CHAPTER

2

MODELS OF SCIENTIFIC SOFTWARE: A STATE-BASED APPROACH

2.1 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 design decisions. Coastal engineers, as just one example, evaluate alternative flood protection measures by subjecting them to large-scale, simulated hurricane events. The performance and fidelity of scientific models are therefore key determinants affecting the quality of those designs.

2.1.1 Challenges

Despite broad and recognized impacts, the field of scientific computation faces a number of challenges. Meeting quality and reproducibility standards is a growing concern [82], as is productivity [32]. 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 [74]. In one [40] and in a more recent follow-up [39], 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 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. [32] 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 stem from fundamental characteristics of the problem domain, along with cultural and development practices within it. To examine these and related issues, researchers from the scientific computation and software engineering communities came together in 2009 for a workshop whose outcomes are summarized by Carver [25]. In an observation about development practices, he 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" [49, 70] and a "communication gap" [32].

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 [74]. 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 [32].

2.1.2 Recommendations

Proposals to address quality and productivity concerns are varied. Storer [74] places new and suggested approaches into broad categories of a) software processes, including agile methods, b) quality assurance practices, including testing, inspections, and continuous integration, and c) design approaches, including component architectures and design patterns. Other recommendations have been made by Faulk et al. [32], who argue for three fundamental software engineering strategies that are successful in other domains: automation, abstraction, and measurement. On the value of providing scientifically relevant computational abstractions, for instance, they state:

> "Important higher-level abstractions will allow scientific programmers to express desired computations in ways that reflect the science and mathematics of the problem domain rather than the computing system."

In the category of quality assurance practices, Storer 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."

2.1.3 Scope and organization

In what follows, we describe a state-based approach for reasoning about scientific software, and show how abstraction and refinement principles can be used 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 Alloy Analyzer [46] which, while increasingly promoted in areas like communication protocols and control systems, may also be exploited in scientific domains. Our focus is on *design thinking* for scientific software systems, an inherently iterative process that, with tool support, is intended to help developers gain a deeper understanding of the structure and behavior of the programs they write.

As a follow-up to a verification study of a large-scale ocean circulation model used in production [11] and a short paper presented at ABZ 2018 [12], the discussion below expands on those ideas and shows how and why state-based formal methods fit into the broader context of scientific computation. The example we present, though small, is chosen to be representative in important ways of scientific software and to highlight limitations of other approaches that have been proposed. By working with models of software, we avoid overcommitting to implementation languages and other details, allowing us to develop and communicate transferable design and quality assurance concepts.

The chapter is organized as follows. Section 2.2 gives some perspective on scientific software, its characteristics, and an approach for utilizing state-based formal methods. Section 2.3 presents a simple, illustrative example that includes elements of the approach and portions of a specification in Alloy that can be found online [1]. Section 2.4 discusses instruction in declarative formalisms like Alloy and a template-based approach for building models. Section 2.5 includes representative selections of related work by ourselves and others. Section 2.6 concludes the chapter.

2.2 Perspective and Approach

The structure and behavior of scientific programs constitute a kind of essential complexity, we believe, that is not merely a byproduct of inconvenient languages or other accidental complexities, to employ a distinction made by Brooks [21] about software engineering concerns. 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 overspecification. 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 our concerns (primarily in Fortran and C++, but also in modern languages like Go).

Below we characterize the nature of scientific programs and give some perspective on a possible role that state-based methods can play in reasoning about them.

2.2.1 About Scientific Programs

We consider the application of state-based methods in a relatively uncharted domain, scientific computation, for which there is little community experience in working with formal methods. We might ask about the essential complexities, what they are, and where formal methods might help. By way of contrast, when computer engineers model systems, they already have some experience in getting at these questions. So, for instance, 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 domains, so it is helpful to take a step back 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. 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.

2.2.2 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. [15] and Siegel et al. [72] 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 [59] can be accommodated as part of the reasoning process.

[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.

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 2.3. 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.

