1 Introduction

Scientists increasingly rely on computational models to explore and understand the world around us, with diverse applications throughout the physical, chemical, and biological sciences. In 2005, The President's Information Technology Advisory Committee (PITAC) issued a report referring to computation as a "third pillar" of science, placing it alongside theory and experimentation. The report underscores qualitative transformations and discoveries throughout the natural sciences, as well as in engineering, manufacturing, and economic processes, which often rely on scientific models to predict the effects of decisions. Coastal engineers, as just one example, evaluate alternative flood protection measures by subjecting them to large-scale, simulated storm events. The performance and fidelity of scientific models are therefore key determinants affecting the quality of those designs.

Despite its importance, the field of scientific computation faces a number of problems. Meeting quality and reproducibility standards is a growing concern [Wilson, 2006], as is productivity [Faulk et al., 2009]. 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 [2017]. In one [Hatton and Roberts, 1994] and in a more recent follow-up [Hatton, 2007], over a dozen independently developed commercial codes for seismic data processing were compared, showing a rate of numerical disagreement between them of about 1% per 4,000 lines of implemented code, and that "even worse, the nature of the disagreement is nonrandom." In addition to cataloging the quality concerns reported in those studies, Storer [2017] further cites their effects, including a widespread inability to reproduce results and subsequent retractions of papers in scientific journals. Productivity problems are also reported, which Faulk et al. [2009] refer to as a *productivity crisis* because of "frustratingly long and troubled software development times" and difficulty achieving portability requirements and other goals.

Sources of difficulty may be the result of fundamental characteristics of the problem domain, as well as cultural and development practices within it. To examine these and related issues, researchers from the scientific computation and software engineering communities came together for a workshop whose outcomes are summarized by Carver [2009]. In an observation about development practices, Carver notes that programmers in scientific areas are often domain experts with a Ph.D. in their respective fields, and that a single scientist may take the lead on a project and then rely on self-education to pick up whatever software development skills are needed. Such practices are long-standing, and represent a disconnect from the software engineering community that has been variously referred to as a "chasm" [Kelly, 2007; Sanders and Kelly, 2008] and a "communication gap" [Faulk et al., 2009].

Other observations, however, point to the unique challenges facing the developers of scientific software. For instance, projects are often undertaken, as one might imagine, for the purpose of advancing scientific goals, so results may represent 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 [Storer, 2017]. 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 to performance and hardware utilization [Faulk et al., 2009].

The overall goal of the proposed research is to address fundamental design and quality assurance challenges that are intrinsic to scientific computation and related types of numerical software. Addressing them may also lead to accompanying gains in productivity, particularly if better designs improve maintainability and extensibility, given the lifespans typical of scientific software products.

In the following sections I propose a state-based approach for reasoning about scientific software, using abstraction and refinement principles to separate numerical concerns from other sources of complexity, such as those introduced to meet performance goals. Elements of the approach include declarative modeling and automatic, push-button analysis using bounded model checking, as embodied in lightweight tools like the Alloy Analyzer [Jackson, 2012] which, while increasingly promoted in areas like communication protocols and control systems, may also be exploited in scientific domains.

The remainder of the proposal is outlined as follows: ...

2 Literature Review

3 Methodology

The objective of this research is to develop modeling approaches using formal methods that allow scientists to represent and reason about the essential structure and behavior of the programs they create. By *structure* I mean static relationships between elements of program state, that is, an assignment of values to variables. By *behavior* I mean dynamic relationships, that is, a sequence of states or steps in a computation, which may include interleaving. Structure and behavior may each find expression in different ways in different programming languages, and so demonstrating the relationship to code is important and also within the scope of the research.

Below I provide background on lightweight formal methods and some of the characteristics of scientific programs that inform the approach, which is based on a separation of concerns that accounts for numerical computations. I also describe what is meant by software abstractions, and provide a simple example that illustrates the rationale for viewing scientific programs as I do. Here I need to talk about everything I'm gonna make.

