The Case for Software Evolution

Claire Le Goues
University of Virginia
legoues@cs.virginia.edu

Stephanie Forrest *
University of New Mexico
forrest@cs.unm.edu

Westley Weimer
University of Virginia
weimer@cs.virginia.edu

ABSTRACT

Many software systems exceed our human ability to comprehend and manage, and they continue to contain unacceptable errors. This is an unintended consequence of Moore's Law, which has led to increases in system size, complexity, and interconnectedness. Yet, software is still primarily created, modified, and maintained by humans. The interactions among heterogeneous programs, machines and human operators has reached a level of complexity rivaling that of some biological ecosystems. By viewing software as an evolving complex system, researchers could incorporate biologically inspired mechanisms and employ the quantitative analysis methods of evolutionary biology. This approach could improve our understanding and analysis of software; it could lead to robust methods for automatically writing, debugging and improving code; and it could improve predictions about functional and structural transitions as scale increases. In the short term, an evolutionary perspective challenges several research assumptions, enabling advances in error detection, correction, and prevention.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; I.2.2 [**Artificial Intelligence**]: Automatic Programming; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

General Terms

Algorithms

Keywords

Software engineering, genetic programming, program repair, evolutionary computation

FoSER 2010 November 7-8, 2010, Santa Fe, New Mexico, USA

1. INTRODUCTION

More than 30 years of research in software engineering and programming languages have given us improved techniques for developing, debugging, maintaining, modifying, and testing software systems. One aim of this research is to allow programmers to more easily design and construct systems that correctly satisfy their requirements when deployed. This research highlights a key aspect of modern software development: Complex software systems are still largely developed, maintained, and verified by humans.

Significant research has been undertaken to ease human-centered software engineering. Topics include testing and test-suite generation (e.g., [15, 31]), languages such as Alloy [19] that help programmers model code more accurately, error detection based on existing specifications (e.g., [5]), mined specifications (e.g., [1]), heuristics and best practices (e.g., [4, 8]), and even source control histories and other development artifacts (e.g., [38]), to outline just a small fraction of existing approaches. The problems of identifying and localizing different types of errors have received individual treatment, from null pointer exceptions (e.g., [17]), to buffer overruns (e.g., [34]), to SQL injection vulnerabilities (e.g., [35]), among many. Other techniques take a different approach, focusing on building programs that are correct by construction (e.g., [22]) or more easily verified (e.g., [37]), or on synthesizing code from specifications (e.g., [32]), reducing the need for post-hoc bug finding and localization.

Despite these advances, however, software today remains, in many ways, far less reliable and more prone to bugs than in the past. The number of reported defects appears to be growing at unsustainable rates. For example, in 2005, one Mozilla developer claimed that, "everyday, almost 300 bugs appear [...] far too much for only the Mozilla programmers to handle" [2, p. 363]. In the absence of sufficient resources to repair them, mature software projects are forced to ship with both known and unknown bugs [23]. Bugs in delivered software are time-consuming to identify, localize, and repair. This can be seen in reports about maintenance, which is defined as any modification made on a system after its delivery. Changing existing code, repairing defects, and otherwise evolving software are major components of maintenance [27], in some cases accounting for up to 90% of the total cost of a typical software project [30], at a total cost of up to $70 billion per year in the US [21, 33]. This time and money add up to serious economic consequences: Deployed programs with incorrect behavior cost billions of dollars each year [25].

As society increases its dependence on software and net-

worked computing infrastructures, and as the complexity of those systems grows, current software engineering practices have become unsustainable. The complexity, diversity, functionality and interconnectedness of software and software systems have increased to the point that they exceed human comprehension. We postulate that the complexity of modern software necessitates a transition to more fully automated software development and maintenance — in short, it is time to "get the human out of the loop."

Rather than viewing software as an externally organized composition of components, we advocate a view similar to that of Ref. [12], that software can be best understood and analyzed as an *evolving complex system*, exemplified by living systems. This perspective allows important insights about how humans or machines can understand, develop, and modify complex systems. Notably, recent advances in fields such as evolutionary algorithms, adaptive systems, and theoretical biology can be profitably applied in the domain of software engineering.

The remainder of this short paper explores the idea that modern software engineering can benefit from viewing software as an evolving complex system.