2.2.3 Abstraction and Refinement

Apart from questions about what to model are those dealing with a specification's level of abstraction. At the highest level are requirements the overall program must satisfy, or approximate, and these may be articulated in prose or in a more formal notation. In scientific domains, the problem statement may already be posed in terms of mathematics. For example, a simple boundary-value problem may be stated as follows:

> Find a function $u = u(x), 0 \leq x \leq N$ that satisfies the following differential equation and boundary conditions:

$$\begin{aligned} -u'' + u &= x, \qquad 0 < x < N \\ u(0) &= 0, \qquad u(N) = 0 \end{aligned} \tag{2.1}$$

For larger scale problems, like simulating ocean currents and waves, one would likely start with a system of hyperbolic partial differential equations in three dimensions that includes terms arising

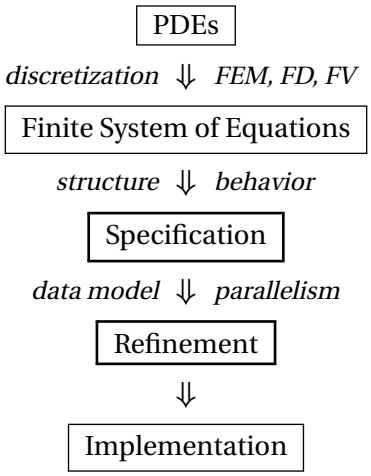

Figure 2.1 A refinement perspective for scientific software, where parts of the highlighted *Specification* and *Refinement* steps are formalized using state-based methods.

from conservation principles, Coriolis and hydrostatic forces, atmospheric pressure gradients, wind and seafloor stresses, tidal potentials, and various types of boundary conditions.

Although we have an obligation to show that programs satisfy these requirements, such high-level descriptions offer little help in structuring programs or reasoning about them, which motivates us to specify what they do at lower levels of abstraction. For example, a developer may choose to recast the problem in variational form and approximate a solution by discretizing a spatial domain using finite element methods. The tools of numerical analysis would then be used to derive or validate such an approach, and in the process yield obligations in the form of invariants that must be satisfied by individual parts of a scientific program. Meeting those obligations can be achieved in any number of ways, often while attempting to exploit hardware capabilities as effectively as possible.

Figure 2.1 outlines a refinement approach for scientific software that might be used to solve partial differential equations (PDEs), which are discretized to produce a finite system of equations that can then be solved by algebraic methods. It is at this intermediate level—between the approximation method and its implementation—that we see a role for state-based methods, i.e., in the specification of the finite system and one or more refinements that successively reduce nondeterminism. Doing so allows us to check that a) the specification is consistent, i.e., it has a model (in the logical sense), and b) the refinement satisfies the specification under a suitable refinement mapping. Because of the central role of refinement, we employ nondeterminism as a means of expressing concurrency [54] in specifications and refinements that are written in an *interleaving* style [55]. The analyses we describe are expressed as *safety* properties for avoiding errors, and additional checks can be performed in a variant of Alloy called Electrum [23] based on Linear Temporal Logic (LTL).

With respect to data, the terms and parameters appearing in a problem statement like Eq. 2.1 represent aspects of the physical and natural world that get pushed down into lower levels of

abstraction. These may be large and varied datasets, and it should come as no surprise that they capture rich state in the form of spatial, geometric, material, topological, and other attributes, which must be represented. Thus, while other modeling approaches might be considered, state-based methods seem particularly appropriate. Here, we focus on Alloy because of its support for conceptual design: it was influenced by modeling languages like OMT and UML and is well-suited for building object models. Perhaps most importantly, Alloy supports *implicitness* in a specification, so problems with rich state and varying spatial discretizations, for instance, can be accommodated in a straightforward manner.

2.2.4 State-based methods and Alloy

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 we employ is Alloy [46], 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 [46], 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 [48] 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 [47]. Our approach may be considered lightweight in an additional sense: we are able to draw useful conclusions about scientific software without simultaneously reproducing the semantic proofs of numerical analysis.

2.3 Illustrative Example

The perspective above draws on experience with various kinds of scientific software used in both research and production, and yet small problems can also be useful objects of study for insights they offer and for communicating principles. The example chosen here, for instance, shares some of the same problem structure found in the simulation of other physical and natural systems, including the data movement and concurrency patterns seen in more complex parallel element-by-

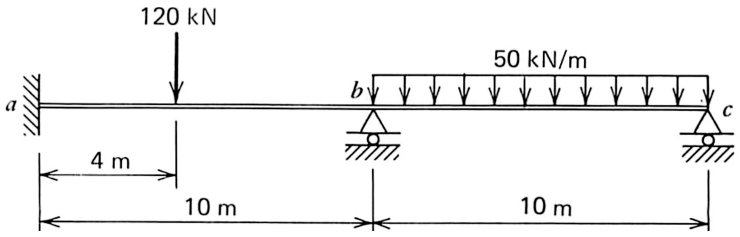

Figure 2.2 Continuous beam with applied loads, supports at labelled joints [80].

element solvers [26]. Just as important, in terms of insight, it demonstrates the relationship between numerical analysis and the role that modeling with state-based methods can play, in conjunction with it, to reason about scientific programs.

