

Applying Machine Learning to Software Analysis to Achieve Trusted, Repeatable Scientific Computing

Producing trusted results from high-performance codes is essential for policy and has significant economic impact. We propose combining rigorous analytical methods with machine learning techniques to achieve the goal of repeatable, trustworthy scientific computing.

Stacy J. Prowell (corresponding) and Christopher T. Symons
Oak Ridge National Laboratory, prowellsj@ornl.gov
+1 (865) 241-8874

“The complexity of computer systems in terms of the amounts of data, the structure of data, and the algorithms, leads to a situation where one loses direct contact with the data. This again offers the possibility of presenting incorrect information, either as a consequence of errors or by deliberately adjusting the results. At the same time, it offers participants an excuse to accept positive data, even if it is clearly wrong” [1].

Economics, law, and public opinion all matter when writing policy, but increasingly scientific evidence informs all of these areas with ever shorter cycles. Scientific models provide the input to judge the economic impact of policy changes. Scientific studies are used to judge the efficacy of current and proposed legislation. Scientific breakthroughs are reported widely in the media and impact public opinion. In short, complex policy issues are increasingly informed by scientific evidence, and policy decisions have near-instantaneous impacts in the marketplace. This is scientific evidence-based policy-making, and it is here to stay.

Increasingly large-scale and high-performance computing provides this scientific evidence. Climate science is perhaps the most visible of these areas, where scientific evidence obtained through large-scale computing directly impacts policy decisions that have significant economic impact. In short, the results of high-performance codes can select winners and losers in the marketplace. This creates economic incentives to tamper with, or at least cast doubt on, high-performance codes.

Trust in these results is critical, and lack of trust has serious consequences. A recent peer-reviewed paper in the Chinese *Science Bulletin* casts doubt on the result of the world’s sizable investment in climate modeling and prediction by challenging the nature of the computations performed [5]. Another study tested a climate code on ten different architectures and found significant variation for different compilers, parallel libraries, and optimization levels [2]. Answering these challenges to reassure policy makers and the public is a serious challenge. Proper science depends on *reproducibility*, a quality that is potentially lacking in experiments performed on high-performance computers [3, 4]. This is a weak point critics can attack to undermine scientific results, and there are cases critics can cite to show the cost of computational errors, even in established algorithms [6].

Fortunately this challenge can be met by increasing our trust in the codes that are run, and several methods hold promise to do just this. *Rigorous verification* can tell us that we are solving the problem correctly; *static analysis*, especially of the compiled software, can tell us that we have (or have not) implemented our methods correctly; and *symbolic execution* can give us a much higher degree of confidence in our results even if we cannot fully analyze the programs by other means.

Not listed above is testing. While testing is certainly necessary to gain assurance that assumptions made during design are true, it is inadequate to demonstrate correctness or to establish trust. Often for the kinds of codes being run there is no oracle to determine if a test passed or failed (what is the true amount of warming we can expect in twenty years?), and large-scale testing of complete codes even when there are known data points is often impractical because of the computational cost of running the codes.

These methods – verification, static analysis, and symbolic execution – all require significant expertise to apply and bring their own challenges. Consider static analysis of compiled software. This is a very powerful tool that allows us to take into account compiler-introduced errors and optimizations that were not present in the original source and has the potential to answer the challenges posed in [2].

Many of the issues limiting the full application of the above methods are amenable to machine learning (ML) [7] solutions. Specifically, several of these challenges can be formulated as a distinction among several possible outputs based on many inputs. Moreover, the relations between said inputs and outputs are either uncertain or too complex to encode in a straightforward manner, but are amenable to expert analysis and decision making, which suggests the application of machine learning. Although these problems can consist of very high-dimensional input spaces, there is repeated structure across examples, shared structure across tasks, and the availability of (or the ability to generate) large example sets over which to discover said structure through advanced ML methods [8,9]. These problems are thus addressable in new ways thanks to advances in large-scale machine learning.

We propose a research program of combining machine learning techniques with rigorous software analysis to achieve both repeatability and trust in scientific computing. We can break static analysis of compiled software into common stages: disassembly, generation of an internal representation, and computation of functionality. Throughout this process theorem provers, model checkers, satisfiability-modulo-theorems (SMT) solvers, and term rewriters are employed to effectively reason about program semantics. Each of these stages has associated challenges that can be addressed through machine learning.

