A Turing Test for Genetic Improvement

Afsoon Afzal, Jeremy Lacomis, Claire Le Goues, Christopher S. Timperley
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
afsoona,jlacomis,clegoues,ctimperley@cs.cmu.edu

ABSTRACT

Genetic improvement is a research field that aims to develop search-based techniques for improving existing code. GI has been used to automatically repair bugs, reduce energy consumption, and to improve run-time performance. In this paper, we reflect on the often-overlooked relationship between GI and developers within the context of continually evolving software systems. We introduce a distinction between *transparent* and *opaque* patches based on intended lifespan and developer interaction. Finally, we outline a Turing test for assessing the ability of a GI system to produce opaque patches that are acceptable to humans. This motivates research into the role GI systems will play in transparent development contexts.

ACM Reference format:
Afsoon Afzal, Jeremy Lacomis, Claire Le Goues, Christopher S. Timperley. 2018. A Turing Test for Genetic Improvement. In *Proceedings of GI'18: IEEE/ACM 4th International Genetic Improvement Workshop, Gothenburg, Sweden, June 2, 2018 (GI'18),* 2 pages.
DOI: 10.1145/3194810.3194817

1 INTRODUCTION

Genetic improvement (GI) applies search techniques to automatically improve existing software artifacts [6]. GI has demonstrated the ability to, amongst other things, automatically repair programs, at both the binary [7] and source-code level [3], and reduce resource consumption (e.g., run-time [8]).

We observe that existing GI work largely considers software systems as *static*, treating correctness with respect to a fixed set of requirements. In reality, software systems and development teams are constantly evolving as requirements are changed and new developers are hired. In these cases, code serves a dual purpose: it both communicates instructions to a computer, and communicates a developer's intent to other humans. We argue that code that is intended to be integrated into a *dynamic* development environment should be easy for developers to understand and modify. We envision a world where GI algorithms are trusted to work alongside humans in a development environment, and generated code is considered to be of equal quality to code written manually.

With this in mind, we propose a Turing test [10] for GI. In his seminal paper, Turing proposed to answer the question "Can machines think?", via an experiment wherein an interrogator is challenged to tell the difference between a hidden human and computer by asking them written questions and examining the answers. We believe that a similar experiment can be constructed to answer the question "Can machines generate code similar to humans?" This vision motivates a research agenda to interrogate the implications of code-generating bots interacting with humans in the socio-technical ecosystem that characterizes modern software development.

GI'18, Gothenburg, Sweden
 978-1-4503-5753-1/18/06…$15.00
DOI: 10.1145/3194810.3194817

2 OPAQUE VS. TRANSPARENT PATCHES

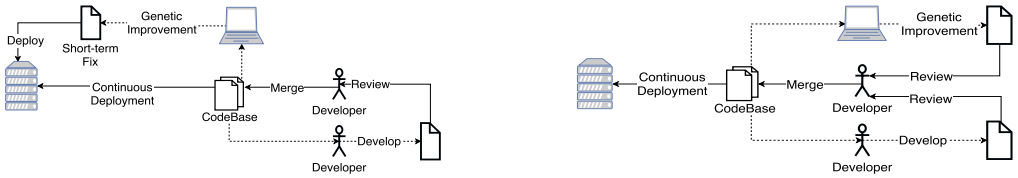
Initial assumptions about the requirements of a codebase influence the acceptability judgment of a generated patch. That is, the quality of GI patches have been measured against the immediate performance of the binary resulting from the improvement process, without considering the long-term impact the automatically-generated patch will have on the code.

When a patch is urgent, such as in security- or uptime-critical scenarios, patch maintainability or readability are indeed less important than semantics. Consider the recent reactionary operating system kernel patches for the Meltdown [4] and Spectre [2] bugs. The current patches for these bugs involve unintuitive strategies such as `retpolines`. In such cases, we agree with Monperrus' assertion that "we should not be afraid of alien ways of writing code." [5], and we should not limit the types of patches GI might generate. We call patches generated without thought to maintainability *opaque*, because they are not primarily intended to be read by humans. Figure 1a depicts an uptime-critical system which benefits from *opaque* patches. A common example of opaque code is compiler output, which can be unintuitive, but is highly performant.

However, rapid repair of serious bugs is only one use case envisioned for GI. Another goal of GI is to reduce development overhead through integration with the development process by, e.g., patching simple bugs or regressions and automatically integrating them into the codebase. This could free up valuable developer time for more difficult problems. Since the goal in this *dynamic* setting is to directly patch actively developed source code, patch understandability is paramount. We refer to these types of patches as *transparent*: their source code is intended to be seen and used by developers, shown by Figure 1b. We focus in this paper on transparent patches.

3 THE DEVELOPER IMITATION GAME

Code that is integrated into a dynamic development environment must be both comprehensible and maintainable to have a long lifetime. Even fully correct code will change as requirements evolve, and so code must be modifiable. Comprehensible code also reduces onboarding time, since new developers can struggle with difficult-to-read code. As such, transparent patches, which are intended to be read and modified by humans, should have qualities similar to code written by humans. In addition to the advantages to human developers, biasing code generation toward human-like patches may increase the subjective "acceptability" of patches generated by program repair tools [1].



