From Problem Landscapes to Language Landscapes: Questions in Genetic Improvement

Brendan Cody-Kenny, Michael Fenton, Michael O'Neill.

ABSTRACT

Managing and curating software is a time consuming process particularly as programming languages, libraries, and execution environments change. To support the engineering of software, we propose applying a GP-based continuous learning system to all "useful" software. We take the position that search-based itemization and analysis of all commonly used software is feasible, in large part, because the requirements that people place on software can be used to bound the search space to software which is of high practical use. By repeatedly reusing the information generated during the search process we hope to attain a higher-level, but also more rigorous, understanding of our engineering material - source code.

KEYWORDS

Software Engineering, Search, Learning

ACM Reference format:
Brendan Cody-Kenny, Michael Fenton, Michael O'Neill.. 2017. From Problem Landscapes to Language Landscapes:
Questions in Genetic Improvement. In *Proceedings of the Genetic and Evolutionary Computation Conference 2017, Berlin, Germany, July 15–19, 2017 (GECCO '17),* 2 pages.
DOI: http://dx.doi.org/10.1145/3067695.3082522

1 UNDERSTANDING SOFTWARE

While software encapsulation, structure, and adherence to coding standards are recommended to make software more navigable and easier to understand, software development remains time consuming. Even simple programs in modern languages rely on many layers of software which may vary in different environments. Larger programs are even less likely to be fully understood by any single person or even team of people. Further complexity from frequently changing and sometimes conflicting requirements makes it difficult to quickly gain and maintain a dependable high-level understanding of software.

Any program functionality deemed "important" should warrant performing a rigorous analysis to broadly aid understanding. Definitions [10] and reference implementations [1] do exist for software but much detail is missing in comparison to the wealth of operational information available for materials in traditional engineering disciplines [3]. What is important here is the *process* of deriving these standards, which for more established engineering domains have been derived manually through centuries of deployment and testing experience. We ask why a similarly rigorous and broad analysis of software can not also be used to aid software development?

In response, we propose a learning system which uses search techniques to explore and curate a continuously expanding library of useful programs (and sub-programs), and that by automatically reusing this knowledge we may more effectively treat source code as an engineering material. What we aim to do by building such a learning system is to teach a computer to recognize requirements and find best matches from existing software based on code previously seen. With such a system, people may describe what they want by either specifying, or even selecting existing, test cases. Alternatively people can make an initial attempt at writing a basic working prototype, with a computer recommending directions in which the program might evolve, either towards more specific functionality or some improvement in non-functional characteristics. As a result, humans can be chiefly concerned with experimenting, prototyping and creating software concepts and architectures in a "grow" phase while an automated support system checks for similarities, merges & simplifies software in a "prune" phase [4].

2 ALWAYS-ON GP

Our proposal is to use Genetic Programming (GP) techniques as part of a continuous learning system[1]. We break this idea into 3 areas - a knowledge base, ways to draw inferences from this knowledge base, and knowledge acquisition. Generally, we feel there is an opportunity to "close the loop" in further automating and integrating many concepts from the extensive existing research for each area to create next generation tooling for software engineering tasks.

2.1 Knowledge Base

Software is complex and ever-evolving but has a number of characteristics that make broad analysis more feasible. Given the seeming repetition and overlap in software implementations, portions of the search space may prove to subsume others, and be more itemisable as a result. While the set of programs which do not exist is infinite, the set of programs which do exist is finite, and within this, the set of programs which are repeatedly considered *useful* is smaller still. We may ask what is it about the software which does exist, in comparison to the software which does not? Unlike engineering of physical things, source code as an engineering material is directly *executable*, making it self-labeling and widely amenable to automated testing.

As such, we feel that human written programs, their test cases, and test results can (and should) be stored for reuse in a knowledge

[1]This idea harks back to work on expert systems, where a knowledge base is (usually manually) curated and used via an inference engine. A major issue in making these systems usable is knowledge acquisition.

GECCO '17, Berlin, Germany

978-1-4503-4939-0/17/07…$15.00
DOI: http://dx.doi.org/10.1145/3067695.3082522

base. By using a shared continuous cache, a vast library of software knowledge could be curated over time including detailed information about multiple operational aspects of executed software, such as test cases passed, execution time, and memory usage. Although testing is computationally expensive, it is *already* widely used in industry to add rigor to software development, however results do not appear to be curated or broadly shared. While test data can be generated indefinitely, some test cases are more important than others. We can consider the question: *How many tests are required?*[2] As a result, gathering, storing and analysing extensive test information may be within reach.