2.3.1 Moment distribution

Well-known among civil engineers, the moment distribution method [29, 80] is an iterative technique for finding the internal member forces that develop in building structures, such as tall office buildings, when external forces are applied to them.[2] The calculations can be performed by hand, and the rapid convergence of the method in practice made it possible for engineers to estimate internal forces in just a few iterations. Although the method has largely been 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 nevertheless displays features that are interesting from a computational point of view, and is applicable to small beams, as shown Fig. 2.2, and to more complex structures, including multistory, multibay, three-dimensional frames. 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.

In its most general form [13], the moment distribution method is similar to the asynchronous, chaotic relaxation algorithms of Baudet [9] and Bertsekas [14], with processes clamping and unclamping joints concurrently. In this scenario, portions of a building structure may converge numerically at different rates as the computation unfolds, depending on process scheduling. The nondeterminism available here is also inherent in the methods commonly used to solve other types of boundary-value problems, including those associated with elliptic partial differential equations, which may exploit nondeterminism in different ways depending on problem characteristics and hardware features. From this we observe that a straightforward combination of model checking and symbolic execution over the reals [72], as a verification approach, is ill-suited to a large class of problems, since such

[2] A *moment* is a rotational force that produces bending in a beam or column in the plane of loading.

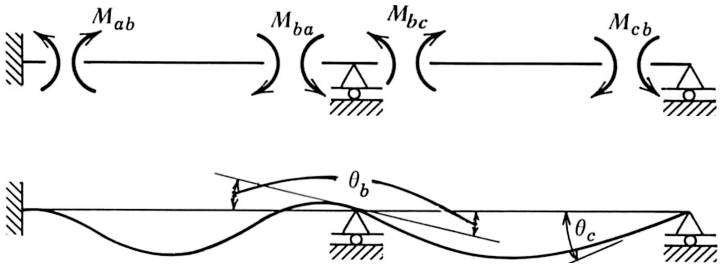

Figure 2.3 Unknown moments (above) and deflected shape (below) [80].

programs appear inequivalent when they in fact converge to the same solution through an entirely different sequence of operations.

2.3.2 Basic procedure

The 120-kN and 50-kN/m loads applied to the beam in Fig. 2.2 induce bending moments whose magnitudes are sought near joints a, b, and c. These internal forces, labelled M_{ij} in Fig. 2.3, are the basic unknowns, occurring at member ends ab, ba, bc, and cb, and producing joint rotations θ_b and θ_c in the deflected shape shown at the bottom of the figure. A boundary condition at joint a prevents rotation at the left end of the beam: such joints are said to be *fixed* against rotation. For those that are not fixed, equilibrium considerations dictate that moments at the joint should *balance*, so that at joint b, for instance, the algebraic sum of its end moments M_{ba} and M_{bc} must equal zero in the final solution.

The method starts with loads removed and all joints clamped to prevent rotation, after which loads are applied in the form of external moments at the member ends. A basic step is that of unclamping, or *releasing* a joint, which allows the joint to rotate slightly and the moments to balance, a redistribution of forces that also produces "carry over" effects at the far ends of adjoining members. Their magnitudes are determined by constants associated with each member end ij, that is, by distribution and carry-over factors D_{ij} and C_{ij} that are based on structural properties particular to the problem at hand. In general, however, we require for any joint that the distribution factors associated with its member ends sum to one, and note that a typical carry-over factor is one-half.

On release, an unbalanced joint becomes balanced, and the amount of the moment to be redistributed is the sum of its end moments. Releasing joint b, for instance, produces the following updates:

$$
\begin{aligned}
M'_{ba} &= M_{ba} - D_{ba}x && \text{(distribution at joint } b\text{)} \\
M'_{bc} &= M_{bc} - D_{bc}x \\
M'_{ab} &= M_{ab} - C_{ba}D_{ba}x && \text{(carry over to joints } a, c\text{)} \\
M'_{cb} &= M_{cb} - C_{bc}D_{bc}x
\end{aligned}
$$

where x is the amount of the imbalance at the joint before it is released, i.e., $x = M_{ba} + M_{bc}$. Starting

