ABSTRACT

DYER, ANDREW TRISTAN. Lightweight Formal Methods in Scientific Computing. (Under the direction of John Baugh.)

Computation is an indispensable tool for scientists and engineers as they seek to understand and shape the world around us. Despite broad and recognized impacts, the field of scientific computing faces challenges related to reliability, reproducibility of results, and productivity. Recent studies confirm what many have suspected or experienced firsthand, that existing practices of constructing scientific software are inadequate and limiting the pace of technological progress. Sources of difficulty may stem from fundamental characteristics of the problem domain, along with cultural and development practices within it. For example, programmers in scientific areas are often domain experts with a Ph.D. in their respective fields, and a single scientist may take the lead on a project, relying 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. Proposals to address these concerns are varied; recommendations ranging from development and quality assurance practices to new design approaches have been suggested.

We develop an approach based on lightweight formal methods, which have received relatively less attention in scientific computing, to address these concerns, allowing us to focus on the essential complexities of software. We employ a state-based formalism called Alloy, which offers rich data modeling features and automatic, push-button analysis performed within a bounded scope using a SAT solver. To evaluate the approach, we begin with a small study of one of the earliest successful algorithms in civil engineering, the moment distribution method. This example, though small, is representative in important ways of scientific software due to its iterative and spatially discrete nature. Building on this experience, we turn to a more in-depth example to better understand the needs of a real-world application. In this case study we reason about the structure and behavior of sparse matrices, which are central to many applications in scientific computation. To express the kind of stateful iteration patterns characteristic of scientific software, a new idiom is presented for loops with incremental updates; it is then applied to sparse matrix-vector multiplication, matrix transpose, and translation between sparse formats. To improve the visualization of instances with spatial and positional features, we introduce a new web-based front-end for Alloy called Sterling. Users enjoy the same immediate visual feedback the Alloy Analyzer provides, but with enhanced Graph and Table Views that extend existing functionality and improve visual consistency between instances that dynamically evolve as part of a trace. In addition, to better facilitate the design and understanding of complex models in science and engineering, Sterling offers a Script View for creating domain-specific visualizations and improving insight. The approach, idiom, and tools we present are intended to help scientists and engineers manage the complexities of the software they write, and to meet fundamental design and quality assurance challenges that are intrinsic to scientific computation and related types of numerical software.

Lightweight Formal Methods in Scientific Computing

by
Andrew Tristan Dyer

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Civil Engineering

Raleigh, North Carolina

2020

APPROVED BY:

Ranji Ranjithan	Kumar Mahinthakumar
Murthy Guddati	John Baugh Chair of Advisory Committee

DEDICATION

To my family.

BIOGRAPHY

Tristan Dyer was born in Fayetteville, North Carolina, in 1988. He studied Civil Engineering as an undergraduate at North Carolina State University and graduated in 2011. He continued his graduate studies there, receiving a Master's degree in 2013 and a Doctorate in 2020. He now resides in Providence, Rhode Island, where he is a postdoctoral researcher in Computer Science at Brown University.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

Computation is an indispensable tool for scientists and engineers. Despite broad and recognized impacts in scientific computing, however, the field is challenged by concerns around reliability, reproducibility of results, and productivity. Recent studies have brought increased attention to existing practices of constructing scientific software, which are shown to be inadequate and limiting the pace of technological advancement. In a comprehensive review of the literature that covers the interaction between software engineering and scientific programming, Storer [74] catalogs numerous empirical studies of software "thwarting attempts at repetition or reproduction of scientific results." Among these studies are attempts to measure the repeatability of results in computing science, one of which found that only around a quarter of the research work from recent computing science conferences could be conveniently reproduced by acquiring, compiling, and running the associated source code [27]. 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."

That these challenges exist is perhaps not surprising; software is "essentially" complex, intangible, and volatile in nature [22], and the typical characteristics of scientific software, including numerical issues, rich data, complex parallelism, and long running times, compound these challenges. We observe, though, that many of the same challenges exist in the broader software industry, and countless methods and tools have been developed to address them. Despite this progress in other domains, the adoption of established software engineering practices and tools in scientific computing is limited at best [77]. Why is this the case, and why does the "communication chasm," as Carver puts it, between software engineering researchers and scientists who develop software exist?

One explanation for this "gap" is a lack of formal training. Developers of scientific software are typically experts in their field, often with a Ph.D., who have little training in software engineering

tools and techniques. According to one study [24], scientists and engineers overestimate their ability to produce high quality software. In a self-assessment of their ability to produce quality software, scientist-developers rated themselves highly despite low awareness of many software engineering practices, and the authors conclude that the results of the survey support their argument that scientist-developers simply "don't know what they don't know." In another survey of almost 2,000 scientists, researchers found that developers of scientific software believe testing is important but typically lack sufficient knowledge about it [38]. Conversely, we observe that computer scientists, who have the training and expertise to produce high quality software, may not have the domain-specific knowledge required to write software in the scientific domain.

We must nevertheless acknowledge the fundamental differences between scientific and other types of software. To examine these differences and related issues, researchers from the scientific computing and software engineering communities came together in 2009 for a workshop whose outcomes are summarized by Carver [25]. In this workshop, researchers identified key characteristics that set apart scientific software. First, a lack of **test oracles** means that validating results can be difficult, if not impossible. It is often the case that for the types of large scale simulations performed by scientific software, such as those simulating natural processes like ocean and atmospheric circulation, there is no known solution that can be used to validate results. In these cases researchers typically have to settle for plausibility checks based on conservation laws or other principles that are expected to hold. Furthermore, because these types of simulations often require extensive computing resources in terms of processing power and data throughput, there tends to be a higher focus placed on **performance** and hardware utilization over code quality. Then, because projects often explore unknown sciences, the **requirements** are not necessarily known a priori and may emerge and evolve throughout development. Finally, the **lifecycle** of a scientific software project is also likely to differ from that of other types of software. In some cases, a piece of software may be written and used only once by a single researcher, and in others it may be adopted by the community and survive decades of continued development and contributions from other scientists, who may also have little formal training in computer science. We must conclude that it is not just a lack of formal training that contributes to the disconnect between software engineering and scientific software development practices.

To address these and other challenges, some have suggested adopting methods and tools from the computer science community. These include development processes such as agile methods, quality assurance practices such as unit testing, inspections, and continuous integration, design approaches such as component architectures and design patterns, and formal methods, where models of software are created to explore and understand design. Attempts to adopt these approaches have been met with varying levels of success, as some transfer more easily than others. Version control systems and continuous integration tools, for example, are easily adopted [8], and some even argue that agile methods are a "better fit" for the dynamic and concurrent nature of scientific software development [3, 83]. Formal methods, on the other hand, have received little attention, possibly due to "the additional challenge of verifying programs that manage floating point data" [74].

If we pause and consider what scientific programs are really like, however, a previously overlooked role for formal methods may begin to emerge. The subject matter of scientific software often involves physical and natural processes where space and time are viewed as being continuous. The software that simulates these processes, however, looks less like a set of purely analytic functions and more like a combination of discrete data structures, algorithms, and of course numerical computations. As a result, one might adopt a perspective that separates concerns, and thereby avoid mixing the types of reasoning best suited for numerical computations and the "interstitial machinery" throughout which they are embedded. State-based formal methods are particularly well suited for reasoning about the structural and behavioral characteristics that are intrinsic to this latter category. We cannot, of course, ignore issues related to numerical computations such as floating point accuracy, but we posit that it is easier to reason about them separately.

The overall goal of this 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. While numerous directions might be taken, our premise and motivating viewpoint is the central role that modeling can and must play in the process of designing and working with complex artifacts, including scientific programs. Culturally, the fit may be a natural one: scientists and engineers are already accustomed to working with models, 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.

This dissertation is organized as follows. In Chapter 2 we present 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 [46] which, while increasingly promoted in areas like communication protocols and control systems, may also be exploited in scientific domains. This chapter is an extension of the work [12] presented at ABZ 2018: The 6th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z.

In Chapter 3 we further develop the approach for reasoning about sparse matrix computations, and show how data abstraction and refinement principles can be used to check invariants and perform bounded verification of safety properties. This work [31] appears as published in the 2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications.

In Chapter 4 we introduce a web-based interface for Alloy called Sterling, with enhanced Graph and Table Views, as well as a Script View for domain specific visualizations. Users receive the same immediate visual feedback the Alloy Visualizer provides, but with added features that better facilitate and embrace an iterative approach.

Finally, in Chapter 5 we conclude with closing remarks and discuss the ongoing development of Sterling.

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

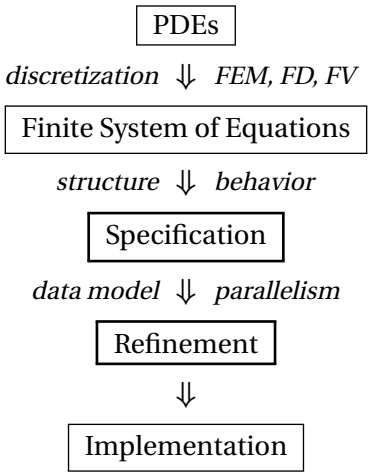


Figure 2.1 A refinement perspective for scientific software, where parts of the highlighted *Specification* and *Refinement* steps are formalized using state-based methods.

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-

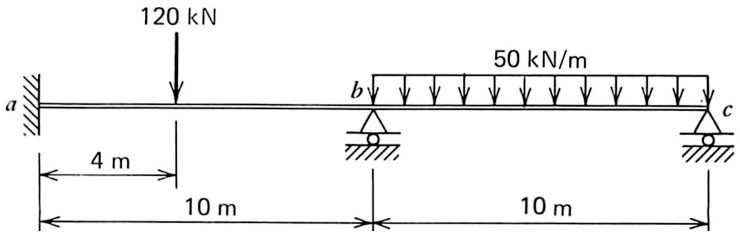


Figure 2.2 Continuous beam with applied loads, supports at labelled joints [80].

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.

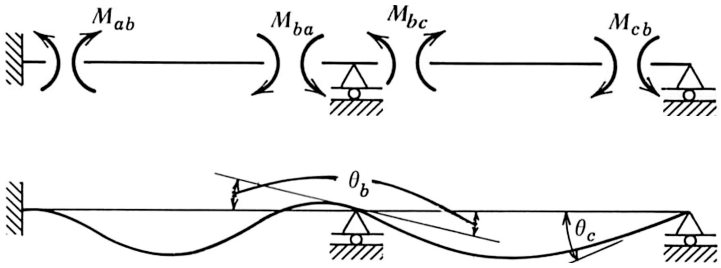


Figure 2.3 Unknown moments (above) and deflected shape (below) [80].

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

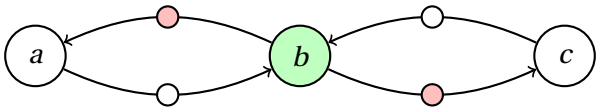


Figure 2.4 Structure represented as a symmetric, directed graph.

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].

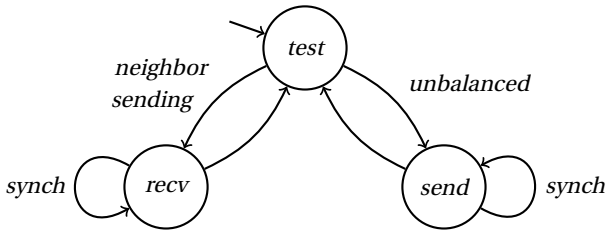


Figure 2.6 Joint processes communicating by synchronous message passing.

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.

CHAPTER

3

BOUNDED VERIFICATION OF SPARSE MATRIX COMPUTATIONS

3.1 Introduction

Sparse matrices are commonly used in scientific and engineering domains to reduce storage requirements and minimize computational effort. For applications in large-scale simulation, signal processing, and machine learning, a variety of formats have been developed—some historical and more widely used, and others of increasing sophistication that track evolving computer architectures. Sparse implementations are realized in popular packages like SuiteSparse, Sparse BLAS, and Sparskit, and as components of larger, more general-purpose libraries and frameworks.

To avoid storing zeros, sparse formats use array indirection and other machinery to encode structure and provide access to non-zero elements, while attempting to exploit hardware characteristics and optimize performance. Memory safety and full functional correctness are obvious concerns, not only for developers of libraries but also for users who work directly with sparse formats, since abstraction boundaries, when present, are often bypassed in the interest of performance.

In ocean circulation modeling—a motivating application for us—sparse matrices figure prominently. Unstructured grid models based on finite element methods use custom assembly routines, impose boundary conditions for wetting and drying to accommodate overland flooding, and perform these and other updates in between calls to linear solvers as they step through time. Preserving representation invariants is a basic safety concern, and dependencies between formats and solvers mean that substituting one solver for another can create ripple effects in the codes that use them.

Though important and challenging, static verification of sparse matrix software has received

little attention. In a study addressing the problem, Arnold et al. [7] describe several attempts to do so, noting that they "failed to verify the functional correctness of even simple formats using several state-of-the-art tools," before creating a variable-free functional language in the style of FP to support verification with Isabelle/HOL.

In this chapter, we develop and present a state-based approach for reasoning about sparse matrix computations and show how data abstraction and refinement principles can be used to check invariants and perform bounded verification of safety properties. We use Alloy [46], a lightweight formal method, to develop models that represent the structure and behavior of sparse matrices, and introduce a new idiom that supports the modeling of imperative loop structure, as is commonly found in scientific software.

To contrast our work with Arnold et al. [7], our models are intended to be more directly relatable to code in imperative programming languages like Fortran and C++; we rely on a formalism and tool whose application is more readily transferable to allied problems in scientific computing [11], allowing for economies of scale in their use; and because verification is bounded, our approach comes with push-button automation that does not require ingenuity in proving theorems.

The chapter is organized as follows. Section 3.2 introduces the approach, the Alloy language, and notions of correctness and refinement. Sections 3.3 and 3.4 show how matrix structure and behavior can be modeled and verified, with examples of ELL and CSR formats. Section 3.5 describes an idiom for bounded iteration and models for translation between sparse formats, matrix transpose, and matrix-vector multiplication. Section 3.6 discusses scope, the ability to detect bugs, and limitations of the approach. Section 3.7 describes related work, and Section 3.8 offers conclusions and directions for future research.

3.2 Approach

We make use of a state-based formalism called Alloy [46], a declarative modeling language that combines first-order logic and relational calculus, and includes 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 Alloy 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.

Because Alloy is a structural modeling language it provides no means of representing real numbers or floating point values, and has only limited support for integers. In this study, we model zero and non-zero values relying only on the property that different numerical values are distinct, and checking for real equivalence of symbolic expressions when and as needed. In related work [11],

our group uses predicate abstraction in Alloy models to factor out numerical concerns, allowing us to show that a performance enhancement made to a popular ocean model is equivalence preserving.

Instead of automatically generating verification conditions from code, we work with "abstract algorithms" that model the array indexing, mutation, and stateful behavior of programs written in imperative languages like Fortran and C++. Doing so makes the approach language agnostic and keeps verification tractable, since fine-grained control can be exercised over model details and scopes. The checks are not exhaustive, but we appeal to the *small scope hypothesis* [46, 6], which suggests that most bugs have small counterexamples.

3.2.1 Structure and Behavior

In state-based formalisms like Alloy, systems are described by defining what constitutes a state and the transitions between states. Though not an obvious choice for scientific software, such an approach is consistent with the perspective we advance: by separating concerns we can direct attention to structural and behavioral complexities that exist apart from the numerical ones [12].

With respect to structure, for instance, complex state is defined implicitly by declarative properties, in terms of sets, relations, and logical formulas. The Alloy Analyzer then serves as a *model finder* in the mathematical sense, finding models of logical formulas. What this means in practice is that fragments of scientific programs, existing or planned, can be put through their paces, with input state automatically generated to drive the model into its corner cases, should they exist. Such state might include, for instance, mesh topologies used in hurricane storm surge simulations, as developed in prior work [11], or sparse matrix formats, the subject of this chapter.

In terms of behavior, operations are modeled as predicates that define transitions between states, which are also defined declaratively. Nondeterminism may be employed as a means of expressing concurrency [54] in models, which may be written in interleaving or noninterleaving styles. For "stateful" algorithms that rely on mutation, Alloy has no fixed idioms, but a common approach is to expand the arity of a "dynamic" relation by introducing a time column and imposing an ordering on time. We later introduce a complementary approach that works well for matrices and similar operations that rely on bounded iteration.