(a) Opaque **(b) Transparent**

Figure 1: Opaque vs. transparent GI. Dashed arrows represent code production; solid arrows, code management.

We propose a Turing-style test to evaluate the "human-ness" of a GI system, as follows: a human interrogator specifies a desired patch to both a human developer and a GI system. Each competitor returns a generated patch to the interrogator, who attempts to determine which of the patches was generated by the GI system, and which was generated by the human. If the GI system successfully convinces the interrogator that the patch was written by a human, then it is has "won" the game. As with the classic Turing test, multiple rounds of interaction would be required. With multiple rounds, and different interrogators, it would be possible to assign a continuous value to the "humanity" of a GI system, based on how often it successfully deceives an interrogator. This could be one metric for comparing GI algorithms.

Much like Turing's original formulation of the test, our version is largely hypothetical, but our idealized version provides guidelines for practical real-world experiments. For example, researchers might examine how automated bots interact with developers in socio-technical ecosystems such as GitHub. GitHub plugins are already increasingly automating previously human-driven tasks (e.g., testing, refactoring, code review); however, social signals remain a critical part of the social coding process [9]. One experimental possibility is to modify human- and machine-generated patches such that each appear to come from the other. Are developers more willing to accept automatically generated patches if they more closely resemble (or are claimed to be) developer-written patches? Similarly, would a human-generated patch be more or less likely to be merged if it appeared automatically generated?

It will always be as important for GI patches to be reviewed as carefully as human-written patches. Both GI techniques and human developers change program semantics when patching; both generated patches and humans can make mistakes. We speculate that code with more human qualities will help mitigate biases of code reviewers and encourage a healthy level of skepticism. It would be unproductive for a reviewer with a deep mistrust in automated repair to reject all GI patches; it would also be ill-advised to blindly accept them as if they were generated by semantics-preserving techniques such as compilation. A sensible middle ground seems to be treating all patches as if they were generated by a trusted, competent, yet still fallible human developer.

4 CONCLUSION

Software systems are constantly evolving. Code is added, removed, and rewritten in response to continually shifting requirements. Notions of performance and even correctness can adjust in response to customer appetites, privacy and security concerns, and changing platforms. However, thus far, GI has mostly concerned itself with the improvement static software systems against a fixed set of requirements (e.g., test cases, benchmark workloads). We argue that GI should embrace the evolving nature of software systems by considering the lifetime of its proposed improvements.

To support this argument, we introduce *patch opacity*. *Opaque* patches are unchecked by the developer, and may affect either source code or a binary. Such patches are similar to compiler optimizations: they should alter program behavior in predictable ways to achieve improvements that are ephemeral to the codebase. By contrast, *transparent* patches will persist in the codebase, alongside changes made by developers. Therefore, transparent patches must also possess qualities that are intrinsic to (good) human-produced patches, such as clarity, maintainability, and stylistic consistency.

To capture the needs of a transparent GI system, we outline a modified Turing test adapted to patches, along with several possible research implications. With it in mind, we challenge the GI community to consider and frame their techniques within the richer context of software evolution.

ACKNOWLEDGEMENTS

This research was partially funded by AFRL (#FA8750-15-2-0075) and DARPA (#FA8750-16-2-0042); the authors are grateful for their support. Any opinions, findings, or recommendations expressed are those of the authors and do not necessarily reflect those of the US Government.

REFERENCES

[1] D. Kim, J. Nam, J. Song, and S. Kim. 2013. Automatic Patch Generation Learned from Human-written Patches. In *ICSE 13*. 802–811.
[2] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints* (Jan. 2018).
[3] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *International Conference on Software Engineering (ICSE '12)*. 3–13.
[4] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. 2018. Meltdown. *ArXiv e-prints* (Jan. 2018).
[5] M. Monperrus. 2014. A Critical Review of "Automatic Patch Generation Learned from Human-Written Patches": Essay on the Problem Statement and the Evaluation of Automatic Software Repair. In *ICSE '14*. 234–242.
[6] J. Petke, S. Haraldsson, M. Harman, W. Langdon, D. White, and J. Woodward. 2018. Genetic Improvement of Software: A Comprehensive Survey. *IEEE Transactions on Evolutionary Computation* PP, 99 (2018), 1–1.
[7] E. Schulte, J. DiLorenzo, W. Weimer, and S. Forrest. 2013. Automated Repair of Binary and Assembly Programs for Cooperating Embedded Devices. In *ASPLOS '13*. 317–328.
[8] P. Sitthi-Amorn, N. Modly, W. Weimer, and J. Lawrence. 2011. Genetic Programming for Shader Simplification. *ACM Trans. Graph.* 30, 6, Article 152 (Dec. 2011), 12 pages.
[9] J. Tsay, L. Dabbish, and J. Herbsleb. 2014. Influence of Social and Technical Factors for Evaluating Contribution in GitHub. In *ICSE '14*. 356–366.
[10] A. M. Turing. 1950. Computing Machinery and Intelligence. *Mind* 59 (1950), 433–460.