Formal Methods at Scale

Clark Barrett

It is an exciting time to be working in formal methods. In the last two decades, there has been a dramatic improvement in techniques, algorithms, and tools, with the result that more than ever before, formal methods are being applied to solve real challenge problems in academia and industry. There has also been a noticable attitude-shift among some (though not all) users. Many non-experts are more open to learning about and using formal methods, believing that they can help make their workflows more productive and less error-prone. As a result, we have a unique opportunity to expand the role and impact of formal methods and to thereby greatly improve the security, reliability, and effectiveness of all kinds of digital systems. In order to take advantage of this opportunity, we must understand the current successes of formal techniques and ensure that we don't lose momentum in those areas. We must also recognize the next set of challenges that will open up new opportunities. In this abstract, I highlight successes and challenges that I believe are crucial to focus on going forward.

Core Verification Engines

Undoubtedly, one of the most important success stories has been the development of fast, robust, and versatile verification "engines." Though there are others, I focus here on solvers for Boolean satisfiability (SAT) and satisfiability modulo theories (SMT), which have gone from theoretical curioisities to industrial workhorses over the last two decades. Trying to describe or quantify the tremendous magnitude of progress in these areas and the resulting impact is nearly impossible. Phrases like "many orders of magnitude improvement" are inadequate. Twenty years ago, we could solve toy problems. Today, we can do full verification of CPU cores, check millions of cloud security policies per day, and automatically find thousands of bugs and vulnerabilities in real code.

I believe it would be a big mistake to think that progress in these engines has matured. Indeed, if anything, progress is being held back by the limited number of people working in this area. My hope for the future would be to find ways to make big investments in supporting the continuing development and improvement of core solvers, and especially in finding ways to attract more researchers to this area.

Some of the most exciting progress in solvers has come as we co-evolve a solver with a specific application. A good example of this is the development of the solver for strings in CVC4. When we started work on this solver, we were motivated by being able to support symbolic execution of string operations for security applications. We worked with security experts to understand the kinds of operations they needed to reason about and to make the solver efficient for their needs. Later, Amazon discovered they could use our solver to reason about security access policies in the cloud. We have been working closely with their team, again to improve the expressive power and efficiency of the solver for their applications. The result of these efforts over many years is a powerful, efficient, and general solver for string constraints that is used millions of times per day, both in academia and industry.

Another example of solver development being driven by a new application area is the development of solvers for neural network verification. When we first looked at this problem, we discovered that state-of-the-art algorithms were not well suited for the problem. However, by leveraging the versatility of the SMT platform, we were able to tailor an existing solver to the specific application. The result was an SMT-based approach that could scale orders of magnitude beyond existing techniques. The general area of AI safety is an exciting but challenging area for formal methods. I believe we have only scratched the surface in developing solvers tailored to application domains.

Key opportunities for future efforts in core solvers include: 1) developing custom solvers for new application domains; 2) leveraging parallel computing to solve large, difficult problems; and 3) leveraging machine learning to selectively improve heuristics. It is important to note that (2) and (3) are not at all straightforward - the first obvious things to try don't work very well. However, I firmly believe that with time and effort, good ideas will surface that will allow (2) and (3) to improve core engines significantly.

Using Proofs for Trustworthiness, Communication, and Compliance

It is clear that solvers will continue to improve and find new and exciting applications. However, a nagging question as solvers proliferate is: "how much should we trust these solvers?" Currently, when a solver produces a model (aka witness or counter-example), we can usually check that model against a real system. But when a solver says that a system is safe, how can we trust it? One answer is that the solver can produce an independently-checkable proof. Instrumenting solvers to produce proofs has many beneficial effects. Besides increasing the robustness and trustworthiness of the solver, it also makes it possible for different tools to communicate in a formal and trustworthy way. For example, the SMTCoq tool developed by the CVC4 team makes it possible to prove Coq subgoals using CVC4. CVC4 produces a proof which is then translated into a native Coq proof. The result is that Coq can leverage the speed and automation of CVC4 without giving up anything in terms of its own trusted computing base. Proofs are becoming standard in SAT solvers, but for other solvers, support is still very limited and often ad hoc. Creating a standard, reliable format for solver proofs is a challenging goal, but one that would have many benefits.

An exciting potential application area for proof-producing solvers is using proofs as auditable certificates for communicating about formal guarantees. For example, ensuring that software complies with various regulatory standards or security practices is currently a largely manual effort requiring hundreds of person-hours. A toolflow based on proof-producing engines could automate many of these tasks by producing auditable and independently-checkable proof certificates certifying that the software is compliant.

Usability of Formal Tools

No matter how fast or powerful formal tools become, they will not be more widely adopted without improving their usability. Clearly, usability by non-experts is crucial to focus on. As an example in this direction, we have developed techniques for formally checking microprocessor cores *without* having to write any formal specification. The technique, called symbolic quick error detection (SQED), takes as input only artifacts that are well-understood by a hardware engineer, namely a description of an instruction set architecture and some design parameters. These are then automatically compiled into a formal checker using a "universal property." The technique has been used successfully to find bugs in a number of open-source (and industrial) processor cores. SQED goes beyond simple "assertion generators." It leverages deep self-consistency properties of systems to provide formal guarantees without having to write formal properties. Similar techniques can be used to find hardware trojans or hardware security vulnerabilities. It remains to be seen how far such techniques can be pushed, but continuing to make progress in the direction of deep guarantees without the need for a hand-written formal specification is, I believe, an important direction.

I would also like to propose that we do not ignore the need for better tools and techniques, even for experts. One reason for this is that, as mentioned above, more people are willing to learn about formal techniques. They are willing to write specifications or learn a new tool. However, they expect tools to be usable, robust, and expressive. As an example, I recently had a student ask me how he could prove some simple properties of his C++ code. I had to admit that there were no good tools for general-purpose verification of C++ code. This despite the fact that C++ is the predominant language used for industrial coding. If we build usable, robust, industrial-strength verification tools for the systems and languages being used today, the experts will use them, and the non-experts may be more motivated to become experts.