# Proofs, Provers, Processes at Scale

Natarajan Shankar
SRI International Computer Science Laboratory
Menlo Park CA 94024

June 22, 2020

> *The science of operations, as derived from mathematics more especially, is a science of itself, and has its own abstract truth and value.*
> Ada Lovelace

Formal methods transform the science of computing into a set of human and automated tools for designing, analyzing, and creating computer systems. Such formal tools address a range of problems that span the design lifecycle from capturing requirements, crafting specifications, evaluating designs, and synthesizing and analyzing code. It used to be believed that formal methods could only be applied to toy problems with intense manual effort, and using them to address the challenges of real software would remain a pipe dream. Several things have happened in the last three decades to challenge this widely-held preconception. One, the problems of software have become more acute and every week we see a fresh wave of catastrophic software failures. Traditional software development techniques have not been particularly effective at delivering reliable software. Two, breakthroughs in decision procedures, theorem provers, model checkers, and static/dynamic analyzers have given us a suite of tools that are making it cost-effective to use formal methods. Three, we have some significant examples of formally verified artifacts. Four, many more people have been trained in the effective use of formal methods. However, there is plenty of room for scaling the power and utility of formal methods.

We first enumerate the dimensions along which we can scale formal methods:

1. Problem size: Code size is a poor measure of problem size, but can we scale deep formal analysis to systems with millions of lines of code?

2. Cost: Developing code through formal methods has a high development cost but can it save cost and effort in testing and maintenance?

3. Quality: Does formal methods yield higher levels of software quality assurance?

4. Performance: Can formal methods yield higher quality code through principled optimization.

5. Productivity: Will the technology for verification, synthesis, and transformation enhance programmer productivity?

6. Usability: Will formal methods tools feature highly usable interfaces that make the technology accessible to much wider audience.

7. Maintainability: Can we control the overhead of maintaining specifications and proofs, and even exploit the technology to support the evolution of the software.

8. Degree of automation: To what extent can we automate specification, code generation, and verification to reduce the manual effort needed to effectively employ formal methods.

9. Degree of specialization: Formal methods need to be customized to individual domains have their own formal models, background theories, and notations.

10. Expressiveness: Formalization and automation need to cope with expressive specification and programming languages while exploiting trade-offs between automation and expressiveness.

11. Abstraction level: Formal methods can be applied at varying levels of abstraction from hardware models to higher-order logic.

12. Background libraries: Programming languages make extensive background libraries available to a programmer who can build on these to script their programs. Formal tools also need such libraries if they are to match the effectiveness of modern programming languages.

We have a range of fairly effective technologies across the above scalability dimensions. Dynamic analysis techniques based on test generation and runtime verification are quite effective at detecting bugs. Static analysis methods are also efficient for detecting and even demonstrating the absence of certain classes of runtime errors. These techniques do not have extensive specification overheads. They operate efficiently, and are accessible to a fairly broad audience. Solvers for Boolean satisfiability and satisfiability modulo theories offer a Swiss army knife of capabilities for software analysis. They can be used for test generation, bug-finding, symbolic analysis, model checking, and type checking. These are not as scalable as testing and static analysis, but are capable of handling quite realistic examples. Model checkers are effective for design time analysis and for demonstrating expressive temporal properties of complex designs. Such model checkers can also be applied to software components on the scale of hundreds of lines. Automated synthesis techniques can be used to construct small programs or program components. Beyond that, more manual techniques for program verification can be used to state and demonstrate expressive properties of complex software systems.

What can we achieve now? The seL4 project performed around ten years ago offers some rough order of magnitude estimates for large-scale software verification. The project estimates that it takes about 3 to 4 weeks per 1000 lines of proof.[1] The seL4 project produced 10KLOC with about 200KLOP (kilo-lines of proof) at 18 person-years. Even though lines of proof is not a meaningful metric since it can vary wildly depending on the available automation, we can assume that these numbers are accurate to within a factor of ten. In terms of (equivalent) lines of C code, if one roughly estimates that each line of code requires an average (with a substantial variance) of 5–10 lines of proof (seL4 was 20), verified code productivity can be around 1.5–3 KLOC/person-year at a conservative estimate. The seL4 proofs were performed by highly trained researchers and their students. Also, in the current state of the art, proof maintenance can be considerably harder than code maintenance. In contrast, for critical system code built without the use of formal methods, software productivity levels are reported to be around 4 to 7 KLOC/person-year. We can expect significant improvements in the productivity, usability, and maintainability of formal methods as the technology matures over the next decade.

What is the single-most important pathway to scalability? We have touched on a number of dimensions of scalability. Obviously, we need to see improvements along all of these dimensions: scale, cost, usability, productivity, automation, maintainability, expressiveness, specialization, etc. However, there is one dimensions that correlates most effectively with the scalability of formal methods, namely, abstraction. Engineers use abstraction in the form of models which are then realized in working artifacts. Formal reasoning must also move up the value chain to capture models that allow software designers to abstract away from low-level implementation details. These models can range from general-purpose models of computation that offer powerful programming abstractions to special-purpose domain-specific tool boxes. Such abstract models can be easily supported by code generation to multiple languages and platforms. Proofs and counterexamples are easier to construct and maintain since abstract models support proof patterns and effective automation through decision procedures and model checking. MATLAB Simulink/Stateflow is an example of the success of model-based software engineering.

In order to leverage the power of abstraction, we need to identify systematic ways of mapping highly reusable mathematical models to executable code/hardware in domains such as

- Grammars and parsers
- Compilers
- Cryptography
- Cyber-physical systems

[1] Productivity for Proof Engineering Mark Staples, Ross Jeffery, June Andronick, Toby Murray, Gerwin Klein, Rafal Kolanski, ESEM'14

- Signal processing
- Machine learning
- File systems, and
- Cryptographic/Distributed protocols.

With such tools, we can empower domain experts to employ formal methods to create tens of thousands of (equivalent) verified lines of code a year. These abstract models will be more reusable than low-level software. They can be more easily maintained and retargetted to new platforms and applications. They are also more amenable to proof automation. Advances in automated formal methods tools and techniques coupled with increasingly specialized use of abstraction in a range of domains opens up several possibilities. In particular, we can revisit the design and construction of the entire hardware/software stack in order to eliminate a range of poor design choices that have made our systems insecure, unreliable, and unmaintainable.