3.2.2 Correctness and Data Refinement

Our notion of conformance is based on substitutability. A computer program written in terms of matrix computations, if correct, should remain correct if sparse matrix formats are used instead to improve performance. The historical origins of data refinement begin with Hoare [42] and proceed in two major veins—based on relational and predicate transformer semantics—with numerous representative examples including Reynolds' stepwise refinement of programs [69], Back and Morgan's refinement calculus [66], and Abadi and Lamport's refinement mappings [2]. More recently, Bolton [17] shows how data types in the Z notation can be translated into Alloy using an explicit encoding that is able to find refinement mappings automatically.

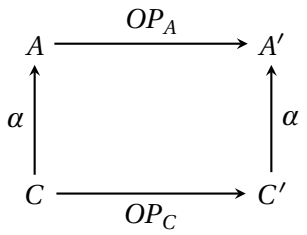


Figure 3.1 Refinement commuting diagram.

To verify sparse matrix formats and operations on them, we adopt a perspective common to state-based formalisms, and use data refinement to show that a more detailed concrete system can simulate a more abstract one. The diagram in Figure 3.1 shows abstract (A) and concrete (C) domains with unprimed and primed terms that correspond to pre- and poststates, respectively, of abstract (OP_A) and concrete (OP_C) operations.[1] A functional relation from concrete to abstract domains, the *abstraction function* α, describes how states satisfying a *concrete invariant* I are interpreted.

We say that a sparse matrix operation OP_C conforms to an abstract one, OP_A, if

$$I(C) \wedge OP_C(C, C') \wedge \alpha(C, A) \wedge \alpha(C', A') \Rightarrow OP_A(A, A') \tag{3.1}$$

a safety property stating that nothing "bad" happens, a type of check well-suited to Alloy. To ensure that something happens at all, a liveness property, is more difficult to formulate in Alloy since it involves unbounded universal quantification over states, as we discuss in Section 3.6.

Abstract matrix operations, then, serve as a specification and are formulated declaratively using Alloy's set comprehension notation, as are abstraction functions. Sparse, concrete operations are often stateful, and for those we use a new idiom for bounded iteration that has intuitive appeal and an obvious relationship to code in imperative programming languages. Concrete invariants have a direct use as heap invariants [45] that must be satisfied by implementing programs.

In subsequent sections, we present portions of Alloy models to illustrate our approach, noting that complete models can be found online [1]. Also, although we introduce most major features of the language as we go, it may be helpful for those unfamiliar with it to consult the Alloy language reference, which is available online.[2]

3.3 Matrix Structure

Two commonly used sparse matrix formats are ELLPACK (ELL) and compressed sparse row (CSR), which we introduce here, looking first at their structure. Figure 3.2 shows a matrix in dense, ELL, and CSR formats.

[1] In subsequent sections, we drop subscripts A and C, in contexts where it is obvious, for operations and abstraction functions. Alloy also supports this type of operator overloading.

[2] alloytools.org/download/alloy-language-reference.pdf

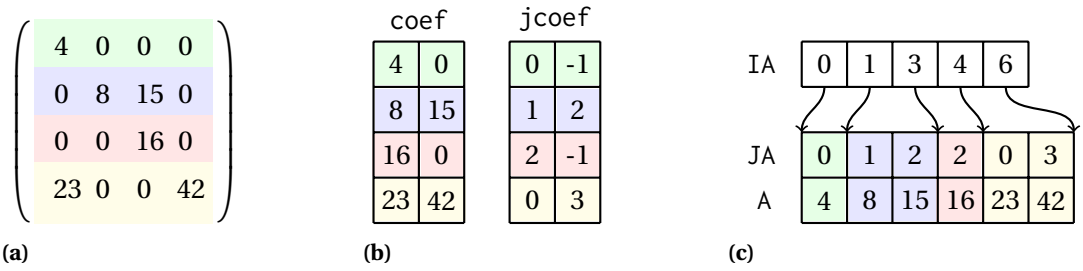


Figure 3.2 A matrix in dense (a), ELL (b), and CSR (c) formats, with rows colored to show how elements are stored across formats.

The ELL format, named for the ELLPACK library from which it originates, uses two two-dimensional arrays, `coef` and `jcoef`, as seen in Figure 3.2b. The `coef` array stores matrix values and the `jcoef` array stores column indices for the corresponding values in `coef`. The dimension of each array is *rows* × *maxnz*, where *maxnz* is the maximum number of non-zero values in any single row of the matrix. Rows that contain fewer than *maxnz* non-zero values are padded with placeholder values—negative one in the `jcoef` array and zero in the `coef` array.

The CSR format offers further compression of the ELL format by removing the values used for padding, as shown in Figure 3.2c. To do so, the `coef` array is flattened into row-major order to produce the one-dimensional array `A`, and the same process is applied to the `jcoef` array to produce `JA`. To access individual rows, an indexing array, `IA`, contains the starting location of each row within the two arrays.

Turning to Alloy, matrix values and dense, ELL, and CSR representations are defined by the signatures shown in Figure 3.3. A *signature* in Alloy introduces both a type and a set of uninterpreted atoms, and may introduce *fields* that define relations over them. In addition to defining a type, a signature's name can also be used within an Alloy expression to denote the set of elements it defines. Subtype signatures using *extends* introduce no new types but instead represent sets of elements that are subsets of their parents, and the *one* keyword denotes a singleton subset.

Since Alloy provides no means of representing reals or floating point values, matrix elements are modeled as some number of distinct non-zero values and zero, depending on scope size. The `Value` and `Zero` signatures introduce the following subscripted, uninterpreted atoms:

$$\texttt{Value} = \{Zero_0, Value_0, Value_1, \ldots, Value_{n-2}\}$$

which are drawn from when a scope of size n is chosen for `Value` (since `Zero` is a subtype). This simple approach suffices for representing the structural properties of matrices, and where more is needed, arithmetic expressions can be built up and checked symbolically, as shown in Section 3.5.4 for matrix-vector multiplication.

Abstract state, from a refinement perspective, is defined by the `Matrix` signature, which includes fields for the number of rows and columns and for dense storage. The `vals` field is a relation that denotes a two-dimensional array, mapping row and column indices to values; the multiplicity

```
sig Value {}
one sig Zero extends Value {}

sig Matrix {
  rows, cols: Int,
  vals: Int → Int → lone Value
}

sig ELL {
  rows, cols, maxnz: Int,
  coef: Int → Int → lone Value,
  jcoef: Int → Int → lone Int
}

sig CSR {
  rows, cols: Int,
  IA, JA: Int → lone Int,
  A: Int → lone Value
}
```

Figure 3.3 Matrix structure in Alloy: signatures for values and matrices in dense, ELL, and CSR formats.

keyword *lone* (less than or equal to one) says there can be at most one such value for any index pair. The combination of Alloy's dot join and box join operators[3] means that, for a matrix `m`, the `i-j` element can be referred to as `m.vals[i][j]`, and if its value is `v`, the tuple `i→j→v` is a member of the `m.vals` relation.

For concrete state, the `ELL` and `CSR` signatures define their respective formats. In the ELL format, `coef` and `jcoef` fields once again denote two-dimensional arrays, as before, and in the CSR format, `IA`, `JA`, and `A` fields denote their respective one-dimensional arrays.

At this point, the collection of signatures defined in Figure 3.3 constitute a complete Alloy model. For validation, the Alloy visualizer can be used to step through and inspect instances, either textually or graphically, that are populated by atoms bounded by the individual scopes of the `Value`, `Matrix`, `ELL`, and `CSR` sets. When doing this, some of the instances produced correspond to valid formats, like those shown in Figure 3.2, and some do not. For instance, some `Matrix` instances have `vals` with `i-j` indices out of bounds, some `ELL` instances have invalid column indices in `jcoef`, and so on.

Constraints on structure can be imposed in Alloy either as *facts*, which must always hold, or as *predicates*, which the Analyzer can check. A concrete invariant for ELL, for instance, might be defined as a predicate to see if it is maintained by an ELL operation, as we illustrate below.

[3] Alloy's dot join operator, relational composition, generalizes the conventional syntax of classes and fields in object-oriented languages. Box join mirrors dot join but with a syntax convenient for indexed lookup.

3.4 Matrix Behavior

To describe dynamic behavior in Alloy, operations are defined as predicates, or relations between states. Figure 3.4 shows examples of some basic predicates, along with assertions to check their behavior.

The model fragment in the top half of Figure 3.4 begins with a predicate `update`, a basic operation of the ELL format, which can be used to change a single value in the matrix—this might be a step in element assembly or matrix construction operations that are either provided by libraries or implemented by users. Parameters include pre- and poststate ELL matrices (`e`, `e'`), an index pair (`i`, `j`) specifying an element of the matrices, and a new value (`v`) for the element in the poststate.

Within `update`, the first line acts as a precondition, or guard, so that pre- and poststates are related only if indices are valid. An implication then makes use of helper predicates, depending on whether the new value for `v` is zero (shown) or non-zero (not shown). Finally, a frame condition [18] is defined by predicate `sameDimensions`: the number of rows and columns is unchanged in the transition. In `toZero`, the expression `k = e.jcoef[i].j` uses relational join to determine the column of `jcoef` that contains index `j` of row `i`, if it exists. The `setAt` predicate overrides the value of the `i-k` elements in `coef` and `jcoef`. To show that `update` preserves the concrete invariant for ELL matrices (not shown), an assertion `preservesInvariant` is provided.

The model fragment in the bottom half of Figure 3.4 defines the abstraction function α for ELL matrices as a predicate, showing the (functional) relationship from concrete to abstract states. It uses a set comprehension to define `m.vals`: for proper `i-j` pairs, the value `v` is in the column of `jcoef`—denoted by `k`—that contains index `j` of row `i`, if it exists; otherwise `v` is zero. For bounded sets of integer indices, an Alloy *function* named `range` is defined.[4] A refinement check can then be performed using `updateRefines` to show that the concrete `update` operation conforms to the abstract one.

3.5 Matrix Computations

As outlined, the essential structure and behavior of sparse matrix computations can be modeled, validated, and checked for conformance. When operations are simple enough, state transitions can be defined declaratively in a straightforward manner using set comprehensions and other basic elements of first-order logic and relational calculus. Operations like sparse matrix transpose and translation between sparse formats, on the other hand, generally involve nested loop structure and rely more commonly on mutation, a natural consequence of using imperative programming languages. Below we describe a new idiom for this style of computation and present several examples of verifying sparse matrix algorithms, including ELL to CSR translation, CSR transpose, and CSR matrix-vector multiplication.

[4] We subsequently overload `range` so that it can accept two parameters: the first being an (inclusive) lower bound, the second an (exclusive) upper bound.

```
pred update [e, e': ELL, i, j: Int, v: Value] {
  i→j in indices[e]
  v = Zero ⇒ toZero[e, e', i, j]
    else toNonZero[e, e', i, j, v]
  sameDimensions[e, e']
}

pred toZero [e, e': ELL, i, j: Int] {
  let k = e.jcoef[i].j |
    e'.jcoef = setAt[e.jcoef, i, k, -1] and
    e'.coef = setAt[e.coef, i, k, Zero]
}

assert preservesInvariant {
  all e, e': ELL, i, j: Int, v: Value | I[e] and update[e, e', i, j, v] ⇒ I[e']
}
```

```
pred α [e: ELL, m: Matrix] {
  m.rows = e.rows
  m.cols = e.cols
  m.vals =
    { i: range[e.rows], j: range[e.cols], v: Value |
        let k = e.jcoef[i].j |
          some k ⇒ v = e.coef[i][k]
            else v = Zero
    }
}

fun range [n: Int]: set Int { { i: Int | 0 ≤ i and i < n } }

assert updateRefines {
  all e, e': ELL, m, m': Matrix, i, j: Int, v: Value |
    I[e] and α[e, m] and α[e', m'] and
      update[e, e', i, j, v] ⇒ update[m, m', i, j, v]
}
```

Figure 3.4 Matrix behavior in Alloy: the ELL update operation and invariant check (above), and ELL abstraction function and refinement check (below).

3.5.1 An Idiom for Stateful Behavior

To accommodate stateful algorithms in declarative languages, idioms are often devised to address the so-called incremental update problem [43], such as state transformers [56] and lazy streams [44] in functional programming, array comprehensions and accumulators in dataflow and single assignment languages [78, 67], and a relational view taken of sparse matrix computations as database queries [51].

Although Alloy itself has no built-in notion of mutation, there are several techniques for modeling it [46, 76]. One common approach—a *local state* idiom—adds a column to a relation to make it "dynamic:" the addition serves as sort of a timestamp for the other columns in the relation. Then, using one of Alloy's built-in modules, a total ordering can be applied to the signature being used as a timestamp. Another approach more common to other state-based formalisms—a *global state* idiom—separates relations by placing them into different signatures based on whether they are considered static or dynamic, and using dynamic signatures for the pre- and poststate parameters of transition predicates.

For matrix computations, these approaches prove to be difficult to make work in practice because of complex interactions between nested loop structure, conditionals, and the exact scopes used by the ordering module. It is particularly important, then, to find idioms and design patterns for formal methods, where possible, that accommodate particular domain needs and broaden their range of applicability [41]. Here we describe a *tabular* idiom for the type of nested, bounded iteration commonly found in matrix computations.

```
some iter: Int → Int → Int, x, y, ...: Int → univ {
  table[{i: ψ, j: ω |...}, iter]
  all i: ψ |
    all j: ω |
      let t = iter[i][j], t' = t.add[1] {
        x[t'] = ... x[t] ...
        y[t'] = ... y[t] ...
        ...
      }
}
```

Figure 3.5 Tabular pattern for nested loops defining an iteration table (`iter`) and time-indexed scalar variables (x, y), where ψ and ω define loop bounds.

Figure 3.5 illustrates the pattern for two nested loops, with boilerplate that controls iteration over an innermost loop body. The existentially quantified expression binds an iteration table `iter` and some number of dynamic variables, such as x and y (of **`univ`** type, the universal set), which are indexed by an **`Int`** timestamp. The `table` predicate establishes the form and order of the table using an argument that defines loop bounds as a binary relation, and to which a time column is added. Time variables `t` and `t'` are used within the loop body to work with current and next values of the

dynamic variables.

While the illustration above makes use of *scalar* dynamic variables, more complex types with array-like indexing, for instance, can be represented by increasing the arity of the relation used to represent the variable, as we later show.

3.5.2 ELL to CSR Translation

The need to translate between one sparse format and another may arise for a number of reasons, including dependencies between formats and solvers, the relative differences in performance between construction and other operations to be performed, and so on.

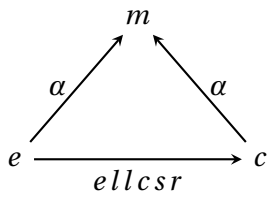
To translate between ELL and CSR formats, as an example, recall from earlier descriptions that the CSR representation removes padding from rows that contain fewer than *maxnz* values in the ELL format, as shown in Figure 3.2. To allow for this, a third array containing the start location for each row, IA, is defined. The translation algorithm, shown in Figure 3.6a, loops through the coef and jcoef arrays used in the ELL format, adding any non-zero values to the A and JA arrays used in the CSR format. A variable *kpos* keeps track of the next available position in the CSR arrays. Once an inner loop completes, the value of *kpos* is the starting location of the next row to be recorded in IA.

$kpos \leftarrow 0$
for i **in** range($rows$) **do**
  for k **in** range($maxnz$) **do**
    if jcoef$[i,k] \neq -1$ **then**
      A$[kpos] \leftarrow$ coef$[i,k]$
      JA$[kpos] \leftarrow$ jcoef$[i,k]$
      $kpos \leftarrow kpos + 1$
  IA$[i+1] \leftarrow kpos$

(a)

```
kpos[0] = 0
all i: range[e.rows] | {
  all k: range[e.maxnz] |
    let t = iter[i][k], t' = t.add[1] |
      e.coef[i][k] != Zero ⇒ {
        c.A[kpos[t]] = e.coef[i][k]
        c.JA[kpos[t]] = e.jcoef[i][k]
        kpos[t'] = kpos[t].add[1]
      } else
        kpos[t'] = kpos[t]
  c.IA[i.add[1]] = kpos[end[iter, i].add[1]]
}
```

(b)



(c)

Figure 3.6 ELL to CSR translation: (a) pseudo-code, (b) fragment of Alloy model, and (c) commuting diagram.

Like other operations in Alloy, the translation from ELL to CSR formats is defined as a predicate:

```
pred ellcsr [e: ELL, c: CSR] { ... }
```

Using the tabular idiom, we distinguish between static and dynamic variables. The A, JA, and IA arrays are static since their elements are set just once. The variable kpos, however, is dynamic, so the following boilerplate is used ahead of the Alloy fragment shown in Figure 3.6b:

```
some iter: Int → Int → Int, kpos: Int → Int {
  table[range[e.rows] → range[e.maxnz], iter]
```

which defines iter from the loop bounds and gives a binding for kpos, the only dynamic variable. Because it is dynamic, kpos is indexed by time, and its current and next values are given by kpos[t] and kpos[t'], respectively. When the conditional test e.coef[i][k] != Zero is false, the expression kpos[t'] = kpos[t] serves as a frame condition for kpos. The function end in the last line of the Alloy fragment obtains kpos at the end of an inner loop.

