Applications of Lightweight Formal Methods in Engineering and Scientific Software Design

Tristan Dyer

May 23, 2019

Abstract

Boop.

1 Introduction

Called a third pillar of science, computation is an indispensible tool for scientists and engineers who simulate physical and natural processes. Recent studies on reliability, reproducibility of results, and productivity have brought forth concern that existing practices of constructing scientific software are inadequate and limiting the pace of technological advancement. A disconnect between existing modern software engineering practice and scientific computation has become apparent and must be addressed. Additionally, the unique challenges facing developers of scientific software, namely the lack of test oracles, software lifetimes and evolving needs that span decades, and the competing objectives of performance, maintainability, and portability, must also be recognized.

I seek to address fundamental design and quality assurance challenges that are intrinsic to scientific computation and engineering software design. While numerous directions might be taken, my premise and motivating viewpoint is the central role that modeling can and must play in the process of designing and working with scientific programs. Culturally, the fit may be a natural one: scientists and engineers are accustomed to working with models anyway, and with the kind of automatic, push-button analysis supported by some state-based formalisms, those who develop software can focus on modeling and design instead of theorem proving.

2 Problem Statement

The goal of this research is to establish the role of lightweight formal methods as an invaluable tool in bridging the gap between modern software engineering practices and the unique challenges presented by scientific and engineering software. This involves the following specific objectives:

1. To clearly define common paradigms of lightweight formal methods within the context of scientific software development. These include but are not limited to: specification, verification and validation, correctness, refinement, and predicate abstraction.

2. To demonstrate the utility of lightweight formal methods through the development of actual models and their respective software. These examples will reflect concepts commonly found in scientific and engineering software, and will span a broad range of abstractions in order to demonstrate the utility and applicability of the approach. Examples are to include the moment distribution method, sparse matrices, the finite element method, web-based analysis tools, and user interface design.

3. To develop tools that aid scientists and engineers in applying these methods in practice. These tools include the a web-based domain specific visualization tool for use in the modeling of finite element software, the development of a model sharing utility that leverages version control and the visualization tool, and the embedding of the visualization tool and the Alloy analyzer into an existing IDE (integrated development environment).

3 Literature Review

Despite broad and recognized impacts, the field of scientific computation faces a number of challenges. Meeting quality and reproducibility standards is a growing concern [], as is productivity []. Not merely anecdotes, numerous empirical studies of software "thwarting attempts at repetition or reproduction of scientific results" have been cataloged in a recent article by Storer [], along with their concomitant effects, including widespread inability to reproduce results and subsequent retractions of papers in scientific journals. Productivity problems are also reported, which Faulk et al. [] refer to as a *productivity crisis* because of "frustratingly long and troubled software development times" and difficulty achieving portability requirements and other goals.

Source of difficulty may stem from fundamental characteristics of the problem domain, along with cultural and development practices within it. For instance, projects are often undertaken, as one might imagine, for the purpose of advancing scientific goals, so results may constitute novel findings that are difficult to validate. In the absence of test oracles, developers may have to settle for plausibility checks based on, say, conservation laws or other principles that are expected to hold. Then, if the software is successful, its lifetime may span a 20 or 30 year period, starting with development and then moving through hardware upgrades and evolving requirements that are intended to keep up with ongoing scientific advancements. Development priorities are such that traditional software engineering concerns, like time to market and producing highly maintainable code, may receive relatively less attention compared with performance and hardware utilization [].

Proposals to address quality and productivity concerns are varied. Storer [] places new and suggested approaches into broad categories of (a) software processes, including agile methods, (b) quality assurance practices, including testing, inspection, and continuous integration, and (c) design approaches, including component architectures and design patterns. In the category of quality assurance practices, he adds formal methods, noting a couple of experience reports, but also observing that such approaches have received considerably less attention in the scientific programming community, possibly due to "the additional challenge of verifying programs that manage floating point data."

4 Methodology

Although the tools and techniques most identified with scientific computation are those of numerical analysis—where error prediction, stability, and convergence are central concerns—such an enterprise offers little guidance in the development process, where early decisions about decomposition and organization establish program structure. The structure and behavior of scientific programs constitute a kind of essential complexity that is not merely a byproduct of inconvenient languages or other accidental complexities, to employ a distinction made by Brooks [**?**] about software engineering. By *structure* we mean static relationships between elements of program state, that is, an assignment of values to variables. By *behavior* we mean dynamic relationships, that is, a sequence of states or steps in a computation, which may include nondeterminism to avoid over-specification. In many cases, this interstitial machinery is itself a complex apparatus, as we find in the case of adaptive, multi-scale, multi-physics applications, for instance, and these are aspects of a program that warrant increased scrutiny and care. I suggest an approach that separates concerns: isolating the structural and behavioral components from the numerics, allowing scientists and engineers to more effectively reason about the programs they create. The approach is well-suited for lightweight tools like Alloy [**?**], a state-based formalism that combines declarative modeling and bounded model checking. The application of state-based methods in scientific computation is relatively uncharted territory, as there is little community experience in working with formal methods.