2. SOFTWARE AS AN EVOLVING SYSTEM

Software today is deployed in highly dynamic environments. Nearly every aspect of a computational system is likely to change during its normal life cycle. For example: new users who interact with software in novel ways, often finding new uses for old code; the specification of what the system is supposed to do; the owner and maintainers of the system; the system software and libraries on which the computation depends, and the hardware on which the system is deployed. These changes are routine and continual:

> "Turning to evolution, we see that the history of manufactured computers is a truly evolutionary history, and evolution does not anticipate, it reacts. To the degree that a system is large enough and distributed enough that there is no effective single point of control for the whole system, we must expect evolutionary forces — or 'market forces' — to be significant. In the case of computing, this happens both at the technical level through unanticipated uses and interactions of components as technology develops and at the social level from cultural and economic pressures. Having humans in the loop of an evolutionary process, with all their marvelous cognitive and predictive abilities, with all their philosophical ability to frame intentions, does not necessarily change the nature of the evolutionary process. There is much to be gained by recognizing and accepting that our computational systems resemble naturally evolving systems much more closely than they resemble engineered artifacts such as bridges or buildings. Specifically, the strategies that we adopt to understand, control, interact with, and influence the design of computational systems will be different once we understand them as ongoing evolutionary processes." [12]

The evolutionary perspective on software is just one example of how technological progress generally can be understood as a product of evolution. Arthur argues in a recent book [3] that new technologies are descended from novel combinations and improvements of older technologies, and Bela et al. show in Ref. [24] that many observed patterns such as Moore's Law or Wright's Law are likely driven by evolutionary processes. Early antecedents of this idea appeared in the work of Rogers [29], who believed that the process by which innovations and ideas spread through culture resembles biological evolution. By extending this line of work, an evolutionary perspective on software development could lead to theoretical explanations and macro-laws, for example, theoretical predictions about distributions of bug sizes [7], scaling of development time with project size, and so forth.

To summarize, programs are already subject to the evolutionary pressures and large-scale patterns of biology, except they are implemented by hand. And humans modify software in ways that resemble the mechanisms of natural evolution — copying code from one program to another (inheritance), making small modifications to existing code (mutation), and combining program modules together in mashups (recombination). When rightfully understood, these processes can potentially be automated using the methods of evolutionary computation, leading to a more automated form software evolution.

3. BIO-INSPIRED COMPUTATION

Viewing software as an evolving complex system suggests promising mechanisms and design principles that can be applied to software, including adaptation and learning, diversity, disposability of components, redundancy, and homeostasis [12].

As a concrete example, recent research has shown the applicability of a computational form of evolution known as *genetic programming* (GP) to the problem of automatic error repair [14, 36]. Although fixing localized bugs in preexisting code is a long way from the vision of an automatically evolving software ecosystem outlined above, it is a first step and proof of principle.

To review, the automatic error repair technique assumes access to a program's source code (in C), a test case that encodes a bug (a *negative* test case that the program currently fails), and a set of test cases that encode required program behavior (*positive* test cases). Fault localization distinguishes between statements executed only by the negative test case and those executed by both positive and negative test cases; this isolates the error and reduces the search space. The repair process randomly mutates and recombines subtrees of the program's abstract syntax tree; fault-localized areas are more likely to undergo change. Mutations are drawn exclusively from elsewhere in the program, and may either insert a statement somewhere along the negative test case's execution path, or delete a statement along that path. These mutations produce variants of the original program, which, if successfully compiled, are evaluated by the provided test cases. The search is successful when it finds a program variant that passes all of the test cases, positive and negative.

This technique has been shown to repair a wide variety of error types, including buffer overruns, denial of service attacks, and format string vulnerabilities, in legacy C software totaling over two millions lines of code, in under five minutes, on average.

The success of this technique says as much about the nature of software as it does about genetic programming. In particular, it highlights the strong analogy between software and living, complex evolving systems. For example, the method assumes that a program "contains the seeds of its own repair"—potential fixes are adapted from other parts of the program. This illustrates redundancy of functionality even in software executing in isolation (and not in a complex ecology). Similarly, many bugs are repaired currently by manually copying code from one location to another, perhaps to propagate fixes between otherwise similar code, or simply because it is easy. In this regard, human evolution of software resembles the mechanisms of biological evolution.

4. CHALLENGED ASSUMPTIONS

Shifting to an evolutionary perspective on software will challenge several assumptions in current software engineering research practice.

