Embedded Dynamic Improvement

Nathan Burles, Jerry Swan, Edward Bowles
Department of Computer Science
University of York
York, YO10 5GH, UK
{first.last}@york.ac.uk

Alexander E. I. Brownlee, Zoltan A. Kocsis, Nadarajen Veerapen
Computing Science and Mathematics
University of Stirling
Stirling, FK9 4LA, UK
{sbr,zak,nve}@cs.stir.ac.uk

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods*; D.2.8 [**Metrics**]: Performance Measures

Keywords

Search-based software engineering; genetic improvement.

ABSTRACT

We discuss the useful role that can be played by a subtype of improvement programming, which we term 'Embedded Dynamic Improvement'. In this approach, developer-specified variation points define the scope of improvement. A search framework is embedded at these variation points, facilitating the creation of adaptive software that can respond online to changes in its execution environment.

1. INTRODUCTION

A recent keynote by Harman [2] asks "whether 'online' genetic improvement may now lie within our grasp?" and gives an illustrative example of 'dreaming devices' that operate in 'normal' and 'learning' modes, with usage distributions being gathered during the former in order that functionality can be optimized by the latter during periods of device inactivity. In this article, we answer this question in the affirmative and describe several implementations of online improvement programming, together with a unifying description of the commonalities that make them effective.

Since there is no requirement for a Genetic Improvement (GI) tool to be written in the same language as the code (source or binary) to be improved, we can make a clear distinction between *variation* and *execution* environments: the variation environment is defined by the language in which the variation tool is written, together with its input (namely, the source or binary which is to be improved). Variants are evaluated within an execution environment which is potentially separate from the variation environment. In the Embedded Dynamic Improvement' (EDI) approach, the developer restricts the scope of improvement to variation points which denote key functionality within a larger system. A search framework is then embedded at these variation points.

GECCO'15 Companion, July 11–15, 2015, Madrid, Spain

ACM 978-1-4503-3488-4/15/07…$15.00
DOI: http://dx.doi.org/10.1145/2739482.2768423

We argue that there are two main advantages to using embedded variant generators to improve software: 1) it enables software to be improved dynamically, and 2) by virtue of the embedding, the variation and execution environments are the same, hence the variant generator can access and exploit information about the running program and its environment that would be unavailable to an offline tool.

2. A COMPARISON OF GP, GI, AND EDI

The EDI approach seeks to gain some of the advantages offered by Genetic Programming (GP) and offline GI while avoiding some of the weakness of these techniques. Table 1 summarizes some of the key differences between GP, GI, and EDI. One of the most profound differences between GP and GI is the nature of the output. GP traditionally produces a program representation (e.g. a Koza tree), which is executed via an interpreter, whereas GI has traditionally operated on a complete system (which may be source- or object-code) and produces a variant copy of its input, typically represented as a patch to the source code. EDI also operates on complete systems, but as the variation occurs online rather than offline, the result is that the original system is modified, rather than replaced. As exemplified by state-of-the-art in offline GI [5, 6], determination of which subsystems to improve is itself part of the search process. While this approach is in principle general, it does not enjoy the full expressiveness of GP: both of these approaches employ 'plastic surgery', i.e. swapping program expressions (modulo variable renaming) rather than generating new expressions. Although recent investigation into the expressiveness of plastic surgery [1] concluded that over 40% of commits to a large codebase could be constructed by plastic surgery from within the codebase itself, this approach cannot take advantage of developer-specific knowledge about variation points ('expressions involving square root are likely to be useful here') and it seems reasonable to conclude that the plastic surgery approach would be less successful for smaller or younger codebases. In addition, the embedded nature of the variant generator facilitates programmatic access to information about variable scope and the type hierarchy: under certain circumstances [3] such detailed knowledge of the type system means that it is possible to replace random mutations with deterministic, semantics-preserving ones.

However, what separates this proposed approach from both GP and GI is its online nature. To best optimise software, it is essential to have a detailed understanding of the problem domain that the software is operating on, the capabilities and limitations of the language used to produce the software, and also the realities of the operating environment. Non-functional properties such as time required to