As shown in Figure 3.6c, the abstraction functions α for both the ELL and CSR formats are used to determine correctness, which we check as follows:

$$I(e) \wedge ellcsr(e,c) \Rightarrow (\alpha(e,m) \Leftrightarrow \alpha(c,m)) \tag{3.2}$$

where the expression is universally quantified over matrices in dense (m), ELL (e), and CSR (c) formats.

3.5.3 CSR Transpose

Matrix transpose swaps the row and column indices of a matrix, so its definition is straightforward for a dense matrix representation. Using a set comprehension for the vals field of poststate m', the transpose of m is:

```
{ j, i: Int, v: Value | i → j → v in m.vals }
```

which swaps i and j indices.

The CSR transpose algorithm is more involved. It consists of four phases: (1) compute row lengths of the transpose, (2) set the starting location of each row in the IA array, (3) copy values and indices into the A and JA arrays, using the content of IA as iteration variables (so IA is destructively modified), and (4) right shift the content of IA one place, returning it to its state at the end of phase 2.

The algorithm uses two sets of arrays: A, JA, and IA arrays as input, and AO, JAO, and IAO arrays as output. Focusing on just the third phase, shown in Figure 3.7a, the algorithm steps through rows of the input matrix, determines the current column j, finds the starting position nxt of that column in the output matrix using the IAO array, and updates those elements of the A and JA arrays. The starting location of that column is then incremented in the IAO array for the next iteration.

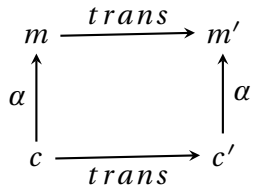
For the Alloy model, a fragment corresponding to phase 3 is shown in Figure 3.7b, where we once again distinguish between static and dynamic variables. The j and nxt variables are temporary

for i **in** range($rows$) **do**
  for k **in** range(IA[i], IA[$i+1$]) **do**
    $j \leftarrow$ JA[k]
    $nxt \leftarrow$ IAO[j]
    AO[nxt] $\leftarrow$ A[k]
    JAO[nxt] $\leftarrow i$
    IAO[j] $\leftarrow nxt+1$

(a)

```
all i: range[c.rows] |
  all k: range[c.IA[i], c.IA[i.add[1]]] |
    let t = iter[i][k], t' = t.add[1],
        j = c.JA[k],
        nxt = iao[t][j] {
      c'.A[nxt] = c.A[k]
      c'.JA[nxt] = i
      iao[t'] = iao[t] ++ j→nxt.add[1]
    }
```

(b)



(c)

Figure 3.7 CSR transpose, phase 3: (a) pseudo-code, (b) fragment of Alloy model, and (c) commuting diagram.

local variables, and the `A` and `JA` arrays are static since their elements are set just once. The `iao` array, however, is dynamic, since the `j` element is accessed and modified in each step of the inner loop. The following boilerplate is used:

```
some iter: Int → Int → Int, iao: Int → Int → Int {
  table[{i: range[c.rows],
         k: range[c.IA[i], c.IA[i.add[1]]]},
        iter]
```

which defines `iter` from the loop bounds and gives a binding for an (intermediate) `iao` array, the only dynamic variable, as a ternary relation, since it is indexed by time. In this case, the inner loop variable `k` depends on the outer loop variable `i`, as shown in the set comprehension that builds the iteration table. To update array `iao` in the inner loop body, Alloy's relational override operator (`++`) is used.

As illustrated in Figure 3.7c, the concrete CSR transpose operation can be shown to conform to the abstract one, which we check as follows:

$$I(c) \wedge \mathit{trans}(c, c') \wedge \alpha(c, m) \wedge \alpha(c', m') \Rightarrow \mathit{trans}(m, m') \tag{3.3}$$

where the expression is universally quantified over matrices in dense (m, m') and CSR (c, c') formats.

3.5.4 CSR Matrix-Vector Multiplication

Sparse matrix-vector multiplication is a basic step in linear and eigenvalue solvers, and is therefore central to many scientific and engineering applications.

With respect to loop structure, the computation Ax is an independent series of dot products, one for each element of b, the result vector. Because incremental updates are not required in the computation, both dense and sparse algorithms can be expressed as set comprehensions, and there is little need for the tabular idiom we define.

To check conformance, however, elements of the resulting vectors must be shown to be equivalent, which effectively calls for a comparison of symbolic expressions. Instead of building general machinery for doing so,[5] however, we take a lightweight approach and model an ordered sum of products as a relation.

The `SumProd` signature defines this particular kind of symbolic expression as a ternary relation of integers and value pairs, as shown below and illustrated in Figure 3.8a.

```
sig SumProd { vals: Int → lone Value → Value }
```

In the `vals` relation, each pair of values represents a product of scalar values, and the entire relation defines the sum of the products. An index associated with each pair corresponds to its position in the associated input vectors.

[5] See, for instance, the work of Siegel et al. [72], who build symbolic expression tables for checking real, IEEE, and Herbrand equivalence of general symbolic expressions using the Spin model checker.

j	p	q
0	$A_{i,0}$	x_0
1	$A_{i,1}$	x_1
2	$A_{i,2}$	x_2
⋮	⋮	⋮

(a)

Dot product $b[i]$ for dense storage, matrix m, sequence x:

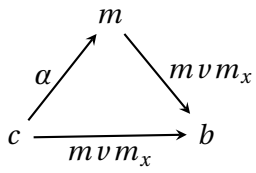
```
{ j: Int, p, q: Value-Zero |
    j in range[m.cols] and p = m.vals[i][j]
      and q = x[j] }
```

(b)

Dot product $b[i]$ for CSR storage, matrix c, sequence x:

```
{ j: Int, p, q: Value-Zero |
    some k: range[c.IA[i], c.IA[i.add[1]]] |
      j = c.JA[k] and p = c.A[k]
        and q = x[c.JA[k]] }
```

(c)



(d)

Figure 3.8 Matrix-vector multiplication: (a) sum of products for row i of matrix A and vector x, (b) dense dot product in Alloy, (c) CSR dot product in Alloy, and (d) commuting diagram.

To model matrix-vector multiplication, then, the result vector b is represented as a sequence of `SumProds`. A basic step in the algorithm that computes dot products is shown Figs. 3.8b and 3.8c for matrices in dense and CSR formats, respectively. In both cases, each product pair `p-q` is comprised of non-zero values (`Value-Zero`) to facilitate a comparison of expressions.

As illustrated in Figure 3.8d, the concrete CSR matrix-vector multiplication operation can be shown to conform to the abstract one, which we check as follows:

$$I(c) \wedge mvm(c,x,b) \wedge \alpha(c,m) \Rightarrow mvm(m,x,b) \tag{3.4}$$

where the expression is universally quantified over matrices in dense (m) and CSR (c) formats, and input (x) and result (b) vectors.

3.6 Discussion

To perform the analyses, Alloy provides a number of SAT solvers. For simulation, we use MiniSat [73], an incremental SAT solver from Chalmers University of Technology, Sweden, and for checking, Lingeling [16] from Johannes Kepler University, Austria. All experiments are performed on a 3.5-GHz-Intel-Core-i7 desktop computer.

By default, Alloy uses a scope of size 3 for signatures and a bitwidth of 4 for integers (i.e., from −8 to 7, inclusive). For values, that means two distinct non-zero values and zero. For matrices, which have integer indices, that means a size of 7×7 for dense storage, for instance.

When using default scopes, simulations in Alloy are produced instantaneously, as are counterexamples when checking assertions, indicating, for instance, array referencing and indirection problems; matrices as small as 2×2 and smaller are typically sufficient. For successful checks, most

are completed in a matter of seconds or under a minute, with the longest being the refinement check for CSR transpose, which takes a couple of hours. In practice, we often use smaller matrix sizes and larger numbers of distinct values.

In terms of limitations of the approach, because only safety is being checked, operations can "do nothing" and still be considered correct, e.g., as a result of inadvertent overconstraint. Ensuring liveness with Alloy is more difficult because the form of the check requires an unbounded universal quantification over states, as in the following:

$$I(c) \Rightarrow \exists\, c' \mid trans(c, c') \tag{3.5}$$

which asserts that every CSR matrix c satisfying its invariant has a transpose. In practice, the applicability of operations can be spot-checked using Alloy's simulator and, for small scopes, generator axioms [46] can be used to populate terms in the poststate.

Because correctness is based on conformance with abstract operations, which serve as a specification, validation is particularly important. Beyond just simulation, we find it helpful to check properties of those operations that should hold. For example, the abstract transpose operation is functional and deterministic, eliciting the following check:

$$I(m) \wedge trans(m, m') \wedge trans(m, m'') \Rightarrow equ(m', m'') \tag{3.6}$$

When first defining the operation, we inadvertently swapped row and column sizes in the poststate, resulting in nondeterminism that was detected in this manner.

3.7 Related Work

Earlier we cite the studies of Arnold et al. [7] and Kotlyar et al. [51], both of which have compilation of sparse matrix codes as their primary objective. Beyond program synthesis, Arnold et al. also take up verification because of its potential role in discovering new formats via inductive synthesis. They write: "We are not aware of previous work on verifying full functional correctness of sparse matrix codes. We are not even aware of work that verified their memory safety without explicitly provided loop invariants."

To verify sparse matrix codes, Arnold et al. design a "little language" (LL) that can be used to specify programs as sequences of high-level transformations on lists. The models are then translated automatically into Isabelle/HOL for verification. The authors verify sparse matrix-vector multiplication operations on jagged diagonals (JAD), coordinate (COO), and sparse CSR (SCSR) formats.

In quantifying proof rule reuse, they find that "on average, fewer than 19% of rules used by a particular format are specific to this format, while over 66% of these rules are used by at least three additional formats, ..." They note, however, that format-specific rules are harder to prove, and believe they can be refactored to increase reuse and improve automation.

3.8 Conclusions

We describe a state-based approach for reasoning about the structure and behavior of sparse matrices. Though declarative, the models resemble imperative programs, sharing basic elements like array indirection and loop structure, with the latter made possible by a new idiom for stateful algorithms. Concrete invariants developed and checked are also directly usable for implementations in conventional languages. The study can be viewed, in a way, as an evaluation of state-based formal methods in the context of scientific computing.

The experience has been positive for our group—to the extent that we now use Alloy to spot check the kinds of numerical codes we work with and develop, both in Fortran and C++. It is straightforward to extract program fragments, model them in Alloy, and check a property of interest. Although we have yet to find bugs in existing code, we have found errors in documentation: misstated or at best ambiguous properties that do not hold in software.

Work with Alloy is proceeding in two major directions: applications and tool support. On the latter, we are developing a framework for sharing and visualizing Alloy instances that includes support for domain-specific customization, with spatial layouts that can accommodate planar embeddings of finite element meshes and matrices of various dimensions. We also imagine but have not implemented a layer of syntactic sugar that could reduce some of the boilerplate needed to express bounded iteration. With respect to applications, we are looking at more complex sparse matrix formats and parallelization, adding meshing and assembly concerns for hybrid and element-by-element solvers, and incorporating moving patches [4] and other types of adaptivity in finite element meshes.

3.9 Acknowledgments

This work was supported at NCSU by U.S. DOE's Consolidated Innovative Nuclear Research Program through a project titled "Development and Application of a Data-Driven Methodology for Validation of Risk Informed Safety Margin Characterization Model" under grant DE-NE0008530, and at NCAR by the National Science Foundation.

CHAPTER

4

STERLING: A WEB-BASED INTERFACE FOR ALLOY

4.1 Introduction

Model finding tools like Alloy [46] enable a lightweight approach to design and reasoning about complex software systems. Such tools provide push-button analysis for both checking assertions within bounded scopes, and for generating instances that satisfy a property of interest. An attractive feature of Alloy is the immediate feedback provided by visualizations, allowing users to inspect instances and counterexamples in order to identify design problems. The ability to communicate visual information *intuitively* therefore plays a key role in determining the effectiveness of interactions with the user [34].

The Alloy Analyzer includes a visualizer that can display an instance as a directed graph, and basic properties of the graph such as labels, visibility, color, and shape can be customized manually or by using themes. This customizability, in combination with interactive features such as the ability to manually reposition graph nodes, allows the user to edit the initial visual representation created by Alloy so that they can better understand the instance. Despite these features, certain limitations of the visualizer make it difficult to use as instances increase in size and complexity. In particular, our experience using Alloy in the field of scientific computing has highlighted the need for a visualization approach capable of expressing *spatial* relationships—not just topological ones—and maintaining *consistency* in those relationships when dynamic updates occur. This chapter describes a web-based visualization interface for Alloy called Sterling, whose development has been motivated by a need for more expressive visualizations in the context of modeling scientific software. Not just limited to

the scientific domain, broader discussions among participants at the 2018 Workshop on the Future of Alloy reinforced the need for a more flexible and versatile approach.

This chapter is organized as follows. In Section 4.2 we demonstrate to the reader why a new visualization approach is needed and how Sterling fills this role. To do so, we introduce both Alloy and Sterling visualizations side-by-side while developing a simple model. By building the model in a series of incremental additions, we demonstrate how shortcomings of the existing visualizations become more apparent as the model becomes more complex, and how Sterling supports the iterative process by addressing these shortcomings. Furthermore, we present relevant details of the Alloy language, where appropriate, for readers who are not familiar with the Alloy language. We then proceed to describe the technical details of the Sterling implementation in the following two sections. In Section 4.3 we describe in some detail the Alloy Visualizer and discuss approaches in the literature that identify and address its limitations. These approaches help to shape our own design goals, outlined in Section 4.4. We then present implementation details for Sterling, discussing how implementation decisions were driven by our design goals, and how they address limitations of the existing Alloy visualizations.

4.2 Model Building in Alloy With Sterling

This section describes the incremental construction of a small model, with a special focus on the visual feedback provided by Alloy and supported by Sterling. The example chosen should be familiar to most readers, particularly those with a background in science or engineering: a two-dimensional matrix representation and associated operations. We approach the design of this model with the intent that it will subsequently be used to model and verify sparse matrix formats—those that attempt to optimize storage and performance by removing zeroes—as we have done in Chapter 3. Indeed, the steps taken and the instances displayed here reflect our experience developing those models.

In the second chapter of Software Abstractions [46], the de facto Alloy reference written by its creator, Daniel Jackson introduces Alloy in a “Whirlwind Tour”, describing the incremental construction and analysis of an email client’s address book. This tour gives readers a good sense for what it is like to explore software design using Alloy without delving into the details of the language. In this section we mirror Jackson’s approach, introducing Sterling alongside Alloy.

By developing the model in a series of incremental additions, we demonstrate not only the lightweight and iterative approach that is typical of Alloy, but also the complexity that can arise in the visual feedback it produces in some scenarios, even on simple models. In doing so, we give the reader a sense of what it is like to incrementally build a model, and how the enhanced visual feedback provided by Sterling further facilitates this iterative approach that is so important in model building. While elements may seem simple at the beginning, this initial effort lays the foundation for the analysis of more complex artifacts in engineering and scientific domains.

To familiarize the reader with Alloy and Sterling we first introduce the user interface of each in

Section 4.2.1. We then start with a model of matrix structure in Section 4.2.2. We introduce basic concepts of the Alloy language, discuss how visualizations fit into the iterative design process, and demonstrate how Sterling can be used to create domain specific visualizations. We expand the model in Section 4.2.3 by introducing dynamic behavior, showing how to model an update operation and demonstrating how Sterling addresses limitations of Alloy visualizations of dynamic operations specifically. In Section 4.2.4 we use a composite design pattern to enable differentiation of zero and nonzero values and explore visualization methods for compound graphs. Finally, we introduce execution traces using a completed model of a sparse matrix format and operation in Section 4.2.5. This section demonstrates both the flexibility of the approach and the important role that Sterling plays in facilitating an iterative approach to modeling.