- Disassembly contains computed jumps and mixed data and code. Determining what is code and what is not often relies on heuristics. This area is ripe for application of ML techniques. Direct observation of the execution of similarly constructed programs yields the classifier for code vs. data needed to inform machine learning models.

- Generation of an internal representation may involve annotating code with partial results, such as reachability analysis, and re-structuring the code for easier analysis. For example, loops among parts of the program may be detected in this stage and the program transformed to identify the loop condition and body. Because loops may contain other loops or arbitrarily complex code, this transformation is not optimal and can result in code that is very difficult to analyze – whereas the discovery of the correct minimal loop predicate and properly-structured body could make loop analysis much simpler. This is an optimization choice among multiple competing solutions and, again, is directly amenable to a ML approach.
- Computation of functionality typically relies on term rewriting, model checking, SMT solvers, etc. Here we consider just term rewriting – which is likely common across all approaches. The system must choose and apply rewrite rules in some order to simplify expressions or prove theorems. Some of these results will recur in the program, and should be captured, and some will not. Some orderings of rules will terminate with a useful result, and some will either fail to terminate or do so without generating useful information. Heuristics are often employed to address this. Here again we have an optimization problem suitable for applying ML.

The application of ML techniques yields additional benefits beyond simply making these methods more tractable. By establishing a program that combines ML techniques with rigorous software analysis, we open new avenues for performance optimization and the discovery of under-exploited parallelism. Introducing ML techniques into term rewriting promises to yield dividends in computer algebra systems and theorem provers.

Establishing trust in computational results is an important and worthy goal. This can be achieved – in part – by rigorous software analysis methods. A research program is needed that addresses the challenges to full-scale application of rigorous methods by employing ML techniques where approximations and heuristics are used today.

- [1] Kai A. Olsen, “Is it the computer’s fault the results are wrong? A case from the Norwegian Ministry of Education and Research,” <http://home.himolde.no/~olsen/artikler/educationalnumbers.pdf> (retrieved on May 4, 2015).
- [2] Song-You Hong, *et. al.*, “An Evaluation of the Software System Dependency of a Global Atmospheric Model,” *Monthly Weather Review*, v. 141, n. 11, November 2013. <http://journals.ametsoc.org/doi/abs/10.1175/MWR-D-12-00352.1?af=R> (retrieved on May 4, 2015).
- [3] Sascha Hunold, “Can I Repeat Your Parallel Computing Experiment? Yes, You Can’t,” Proc. TUD ZIH Colloquium, Dresden, Germany. July 2013. https://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/veranstaltungen/zih_kolloquium/dateien/2013_11_07_SaschaHunold (retrieved on May 4, 2015).

- [4] Robert W. Robey, "Reproducibility for Parallel Computing," Proc. Supercomputing (SC13), Denver, Colorado. November 17-22, 2013. <http://sc13.supercomputing.org/sites/default/files/WorkshopsArchive/pdfs/wp110s1.pdf> (retrieved on May 4, 2015).
- [5] Christopher Monckton, *et. al.*, "Why Models Run Hot: Results From an Irreducibly Simple Climate Model," *Science Bulletin*, v. 60, n. 1, pp. 122-135, January 2015. <http://link.springer.com/article/10.1007/s11434-014-0699-2> (retrieved on May 4, 2015).
- [6] B. Jakobsen and F. Rosendahl, "The Sleipner Platform Accident," *Structural Engineering International*, v. 4, n. 3, pp. 190-193, 1994. http://www5.in.tum.de/~huckle/sleipner_Jakobsen.pdf (retrieved on May 4, 2015).
- [7] T. M. Mitchell, *Machine Learning*. McGraw-Hill, 1997.
- [8] O. Chapelle, B. Scholkopf, A. Zien, Eds., *Semi-Supervised Learning*. MIT Press, 2006.
- [9] R. K. Ando, T. Zhang, "A Framework for Learning Predictive Structures from Multiple Tasks and Unlabeled Data," *Journal of Machine Learning Research*, v. 6, pp. 1817-1853, 2005.