3.1 Lightweight Formal Methods

The fundamental idea is to characterize structure, behavior, and properties of interest in a way that allows one to frame questions and get answers to them using tools like Alloy Analyzer [Jackson, 2012] and TLA+/TLC [Lamport, 2002], which support automatic, push-button analysis. What I propose, then, is a *lightweight* approach to formal methods [Jackson and Wing, 1996], in which there is both *partiality in modeling*, since we do not commit to modeling entire systems, and *partiality in analysis*, since the analyses being performed are bounded. An additional aspect of lightweight formal methods is an incremental style of modeling, which tools like Alloy support by offering immediate feedback while models are being constructed: we start with a minimal set of constraints and "grow" them via conjunction.

From their declarative underpinnings comes expressive power, along with an effective means of reducing complexity and probing designs. As Jackson [2012] writes, code is a poor medium for exploring abstractions, and lightweight tools offer modeling environments that support an iterative design scenario akin to what might be performed by, say, a civil engineer designing a bridge. As with an algebraic approach to specification, as has been found [Baugh, 1992], not all aspects of a software system warrant the same level of attention from a modeling perspective.

With respect to a bounded or partial analysis, the merit of performing one springs from an appeal to the *small scope hypothesis:* if an assertion is invalid, it probably has a small counterexample [Jackson and Damon, 1996], as has been observed in a prior case study [Baugh and Altuntas, 2017]. In addition, in some cases, *domain-specific* arguments can be used to bolster the conclusions of such analyses, e.g., by calling on the so-called Courant-Friedrichs-Lewy (CFL) condition when solving hyperbolic partial differential equations [Baugh and Altuntas, 2017; Altuntas and Baugh, 2017].

Alloy Analyzer. For background on the primary tool I expect to use, Alloy is a declarative modeling language combining first-order logic with relational calculus and associated quantifiers and operators, along with 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 properties of interest and assertions. Alloy supports expressions with integer values and basic arithmetic operations.

3.2 Scientific Programs

The proposed work is being undertaken in a relatively uncharted domain, scientific computation, for which there is little community experience in applying formal methods. We might ask about essential complexities, what they are, and where formal methods might help. In contrast, when computer engineers model systems, for instance, they already have some experience in getting at these questions. So when specifying a two-phase handshake protocol they know whether they can ignore what's going through the pipe: they generally have some sense of how and what to specify, and what to ignore. There is far less of this kind of experience with programs in scientific areas, so it is helpful to pause and characterize what they are like.

The subject matter of scientific programs includes the physical and natural processes that surround us, where space and time are traditionally viewed as being continuous. Circulation of currents within the atmosphere and oceans, for instance, involves state that is continuously varying and where, indeed, continuity arguments are used to "fill in" gaps that may be associated with sampled data. The computational apparatus underlying ocean circulation models, however, looks less like purely analytic functions and more like an amalgam of discrete and continuous constructions that allow, as one example, the representation of irregular land and seafloor geometries as piecewise polynomial surfaces. 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.

That problems such as these are discretized implies there is a level beneath which certain physical features and processes are not resolved—a modeling concern to be managed. But more than that, discretizing time and space means that contingent processes, like the continuous movement of the shoreline due to tides and storms, require special handling to account for them in physically meaningful and often nontrivial ways. As a result, it may be helpful to separate concerns and avoid mixing the types of reasoning best suited to the respective types of processes, whether discrete or continuous.

3.3 Separating Concerns

Related studies on formal methods, though few, have targeted somewhat different aspects of scientific programs, allowing us to make observations and draw some comparisons. Most have focused on numerics—of which the works of Bientinesi et al. [2005] and Siegel et al. [2008] are representative—by combining model checking and the reals with applications to some of the direct methods of linear algebra.[1] However, the scope of systems that can be so analyzed will remain rather limited until the deep, semantic proofs of numerical analysis [Linz, 1988] can be accommodated as part of the reasoning process.

We take the position that a separation of concerns is in order, and propose something akin to the two-phase handshake protocol analogy, where the data going through the pipe are, in this case, numerical expressions. We cannot ignore them, of course, but we aim to consider them separately using the conventional tools of numerical analysis.