Table 1: Characteristics of GP, Offline GI and EDI

Characteristic	GP	Offline GI	EDI
Output	Expression tree	Patch for existing system	Updated system state
Target Language	Expression tree	Source or executable	Source or executable
Expressiveness	Arbitrary expressions	Defined by patterns	Arbitrary expressions possible
In situ	No	No	Yes
Scale	Single expression	Entire system	Developer-specified
Processing	Offline	Offline	Online
Execution Frequency	Once	Once	Once or periodically
Demarcation	Researcher-defined	Tool-defined	Developer-defined

execute, size of the executable, and power consumption have been popular targets for the GI community. It is self-evident that measuring non-functional characteristics requires some execution of the variant in order to evaluate the effectiveness of the transformation. We argue that in order to realistically gauge the effectiveness of a transformation, it is important to run the software *in-situ*, that is within the environment that the improved software should be run. By having an embedded variant generator within the application to be improved, we can get a very realistic idea of the true performance of an application. For example, power consumption is dependent on which opcodes are executed on the hardware, on the parameters passed to those opcodes, the order in which they run, and how many instructions are executed. Performance can be heavily impacted by the degree to which variable sizes are aligned to cache lines, as well as general contention around computational resources. By having an online optimisation process using an embedded variant generator, we can gain access to 'real' performance characteristic data rather than the simulations to which we are currently limited.

3. EXAMPLES OF THE EDI APPROACH

Gen-O-Fix [9] acts as an embedded monitor system for improving programs hosted on the Java™ Virtual Machine. As a prototypical exemplar of online GI, it can improve subsystems while the host system is running and output source and binary snapshots of the current state of the variant subsystem. The inputs provided to Gen-O-Fix by the developer are a callback to the subsystem functionality to be improved and a fitness function that can determine the quality of the variant.

Templar [8] is a generic framework for *template method hyper-heuristics*: the developer provides an algorithm skeleton, together with a list of the variation points. The framework then uses a hyper-heuristic training phase which operates above traditional GP to generate optimized implementations for these variation points. A Templar invocation can be embedded directly into the code of any Java-compatible language. As specified by a single parameter when the embedding is constructed, the developer can elect to have the hyper-heuristic layer re-train on an online distribution in one of several ways: when the embedding is performed; when the algorithm itself is first invoked; synchronously, every user-specified number of invocations; asynchronously, every user-specified period of time.

AntBox [4] is a framework for the search-based construction of Java objects within an existing program. Part of the motivation for AntBox is to help developers make an easier transition to Search Based Software Engineering techniques, so the API for the search process is presented in terms of the popular dependency injection framework Google Guice (`https://github.com/google/guice`). By this means, the mechanics of the search process (which uses Ant Programming [7] to build the object construction graph) are hidden from developers, necessitating them only to specify the properties of the object to be constructed as a constraint DSL.

4. CONCLUSION

We have outlined an approach for online improvement of existing programs which attempts to reconcile some of the benefits of Genetic Programming (e.g. creation of arbitrary expressions *ex nihilo*) with the ability to achieve modification of a sizeable body of program code. This is achieved by two main mechanisms: firstly contiguous subsets of program code to be varied are explicitly denoted by the developer and secondly the variant generator is embedded within the running application. This tight integration between variation and execution environments allows adaptive learning to take place with developer-specified periodicity. We believe that there are interesting historical parallels with the automated background activity of garbage collection: while this doubtless appeared exotic to first-generation 'C' programmers, it is now a standard part of all modern languages. In the same manner (and irrespective of the specific learning or variation mechanisms employed), one might envisage that explicit annotation of subsystems for automated online improvement could easily become an established part of the development toolchain.

Acknowledgements

Work funded by UK EPSRC grant EP/J017515/1 (DAASE).

5. REFERENCES

[1] E. T. Barr et al. The plastic surgery hypothesis. Foundations of Software Engineering. ACM, 2014.
[2] M. Harman et al. Genetic improvement for adaptive software engineering. SEAMS. ACM, 2014.
[3] Z. A. Kocsis and J. Swan. Asymptotic genetic improvement programming via type functors and catamorphisms. In *Semantic GP Workshop, PPSN*, 2014.
[4] Z. A. Kocsis and J. Swan. AntBox - a dependency injection container for embedded automatic improvement programming. Technical Report YCS-2015-497, Uni. of York, Apr 2015.
[5] W. B. Langdon and M. Harman. Optimising existing software with genetic programming. *IEEE TEC*, 2013.
[6] C. Le Goues, S. Forrest, and W. Weimer. Current challenges in automatic software repair. *Software Quality Jornal*, 21:421–443, 2013.
[7] O. Roux and C. Fonlupt. Ant programming. In M. Dorigo, editor, *ANTS*, 2000.
[8] J. Swan and N. Burles. Templar - A framework for Template-Method Hyper-Heuristics. In P. Machado et al., editors, *Genetic Programming, LNCS 9025*. 2015.
[9] J. Swan, M. G. Epitropakis, and J. R. Woodward. Gen-O-Fix: An embeddable framework for Dynamic Adaptive Genetic Improvement Programming. Technical Report CSM-195, Uni. of Stirling, Jan 2014.