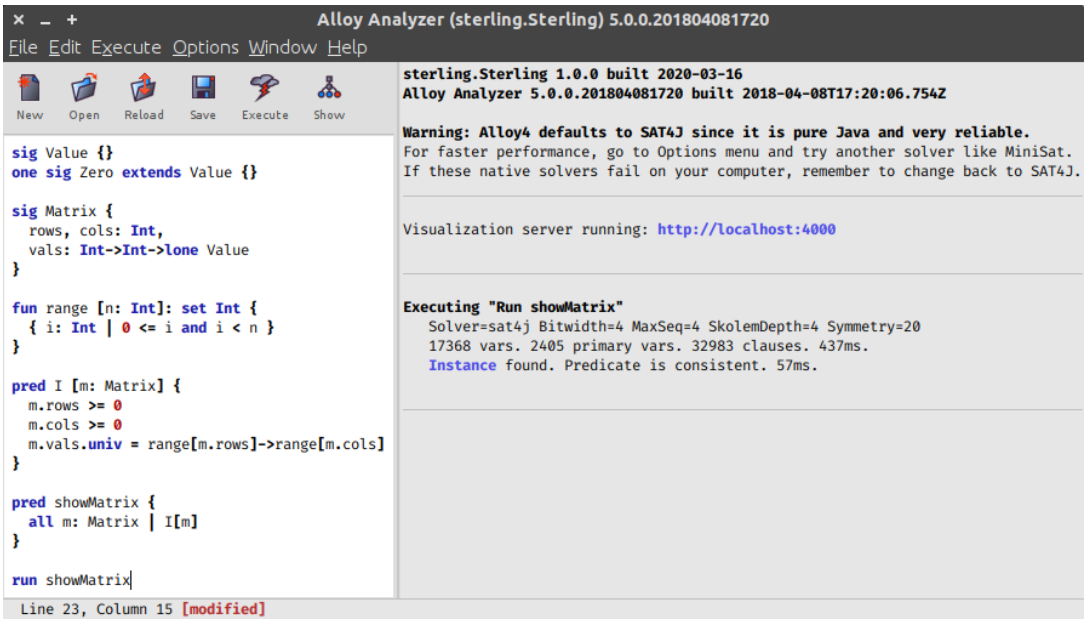
4.2.1 The Alloy and Sterling User Interfaces

At the core of the lightweight and incremental spirit of Alloy is its IDE, which brings together a model editor, the Alloy Analyzer, and the Alloy Visualizer. Models are built up incrementally in the model editor, and the Alloy Analyzer is used to perform simulations and check properties along the way. Instances and counterexamples generated by the Analyzer are then displayed in the Alloy Visualizer, providing instant, visual feedback that can be used in turn to inform the continued development of the model. Sterling itself is a web-based visualizer that extends the functionality of the Alloy Visualizer. Instances and counterexamples generated by the Alloy Analyzer can be viewed in the Alloy Visualizer, in Sterling, or both. As shown in the coming sections, each has its own strengths, and so the choice of visualizer typically depends on the needs of the modeler.

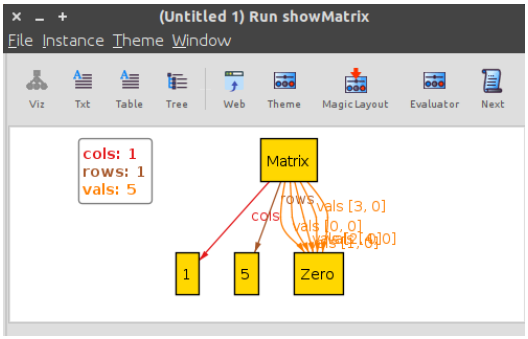
Figure 4.1a shows a complete model in the Alloy IDE. The window is split into two panes—the left pane is the model editor and the right pane displays messages generated by the Alloy Analyzer. In this example, the `run showMatrix` command has been executed, and the Analyzer has produced an instance—an assignment of values to the model's variables—that satisfies the model's constraints. The Alloy Visualizer, in Figure 4.1b, is opened automatically when an instance is found, and we can explore the instance data in any of the four views, accessible via named buttons in the Visualizer toolbar: Viz, Txt, Table, and Tree. Here, the Viz or "Graph" view displays the instance as a directed graph.

The same instance can also be visualized in Sterling's Graph View, as shown in Figure 4.1c. Graph and Table Views extend the functionality of the corresponding views in Alloy, and the Script View is used to create domain specific visualizations. Sterling can be opened from the Alloy Visualizer by clicking the "Web" button, or in a web browser by navigating to http://localhost:4000. Mirroring the Alloy Visualizer's interface, views are accessible via named buttons in the toolbar: Graph, Table, and Script.

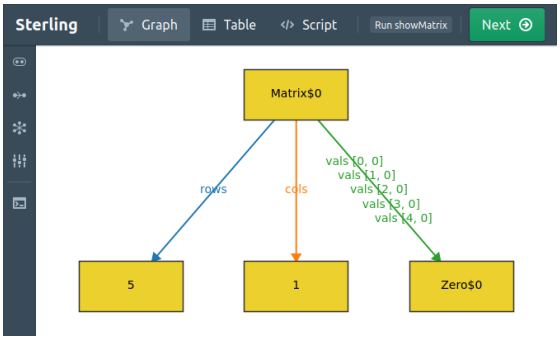
As demonstrated in upcoming sections, writing models in Alloy is an iterative process. We write a small portion of the model, ask the Analyzer to generate an instance or check some property, and then use the results of the analysis to make modifications to the model. A model like this one will have likely undergone multiple iterations of writing model code and generating instances



(a)



(b)



(c)

Figure 4.1 Components of the Alloy user interface: (a) the Alloy IDE, (b) the Alloy Visualizer, and (c) Sterling.

before reaching this point. We will also see that a small model does not necessarily produce small instances and that while the Alloy Visualizer is well-suited to display relatively small instances, its visualizations can become cumbersome and difficult to interpret as instances grow in size and models grow in complexity.

4.2.2 Structure

We begin with a model of matrix state.

```
sig Value {}
sig Matrix {
  rows, cols: Int,
  vals: Int → Int → lone Value
}
```

This model defines the complete state—the variables to which values may be assigned—that is needed to represent a matrix. It introduces two signatures, `Value` and `Matrix`. Signatures in Alloy introduce sets of objects called atoms. Atoms are the basic building blocks of our model: a `Matrix` atom represents a matrix and a `Value` atom represents a value that can reside in a matrix. Atom names never appear in Alloy models themselves, rather the Alloy Analyzer populates sets with atoms when it generates an instance or counterexample. For example, an instance of this model could be populated as follows.

$$\texttt{Value} = \{\texttt{Value\$0}\}$$
$$\texttt{Matrix} = \{\texttt{Matrix\$0}\}$$
$$\texttt{Int} = \{-4,-3,-2,-1,0,1,2,3\}$$

Where `Value$0`, `Matrix$0`, and the integers in the range $[-4,3]$ are atoms. Alloy has native support for integers, which are included in the "Int" set by default in every model. The set of integers is limited by the scope of the analysis, where the scope setting gives the maximum bit-width for integers—here it is set to 3, so the set includes integers in the range $[-4,3]$. Alloy does not support floating point values.

To introduce structure to the model, we relate atoms using fields. The `Matrix` signature has three fields: `rows`, `cols`, and `vals`. The `rows` and `cols` fields, which map matrices to integers, represent the number of rows and columns in a matrix. The `vals` field is a four-way mapping, containing the tuple `m→i→j→v` when the matrix `m` contains the value `v` at the index `(i, j)`. It is often helpful to think of a field as a table, populated by atoms, in which the order of the columns matters but the order of the rows does not. For example, using the sets defined above, a 2×2 matrix would look like the tables in Table 4.1.

One may wonder why we use a signature to represent matrix values instead of integers. Considering that this model will be used to verify sparse matrix format, we will need to be able to distinguish between zero and nonzero values. While this is possible using Alloy's integers, we note that integers

rows

Matrix	Int
Matrix$0	2

cols

Matrix	Int
Matrix$0	2

vals

Matrix	Int	Int	Value
Matrix$0	0	0	Value$0
Matrix$0	0	1	Value$0
Matrix$0	1	0	Value$0
Matrix$0	1	1	Value$0

Table 4.1 Fields of the matrix model, interpreted as tables, populated by a 2×2 matrix.

are also used in the vals field to represent matrix indices and in the rows and cols fields to represent matrix dimensions. As such, a separate signature for matrix values gives us fine grain control over the scope of possible values without the need to consider unintentional side effects that could limit the size of matrices that can be represented by the model.

Currently this model does not contain any commands, and so there is no way to perform analysis or explore states. To do the latter, we introduce a predicate—a named, reusable constraint—and a command to find an instance, an assignment of values to variables.

```
pred showMatrix {}
run showMatrix for 3
```

Executing the "run showMatrix for 3" command instructs the Alloy Analyzer to search for instances that satisfy the showMatrix predicate and will limit the number of atoms of each signature to a maximum of 3. For now the showMatrix predicate is empty so we can explore the unconstrained state space. Running the command produces the instance in Figure 4.2.

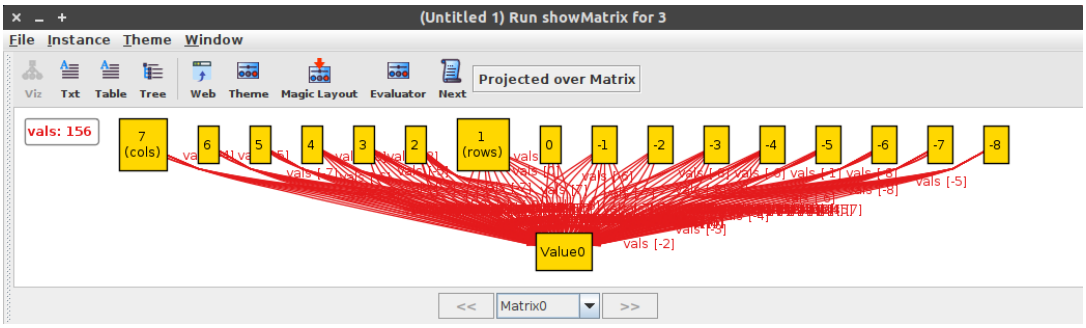


Figure 4.2 An unconstrained instance of the matrix model in the Alloy Graph View.

In the Graph View, instances are displayed as directed graphs in which graph nodes represent atoms and labelled edges connecting nodes represent the tuples of a relation. For relations with arity greater than two, such as the vals relation, the first and last atoms in a tuple are connected by an edge and the intermediate ones are displayed in brackets in the edge label.

This graph is projected over the Matrix signature, as indicated by the message in the toolbar to the right of the "Next" button. Projection is a feature of the Graph View that is used to display an

instance from the perspective of an atom or set of atoms. For each projected signature, the user chooses a single atom from that signature over which to project; here the Matrix0 atom is selected in the dropdown menu below the graph. For each relation that includes a projected signature, the column that contains the signature is moved to the front of the relation. Then, for each atom that appears in the first column, the tuples that begin with that atom are associated with the atom, but with that atom removed. By projecting over the Matrix0 atom, the vals relation includes only tuples that begin with Matrix0, but with Matrix0 removed. Under this projection, then, the vals relation is a three-way mapping (Int→Int→Value) rather than a four way mapping (Matrix→Int→Int→Value). Similarly, the rows and cols relations have been reduced from binary relations to unary ones. A unary relation is displayed as a label on each atom that belongs to the relation—the (cols) label indicates that the 7 atom is in a tuple of the cols field, and the (rows) label indicates that the 1 atom is in a tuple of the rows relation.

This graph shows that the model is underconstrained, meaning that while the instance satisfies the constraints of the model, it does not necessarily represent a realizable matrix. In this instance, the Matrix0 atom represents a 1×7 matrix but contains 156 values, as indicated by the key at the top-left of the graph which shows the size of vals relation. Furthermore, edges connecting negative integers to the Value0 atom, such as the edge labeled "vals[-5]" connecting the -8 atom to Value0, indicate that negative integers are used for matrix indices.

Because we have not constrained the model, the Alloy Analyzer is free to assign any correctly typed values to the model variables. Clicking the "Next" button in the toolbar instructs the Alloy Analyzer to generate another instance with a different set of values, and doing so generates many similarly unconstrained instances. The particular assignment of values to variables in an instance will vary depending on the Alloy preferences and is highly dependent on the solver being used.

Based on the underconstraints identified above, we introduce the following constraints to the model: (1) all matrix indices must be non-negative integers, (2) the number of rows and columns must be non-negative, and (3) the total number of values in a matrix is equal to the product of the the number of rows and columns. Rather than add the constraints to the showMatrix predicate, we take a more modular approach and create a new predicate, I (short for "Invariant"), that takes a matrix as an argument, and "invoke" it from showMatrix.

```
pred I [m: Matrix] {
  m.rows ≥ 0
  m.cols ≥ 0
  all i: m.vals.Value.Int | i ≥ 0
  all j: m.vals.Value[Int] | j ≥ 0
  #m.vals = mul[m.rows, m.cols]
}

pred showMatrix{
  all m: Matrix | I[m]
}
```

Running the command again now gives the instance in Figure 4.3. In this instance all indices are positive, the matrix has 6 rows and 1 column (as indicated by the placement of the “rows” and “cols” labels) and there are 6 tuples in the vals relation. At first glance this may appear to represent a valid matrix, but further inspection reveals that certain index pairs, such as (1, 7), fall outside the bounds of a 6x1 matrix.

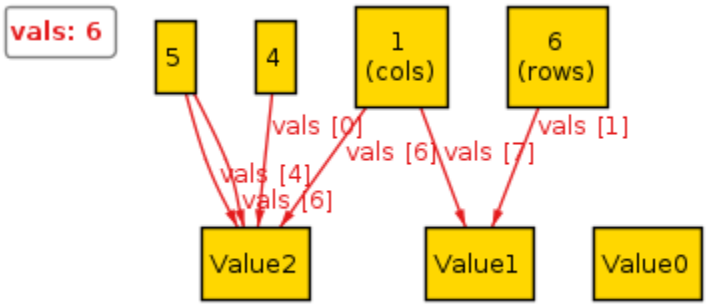


Figure 4.3 An instance of the matrix model constrained by the `I` predicate, shown in the Alloy Graph View.

Already we see certain limitations of the Alloy graph view. Even though our model is small, the first instance generated was cluttered with edges. Atoms can be manually repositioned to make the graph more readable, but their movement is restricted to the rows in which they are initially placed. Adding constraints reduced the size of the instance and made the graph more readable, as we saw in the second instance, but individual edges are still difficult to interpret. Tuples of arity higher than two, like those in the vals relation, do not have a natural visual representation in a directed graph, so their atoms are split between graph nodes and labels.

But more than that, it is not clear how the topological relationships of the graph translate to the spatial relationships of a two-dimensional matrix. We must inspect each graph edge individually to verify that the matrix indices are properly assigned; for small matrices this is easy to do, but the process becomes tedious as Alloy generates larger ones. This is a particularly common theme, we have found, when modeling scientific software. The basic character of physical and other natural processes suggests that special attention must be paid to spatial relationships, not just topological ones.

We turn to Sterling to create visualizations that more clearly express these spatial relationships. Scripts written in the Script View are executed in an JavaScript environment that contains, as global variables, the instance, a rendering stage, and access to a large ecosystem of JavaScript libraries. The script in Figure 4.4 displays the tuples of the vals relation as labelled squares. The position of each square is determined by the index pair and the label corresponds to the Value atom. The bounds of the matrix are displayed as a bolded rectangle.

In the image generated by the script, the spatial relationships are immediately clear, revealing that only a single tuple in vals represents a valid value in a 6x1 matrix. When stepping through

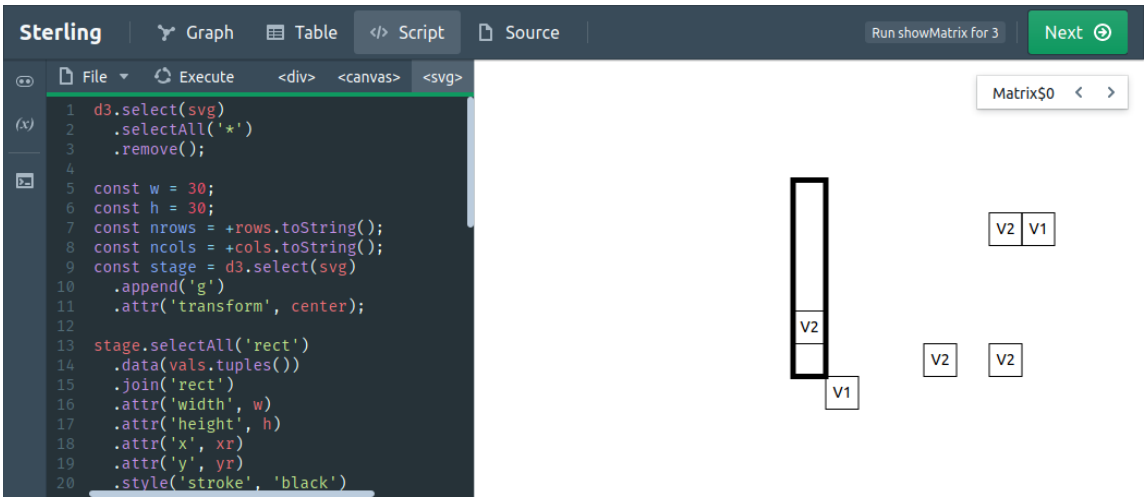


Figure 4.4 An instance of the matrix model constrained by the `I` predicate, shown in the Sterling Script View.

instances by clicking the Next button, the script is automatically rerun and a new image is generated each time; this shows many instances in which values are not placed within the bounds of the matrix.

