In *In the Doctoral Symposium of the IEEE International Conference on Software Engineering (ICSE)*, pages 1003–1006, 2008.
- [3] A. Arcuri, D. R. White, J. Clark, and X. Yao. Multi-objective improvement of software using co-evolution and smart seeding. In *International Conference on Simulated Evolution And Learning (SEAL)*, pages 61–70, 2008.
- [4] A. Arcuri and X. Yao. Coevolving programs and unit tests from their specification. In *IEEE International Conference on Automated Software Engineering (ASE)*, pages 397–400, 2007.
- [5] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 162–168, 2008.
- [6] D. Ashlock and S. Willson. Coevolution and tartarus. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 1618–624, 2004.
- [7] D. Cliff and G. F. Miller. Tracking the red queen: Measurements of adaptive progress in co-evolutionary simulations. In *European Conference on Artificial Life*, pages 200–218, 1995.
- [8] R. A. DeMillo, R. J. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.
- [9] M. Harman. The current state and future of search based software engineering. In *Future of Software Engineering (FOSE)*, pages 342–357, 2007.
- [10] M. Harman and B. F. Jones. Search-based software engineering. *Journal of Information & Software Technology*, 43(14):833–839, 2001.
- [11] W. D. Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D*, 42(1-3):228–234, 1990.
- [12] M. Jazayeri. Formal specification and automatic programming. In *IEEE International Conference on Software Engineering (ICSE)*, pages 293–296, 1976.
- [13] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the ACM conference on Computer and Communications Security*, pages 290–299, 2003.
- [14] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.
- [15] G. Myers. *The Art of Software Testing*. Wiley, New York, 1979.
- [16] J. Paredis. Coevolving cellular automata: Be aware of the red queen. In *Proceedings of the International Conference on Genetic Algorithms (ICGA)*, pages 393–400, 1997.
- [17] R. Poli, W. B. Langdon, and N. F. McPhee. *A field guide to genetic programming*. Published via <http://lulu.com> and freely available at <http://www.gp-field-guide.org.uk>, 2008.
- [18] R. Purushothaman and D. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511–526, 2005.
- [19] C. Rich and R. C. Waters. Automatic programming: myths and prospects. *Computer*, 21(8):40–51, 1988.
- [20] A. Rong and M. G. Nordahl. Genetic programs and co-evolution. developing robust general purpose controllers using local mating in two dimensional populations. In *Parallel Problem Solving from Nature IV, Proceedings of the International Conference on Evolutionary Computation*, pages 81–90, 1996.
- [21] J. M. Spivey. *The Z Notation, A Reference Manual. Second Edition*. Prentice Hall, 1992.
- [22] P. Tonella, M. Torchiano, B. D. Bois, and T. Systä. Empirical studies in reverse engineering: state of the art and future trends. *Empirical Software Engineering*, 12(5):551–571, 2007.
- [23] W. Weimer. Patches as better bug reports. In *International conference on Generative programming and component engineering*, pages 181–190, 2006.