with the following moments, in units of kN·m,

$$M_{ab}, M_{ba}, M_{bc}, M_{cb} = \langle -172.8, 115.2, -416.7, 416.7 \rangle$$

the unbalanced moment x at joint b is −301.5 kN·m. For distribution and carry-over factors of 0.5, releasing joint b then yields

$$M'_{ab}, M'_{ba}, M'_{bc}, M'_{cb} = \langle -97.425, 265.95, -265.95, 492.075 \rangle$$

for the member end moments, and we note that $M'_{ba} + M'_{bc} = 0$, so the moments at joint b are now balanced. In general, joints b and c, not being fixed, are eligible for release (when they are not balanced), and this operation may be performed successively until convergence is obtained. For distribution and carry-over factors at joint c of 1 and 0.5, respectively, such a process yields the following solution

$$M^*_{ab}, M^*_{ba}, M^*_{bc}, M^*_{cb} = \langle -27.129, 406.54, -406.54, 0 \rangle$$

to 5 significant digits and in units of kN·m. We note that joints b and c are balanced: each is in equilibrium, as required.

2.3.3 Numerical analysis

A consecutive joint balancing approach, as described above, is similar to an incremental form of the Gauss-Seidel method [36], which uses the most recently updated estimates at each iteration to solve a linear system of equations. Other schemes are possible, however, and the procedure is generalized by Baugh and Liu [13], who present nondeterministic sequential and concurrent algorithms that exhibit the full range of behavior allowed by the method. To perform the analysis, iteration matrices are constructed where an i, j entry corresponds to the effect of joint i on the moments at joint j. Elementary operations are then defined in those terms, and matrix structure is used to show convergence to the exact solution under a mild fairness constraint that a joint is released infinitely often.

2.3.4 Specification in Alloy

Results of the study above allow basic steps to be defined without overspecifying their order. To do so, it is convenient to begin by representing a building structure as a symmetric, directed graph, as shown in Fig. 2.4, with vertices for joints and pairs of anti-parallel edges for the beams or columns that connect them. Making use of predicate abstraction [37], the representation indicates whether or not a joint (a vertex) is balanced, and whether or not a moment is to be carried over from one joint to another (on an edge). In the figure, for instance, the prior beam is shown midway through

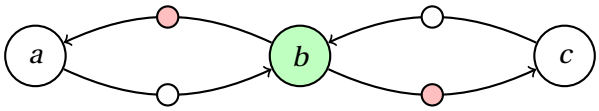

Figure 2.4 Structure represented as a symmetric, directed graph.

its initial release-b operation: distribution has been performed so joint b is balanced (shown as a green vertex), and pending moments have yet to be carried over from b to a and from b to c (shown as red edge tokens).

A specification in Alloy is shown in Fig. 2.5, where a *Joint* signature introduces both a type and a set of uninterpreted atoms. It contains a field *neighbors* that defines a binary relation on joints; a *topology* fact ensures that it has no self loops and is symmetric and strongly connected. A *Fixed* signature defines a subset of joints that are prevented from rotating. Together, the signatures and fact above constitute an implicit, static description of the structure of the problem.

For modeling dynamic behavior, a *State* signature defines fields *balanced* and *pending* that record at each step whether or not a joint is balanced, and whether or not a moment is to be carried over on an edge, consistent with the interpretation of the graph shown in Fig. 2.4. We employ an idiom common to Alloy in predicate *show* that defines a total ordering on *State* atoms to create a global execution trace in terms of the *init* and *step* predicates, whose behavior is consistent with the numerical study of Baugh and Liu [13].

A step is either the release of a joint that is not fixed, a carry-over operation, or a stutter step, which leaves the state unchanged. The *release* predicate is enabled on a joint when it is unbalanced and has no pending carry-over operations to perform, and produces a balanced joint since we view release and distribution as occurring in a single step; operators + and <: are set union and domain restriction, respectively. The *carryover* predicate carries a moment from joint u over to v, making v unbalanced; operator $-$ is set difference and $u \rightarrow v$ in this context is a pair. Additional details of the Alloy language can be found in the text by Jackson [46], and the complete specification and refinement below are available online [1].

2.3.5 Refinement

Moving closer to an implementation, we consider a procedural refinement in which each joint in a structure is a separate process, and where synchronous message passing is used to carry over moments. The idea is to experiment with and check the correctness of different forms of concurrency, communication, and synchronization apart from the numerics, thereby increasing confidence that a corresponding implementation in a conventional programming language is based on sound concepts.