First, viewing software as an evolving system can **release program analyses from the "shackles of soundness"**, thereby influencing their development and application. The complexity of software systems is one reason that bugs and errors that are so often not discovered in pre-release testing or development phases. This complexity arises from non-linearities and interactions with continuously evolving computation environments; in this setting, the idea of finding *a priori*, mathematically precise proofs of program correctness, either using testing or verification techniques, is problematic. Biological complex systems, similarly, do not rely on *a priori* correctness for continued success. We conjecture that program analysis techniques could profitably focus on fulfilling alternative definitions of utility and on features that enable adaptation in practice. Relaxed assumptions about soundness may explain the success and industrial uptake of several research approaches that make no guarantees about false positive or negative rates. Instead, these systems focus on finding useful, actionable bug reports based on heuristics and common coding error patterns [4, 8, 10].

As an alternative to the heuristics used by these and similar approaches, correct behavior for a given program could be encoded in **evolving test suites**. Such test suites, found much more commonly in practice than formal specifications, may act as a useful proxy for specifications and are already used by a number of recent program analysis techniques (e.g. [20]). This is the approach taken by evolutionary program repair (Section 3), which does not require special program annotations or coding practices [36].

Before evolving test suites can gain acceptance as a proxy for correctness, we need additional support for constructing them. Although automated test case generation is a popular topic, in practice, scalable approaches produce test inputs, not full test cases (e.g., [15, 31]). They may generate successive inputs that help maximize code coverage, for example, but, in the terminology of the oracle-comparator model [9], they do not produce comparators. Unless the program crashes, it is often difficult to determine automatically whether the program passes the test case when presented with the input. We believe that this is a fruitful area for future research, one that would enhance our existing methods for assessing the fitness of evolving software.

Shifting focus away from soundness as the program analysis goal challenges common definitions of **acceptability** and presents a need for different metrics and benchmarks. This is the general approach taken by Rinard *et al.* in their failure-oblivious computing models, where programs are dynamically modified to continue execution in the face of an error such as a buffer overrun [26, 28]. Continued execution of an adapted or evolved system may often be a better outcome than complete failure.

An evolutionary approach to software challenges the principle of **separation of concerns**. Although biological systems have evolved many boundaries and interfaces for enforcing modularity (e.g., the cell wall), biological implementations are considerably richer than is typically seen in computing. For example, the strict hardware/software abstraction, on which so much of computer science relies, is likely ill-suited to robust computational behavior in the dynamic and energy-constrained, environments we project for the future. We view research to use multi-core architectures to gain correctness, rather than speed, as a positive step in this direction (e.g., [16]).

Finally, viewing software as an evolving, complex system will encourage researchers and practitioners alike to **accept software diversity** as both an inevitable result of biologically-inspired analyses and program modification techniques and as a desirable state of affairs for a system of software deployments. For example, automatic, individualized software evolution in response to errors—a potential side-effect of evolutionary program repair [36]— is likely to compromise the consistency of a single program across multiple deployments in different environments. Although such diversity may initially seem unappealing, n-variant systems have security advantages [11, 13]. Diversity is an important source of robustness in biological systems, providing protection against the spread of disease in populations and alternative pathways that allow functionality to be maintained when one particular pathway is disrupted. Diversity has already proven valuable in practice in security research: instruction set randomization is an effective prevention for many common attacks [6, 18].

5. CONCLUSIONS

The overwhelming size and complexity of modern software systems, taken together with recent advances in fields such as adaptive systems, evolutionary computation and theoretical biology, suggest that it may be time to revisit the dream of 'automatic programming'—a vision dating back to earliest days of computing—with the goal of automating many aspects of the software development process.

Rethinking software as an evolving, complex system has the potential to dramatically change how software is developed and maintained. It can also provide theoretical predictions and analyses of how the software process is likely to operate over long time scales. More immediately, shifting our perspective on software could enable new approaches by challenging several long-held assumptions in current software engineering research practice. We believe that software engineering researchers have much to gain by viewing software systems through the lens of biological adaptive systems.

6. ACKNOWLEDGMENTS

The authors gratefully acknowledge the support of NSF grants CCF 0621900 (SF), CCR 0331580 (SF), CNS 0716478 (WW), CNS 0905373 (WW), CCF 0954024 (WW), and SHF 0905236; AFOSR MURI grant FA9550-07-1-0532, and the Santa Fe Institute.

