Towards Modular Large-Scale Darwinian Software Improvement

Michael Orlov
Shamoon College of Engineering
Beer Sheva, Israel
orlovm@noexec.org

ABSTRACT

This paper proposes to explore a software engineer-assisted method for evolutionarily improving large-scale software systems. A framework is outlined for selecting and evolving specific components of such systems, while avoiding treating the complete software as a single independent individual in the population, thereby forgoing the high costs of that approach.

CCS CONCEPTS

• **Software and its engineering → Search-based software engineering**; **Genetic programming**; *Ultra-large-scale systems*;

KEYWORDS

Genetic improvement, genetic programming, large-scale software systems

ACM Reference Format:
Michael Orlov. 2018. Towards Modular Large-Scale Darwinian Software Improvement. In *GECCO '18 Companion: Genetic and Evolutionary Computation Conference Companion, July 15–19, 2018, Kyoto, Japan.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3205651.3208311

1 POSITION

Software complexity and size is constantly increasing [2]. If we are ever able to automatically improve large-scale software systems using evolutionary computation, we need to explicitly address the difficulty of applying traditional evolutionary methods to such systems. At present, there exist only few examples of improving large-scale software that are not limited to specific use cases like bug repair [1]. This paper attempts to analyze why this is the case, and to propose a practical pathway for extending the approaches that are in use at present.

2 LARGE-SCALE SYSTEMS

When working with large-scale software systems [2], it is typical for them to have the following properties:

- very large amount of code;
- long startup and shutdown times;
- complex logic that is hard to formalize algorithmically or via unit tests;
- dependency on external systems with state, such as databases.

These features, not exhaustive by any means, make it hard to evolve the system as-is, by representing all of its code as a single individual in a large population. Large amount of code makes the search space impractical, unless focusing on established methodologies like bug fixing [5]. Long startup and shutdown affect evaluation time, prohibiting use of non-trivial population sizes, and demanding methods like grid computing for even the smallest of runs. Unformalizable logic makes it impossible to produce reliable code for critical infrastructure, although it might be possible to alleviate the issue with automatically evolved test cases [3]. Finally, dependency on external systems does not allow treating the code as a single, independent evolutionary unit due to introduction of side effects.

There are techniques to automatically limit the search space [4], so code size per se is not an insurmountable problem. However, the other issues mentioned above pose a barrier to adopting automatic software improvement for software that cannot be formalized as a black box with negligible cost of evaluating modifications. Below, we propose a framework that allows a software engineer to integrate automatic evolution into existing projects, without having to face aforementioned drawbacks.

3 MODULAR EVOLUTION

In order to be able to automatically improve large-scale software systems, we need to accept that a software engineer must adapt the system to genetic improvement at design stage. The problems described in Section 2 stem from taking an engineered system, and attempting to evolve it by supplying exclusively external constraints such as: functional or non-functional fitness function; bias towards modification of certain parts of code; bias towards allowed types of modifications, and so on. If a method were devised for evolution of designated components in a *live* (i.e., running) software system, it is possible that the problematic issues could be sidestepped altogether.

Evolving software components in a live system requires a way to evaluate individual modified components independently, without having to restart the complete system and all its dependent external modules. Thus, the engineer needs to employ a specific API for modules to be evolved, covering:

- module initialization and shutdown logic;
- module evaluation function;
- strict functional restrictions;
- strict and soft non-functional restrictions.

The purpose of this API, as illustrated in Figure 1, is a trilateral separation of the live large-scale software system, the requirements of component(s) that are to be automatically improved, and the evolutionary engine that must not concern itself with the large-scale

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GECCO '18 Companion, July 15–19, 2018, Kyoto, Japan
© 2018 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5764-7/18/07…$15.00
https://doi.org/10.1145/3205651.3208311

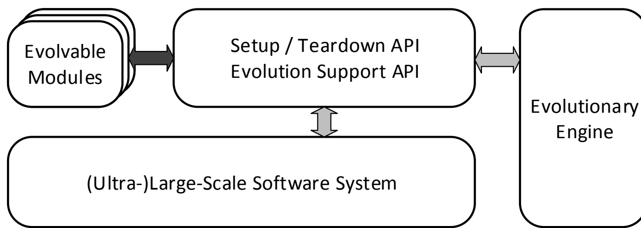

Figure 1: Conceptual architecture of modular large-scale software improvement.

system as a whole, but only with its smaller designated modules. It should be noted that the API is somewhat similar to those used in unit testing frameworks, and could perhaps be integrated with an existing framework like JUnit and its variants[1].

4 USE CASES

Consider a typical evolutionary benchmark problem: evolving a car racing controller. Loiacono et al. [6] set up a challenge based on TORCS open-source 3D car racing simulator[2]. TORCS is a complex engine requiring elaborate and time-consuming initialization with each different controller that we are supposedly trying to evolve. However, the engine code could probably be modified to support dynamic loading of controllers, allowing for significantly faster evaluation of individuals. Functional and non-functional restrictions would not be necessary, since this challenge was specifically designed to make applying evolutionary methods as easy as possible.