So we rewrite the I predicate to constrain the set of integer pairs that appear in a matrix based on its dimensions. In this way, we specify the total number of values and the allowable index pairs using a single expression.

```
pred I [m: Matrix] {
  m.rows ≥ 0
  m.cols ≥ 0
  m.vals.univ = range[m.rows]→range[m.cols]
}

-- the set of integers such that for every integer i, 0 ≤ i and i < n
fun range [n: Int]: set Int {
  { i: Int | 0 ≤ i and i < n }
}
```

Alloy functions are named expressions; here we introduce the range function which generates a set of non-negative integers within a specified range. The expression range[m.rows]→range[m.cols], using the arrow product, creates a set of tuples that contains every combination of row and column index values. We rerun the command, generating and exploring instances to visually verify that the new constraints describe valid matrices. In Figure 4.5 we compare Alloy visualizations in the left column with the corresponding Sterling visualizations in the right column.

This sequence of images conveys the role of Sterling in the iterative design process. If we imagine stepping through instances with only Alloy visualizations, we see that significant effort is required to interpret the display and look for design errors. The Sterling visualizations, on the other hand,

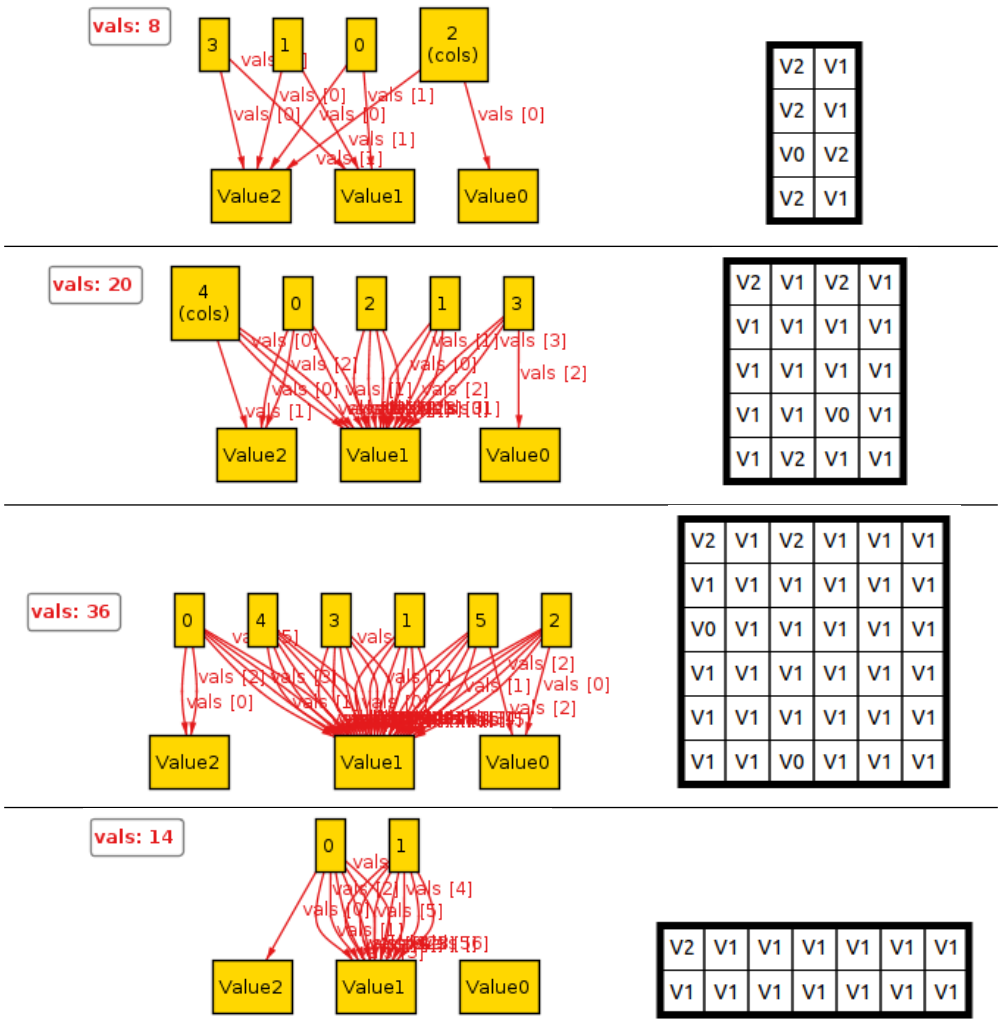


Figure 4.5 A variety of matrix instances displayed in the Alloy Graph View (left) and the Sterling Script View (right).

match the structure of a matrix and so design errors are highlighted more effectively. Not only that, but because interpretation is quicker, we can explore more instances and are therefore more likely to encounter one that exposes a case we may not have considered.

As the model continues to grow, so will the visualization script. In practice we have found that model building and script building complement each other well; the upfront cost of updating a script when a model has changed tends to be offset by the speed with which visualizations are interpreted and the number of instances that can be explored.

4.2.3 Behavior

We continue building our model by describing an update operation.

```
pred update [m, m': Matrix, i, j: Int, v: Value] {
  let u = m.vals[i][j] |
    m'.vals = m.vals - i→j→u + i→j→v
}
```

The update predicate, like the I predicate, defines a constraint. In this case, however, the constraint describes dynamic behavior. Its arguments are a matrix before the update (m), the matrix after the update (m'), the row and column indices that are updated (i, j), and the new value (v). The constraint states that the set of values in the matrix after the update occurs is equal to the set before the update with the value at location (i, j) replaced by the new value, v.

This method of describing an operation is declarative and so there is no explicit mutation. Rather, two matrices are created and the effect of the operation is captured by relating them. The Alloy Analyzer, then, can check whether an operation is valid by comparing the before and after states. Contrast this with an imperative language, in which a procedure is operational and describes how to produce changes by modifying state.

To simulate the update operation we could run the command "run update for 2" and Alloy would generate an instance that satisfies the update predicate. We note, however, that the I predicate is not included as a constraint and so the operation could be applied to an invalid matrix. Rather than add I to the update predicate itself, we instead create a new predicate. This is a more modular approach that creates a clear distinction between behavioral and stateful constraints.

```
pred showUpdate {
  some m, m': Matrix, i, j: Int, v: Value |
    I[m] and update[m, m', i, j, v]
}
run showUpdate for 3 but 2 Matrix
```

The showUpdate predicate "invokes" the update predicate and uses the I predicate to ensure that the matrix representing the initial state is valid. Matrix scope for the showUpdate command is limited to two, for the pre-state (the state of the matrix before the update operation) and the post-state (the state of the matrix after the update operation).

Figure 4.6 compares the Alloy (left column) and Sterling (right column) Graph Views of the instance generated by this command. Each column shows the pre-state above and the post-state below. Examining the graph, we can determine which value in the matrix has been updated. The labels $showUpdate_i, $showUpdate_j, and $showUpdate_v indicate which atoms are used as arguments to the update predicate. Comparing the edge sets we see that the edge "vals[0]" connects the "0" atom to the "Value$0" atom in the pre-state but connects it to the "Value$1" atom in the post-state. So, in this instance, the matrix has been updated with the value "Value$1" at index (0, 0).

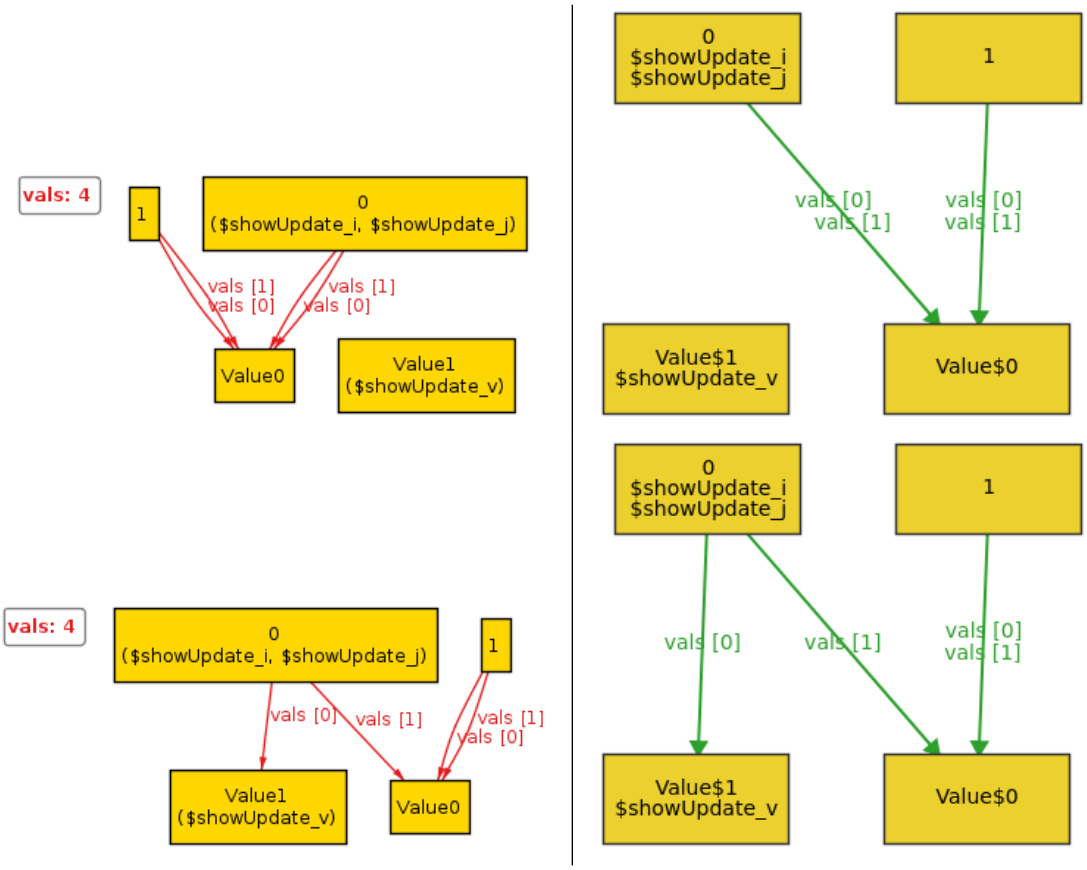


Figure 4.6 A dynamic operation displayed in the Alloy (left) and Sterling (right) Graph Views. In each, the pre-state is above the post-state.

This comparison of the Alloy and Sterling graphs highlights differences in the positions of graphical elements. In both Alloy and Sterling, only a single state is displayed at a time, and the user can manually toggle back and forth between states to make comparisons. In Alloy (left column), the graph layout for each state is calculated independently to minimize edge crossing, and so the positions of atoms are not guaranteed to be consistent when toggling back and forth. We see that this is the case here—the "0" and "1" atoms have switched positions, as have the "Value0" and "Value1" atoms. Compare this with Sterling (right column), where the layout of atoms is consistent between

the two states. When toggling back and forth, then, differences between the two are highlighted.

The graph view is an indispensable tool for interpreting Alloy instances, particularly in the early stages of developing a model when instances tend to be relatively small. As dynamic operations are introduced, especially those that span many states, spatial inconsistencies in the Alloy Graph View can make instances difficult to interpret. And while the Sterling Graph View addresses this shortcoming, among others, the directed graph representation can become difficult to use as the model grows, and a more appropriate visual abstraction may be more useful, just as we saw in SECTION(statics).

In fact, we can visualize the update operation using the script from SECTION by projecting over the Matrix signature in the Sterling Script View. In doing so, we can toggle back and forth between the pre- and post-state matrices, shown in Figure 4.7.

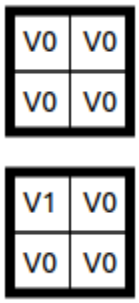


Figure 4.7 A dynamic operation displayed in the Script View (pre-state above, post-state below).

This instance shows a valid matrix update, but it is entirely possible that the Alloy Analyzer happened to choose an instance of a valid update operation despite the model being underconstrained. Rather than stepping through instances to manually look for design errors, we make an assertion about how the update operation behaves.

```
assert updateValid {
  all m, m': Matrix, i, j: Int, v: Value |
    I[m] and update[m, m', i, j, v] ⇒ I[m']
}
```

An *assertion* is a constraint that is intended to be valid. In other words, it is a constraint that we expect to hold true in all cases. This one states that if the update operation is applied to a valid matrix, the resulting matrix will also be valid. To check the assertion, we issue the following command to the Analyzer:

```
check updateValid for 3 but 2 Matrix
```

This instructs the Analyzer to search for a counterexample—a scenario in which the assertion is violated—and indeed it finds one, shown in Figure 4.8. Again we show the Alloy visualization in the left column and the Sterling script-generated visualization in the right column.

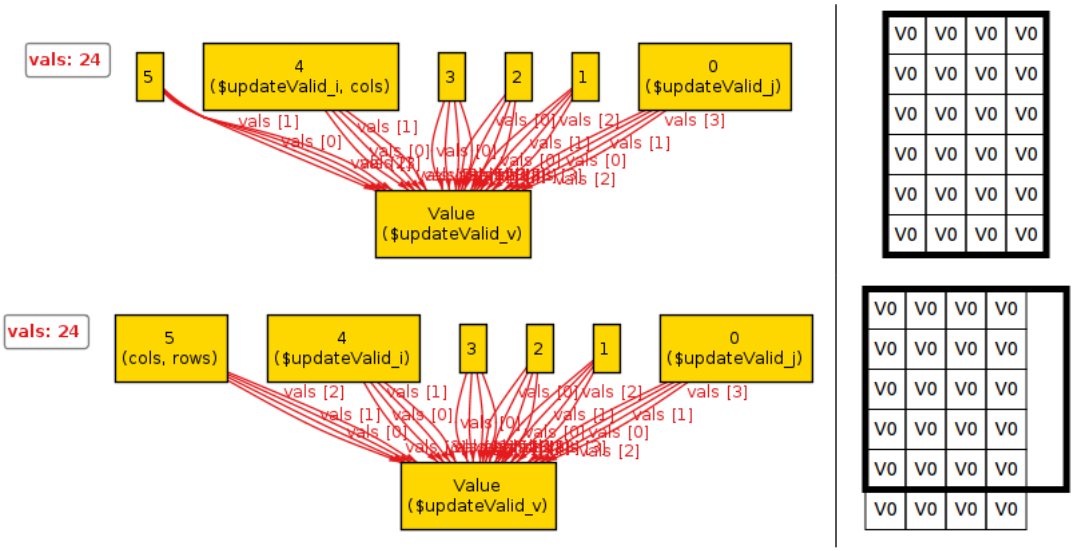


Figure 4.8 A counterexample displayed in the Alloy Graph View (left) and the Sterling Script View (right). In each, the pre-state is above the post-state.

Just as in SECTION(statics), this comparison demonstrates the importance of expressing *spatial* properties embedded throughout the model, not just topological ones. In this case, the Graph View gives little insight into how the model is underconstrained, but the Script View clearly shows that the dimensions of the matrix have changed. Because we don't explicitly state in the update predicate that the dimensions of the matrix do not change as a result of the operation, the Analyzer is free to do exactly that. So we update the predicate to include this constraint.

```
pred update [m, m': Matrix, i, j: Int, v: Value] {
  let u = m.vals[i][j] |
    m'.vals = m.vals - i→j→u + i→j→v
  m'.rows = m.rows
  m'.cols = m.cols
}
```

Executing the check finds another counterexample, shown in Figure 4.9.

