Computer Science Can Use More Science

Clayton T. Morrison and Richard T. Snodgrass
University of Arizona

October 27, 2010

1 Introduction

The so-called LAMP stack (Linux OS/Apache internet server/MySQL DBMS/Perl PL) consists of 10 million lines of code [Neville-Neil 2008], interacting in myriad ways to achieve impressive functionality and performance. This approaches the intellectual complexity of the Saturn V rocket (with three million parts) which took man to the moon. The similarities between these two enormous engineering feats are important: good engineering design practices have been followed; requirements were defined and met; rigorous testing and debugging has taken place. Yet, to date, the LAMP stack is much less well understood than the Saturn V rocket. It is much harder to predict how the LAMP stack will perform under varying conditions and where things might go wrong than it is to consider how the Saturn V may behave in different operating environments.

What the engineers at NASA have that the developers and users of the LAMP stack do not is an understanding of how configurations of system components (in NASA's case, physical materials) will behave in a variety of (physical) contexts and an idea of where the boundary conditions of those behaviors lie. This understanding is the product of physical theories about universal laws of nature—laws that have been identified through a tradition of model construction and empirical testing to produce general principles. Developers of the LAMP stack do not approach this level of understanding.

If computer science can achieve this level, that is, uncover the kinds of scientific theories and laws that physics provides, software developers building artifacts could do what NASA engineers do now: analyze their designs according to underlying theories and predict how a working system will behave. Extracting such laws and principles is the goal of science.

Whether and how computer science is a science has been a topic of discussion since the beginnings of the field. Thinkers including Herbert Simon [Simon 1996] have offered deep insight into the special nature of computation and how it exists as a phenomenon—in part, it's artificial, something we create. However, the nature of what computer science is studying, or how best to study it, is by no means a settled topic. Just because we design and build computational systems does not mean we understand them; special care and much more work is needed to correctly characterize computation as a scientific endeavor.

The early challenges of the field were engineering in nature, leading to a de-emphasis of empirical methods. In recent years, with increasing complexity of computational systems, empirical methods are now needed to discover system limits and predict future behavior.

2 The Methodology of Debugging

Let's start with the parts of the scientific method we do well. Programmers engage naturally in one of the purest forms of empirical investigation: debugging. Whenever a program exhibits a fault, in which a run over particular input data yields an exception or incorrect output, the programmer generally follows a series of steps. First, the programmer develops a *hypothesis* of where the fault lies (whether in a particular statement or more generally with the logic of the design) and considers how to correct it. This hypothesis is based on the programmer's mental model of how the programming environment, referenced libraries, operating system, and environment (input data, system events, etc.) function and interact. This model is generated and refined through additional runs of the program with print statements or through interaction with a debugger in which the program state can be examined as the program is single-stepped. The programmer then *tests this hypothesis* by making changes believed to fix the problem and rerunning the program, predicting that the change will result in the correct output for that particular test input. If the program still faults, the programmer refines the hypothesis, updating their mental model of how they believe the underlying system is behaving, and further investigates possible causes. Programmers frequently demonstrate that they are highly skilled at understanding complex interactions by forming models and testing and revising them.

3 Toward General Predictive Models

Why, then, do we not have the same depth of understanding of the LAMP stack as the Saturn V rocket? Formulating hypotheses, devising tests and carrying them out, and then revising hypotheses based on test outcomes are critical components of the basic methodology of empirical science. But these activities are not all that science does. What is missing from the debugging picture is searching for generalizations: attempting to extract general patterns and identify the factors that govern their behavior. This requires looking past individual systems, past simply getting the program to run in a particular context, to characterizing general principles of system behavior. The predictive model underlying the hypothesis testing undertaken by our programmer is specific to the extreme.

Empirical generalization [Cohen 1995] progresses from description of particular instances to prediction (via models) of classes of systems. This progression is advanced to an explanation that connects what is going on in particular observed cases to a *general class*, in this case of certain kinds of software systems in certain kinds of configurations. The models and the kinds of systems they describe should both be broadly construed to render a true understanding of these systems. Are such general predictive models even possible for software systems? Experience indicates that when such systems are examined experimentally, with an explicit goal of discovering causal models, such models can in fact be found and highlight deep structure [Snodgrass 2010].

The articulation of scientific theories as predictive causal models, the methodology of empirical generalization, and the evaluation of such theories via hypothesis testing is prevalent in isolated sub-disciplines of computer science, including HCI, empirical software engineering, web science, and data mining. It is however rarely used in other sub-disciplines of computer science, especially those concerned with software systems artifacts, such as compilers, databases, networks, operating systems, and programming languages.

4 Examples of Computational Models

One example of a computational model that is the product of empirical generalization is the *Theory of Locality* [Denning 2005]. This model arose from a study of the cost of managing page transfers between main memory and a much slower disk drive, within the general area of operating systems. What Denning found was that data relevant to the current context of a running program tended to be grouped in space and time to local chunks, and this was the product of the way programs are written by humans, rather than due to any underlying constraints of the computing system. When our memory management algorithms respect this general pattern, performance improves dramatically. The resulting theory—that human-constructed information processing systems will exhibit locality—is inherently predictive and has been tested many, many times. This theory has subsequently been generalized to apply to computational systems of all kinds:

"in virtual memory to organize caches for address translation and to design the replacement algorithms, ... in buffers between computers and networks, ... in web browsers to hold recent web pages, ... in spread spectrum video streaming that bypasses network congestion" [Denning 2005, pp. 23–24].