Fig. 2.6 shows a state machine, executed by each process, that begins in *test*, and where a transition to *send* is enabled when a joint is unbalanced, causing it to balance, and where a transition to *recv* is enabled when a neighbor is attempting to carry over a moment, causing the receiving joint to become unbalanced. After either sending or receiving a message, a joint returns to its test state.

```
open util/graph [Joint]
open util/ordering [State] as so

sig Joint { neighbors: some Joint }
sig Fixed in Joint {}            -- some joints may be fixed
sig State {
   balanced: set Joint,          -- joints that are balanced
   pending: Joint → Joint        -- a pending carry over
}

fact topology {
   noSelfLoops[neighbors] and undirected[neighbors]
      and stronglyConnected[neighbors]
}

fact eligible { all s: State | s.pending in neighbors }

-- start with no moments to carry over
pred init [s: State] { no s.pending }

-- release a joint, carry over a moment, or stutter
pred step [s, s': State] {
   (some u: Joint | u not in Fixed and release[u, s, s'])
      or (some u, v: Joint | carryover[u, v, s, s'])
         or stutter[s, s']       -- leave state unchanged
}

-- release and balance a joint u, set up carry overs
pred release [u: Joint, s, s': State] {
   u not in s.balanced and no s.pending[u]
   s'.balanced = s.balanced + u
   s'.pending = s.pending + u <: neighbors
}

-- carry a moment from u over to v, making v unbalanced
pred carryover [u, v: Joint, s, s': State] {
   u→v in s.pending
   s'.balanced = s.balanced - v
   s'.pending = s.pending - u→v
}

pred show {
   init[so/first]
   all s: State - so/last | step[s, s.so/next]
}
```

Figure 2.5 A specification of the moment distribution method in Alloy [1].

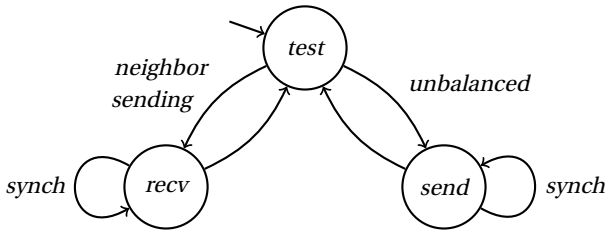

Figure 2.6 Joint processes communicating by synchronous message passing.

The protocol is akin to a rendezvous in CSP, where senders block until there is a matching receive, and where nondeterministic choice allows processes to communicate with neighbors in an order that is left undefined.

To model the refinement in Alloy, a program counter, *pc*, is added to a subsignature $State_R$ of *State* as a field, allowing each process to record its current mode: *test*, *send*, or *recv*. Given $init_R$ and $step_R$ predicates that describe this behavior, refinement R can be shown to satisfy the specification as follows over all states r and r' in $State_R$:

$$init_R[r] \Rightarrow init[\alpha[r]]$$
$$inv_R[r] \textbf{ and } step_R[r, r'] \Rightarrow step[\alpha[r], \alpha[r']]$$

under refinement mapping α, an abstraction function that maps every element of $State_R$ to at most one element in *State*, and where inv_R is an inductive invariant that eliminates unreachable states in refinement R.[3] Doing so checks all building topologies (within bounds), and is in this case an over-approximation, since the specification admits more topologies than we expect to find in actual building structures.

Embellishments can easily be imagined that bring additional concerns into the picture. Once the method reaches convergence, for instance, a carry-over operation no longer causes the receiving joint, within some tolerance, to become unbalanced. A parallel implementation would ideally detect a state of global quiescence using a separate, lower-priority process or perhaps using a more general technique like Dijkstra's ring-based detection algorithm [30]. Other refinements might include red-black ordering for joint processes and other groupings, both static and dynamic, that are commonly used in domain decomposition methods to improve performance. In addition, the particulars of MPI-style communication [58] and other library frameworks may already be specified, and could be included in the modeling exercise.

Finally, with respect to code, the approach taken above has been implemented in Go using parallel goroutines and message passing, and with guarded select statements that mimic similar capabilities found in CSP. Both this and another implementation that includes Dijkstra's ring-based detection algorithm are available along with the Alloy specifications online [1].

[3] Since a process associated with a joint cannot have moments to carry over unless it is in *send* mode. More generally, because the refinement mapping is functional, the proof obligations are simplified [84].