Again the Script View shows what the Graph View cannot: that the indices of the update operation fall outside the bounds of the matrix. Note that the Graph View is so cluttered with edges that it isn't possible to see the addition of a tuple to the vals relation. Again we update the predicate with an additional constraint.

```
pred update [m, m': Matrix, i, j: Int, v: Value] {
  i in range[m.rows]
  j in range[m.cols]
  let u = m.vals[i][j] |
    m'.vals = m.vals - i→j→u + i→j→v
  m'.rows = m.rows
  m'.cols = m.cols
```

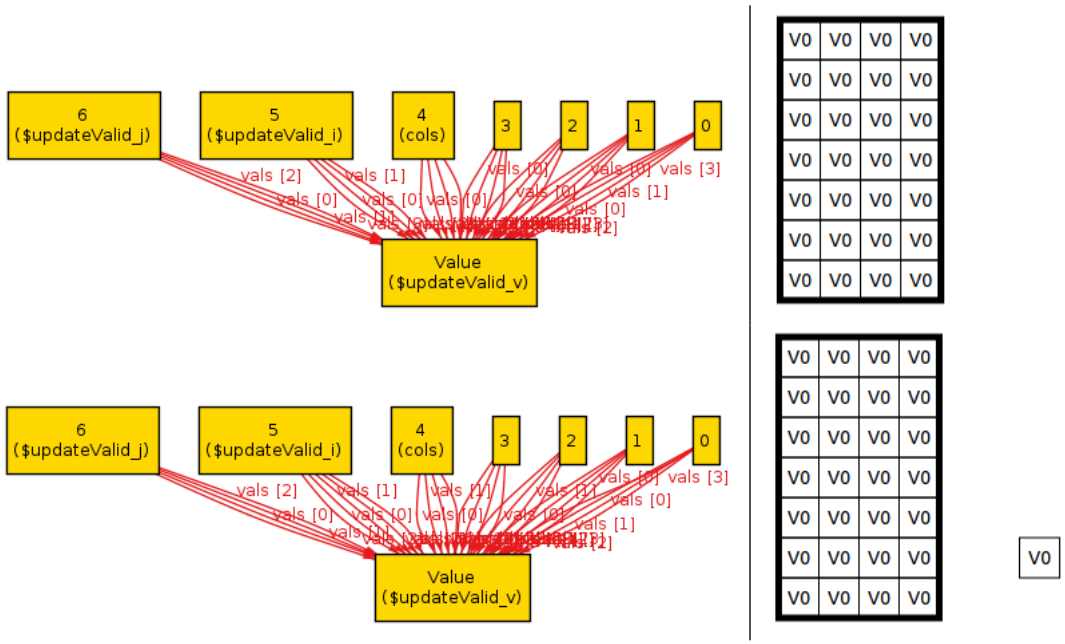


Figure 4.9 A counterexample displayed in the Alloy Graph View (left) and the Sterling Script View (right). In each, the pre-state is above the post-state.

}

Executing the check now finds no counterexample. The assertion may still be invalid, though, since the Analyzer only considered matrices up to size 7x7 containing at most 3 different values. So we increase the scope to check more cases. There's no point considering more than two matrices, but we allow 10 values and larger matrices by increasing the Integer scope explicitly.

```
check updateValid for 10 but 2 Matrix, 6 Int
```

Executing this command takes a bit longer, just under a minute on a desktop computer, as there are more cases to consider. By increasing the Int scope to 6, we've instructed the Analyzer to use integers in the range [-32, 31]. This check, then, has confirmed that for all possible matrices up to size 31x31, the update operation is valid. Not only that, but for every matrix size checked, it has considered every possible way to populate the matrix with up to 10 different values. Given that for the 31x31 matrix alone there are 10^{961} possible combinations, we can begin to see why this kind of analysis is more effective than testing. The Analyzer does not, of course, construct and test each case individually, but instead relies on pruning the tree of possibilities to rule out large subspaces without examining them fully. And while we have not proven that the assertion is valid, our intuition tells us that it is unlikely for a problem to exist that cannot be shown using a matrix size up to 31x31.

Now that the Analyzer finds no counterexamples we may wish to visually verify our model once more. As a final update to our script, we now highlight the value that is updated so that we can more easily see the effect of the operation. The sequence of images in Figure 4.10 shows two instances. In each we've included the pre- and post-states in Alloy (left), the Sterling Graph View (center), and the

Sterling Script View (right).

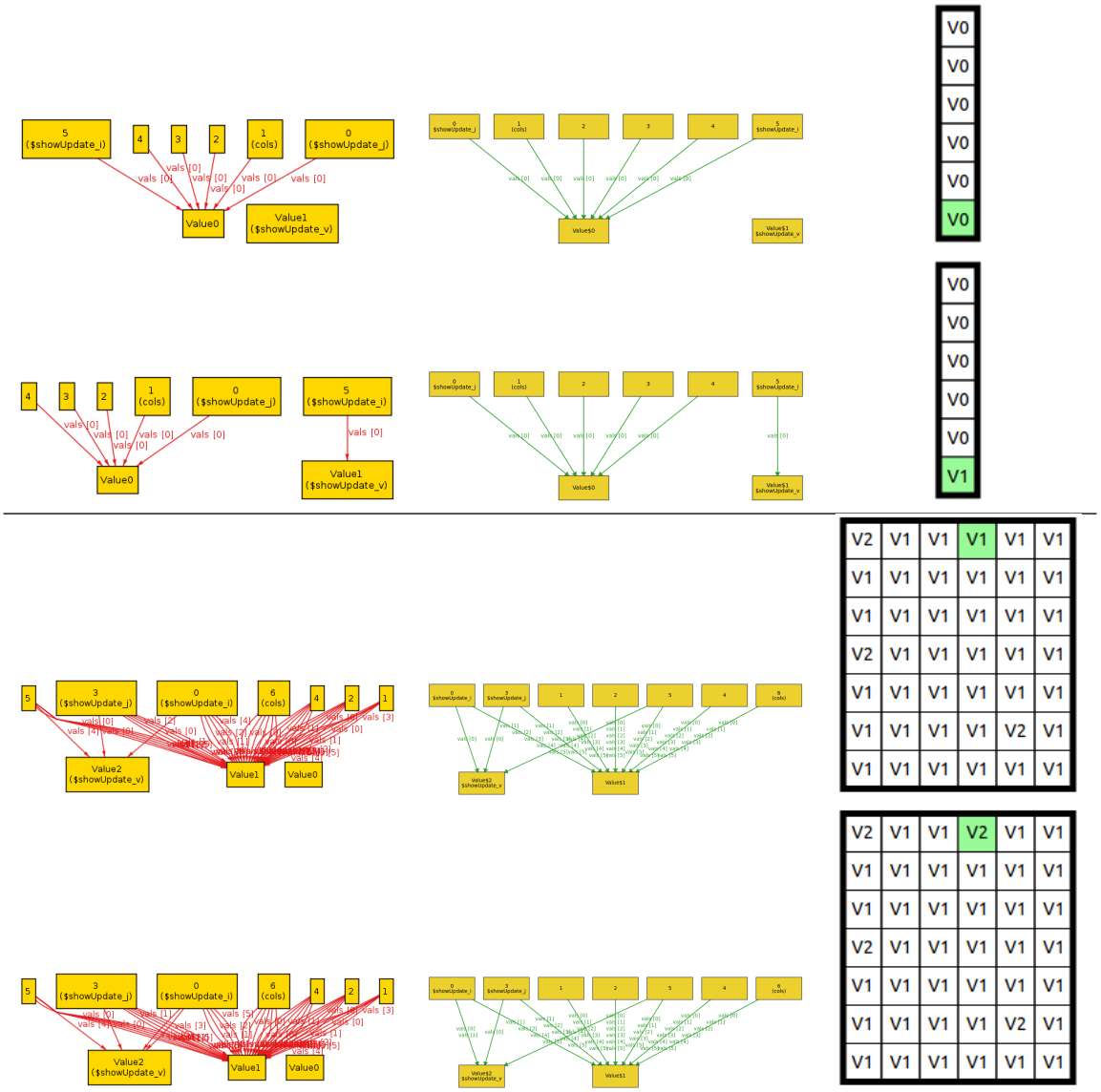


Figure 4.10 A comparison of the Alloy Graph View (left), the Sterling Graph View (center), and the Sterling Script View (right). Two instances are displayed, each with the pre-state of the update operation above and the post-state below.

4.2.4 Classification Hierarchy

With the matrix structure and update operation modeled, we now consider matrix sparsity. The intent is to model a sparse matrix format and to use the model we just created to verify that the implementation is correct. To perform verification, we use data refinement to show that the more

detailed concrete sparse matrix format can simulate the more abstract dense matrix representation. To show that a dense matrix and sparse matrix representation are equivalent, we will of course need to be able to distinguish between zero and nonzero values. In this section we demonstrate how to expand the model to support this requirement and discuss the visualization challenges that are introduced in turn.

Currently we use the Value signature to represent any numerical value in a matrix, zero included. As it stands, we could simply interpret a specific Value atom, perhaps the one labelled Value$0, as zero when interpreting an individual instance. This approach, however, gives us no way to distinguish zero from nonzero values in the model proper, and so we cannot guarantee that zero will not be included in the models of our sparse formats. So instead, we extend Value to introduce the Zero signature.

```
sig Value {}
one sig Zero extends Value {}

sig Matrix {
  rows, cols: Int,
  vals: Int → Int → lone Value
}
```

By using the extends keyword, we introduce Zero as a subset of Value. This means that Zero is a Value, and can be used anywhere a Value is used. The keyword one indicates that there is exactly one atom in the Zero set. So, depending on the scope, the complete Value set will look like

```
{Zero,Value0,Value1,Value2}
```

Figure 4.11 shows a model diagram, a graphical representation of the model's declarations, generated by the Alloy Analyzer. Note that the vals field still maps Matrices to Values, and that Zero is an extension of Value.

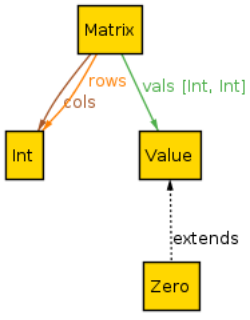


Figure 4.11 A model diagram depicting the matrix model.

Now we can rerun the commands in our model to check assertions and generate instances that include Zero in our matrices. Executing the “check updateValid” command reveals that there are still no counterexamples, and executing the “run showMatrix” command gives us the instance in Figure 4.12, a 2x2 matrix that contains two zeros and two nonzero values.

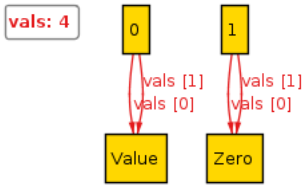


Figure 4.12 An instance of the matrix model with zero and nonzero values.

The Graph View, as we have seen, displays the topological relationships created by our model’s fields. A node is created for each atom, and arcs are drawn for each tuple connecting the nodes corresponding to the first and last atoms in the tuple. The graph does not, however, express the hierarchical relationships of the model. There is no indication that the Value atom and the Zero atom belong to the same set, likewise for the 0 and 1 atoms. Certain methods can help express these relationships, such as node coloring and labelling, but they tend to introduce more clutter than clarity when there are multiple levels of extensions.

In Software Abstractions, Jackson describes a snapshot view, in which the graph is drawn as above, but with the addition of labelled contours surrounding sets. These contours can express the hierarchical relationships embedded within a model, even with many levels of subsets, but as Jackson notes, the Alloy Visualizer is incapable of producing them.

Not just useful for creating domain-specific visualizations, the Sterling Script View can likewise be used to create these snapshot views. To do so, we leverage Sterling’s access to the npm package manager, which at the time of writing contains over 1.3 million open source libraries; many are well-established libraries that can be used to quickly create visualizations in the Script View. In FIG we have imported Cytoscape.js [33], a graph theory library for visualization and analysis, and it is included in the list of variables that are globally available in the scripting environment (left). Cytoscape.js natively supports rendering compound graphs—graphs with embedded hierarchical structure—using a variety of layout algorithms. The short script (center) in Figure 4.13 uses this library to generate the snapshot (right).

Now we can see both the topological and hierarchical relationships of our instance. The Zero$0 atom is in the “this/Zero” set, which itself is in the “this/Value” set, as indicated by the labeled contour lines surrounding each.

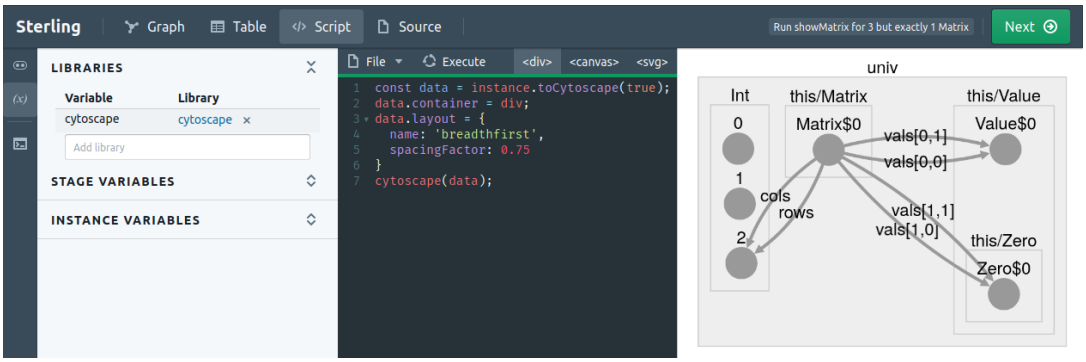


Figure 4.13 An instance of the matrix model displayed as a snapshot using the Cytoscape.js library in the Sterling Script View.

$$\begin{pmatrix} 4 & 0 & 0 & 0 \\ 0 & 8 & 15 & 0 \\ 0 & 0 & 16 & 0 \\ 23 & 0 & 0 & 42 \end{pmatrix}$$

IA	0	1	3	4	6	
JA	0	1	2	2	0	3
A	4	8	15	16	23	42

Figure 4.14 A dense matrix (left) in the CSR format (right).

4.2.5 Execution Traces

Now that we are able to distinguish zero from nonzero values, we could begin developing a model of a sparse format. However, the intent here is not to delve into the details of sparse formats and how they are modeled in Alloy; rather, we give the reader insight into what it is like to work with a model that has been through dozens of iterations of modifications that add detail and model complex operations. In doing so we demonstrate the challenges associated with interpreting the instances and counterexamples generated by these models and how Sterling is used to address these challenges. The model we present is the matrix transpose operation for the Compressed Sparse Row (CSR) format. Before discussing instance visualization for this model we first present relevant details of the CSR format and the model itself.

The CSR format is composed of three one-dimensional arrays. The A and JA arrays store nonzero values and column indices, respectively. The IA array is an indexing array that contains the starting location within the A and JA arrays of each matrix row. For example, the matrix in Figure 4.14, which shows the dense matrix representation on the left and CSR on the right, contains only a single nonzero value, 4, in the first row. This value is the first entry in the A array and its column index, 0, is in the same position in the JA array. The first value in the IA array, then, is the starting index within A and JA of the values of the first matrix row, as indicated by the arrow.

The matrix transpose operation flips the values of a matrix over its diagonal. To model the operation in our dense matrix representation, we can use set comprehension to flip the row and column indices. The CSR transpose algorithm, however, is more involved. It consists of four discrete

steps, each modifying in place an array, IAO, that becomes the IA array for the transposed CSR matrix. The details of each step, which can be found in Chapter 3 along with the complete model online [1], are not important here. Rather, we focus on how the individual steps are composed in Alloy to form a trace, an entire execution involving a series of operations.

To create the transpose execution trace, we model each of the four steps as a predicate, and invoke each one individually within the transpose predicate, using the "output" of one step as the "input" for the next.

```
pred step_1 [c: CSR, iao: seq Int] { ... }
pred step_2 [iao, iao': seq Int] { ... }
pred step_3 [iao, iao': seq Int] { ... }
pred step_4 [iao, iao': seq Int] { ... }

pred transpose [c, c': CSR] {
  ...
  some iao, iao', iao'', iao''': seq Int {
    step_1[c, iao]
    step_2[iao, iao']
    step_3[iao', iao'']
    step_4[iao'', iao''']
    c'.IA = iao'''
  }
}
```

The Alloy keyword seq declares a field as a sequence of atoms. In this partial model we use seq Int, whose type is Int→Int, to represent arrays of integers. The first column of the relation contains the index of the corresponding value in the second column. Modeling arrays this way is convenient, as values can be accessed using the box join operator (e.g. iao[0] gives the first value in the iao array) and the seq module provides many other useful helper functions. The existential quantifier some binds four integer sequences, iao, iao', iao", and iao"', representing the state of the IAO array after the execution of each step. The step_1 predicate, modeling the first step of the algorithm, establishes the initial state of the IAO array by relating the variables of the initial CSR matrix, c, to the variable iao. The step_2 predicate models the second step of the algorithm by relating iao to iao', and this pattern continues for the remaining two steps. Finally, we establish that the IA array of the transposed matrix is equivalent to the IAO array after the final step, iao"'.