7. REFERENCES

[1] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Principles of Programming Languages*, pages 4–16, 2002.

[2] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *International Conference on Software Engineering*, pages 361–370, 2006.

[3] W. B. Arthur. *The Nature of Technology: What it is and How it Evolves*. Simon and Schuster, 2009.

[4] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.

[5] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *Principles of Programming Languages*, pages 1–3, 2002.

[6] G. Barrantes, D. Ackley, S. Forrest, T. Palmer, D. Stefanovic, and D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Computer and Communications Security*, 2003.

[7] T. V. Belle. *Modularity and the Evolution of Software Evolvability*. PhD thesis, University of New Mexico, Albuquerque, NM, 2004.

[8] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. R. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, 2010.

[9] R. V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley, 1999.

[10] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw., Pract. Exper.*, 30(7):775–802, 2000.

[11] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: a secretless framework for security through diversity. In *USENIX Security Symposium*, 2006.

[12] S. Forrest, J. Balthrop, M. Glickman, and D. Ackley. Computation in the wild. In E. Jen, editor, *Robust Design: A Repertoire of Biological, Ecological, and Engineering Case Studies*, pages 207–230. Oxford University Press, 2004. Reprinted in K. Park and W. Willinger Eds. *The Internet as a Large-Scale Complex System*, pp. 227-250. Oxford University Press (2005).

[13] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Sixth Workshop on Hot Topics in Operating Systems*, 1998.

[14] S. Forrest, W. Weimer, T. Nguyen, and C. Le Goues. A genetic programming approach to automated software repair. In *Genetic and Evolutionary Computing Conference*, pages 947–954, 2009.

[15] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Programming Language Design and Implementation*, pages 213–223, 2005.

[16] J. Ha, M. Arnold, S. M. Blackburn, and K. S. McKinley. A concurrent dynamic analysis framework for multicore hardware. In *Object oriented programming systems languages and applications*, pages 155–174, 2009.

[17] D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and tuning a static analysis to find null pointer bugs. *SIGSOFT Softw. Eng. Notes*, 31(1):13–19, 2006.

[18] W. Hu, J. Hiser, D. Williams, A. Filipi, J. W. Davidson, D. Evans, J. C. Knight, A. Nguyen-Tuong, and J. C. Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. In *Virtual Execution Environments*, pages 2–12, 2006.

[19] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[20] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Automated software engineering*, pages 273–282, 2005.

[21] M. Jorgensen and M. Shepperd. A systematic review of software development cost estimation studies. *IEEE Transactions on Software Engineering*, 33(1):33–53, 2007.

[22] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Principles of Programming Languages*, pages 42–54, 2006.

[23] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Programming language design and implementation*, pages 141–154, 2003.

[24] B. Nagy, J. D. Farmer, J. E. Trancik, and Q. M. Bui. Testing for laws of technological progress. Santa Fe Institute Technical Report, (2010).

[25] NIST. The economic impacts of inadequate infrastructure for software testing. Technical Report NIST Planning Report 02-3, NIST, May 2002.

[26] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Symposium on Operating Systems Principles*, pages 87–102, October 2009.

[27] C. V. Ramamoothy and W.-T. Tsai. Advances in software engineering. *IEEE Computer*, 29(10):47–58, 1996.

[28] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *Operating Systems Design and Implementation*, pages 303–316, 2004.

[29] E. M. Rogers. *Diffusion of Innovations*. Glencoe: Free Press, 1962.

[30] R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley, 2003.

[31] K. Sen. Concolic testing. In *Automated Software Engineering*, pages 571–572, 2007.

[32] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *Principles of Programming Languages*, pages 313–326, 2010.

[33] J. Sutherland. Business objects in corporate information systems. *ACM Comput. Surv.*, 27(2):274–276, 1995.

[34] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Networking and Distributed System Security Symposium*, Feb. 2000.

[35] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Programming Languages Design and Implementation*, pages 32–41, 2007.

[36] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering*, pages 364–367, 2009.

[37] X. Yin, J. C. Knight, E. A. Nguyen, and W. Weimer. Formal verification by reverse synthesis. In *Computer Safety, Reliability, and Security*, pages 305–319, 2008.

[38] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Foundations of Software Engineering*, pages 91–100, 2009.