2.4 Instruction and Templates

Declarative languages like Alloy are expressive, but it is not always clear how they can be used to specify a system. There are no special constructs for parallelism, message-passing, synchronization or other mechanisms that give some insight into what one is "supposed" to do with it. In other words, there are few affordances or "action possibilities" that are readily perceivable.

To make this style of modeling more accessible, we are working on a collection of "templates," small Alloy models that can be used as a starting point for model building. The idea is to group together examples that share common features into a category, and to define what may become a dozen or more categories, each with several examples and variations on them that help reveal an abstract concept through a series of "concrete" examples.

In designing the collection, we are beginning with simple building blocks, categorizing problems in scientific computation according to their structure and behavior. We are also including examples that demonstrate Alloy idioms for state changes, traces, and other structuring mechanisms in the context of our domain, primarily civil and environmental engineering and science.

This perspective on templates and model building is informed by firsthand experiences in an undergraduate engineering course on mathematical programming, which involves setting up and solving declarative, algebraic models that include objectives and constraints in terms of decision variables. The idea of using templates for instruction is due to Schrage [71] and probably others in the field of operations research, where the approach is commonly used.

One could argue that the value of state-based approaches must first be demonstrated in practice, though we might also reimagine what instruction in computation *should* look like for scientists and engineers, particularly at the graduate level. An experience of building models and reasoning about them is likely to be beneficial in the long run, even in situations where formal models are not developed. Our own anecdotal evidence suggests that it is.

2.5 Related Work

In related work on state-based modeling [11, 10], we look at a large-scale ocean circulation model used in production and an extension called subdomain modeling that offers substantial performance gains. Models developed in Alloy allow us to draw useful conclusions about implementation choices and guarantees about the extension, in particular that it is equivalence preserving. In one of its earliest applications, subdomain modeling was used in post-Katrina studies by the U.S. Army Corps of Engineers, where the technique "yielded considerable time and cost savings in the calculations" when analyzing the Western Closure Complex, a key component of the hurricane storm damage reduction system for New Orleans.

As an example of reasoning about numerical concerns, in another study [5] we apply hybrid theorem proving from the field of cyber-physical systems to problems in scientific computation, and show how to verify the correctness of discrete updates that appear in the simulation of contin-

uous physical systems. By viewing numerical software as a hybrid system that combines discrete and continuous behavior, test coverage and confidence in findings can be increased. We describe abstraction approaches for modeling numerical software and demonstrate the applicability of the approach in a case study that reproduces undesirable behavior encountered in ocean components of large-scale climate models.

In representative work by others, Bientinesi et al. [15] use Floyd-Hoare logic to derive and prove the correctness of dense linear algebra algorithms. Their approach is a correct-by-construction technique that uses a simplified matrix notation to hide indexing details. The authors show how invariants for nested loops can be found systematically for a broad class of dense linear algebra routines.

Siegel et al. [72] present a framework that tests small numerical programs for *real equivalence*, meaning that one program can be transformed into the other using the identities of real numbers. The approach is based on creating a sequential program that serves as a specification and then using it as the measure against which an implementation is compared, such as a more complex MPI-based parallel program. Equivalence is checked by building up symbolic expressions in both programs and comparing them using the SPIN model checker.

2.6 Conclusions

The extent to which formal methods might become more broadly used in the development of scientific software is an open question. Some of the techniques that automatically extract finite-state models from code may make such approaches more accessible to a wider community, though we wonder if tethering them to conventional programming languages is the ideal means of doing so. The approach described here, by way of contrast, involves us in the development of *models* of software, which may offer some advantages: scientists and engineers are accustomed to working with models anyway, and with automatic, push-button analysis as an alternative to theorem proving, they can focus on modeling and design aspects. In addition, by not calling for language-specific modules, interfaces, or classes, the approaches we advance do not presuppose that scientific programmers commit to a certain kind of programming language or environment to enjoy the benefits of creating and working with models of software.

Given the fundamental role of computation in modern science, the development and adoption of better design practices could have far-reaching benefits. Toward that end, we suggest a focus on essential complexities and scientifically relevant computational abstractions, as advocated by Faulk et al. [32], using precise and expressive notations that support exploration and analysis. Future work in this direction may lead to new insights and deeper understanding, as well as auxiliary tools and instructional materials that make these advances more accessible to scientists and engineers in traditional disciplines.