Having also defined a valid CSR state in an invariant predicate, I, we can now generate an instance of the CSR transpose operation.

```
pred showTranspose {
  some c, c': CSR | I[c] and transpose[c, c']
}

run showTranspose for 2 CSR, 2 Value, 7 seq
```

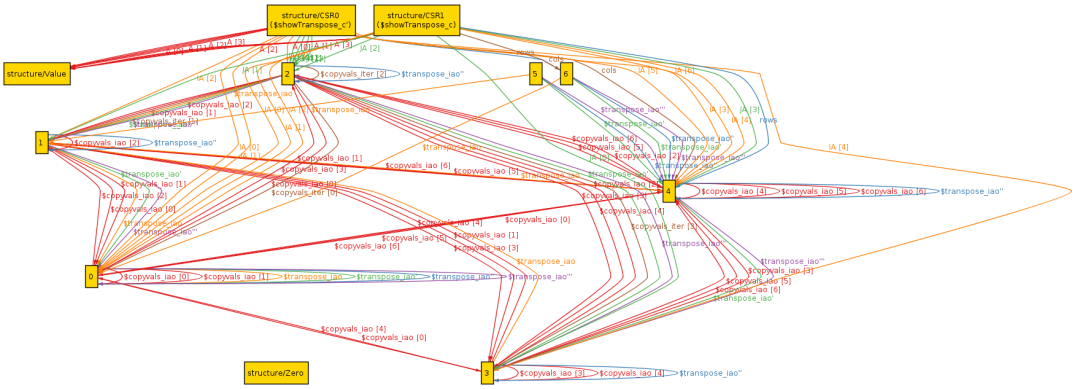


Figure 4.15 An instance of the CSR transpose operation displayed in the Alloy Graph View.

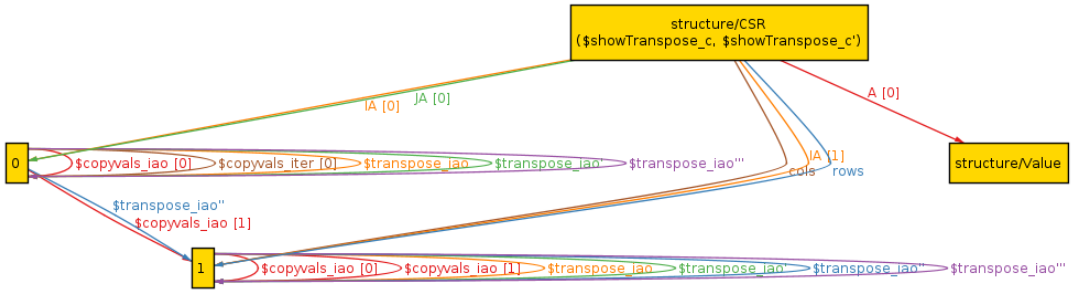


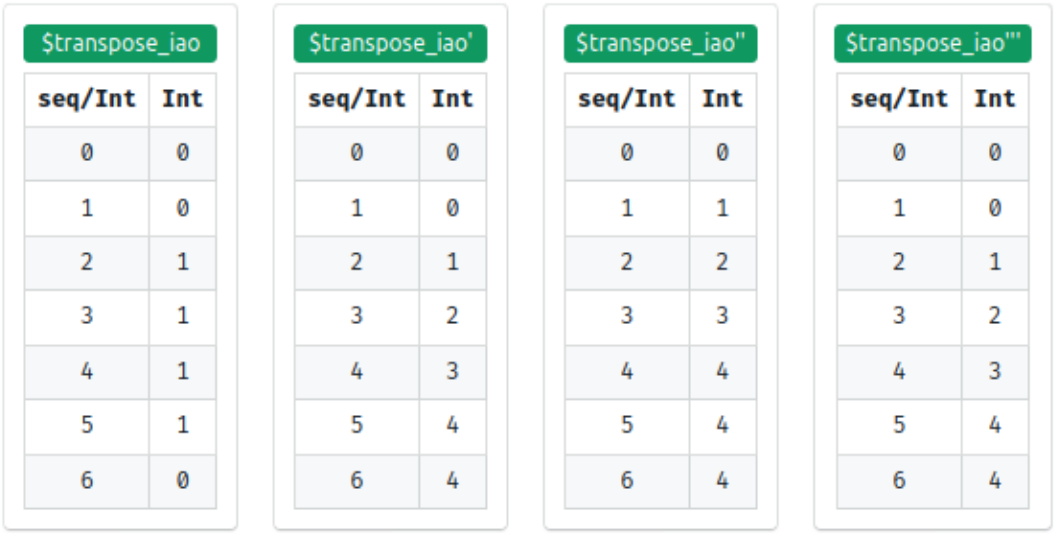
Figure 4.16 An instance of the CSR transpose operation applied to a 1 × 1 matrix, displayed in the Alloy Graph View.

In the showTranspose predicate we have instructed the Alloy Analyzer to find a pair of CSR matrices, c and c', such that c is a valid CSR matrix and the two are related through the transpose operation. Running the command produces the instance in Figure 4.15.

The directed graph representation of this instance is illegible, and there are no settings or built-in tools that make it easier to interpret. By projecting over the CSR signature we can toggle between the two matrices, but the individual graphs exhibit similar levels of complexity. Furthermore, there is no way to display the individual steps of the transpose algorithm itself. The four intermediate states of the IAO array are present in the instance as witnesses—values for quantified variables that make the body of the quantified formula true. Reading a sequence in the graph requires locating each index atom, searching for the edge label corresponding to that specific sequence, and locating the value it connects to. Even in the case of a 1x1 matrix, shown in Figure 4.16, this process proves cumbersome.

For this model, the Table View is better suited to exploring instance data. Tables more closely resemble the structure of an array, and so we can see how the values of the IAO array evolve during an execution of the transpose algorithm; the tables in Figure 4.17 show the four states of the IAO array for the instance in Figure 4.15. Note that we must use the Sterling Table View, as the Alloy one does not display witnesses.

Imagine, however, that the instance is instead a counterexample that expresses an error in the



$transpose_iao

seq/Int	Int
0	0
1	0
2	1
3	1
4	1
5	1
6	0

$transpose_iao'

seq/Int	Int
0	0
1	0
2	1
3	2
4	3
5	4
6	4

$transpose_iao''

seq/Int	Int
0	0
1	1
2	2
3	3
4	4
5	4
6	4

$transpose_iao'''

seq/Int	Int
0	0
1	0
2	1
3	2
4	3
5	4
6	4

Figure 4.17 Tables showing the execution trace of the IAO array in the CSR transpose operation.

model. There are two CSR matrices, each consisting of three arrays, and four additional arrays that represent the intermediate states of the IAO array. Considering the web of dependencies among these arrays, it is likely that the effects of an error in the model will propagate through the entire instance. Tracking down the source of the error, then, requires starting with the initial CSR matrix and manually matching the execution trace with the expected behavior of the operation. Certain subtle errors may even require that the pre- or post-state CSR matrix be converted to a dense matrix format to provide important contextual information. Indeed, we found this to be the case when writing these models.

Using the Sterling Script View, we can create a visualization that presents the relevant arrays in a well organized and structured fashion *and* provides useful contextual information. Figure 4.18 shows the same instance displayed in Figure 4.15.

The image is divided into three rows: the top row shows the initial state of the matrix, the middle row shows the complete execution trace for the transpose algorithm, and the bottom row shows the state of the matrix after the transpose operation. The initial and final states of the matrix are displayed in their CSR representations on the left and have been translated to their dense matrix representations on the right. Nonzero values are highlighted in the dense matrices to highlight the effects of the transpose operation.

Given that the script is over 150 lines long, the reader may question whether taking the time to create this visualization is worthwhile, and so we emphasize that the script as shown was not written in a single sitting. Rather it was created incrementally in a series of small additions in lockstep with the model itself. Individual additions typically included short functions, many of which are shown collapsed in the editor, that parse instance variables or render some small portion of the model. The function that renders dense matrices, for example, is the same one we used in Section 4.2.2

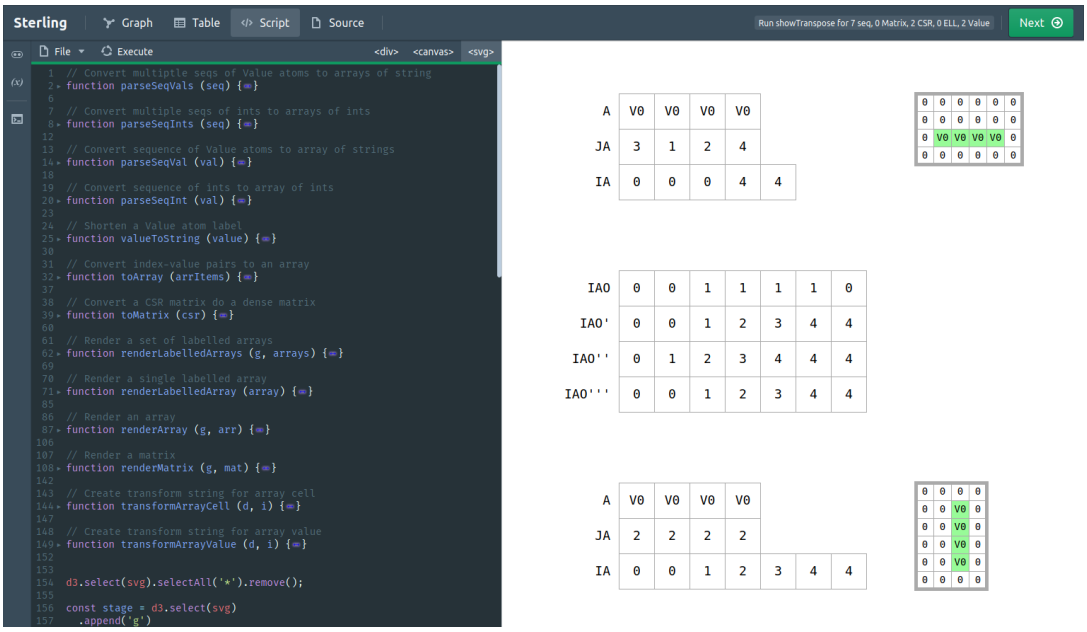


Figure 4.18 An instance of the CSR transpose operation displayed in the Sterling Script View.

with a small modification to highlight nonzero values. In building the script this way we gradually assemble a sort of "library" of functions that can be called as needed as the model changes and the visualization requirements evolve.

Complex models like this one can make an iterative approach to modeling less tractable. Compounding factors of increased analysis time and instances that are difficult to interpret result in increased time between iterations. The modeler, then, is forced to spend more time tracking down modeling errors and waiting for analyses to complete than thinking critically about the model itself. Sterling, as we have shown, addresses the issue of instance complexity by providing tools that make instances easier to interpret at all stages of model development.

4.3 Background and Related Work

Having introduced visualization in Alloy and Sterling in Section 4.2, we now give additional background on the Alloy Visualizer and discuss related work addressing the shortcomings of the visualizations it provides.

Perhaps the most frequently used of the three views in the Alloy Visualizer, the Graph View displays an instance as a directed graph in which each node represents an atom and each edge represents a tuple in a relation. To help improve readability, basic properties of the graph, such as labels, visibility, color, and node shape, can be customized manually or by using themes, and the positions of the graph elements can be manually repositioned. Relations can optionally be displayed as attribute labels, a feature particularly useful for binary relations that map to enumerations. Take, for example, a model of a traffic light system: a relation that maps lights to their current color state

could be displayed as a label such as “color: green” on the light nodes [68]. The graph view also supports *projection*, a feature most commonly used with models of dynamic systems. Since Alloy has no built-in notion of time, models involving state change often explicitly model time using higher-order relations. The graphical view of such a model, then, includes every state simultaneously. When an instance of such a model is projected over time, however, Alloy constructs a graph for each atom of the time signature, and the user is able to interactively step through snapshots of individual states in a sequence, as demonstrated in Section 4.2.3.

The Graph View presents three commonly identified limitations [28], each related to the layout. First, the layout algorithm, which automatically generates a Sugiyama-style graph [75], provides no option for customization by the user and rigidly arranges graph nodes into rows. While it is possible to interactively move nodes within a row by clicking and dragging, they cannot be moved to different rows, making it particularly difficult for the user to manually improve the layout if the calculated one is difficult to read [28, 63, 86]. Second, the graph layout is recalculated any time a new instance is generated or the projection is changed, and so the user must reinterpret the entire graph if, for example, they are stepping through state atoms in sequence [28, 65, 85]. Finally, difficulties in understanding instances that stem from limitations of the layout algorithm are compounded by the low-level nature of the directed graph representation, which can become cluttered with edges as models become more complex [19, 34].

Various approaches have been proposed to address these issues, either by extending the Alloy Visualizer or by introducing new tools. The *Magic Layout* tool, included in the Alloy Visualizer, is used to automatically infer a better initial layout and theme based on the structure of the instance itself. In doing so it addresses perceived difficulty of customization and prevents the user from having to learn the intricacies of theme development by providing an automatic method for their creation [68]. Improvements to the Magic Layout tool, such as static node positioning through projected frames, coloring to express state change, and more, have been proposed [85].

Electrum [61] is an Alloy extension, adding connectives from linear temporal logic, that is used to model and verify dynamic systems with rich configurations in bounded and unbounded scopes. It tailors the standard Alloy visualizer to the needs of dynamic systems by adding visual components to the UI; the trace graph indicates which state(s) are currently visible, and a split pane visualizer presents two states simultaneously, as shown in Figure 4.19.

The Alloy4Fun tool, a web application built primarily for use in educational an context, enables online editing and sharing of Alloy models and instances [62]. It includes a built-in visualizer that generates directed graphs using the Cytoscape.js [33] library. Customizations such as node color and shape are possible through a right-click menu, and the user can choose from multiple layout algorithms. Nodes can be manually repositioned and are not restricted to rows, and the positions of nodes are preserved between frames of projected instances. However, the visualization approach is more lightweight compared to Alloy, allowing only the most common theme customizations. Furthermore, the tool itself lacks a number of key features commonly used when writing complex models, such as the Alloy module system which allows models to be organized in a modular fashion

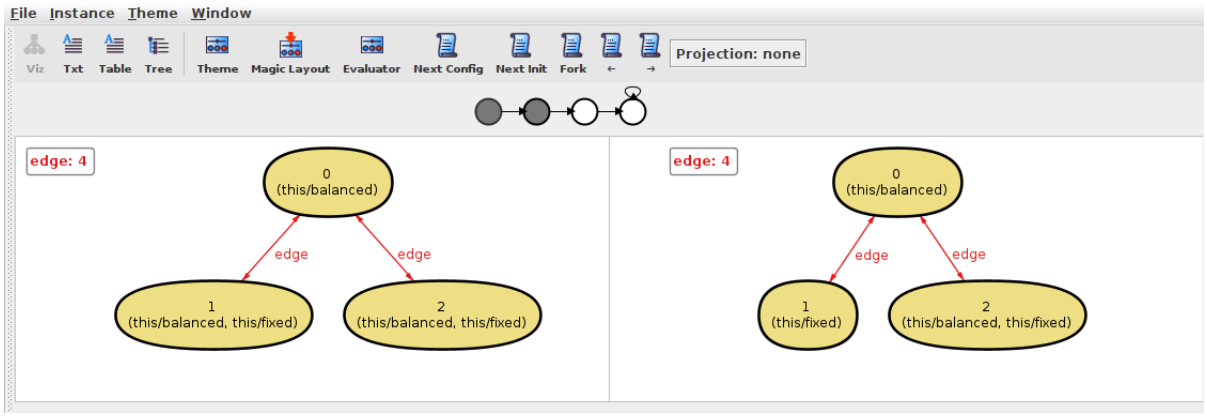


Figure 4.19 Two states of a trace in the Electrum Graph View.

among many files, and the ability to choose the particular SAT solver performing a given analysis.

In the Lightning tool [34], an Eclipse IDE plugin, users can create domain specific visualizations using a Visual Language Model, which itself is written in Alloy. The VLM contains basic visual elements such as shapes and text which can be composed or related to each other via connectors and can be arranged through the definition of layouts.

In related work in other formalisms we look at several visualization tools based on ProB [57], an animator, constraint solver, and model checker that supports many formal methods. BMotion-Web [53] is a ProB-based tool for creating interactive visualizations of B, Event-B, and CSP models based on web technologies. The learning curve for this tool is quite steep, however, and it is no longer supported by the ProB team [79]. VisB [79], the successor to BMotionStudio [52], is a plugin for ProB that enables users to visualize models in B, Event-B, Z, TLA+, and Alloy by composing SVG graphics. A kernel for Jupyter [50] allows B and Pro-B models to be executed and visualized from directly within Jupyter notebooks [35]. Outside of ProB we look to tools such as PVSio-web for model-based development of human-machine interfaces [64] in PVS.

