Applications of Lightweight Formal Methods in Engineering and Scientific Software Design

Tristan Dyer

May 31, 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 assur-

ance 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 ex-

pressed 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 formalisms 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, and Electrum [], a temporal extension to Alloy that introduces the concepts of linear temporal logic into the Alloy language. Alloy 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 model-*

ing—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 []. Our 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

The following are specific applications of lightweight formal methods that will be explored in this research.

4.2.1 Moment Distribution

Moment distribution [] is an iterative technique, well-known among civil engineers, for finding the internal member forces that develop in building structures when external forces are applied to them. In its most general form, the method is similar to asynchronous, chaotic relaxation algorithms, where portions of a building structure converge numerically at differing rates as the computation unfolds, depending on the process scheduling. The nondeterminism available here is also inherent in methods used to solve elliptic partial differential equations, which may exploit nondeterminism in different ways depending on problem characteristics and hardware features.

Although it could be considered a 'toy' problem, the moment distribution method displays structural and behavioral features that are interesting from a computational perspective and commonly found in broad range of scientific software. The structural components of the problem and the topological relationships between those components are concepts that are found in, for example, methods that require a spatial discretization such as finite volume and finite elements. The implicitness supported by Alloy allows us to work with arbitrary spatial discretizations, and starting with a simple problem such at this will al-

low us to build upon the conclusions drawn as more sophisticated models are developed through further abstraction and refinement.

WORK ON THIS: Similarly, the behavioral components of moment distribution can be found in a large class of problems that exhibit similar levels of inherent parallelism. Alloy does not impose fixed idioms and can

* common structural components: data at discrete spatial points (topology)
* common communication patterns: transfer of data between discrete spatial points (implicitly parallel)

4.2.2 Sparse Matrices

The types of problems found in scientific and engineering software frequently require a discretization over some continuous domain, resulting in a system of equations that can be solved using linear algebraic methods. As a result, it is not uncommon to find powerful linear algebra libraries at the core of some piece of scientific software. These libraries, some of which have existed for decades, provide us with a unique perspective, one that lends itself to software modeling: while the underlying mathematics have remained constant, the structural and behavioral components of the libraries have had to adapt as evolving technologies have created potentially new performance benefits. For example, the advent of massively parallel systems, in concert with the introduction of general purpose computing on GPUs, has resulted in systems that are capable of incredible data throughput. These systems can employ numerous types of processors, each built on different memory models and each capable of leveraging multiple types of parallelism. As a result, many linear algebra libraries have adapted in order to properly leverage the performance benefits made available by these types of systems.

Often, the methods of discretization that are employed in scientific software, such as in the finite element method, will result in a system of equations that can be represented using sparse matrices—matrices in which the majority of the values they contain are zero. As such, most linear algebra libraries provide very fast

and efficient computational methods for representing and using sparse matrices. From the structural perspective, there are many different ways of representing a sparse matrix in memory, each with strengths in some applications and weaknesses in others. For example, the DOK (dictionary of keys) format excels at matrix assembly, but is slower compared to other formats when being used in some algebraic operation. Conversely, the CSR (compressed sparse row) format excels when used in certain matrix operations, such as matrix-vector multiplication, but is slower to assemble than others. From a behavioral perspective, the algorithmic implementation used to perform an algebraic operation is highly dependent on the format, i.e. the memory layout, used to represent the sparse matrix.

Nonetheless, all sparse matrix formats and the corresponding algorithms used in algebraic expressions represent exactly the same thing at an abstract, mathematical level. This presents us with a valuable opportunity to model these formats and operations, the benefits of which are threefold:

1. To provide a clear and useful example of how both data and procedural refinement can be used in scientific and engineering software development.

2. To verify that all formats and operations are accurate representations of the underlying mathematics.

3. To provide model that can be built upon as new technologies and techniques are developed, in order to ease development and instill confidence in implementations.

In this research, I will create an abstract model of a sparse matrix as well as the sparse matrix-vector multiplication operation. I will then create models of the DOK and CSR sparse matrix formats and use abstractions functions along with implication to verify that the data refinement is sound. Additionally, for both formats I will model the sparse matrix-vector multiplication operation and similarly use abstraction functions and implications to verify that the procedural refinements are sound. Finally, I will create a concrete implementation of both

formats and their corresponding operations in order to show the complete story as a guide to engineers and scientists—beginning with a mathematical problem, how might one go through the process of incrementally stepping through the modeling process, moving closer to an implementation at each step, and finally writing a concrete implementation.

4.2.3 Finite Element Meshes

The finite element method is extremely common in modern scientific software and can be used to solve a wide range of mathematical problems.

Things to model in meshes: * Element shape * Topology requirements: connectivity limitations, dimensions, continuity, etc. * Element order: linear, quadratic, etc. * Features: adaptivity * Abstractions levels: node/element, arrays, etc.

4.2.4 Data Visualization and UI Design

* Finite element visualization and exploration in a modern ecosystem, i.e. web browser. * Getting data organized * Communication between various components: GPU, worker threads, main thread * Communicating effectively with the user about what data is available, what the tool is actively doing, how to interact with data, etc. * Lightweight formal methods can even be used to ensure that simple user interfaces remain correct – ellipse tool case in point.

4.3 Applications and Tools

Alloy is a general purpose tool - not always the best visualizations. Created a few tools to make Alloy a little more suitable to the scientific computation and the types of problems I've modeled (meshes mostly)

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.