Accordingly, we advance the following perspective:

$$\boxed{\begin{array}{c}\text{scientific}\\ \text{programs}\end{array}} = \boxed{\begin{array}{c}\text{numerical}\\ \text{expressions}\end{array}} + \boxed{\begin{array}{c}\text{interstitial}\\ \text{machinery}\end{array}}$$

By *interstitial machinery* we mean the data structures and algorithms throughout which numerical expressions are embedded. In many cases, the interstitial machinery is itself a complex apparatus, as we find in the case of adaptive, multiscale, multiphysics applications, for instance, and these are aspects of a program that warrant increased scrutiny and care. Correctness arguments for this part of scientific programs can be made without simultaneously reasoning about, say, elements of the Gershgorin circle theorem for eigenvalues. Instead, results of numerical analyses may be brought into the modeling process in the form of invariants and other structural properties.

With respect to numerical analysis, the form of the results needed from the effort may vary, but this type of work may be conducted independently and used to inform the process undertaken when reasoning about the interstitial machinery so that the overall conclusions are sound, as we illustrate in Section **??**. Beyond that example and an appeal to experience, a supporting idea for the claim is the following:

> The numerical analyses performed for scientific computations often apply, unchanged, throughout a broad range of implementation choices and modifications, changes in libraries and solvers, and diverse hardware upgrades, over the life of the program.

References

Altuntas, A. and Baugh, J. (2017). Adaptive subdomain modeling: A multi-analysis technique for ocean circulation models. *Ocean Modelling*, 115:86–104. https://doi.org/10.1016/j.ocemod.2017.05.009.

Baugh, J. and Altuntas, A. (2017). Formal methods and finite element analysis of hurricane storm surge: A case study in software verification. *Science of Computer Programming.* https://doi.org/10.1016/j.scico.2017.08.012.

Baugh, J. W. (1992). Using formal methods to specify the functional properties of engineering software. *Computers & Structures*, 45(3):557–570. https://doi.org/10.1016/0045-7949(92)90440-B.

[1]By direct methods, we mean those that produce an exact solution, modulo rounding errors, in a finite number of steps. Iterative methods, on the other hand, successively approximate a solution, gradually improving it until convergence is reached.

Bientinesi, P., Gunnels, J. A., Myers, M. E., Quintana-Ortí, E. S., and Geijn, R. A. (2005). The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software (TOMS)*, 31(1):1–26.

Carver, J. C. (2009). Report: The second international workshop on software engineering for CSE. *Computing in Science & Engineering*, 11(6):14–19.

Faulk, S., Loh, E., Van De Vanter, M. L., Squires, S., and Votta, L. G. (2009). Scientific computing's productivity gridlock: How software engineering can help. *Computing in Science & Engineering*, 11(6):30–39.

Hatton, L. (2007). The chimera of software quality. *Computer*, 40(8).

Hatton, L. and Roberts, A. (1994). How accurate is scientific software? *IEEE Transactions on Software Engineering*, 20(10):785–797.

Jackson, D. (2012). *Software Abstractions: Logic, Language, and Analysis*. The MIT Press.

Jackson, D. and Damon, C. A. (1996). Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering*, 22(7):484–495.

Jackson, D. and Wing, J. (1996). Lightweight formal methods. *Computer*, 29(4):21.

Kelly, D. F. (2007). A software chasm: Software engineering and scientific computing. *IEEE Software*, 24(6).

Lamport, L. (2002). *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc.

Linz, P. (1988). A critique of numerical analysis. *Bulletin of the American Mathematical Society*, 19(2):407–416.

Sanders, R. and Kelly, D. (2008). Dealing with risk in scientific software development. *IEEE Software*, 25(4).

Siegel, S. F., Mironova, A., Avrunin, G. S., and Clarke, L. A. (2008). Combining symbolic execution with model checking to verify parallel numerical programs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):10.

Storer, T. (2017). Bridging the chasm: A survey of software engineering practice in scientific programming. *ACM Computing Surveys (CSUR)*, 50(4):47.

Wilson, G. V. (2006). Where's the real bottleneck in scientific computing? *American Scientist*, 94(1):5.