When we refer to scientific software, we think primarily of problems expressed as mathematical models, where approximate solutions are sought for differential or integral equations that have no closed form solution. As a result, they must be discretized to produce a finite system of equations that can then be solved by algebraic methods. Ocean circulation models, for instance, may be expressed as a system of hyperbolic partial differential equations, and solved by finite element or other numeric schemes. Because they represent aspects of the physical and natural world, the terms and parameters appearing in the equations capture rich state in the form of spatial, geometric, material, topological, and other attributes. The types of discretizations that may be employed in both time and space are varied, and each has its own performance, accuracy, and ease-of-development implications.

In addition to the complexities involved in representing and solving a mathematical model in computer software, we must also recognize that these artifacts do not exist in a vacuum. Innovation in both the hardware and software industries often drive change at a pace that developers of scientific software find difficult to maintain. Examples include new applications of hardware such as GPUs in general purpose computation, new highly performant software platforms such as web-based computation, evolving user interface capabilities, and new and evolving languages that offer varying levels of convenience and performance. Approaching the development of scientific software with the knowledge that the software must exist in a modern ecosystem of emerging and evolving technologies requires a deeper understand of how the machinery of that software might also evolve, without effecting the underlying numerics.

The approach taken in this research is to identify the essential complexities of common paradigms found in scientific software and to determine whether formal methods might help in reasoning about these complexities. The following sections are organized as follows. First, to give context and some background, we discuss the elements of formal methods that will be used in this research. Second, I give brief descriptions of specific applications that will be explored, describing the inherent structural complexities presented, as well as the elements of formal methods I expect to use in modeling each case. These applications were chosen because, to varying extents, they each present challenges to the developer, both from the perspective of accurately representing an inherently mathematical problem, as well as the perspective of existing in an ecosystem that is constantly evolving. Finally, we discuss the applications and tools that will be developed in order to aid scientists and engineers in the application of these methods in practice.

4.1 Lightweight Formal Methods

State-based or model-oriented approaches describe a system by defining what constitutes a state and the transitions between states, or operations. The state-based formalism employed in this research will be Alloy [**?**], a declarative modeling language that combines first-order logic with relational calculus and associated operators, as well as transitive closure. It offers rich data modeling features based on class-like structures and an automatic form of analysis that is performed within a bounded scope using a SAT solver. For *simulation*, the analyzer can be directed to look for instances satisfying a property of interest. For *checking*, it looks for an instance violating an assertion: a counterexample. The approach is *scope complete* in the sense that all cases are checked within user-specified bounds. Alloy's logic supports three distinct styles of expression, that of predicate calculus, navigation expressions, and relational calculus. The language used for modeling is also used for specifying the properties to be checked.

From Alloy's declarative underpinnings comes expressive power and an effective means of reducing complexity and probing designs. As Jackson writes [**?**], code is a poor medium for exploring abstractions, and tools like Alloy offer modeling environments that support an iterative design scenario akin to what might be performed by, say, a civil engineer designing a bridge. The approach is sometimes referred to as lightweight [] because there is *partiality in modeling*—a focused application of the method—and *partiality in analysis*, since the verification being performed is bounded. With respect to the latter, and on the value of the approach, we appeal to the *small scope hypothesis*: if an assertion is invalid, it probably has a small counterexample []. out approach may be considered lightweight in an additional sense: we are able to draw useful conclusions about scientific software without simultaneously reproducing semantic proofs of numerical analysis.

4.2 Applications of Lightweight Formal Methods

4.2.1 Moment Distribution

Well-known among civil engineers, the moment distribution method [] is an iterative technique for finding the internal member forces that develop in building structures when external forces are applied to them. The calculations can be performed by hand, and the rapid convergence of the method in practice makes it possible for engineers to estimate internal forces in just a few iterations. Although the method has been largely superseded by the convenience and availability of more general computational approaches, it was the primary tool used by engineers to analyze reinforced concrete structures well into the 1960s.

Conceived in the 1920s, before the advent of computers, the method nonetheless displays features that are interesting from a computational perspective. Intuitively, the method works by "clamping" joints, applying external loads, and then successively releasing them, allowing them to rotate, and reclamping them. Each time, the internal forces at the joints are distributed based on the relative stiffnesses of the adjoining members. The method converges under a variety of distribution sequences, e.g. varying the order in which joints are unclamped. In addition, there is inherent concurrency in the method since internal forces can be distributed simultaneously and summed.

4.2.2 Sparse Matrices

4.2.3 Finite Element Meshes

4.2.4 Data Visualization and UI Design

4.3 Applications and Tools

5 Overview of Chapters

- Moment distribution
- Sparse matrices
- Finite element
- User interface design
- Mesh editing and rendering
- Utilities for Alloy

6 Plan of Work

Get it all done this year.

7 Bibliography

Citations.

8 Appendices

Papers we've already written.