A more realistic use case scenario, such as an industrial software-based factory control system would require a much more sophisticated setup. Certainly, no one would (hopefully) start up and shutdown the control system just for the sake of evaluating its single modification, with obvious efficiency and security implications. However, an engineer could designate specific modules in the system that need to be automatically improved — for instance, a controller for a system of valves that is tasked with maximizing the throughput of a high-viscosity fluid through a set of pipes.

The engineer would then likely need to implement the following functionality in order to allow for modular software improvement of the selected component:

- quick setup and release of valves access during initialization and shutdown;
- logging of current component specifications and outcomes;
- fitness evaluation of current component performance during and after its operation;
- restrictions on allowed operations by the component, preventing it from causing physical damage to the system;
- security restrictions on the component to comply with regulatory requirements on unverified code — e.g., a sandboxed execution;
- code size, memory and other resource-related strict and soft restrictions stemming from system limits.

[1]See https://junit.org/
[2]See http://torcs.org

Viability of this approach might be the one to make the difference between "let's not!" and "what is the procedure?" outlook of managers and engineers on genetic improvement of complex and critical large-scale infrastructure systems.

5 IMPLEMENTATION

As mentioned in Section 3, unit-testing frameworks are good candidates for adding modular software improvement functionality. Frameworks like JUnit typically allow annotating code with having relevance to certain functions and classes, which can be used to define the API described previously in a developer-friendly fashion. The developers need not concern themselves with evolutionary engine details, as that part can be handled by the extended framework. There is no reason why a researcher-friendly framework like ECJ [7] cannot be used in such a setup.

6 CONCLUSION

This paper described an early-stage practical approach to evolutionarily improving large-scale and ultra-large-scale software systems. With software systems growing in size with no limit in sight, it is clear that we will eventually have to face the need to automatically improve systems that are actually of scale. It is, however, unclear which method of handling that unique problem is preferred by the genetic improvement community. Should the evolutionary engine locate the code parts to improve or repair by itself, or should the software engineer assist with that task [8, 9]? This question has both practical and fundamental considerations, due to the inherent conflict presented by applying a nature-inspired method to an engineered system. Hopefully, a proof of concept of the approach presented here will soon allow to make progress in this debate.

REFERENCES

[1] Andrea Arcuri. 2008. On the Automation of Fixing Software Bugs. In *Companion of the 30th International Conference on Software Engineering (ICSE Companion '08)*. ACM, New York, NY, USA, 1003–1006. https://doi.org/10.1145/1370175.1370223
[2] Peter Feiler, Richard P. Gabriel, John Goodenough, Rick Linger, Tom Longstaff, Rick Kazman, Mark Klein, Linda Northrop, Douglas Schmidt, Kevin Sullivan, and Kurt Wallnau. 2006. *Ultra-Large-Scale Systems – The Software Challenge of the Future*. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213-3890, USA. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=30519
[3] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (Sept. 2011), 649–678. https://doi.org/10.1109/TSE.2010.62
[4] William B. Langdon and Mark Harman. 2015. Optimizing Existing Software with Genetic Programming. *IEEE Transactions on Evolutionary Computation* 19, 1 (Feb. 2015), 118–135. https://doi.org/10.1109/TEVC.2013.2281544
[5] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 bugs for $8 Each. In *34th International Conference on Software Engineering (ICSE 2012)*, Martin Glinz (Ed.). IEEE, Zurich, 3–13. https://doi.org/10.1109/ICSE.2012.6227211
[6] Daniele Loiacono, Julian Togelius, Pier Luca Lanzi, Leonard Kinnaird-Heether, Simon M Lucas, Matt Simmerson, Diego Perez, Robert G Reynolds, and Yago Saez. 2008. The WCCI 2008 Simulated Car Racing competition. In *2008 IEEE Symposium On Computational Intelligence and Games*, Philip Hingston and Luigi Barone (Eds.). IEEE, Perth, WA, Australia, 119–126. https://doi.org/10.1109/CIG.2008.5035630
[7] Sean Luke. 2017. ECJ then and Now. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '17)*. ACM, New York, NY, USA, 1223–1230. https://doi.org/10.1145/3067695.3082467
[8] Kwaku Yeboah-Antwi and Benoit Baudry. 2017. Online Genetic Improvement on the Java Virtual Machine with ECSELR. *Genetic Programming and Evolvable Machines* 18, 1 (March 2017), 83–109. https://doi.org/10.1007/s10710-016-9278-4
[9] Shin Yoo. 2017. Embedding Genetic Improvement into Programming Languages. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '17)*. ACM, New York, NY, USA, 1551–1552. https://doi.org/10.1145/3067695.3082516