2.2 Making Inferences

By comparing program variants and their operational characteristics, we can extract code snippets which are found to improve programs [9]. Further to this, structure and linkages can be found between software artifacts with the aim of allowing for automated inferences to be made. We argue that there is overlap in the search space as certain functionality is shared among programs. This should allow us to build a functionality map of frequently required software, navigable by linking multiple software implementations which measurably fulfill the same or similar requirements. Problem landscape analysis is widely used by the Evolutionary Computation (EC) community to understand specific problems and solution structures [8]. Similar analysis in a Software Engineering context could also provide high level understanding of what trade-offs are possible from an existing program [6]. If we are already performing the expensive process of mutation and testing across many programs anyway, there is potential to analyse this information to provide a view across the landscape of all useful programs in a, or even across more than one, language.

Since there may be pre-existing solutions which meet the same requirements (and pass the same test cases [2]), a simple look-up of the cache could return a range of existing alternative implementations which hopefully includes an improvement. For more complex functionality where test cases can not be directly comparable, more advanced techniques such as neural networks could be used to recognise when the input-output values of a program represent similar functionality as an input-output pair containing different values. In effect, this could be treated as a "Big Data" mining issue across semantic information [7].

We see boundary value testing as potentially useful in capturing program equivalence. Boundary testing involves making incremental modifications to input data until a test case no longer passes. When a small change to a program causes a previously passing test to fail, we can say we have found the boundary point of correctness. We can pair these test cases which mark the edge of correctness. By collecting many input data pairs, (or "boundary value pairs"), we can build up a set which may capture the "contours" of program functionality. With a large and well connect graph of previously seen code, new code presented to the system can be analysed for similarities, alternative implementations and potential improvements.

[2]This question is paraphrased from one asked by Mark Harman during the 50th Crest Open Workshop at UCL

2.3 Knowledge Acquisition

While data mining existing software modifications can instruct GP search [9], detailed knowledge can also be acquired from the GP process itself and added to the library. Where and unseen program is not well placed within the existing corpus of information, we may use GP to find a lineage of program transformations which better connects an unseen program to the body of knowledge that has already been built up. If GP is not tackling some unseen problem it can still run to generate useful improvement. GP modifications which degrade existing programs can be instructive for what *not* to do. The most deleterious program transforms can be applied in reverse to unseen programs, in the hope of improvement. While the idea of automatically reusing human-written code through automated decomposition and recomposition is not new [5], we feel it is becoming more feasible.

3 WHERE TO BEGIN

For an initial experiment, ideally we would like a language which is simple, well defined, easy to test but is also in widespread industry use. Fortunately Regular Expressions (regexes) fit this description. There is good evidence that in a large sample of regexes there is considerable semantic similarity and that linkages can be drawn between them [2]. We are interested here to map out the relation between existing regexes and their test cases, as opposed to "evolving" or "growing" regexes.

Other problems, particularly those that are well studied in the GP literature, could also be reconsidered in terms of reuse. Making the results of a GP run as reusable as possible for subsequent GP runs may make interesting future work on benchmark GP problems.

ACKNOWLEDGMENTS

This research is based on works supported by Science Foundation Ireland under grant 13/IA/1850 and 13/RC/2094 which is co-funded by Lero - the Irish Software Research Centre (www.lero.ie).

REFERENCES

[1] 2017. World Wide Web Consortium. https://www.w3c.com. (2017). Accessed: 28th March.
[2] Carl Chapman and Kathryn T Stolee. 2016. Exploring regular expression usage and context in Python. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 282–293.
[3] EN 1990 2001. *Eurocode - Basis of Structural Design.* Standard. European Committee for Standardization.
[4] David Faust and Chris Verhoef. 2003. Software product line migration and deployment. *Software: Practice and Experience* 33, 10 (2003), 933–955.
[5] Keith Gallagher and David Binkley. 2008. Program slicing. In *Frontiers of Software Maintenance, 2008. FoSM 2008.* IEEE, 58–67.
[6] Mark Harman, William B Langdon, Yue Jia, David R White, Andrea Arcuri, and John A Clark. 2012. The GISMOE challenge: Constructing the Pareto Program Surface Using Genetic Programming to Find Better Programs (keynote paper). In *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on.* IEEE, 1–14.
[7] Yalin Ke, Kathryn T Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing programs with semantic code search. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on.* IEEE, 295–306.
[8] William B Langdon, Nadarajen Veerapen, and Gabriela Ochoa. 2017. Visualising the Search Landscape of the Triangle Program. In *European Conference on Genetic Programming*. Springer, 96–113.
[9] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, Vol. 1. IEEE, 213–224.
[10] James W Moore. 1999. An integrated collection of software engineering standards. *IEEE software* 16, 6 (1999), 51–57.