SIGEVO SUMMER SCHOOL 2017 – Summaries of papers provided by Dr. Justyna Petke

Summarized by Shashank Srikant
shash@mit.edu

Two papers from the field of Genetic Improvement are summarized here. The first is a position paper which advocates the integration of GI in the design of programming language. The other is a paper on GI being applied to fixing bugs in real-time software systems.

Paper 1:

Embedding Genetic Improvement into Programming Languages by Shin Yoo, KAIST

Summary:

This is a position-paper which emphasizes the role Genetic Improvement can have in the design of programming languages. The author describes how GI can play a pivotal role in automatically refining functional and non-functional components of software codebases. The author foresees GI to gain prominence in the programming languages domain, given some of the substantial results it is able to contribute to. Given the growing popularity, the author wishes to put forward a vision in how the technology ought to be adopted. Having said that, he also draws caution to the difficulties which the subject matter brings along with it. It lacks a rigorous theoretical framework and has a steep learning curve.

The primary point driven by the author is to consider GI as an integral part a programming language or its compiler/interpreter-system, a component "embedded in the system" as against being an add-on or a layer above it. The author prefers embedding this technology as against treating it as a framework or a library because – (a) given its strengths, it can seamlessly provide the end-user (a programmer, software engineer/designer) control over any quantifiable metric regarding a program, such as memory, run-time etc. Also, this is best done at a programming language level rather than any library/framework can have access to, and (b) will ensure the technology is adapted easily, given only its front-end would be visible to the end-user.

In doing so, the author also additionally talks about some programming constructs which are likely to become common-place once GI is integrated into languages. He describes constructs like @optimize, a construct akin to the '@assert' construct, which will allow GI to perform to the best of its design. He also envisions constructs like @optional and @approximate, both of which allow the end-user flexibility such as disallowing loading something onto the memory or trading off accuracy (or some such functional metric) to execution speed-up or energy minimization.

In summary, the author's holds a bright view of GI for the field of programming languages. He advocates its integration with the low-level design of a programming language so as to get an end-user to easily make use of its strengths and features.

Paper 2:

Fixing bugs in your sleep: How GI became an overnight success by Haraldsson et al., GECCO 2017

Summary:

The paper describes a GI system used in a live-software system. It describes how the designers are able to integrate GI in their software, enabling it to automatically fix bugs and improve its code base. The paper describes a software system which, on the basis of a day's error-log, improves its own code-base during the night.

The paper describes a data management software on which GI has been applied. Called Janus Management, it is a CRM internal to an Icelandic rehabilitation center (called JR). It possess different functionality like record maintenance, obtaining summary statistics of the various internal processes and transactions happening via the system etc. It is a system spanning 25K LOC in Python, spread over 300 classes and 600 methods. Most of these have an internal unit/class test available.

The key component of the system design is the analysis of the software logs that are generated when the system is used through the day. All exceptions raised in the code-base become inputs to the GI system. The fix is a two-fold process – first, once the exception and the part of the affected code has been identified, test cases need to be synthesized which reproduce the error. The subsequent step is using GI to improve the code in order for the evolved code to robustly handle all the synthesized test cases. Doing so tacitly fuels the GI system with only exceptions and not functional errors, i.e. errors which raised no exceptions.

The technique to synthesize test cases is ad-hoc, in that there are a few rules that have been defined to modify test cases in a hope to resemble similar pathological cases. It does not apply program synthesis and other sophisticated tools. Neither is it empirically justified.

The GI is founded on a grammar of edits which involve basic syntax like binary operators and numerical, string literals. This is interesting in that it does not apply any BNF-grammar level or AST-level changes which Petke et al. [1] and Le Goius et al. [2] respectively apply. This is an intuitive and a stronger baseline which can in fact compare the efficacy of both, [1] and [2]. The operators for GI is restrained to mutation. In addition to being intuitive, it reinforces [1], where the authors too speculate the utility of having cross-over operations. Specifically, the mutation operators are allowed to (a) *grow*, meaning a delete, replace or copy/swap of lines from within the source code is allowed (which [1] uses) or, (b) one specific operator in the source code, at random, is changed. Besides intuition, the authors provide no formal justification to adopting/efficacy of these strategies.

The results describe 22 bug fixes made over six months. They are restrained to 10 edits per each fix.

Critique:

- It is not clear whether the bug-fixes made were parts that were specifically raised by the exception. If it is, then the system at most reduces the man-effort to fix the minor-nature of the bug that was raised by the exception. It would be great if the system, in the process of fixing the

exception issue, managed to fix other bugs in the source code as well. The current implementation of the GI system is only a partial automation of human-intensive effort.

- The authors involve software developers to look at test cases and discard duplicates. That same effort can possibly be used to fix the bugs themselves, given the minor nature of changes that are required to fix them.
- The grammar employed to instruct the evolution suggests that there is only a restricted family of fixes which the software can make.

In summary, the system is demonstrative of some of the very interesting applications which GI can command. It demonstrates how a more sophisticated system can actually be deployed in a real-time software, wherein program repair can be performed automatically on itself. This is a first step towards a suite of such applications hopefully.

References:

[1] Petke, Justyna, et al. "Using genetic improvement and code transplants to specialise a C++ program to a problem class." *European Conference on Genetic Programming*. Springer, Berlin, Heidelberg, 2014.

[2] Le Goues, Claire, et al. "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each." *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012.