Note what Denning did: He recognized a kind of behavior that spans a class of systems. He searched for the factors responsible for the behavior he and others had observed. And, *how* he searched is of utmost importance: he conducted experiments, adjusting the candidate factors responsible for producing the behavior, collected data and carefully analyzed it and used the results to update his hypotheses. Denning wasn't debugging and he wasn't deriving proofs; he was revising his model based on experimental data while expanding the model to characterize a larger class of computational systems that exhibit locality. In doing so, Denning was able to achieve greater clarity about this phenomenon and how and why a computational system will exhibit this phenomenon.

We don't yet know whether the laws of computational systems will cohere into explanatory frameworks like other natural sciences, or as elegantly as the theory of locality. For example, many physical systems are continuous in nature while most computer science domains are discrete. Discrete systems tend to exhibit chaotic behavior, in which small changes in inputs or initial conditions lead to dramatic differences in outcomes. It is not known what portion of computer science phenomena are chaotic. But even here, empirical and analytical tools are available to help, such as dynamic systems theory (Bradley provides an example of applying dynamic systems theory in a computer science context [Bradley 1999]). Discrete systems are analytically very challenging, but incorporating empirical tools will help expand our modes of analysis.

Another notable episode of empirical investigation in computer science, still unfolding, is in the study of where the hard problems lie within the parameter space of computationally hard (NP-complete) problems. Many of the problem instances turn out to be relatively easy to solve, while others are insurmountably difficult. Early models proposed that NP-complete problems would transition from easy to hard then back to easy around a threshold value of certain parameters. Subsequent studies have since systematically explored this phenomenon, and have discovered that where these transitions occur may depend on a variety of additional factors, including the kind of problem solving method used. Different problem solving methods provide us with different tools with which to study the complexity of these problems; an apt analogy has been made to astronomers using telescopes that operate on different light wavelengths to provide different perspectives on the structure of the cosmos [Coarfa *et al.* 2000]. The jury is still out on how to best characterize the transition behavior, but the key point for our purposes is to again focus on the methodology employed: these researchers conduct experiments, take measurements, and refine their models, while striving for generality.

5 Realizing the Benefits of More Science in CS

These examples and other extant computer science theories emphasize that by embracing the methodology of developing and evaluating predictive models through experimentation over multiple members of a class of software systems, a more complete understanding of such artifacts will emerge. In addition, this observational and experimental scientific perspective encourages the computer scientist to actively look for relevant phenomena that may have been missed because they aren't currently described by existing, closed-form analytical descriptions [Tichy 1998]. The developed explanatory model provides better prediction and ultimately more dependable products built on those principles. Finally, good scientific models will generate new questions to be answered and will drive our field to yet deeper understanding.

How can these benefits be realized? How might we change what we do? We can adapt our already very skilled hypothesis testing in debugging and broaden it by asking more general questions, identifying the classes of system properties that contribute to behavior, identifying their boundary conditions, and working to fit them into a unified picture. We can also more broadly adopt additional methodological tools, such as from statistics and dynamic systems; a number of computer science subfields already profitably do so.

The pristine presentations of scientific reasoning and the tremendous successes of such reasoning in other fields may appear to the practicing computer scientist as out of reach. But many of our colleagues have started down this path, the tools are accessible, and the promise great.

6 Acknowledgements

We thank the reviewers and editors for an enjoyable and vigorous conversation around the topic of this essay. The reviewers will not agree with every detail, but they have certainly helped us make this a better commentary.

References

[Bradley 1999] E. Bradley, "Time-series analysis," in M. Berthold and D. Hand, editors, **Intelligent Data Analysis: An Introduction**, Springer Verlag, 1999.

[Coarfa *et al.* 2000] C. Coarfa, D. D. Demopoulos, A. San Miguel Aguirre, D. Subramanian and M. Y. Vardi. "Random 3-SAT: The Plot Thickens," in *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming*, Springer LNCS, 2000, pp. 143-159.

[Cohen 1995] P. Cohen, **Empirical Methods for Artificial Intelligence**, MIT Press, 1995.

[Denning 2005] P. J. Denning, "The Locality Principle," *CACM* 48(7):19–24, July 2005.

[Neville-Neil 2008] G. V. Neville-Neil, "Code Spelunking Redux," *CACM* 51(10):36–41, October 2008.

[Simon 1996] H. A. Simon, **The Sciences of the Artificial**, Third Edition, The MIT Press, 1996.

[Snodgrass 2010] R. T. Snodgrass, "*Ergalics*: A Natural Science of Computation," February 2010. `http://www.cs.arizona.edu/projects/ergalics/whatis.html`

[Tichy 1998] W. F. Tichy. "Should Computer Scientists Experiment More?" *IEEE Computer* 31(5): 32-40, May 1998.

Clayton Morrison is a Research Assistant Professor of Computer Science at the University of Arizona. He has more than a decade of experience building artificial intelligence systems for DARPA, AFOSR and NSF projects. He holds a doctorate in philosophy.

Richard Snodgrass is a Professor of Computer Science at the University of Arizona. He is an ACM Fellow, has chaired the ACM Publications Board and SIGMOD, and has served as Editor-in-Chief of ACM Transactions on Database Systems.

`{clayton,rts}@cs.arizona.edu`
`http://www.arizona.edu/people/{clayton, rts}`