4.4 Sterling

Our experience using Alloy in the field of scientific computing has highlighted the need for Sterling. The basic character of physical and other natural processes suggests that attention must be paid to *spatial* relationships—not just topological ones—and to *consistency* in those relationships when dynamic updates occur, as they do in problems with time-varying state. A lack of consistency in the layout of graphs provided by Alloy, then, inhibits the incremental modeling process when developing models in the scientific domain, and lack of tool support means we must resort to hand-drawn figures when the provided visualizations prove too difficult to interpret.

We have identified a set of design goals, outlined in Section 4.4.1, which encompass a better approach to visualization of state based models. Adhering to these design goals as the foundation of the Sterling architecture, described in Section 4.4.2, has made possible the development of new

and enhanced views, described in Section 4.4.3, that address shortcomings of the existing Alloy visualizations and enable an iterative modeling approach.

4.4.1 Design Goals

The following design goals define a good approach to instance visualization based on both the strengths and shortcomings of the existing visualizations, and from our experience modeling scientific software using Alloy. Drawing from research in the field of human-computer interaction, many of these design goals draw inspiration from the Thirteen Principles of Display Design [81], as indicated by *italics*.

- Based on the *principle of consistency*, a tool supporting Alloy visualization by extending existing functionality should provide a user interface similar to that of the Alloy Visualizer when possible.
- A tool should *minimize information access cost* by providing immediate visual feedback and minimizing the effort required to create legible visualizations.
- Based on the *principle of pictoral realism*, which states that a display should look like the variable that it represents, a tool should provide functionality for creating domain specific visualizations.
- A tool should natively support visualization of dynamic models by providing interactive control over projections and, based on the *principle of the moving part* should support animation and other methods to aid users in tracking changes between state.
- A tool should be extensible in terms of both visualization techniques and data providers. It should enable the rapid prototyping and development of visualization techniques in formal methods, and should not be strictly tied to a single formalism such as Alloy.

These design goals may be conflicting at times and may or may not be applicable in certain contexts. As such, the architecture of Sterling and the design of its individual components attempt to strike a functional balance among these goals to create an effective visualization tool.

4.4.2 Architecture and Design

Sterling is a React web application, packaged with a custom build of Alloy, that provides views of instances in the same vein as the Alloy Visualizer. Drawing on the successes of tools such as Alloy4Fun and BMotionWeb, a web-based platform was chosen due to the availability of robust data visualization and user interface libraries as well as the popularity of the JavaScript programming language. The custom build of Alloy, called Alloy Sterling, consists of two components: the Alloy IDE and the Sterling web application. From the user's perspective, writing models in Alloy Sterling is no different than writing models in Alloy. The Alloy Visualizer is still available and the

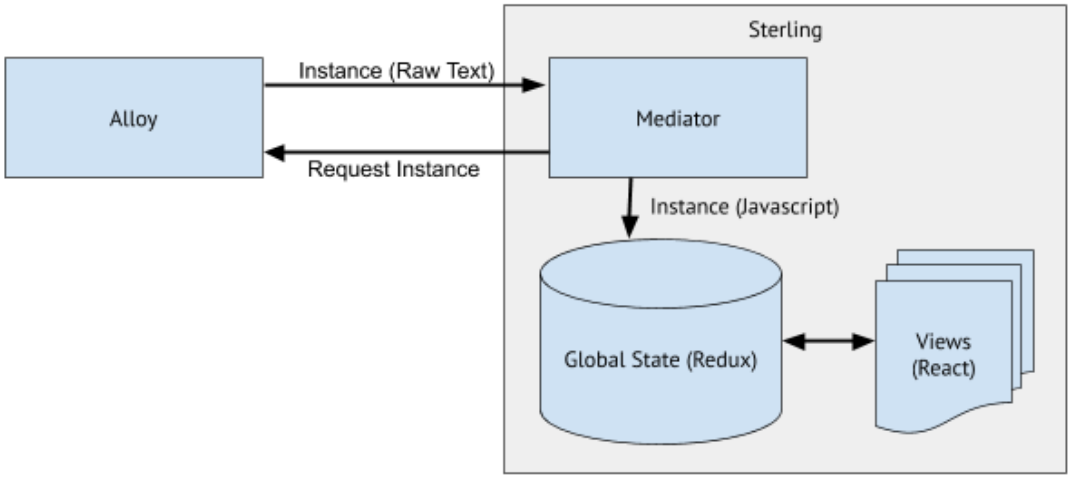


Figure 4.20 The Sterling communication architecture.

user has the option to choose either or both tools to display an instance or counterexample. For brevity, when we refer to Alloy in this Section, we mean this custom build of Alloy called Alloy Sterling.

Communication. A client-server relationship is established between Sterling and Alloy, respectively, by an embedded web server in Alloy, enabling two-way communication. Within Sterling, communication between Alloy and the individual Sterling views is managed using a mediator pattern. These communication patterns are depicted in Figure 4.20.

The mediator, which communicates with Alloy using the JavaScript WebSockets API, is responsible for sending requests to and handling responses from Alloy. Messages sent using WebSockets are text strings, and all communication between Sterling and Alloy adheres to the protocol outlined in Table 4.2.

Using this communication pattern, Sterling can request from Alloy the most recently generated instance or the result of evaluating an Alloy expression. Conversely, Sterling is always listening for messages from Alloy and will display the most recently received instance. This way the user can, for example, request the next solution from the Alloy IDE or the web browser, and Sterling will always display the correct instance.

Addressing our design goals, direct communication between Alloy and Sterling minimizes information access cost by enabling instant visual feedback when an instance or counterexample is generated. Furthermore, the mediator pattern provides flexibility in terms of data providers and visualizations. Regarding data providers, Alloy need not be the source of data so long as the data being supplied to Sterling adheres to the XML format of an Alloy instance and communication is in adherence with the protocol outlined above. As such, any model finder can be used to supply data for visualization. Indeed, an Alloy-like model finder called Forge, developed at Brown University for a Logic for Systems class, makes use of Sterling for visualizations. Regarding visualizations, Sterling views can be developed in isolation from concerns of communication or evolving data formats. Raw

Message protocol: Sterling → Alloy

Message	Description
`current`	Request the current instance
`next`	Request the next solution
`EVL:{id}:{expression}`	Request evaluation of an Alloy expression • `{id}` - a unique string identifier • `{expression}` - the expression to be evaluated

Message protocol: Alloy → Sterling

Message	Description
`EVL:{id}:{result}`	The result of evaluating an expression • `{id}` - the unique string identifier submitted with the request • `{result}` - the result of evaluating the submitted expression
`XML:{instance}`	An Alloy instance in XML format • `{instance}` - the XML data

Table 4.2 The Sterling communication protocol.

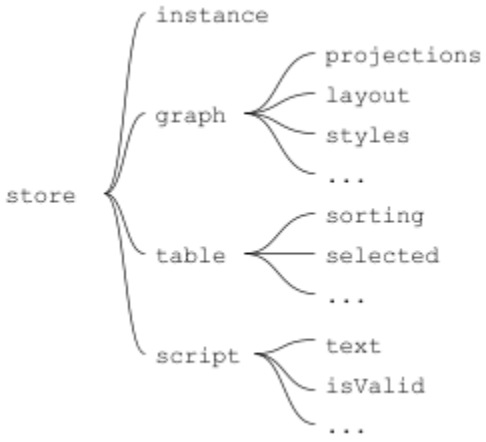


Figure 4.21 The Sterling application store.

instance data received by the mediator is translated to a common format that is distributed to each of the individual views for display.

User Interface. The Sterling user interface is implemented using React, a JavaScript library for building component-based declarative views. The React model encourages composition over inheritance, where individual visual components are assembled into a single, cohesive interface. As such, there is a rich ecosystem of component libraries. Sterling employs the Blueprint JS library for core UI components such as navigation bars, buttons, toggles, icons, and more.

In line with our design goals, the layout of the Sterling user interface intentionally mirrors that of the Alloy visualizer. As in Alloy, there is at most a single instance visible at any given time. The user can toggle between views using buttons in the navigation bar and request more solutions from the Alloy Analyzer using a "Next" button. Each Sterling view is independent of the others, containing its own settings, which are presented in a collapsible sidebar. As in Alloy, an evaluator in Sterling is used to evaluate individual expressions for the current instance in a REPL.

State Management. The application state of a visualization tool like Sterling, which provides numerous views and settings in an interactive user interface, is inherently complex. The nature of the tool gives rise to complex state dependencies that can be difficult to maintain as the tool grows. To manage application state, Sterling employs Redux, a JavaScript library which provides a predictable container, operating in a similar fashion to a reduce function. With Redux, the complete state of the Sterling application is stored in a single object tree called the store; this includes the current instance, the connection status with Alloy, and the complete state of all settings for each view. To change the application state, we specify mutations using actions that describe what we want to happen rather than directly performing mutations. Then, pure functions called reducers specify how each action transforms the entire state of the application.

The state tree is partitioned such that each view occupies a separate branch, as shown in Fig-

ure 4.21. When an instance is received by Sterling, the mediator parses the XML data and places the instance data into the application store using an action called `setInstance`. Each view, then, has registered a reducer function that specifies how that view's branch of the application store is transformed as a result of the `setInstance` action.

Redux was chosen because it enables the mediator pattern employed by Sterling and provides a scalable approach to application state, making Sterling a viable platform for rapid prototyping and development of new visualization techniques. As Redux provides a single source of truth for application state, adding interactive views and UI components is straightforward.

4.4.3 Views

Sterling provides three views: Graph, Table, and Script. The Graph and Table Views extend the functionality of their counterparts in the Alloy Visualizer, and the Script View is an original contribution that enables the creation of domain specific visualizations as part of the iterative design process. Here we discuss the implementation details for each of the three views.

Graph View. The Sterling Graph View provides all of the same functionality of the Alloy Graph View, but is supported by a few key extensions. Guided by our design goals, these extensions address numerous issues commonly discussed in the literature and those encountered in our experience modeling scientific software. Here we describe these extensions in detail.

First, graph elements are not restricted to rows; users may freely arrange graphical elements to make the display more readable. Second, the layout algorithm is not automatically executed when the projection is updated or when a new instance is generated, unless the graphs do not have any elements in common. As such, graphical elements remain static as users step through stateful models and generate instances. Furthermore, Sterling provides multiple configurable layout algorithms, including circle, grid, row, and Sugiyama style, that the user may choose to apply manually at any time. Finally, in an effort to address crowding issues created by fields with many tuples, the Graph View supports multiple methods for edge collapsing as well as semantic zooming. Edge collapsing allows a user to display multiple edges between the same two elements as a single edge with multiple labels. Semantic zooming, controlled by the scroll wheel, modifies the distance between graphical elements in much the same way that a map is zoomed, keeping their sizes and relative distances constant. Contrast this with geometric zooming, used in Alloy, which modifies the dimensions of graphical elements to achieve a zooming effect.

This combination of features results in a more interactive system when compared to the Alloy Graph View. In Sterling, the initial layout of the graph acts more as a "good" starting point rather than attempting to serve as the best possible arrangement, and repositioning atoms manually or by applying layouts is encouraged. Panning and zooming using the mouse encourages direct interaction with the graph and the combination of the two is particularly useful for exploring very large graphs. Contrast this with the Alloy Graph View, which has a more static feel despite including certain interactive features. The inability to move nodes outside of the rows in which they are initially

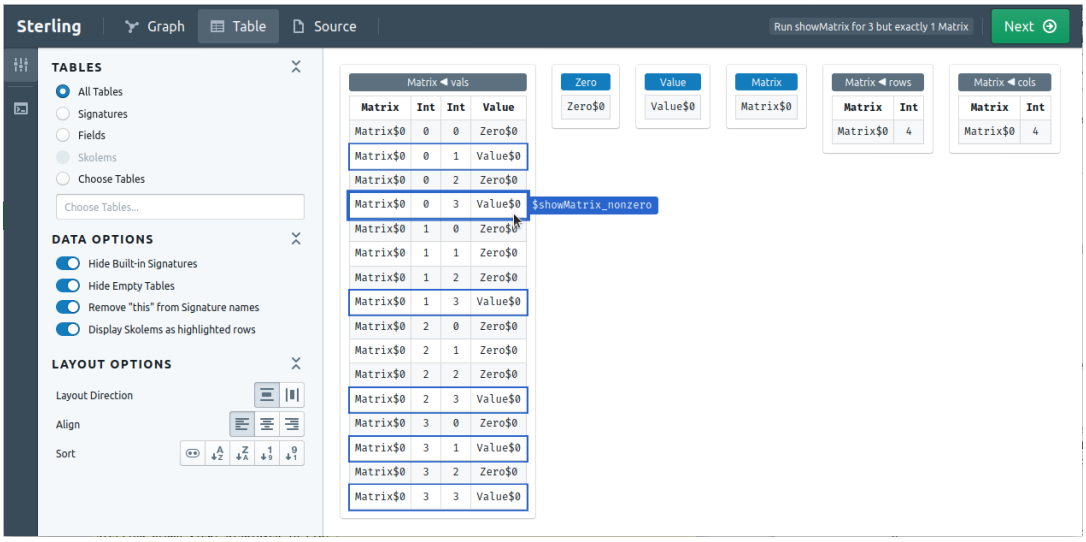


Figure 4.22 A dense matrix instance in the Sterling Table View.

placed discourages use in all but the most simple cases, and so the initial layout tends to be close to the best possible arrangement of atoms. Furthermore, zooming serves only to make the graph image larger in size, and panning is only supported when the image overflows the window. In practice we have found that, particularly as instances grow in size, the interactive nature of the Sterling Graph View facilitates a quicker understanding of the instances themselves, and therefore better supports the iterative design process.

Table View. Though rarely discussed in the literature, the Table View provides a unique and useful perspective of instance data. As we demonstrated in Section 4.2.5 by modeling execution traces of the matrix transpose operation, certain problems are better suited for display in a structured grid than in a directed graph.

In the Table View, all signatures and fields are displayed as tables. The Alloy Table View is basic, providing no functionality beyond display of simple tables. The Sterling Table View, on the other hand, provides features and tools to aid the user in exploring instance data more effectively. These include color categorization of tables by type, filtering, sorting, and alignment tools, and the ability to display skolemized variables (witnesses) as highlighted rows. This last feature has proven most useful, as it provides useful context for witnesses that are subsets of other relations. For example, Figure 4.22 shows an instance of our dense matrix model in the Table View. For this particular instance, we have also instructed Alloy to determine the subset of the values in the matrix that are nonzero using the existential quantifier. This subset is bound to a witness variable, `$showMatrix_nonzero`, which is highlighted in blue in the `vals` relation, rather than displayed as a separate table.

Script View. The Script View, a view introduced by Sterling, provides a scripting environment for the development of domain specific visualizations and the rapid prototyping of visualization techniques

that may see deeper integration into Sterling by improving existing views or creating new ones altogether. This environment is enabled by the combination of three components: a scripting language, a rendering stage, and the instance data. Here we introduce each component individually and then discuss how the view has found applications in scientific computing.

Scripts in the Script View are written in JavaScript using a built-in text editor that supports syntax highlighting and code completion. JavaScript was chosen as the preferred language due to its popularity, the large ecosystem of graphics rendering libraries, and the native, cross-platform execution environment provided by web browsers. Within the Script View, users have direct access to npm, a JavaScript package repository containing over 1.2 million JavaScript packages at the time of writing. Users specify the libraries they require from the settings sidebar and Sterling retrieves them from npm using the jsDelivr service.

The rendering stage, adjacent to the text editor, serves as a blank canvas onto which custom visualizations are rendered. Using a toggle button in the user interface, the user can choose from three different types of rendering stage: HTML div, SVG, and HTML Canvas. The choice of rendering platform is typically dependent on the type of visualization the user wishes to create, as each has its own strengths and weaknesses. The HTML div element is a generic container for HTML elements and can be used, for example, to create custom tables to display instance relations or add user interface components such as buttons and menus. SVG, or Scalable Vector Graphics, is a declarative, XML-based image format that supports 2D vector graphics. It provides convenient methods for drawing generic shapes such as rectangles, circles, and text, but also supports the creation of arbitrary shapes using paths. HTML Canvas, compared to SVG, provides a lower-level, imperative rendering API capable of rendering 2D and 3D graphics. The 2D rendering API supports only two basic shapes, rectangles and paths, and so visualizations are typically created by combining multiple paths using commands similar to those found in turtle graphics. The 3D rendering API exposes a WebGL context that the user can use to create interactive 3D visualizations. A 3D graphics library such as THREE.js is typically used to create visual elements, as the WebGL API is relatively low-level and does not provide methods for the creation of generic shapes or manipulation of 3D views.

A JavaScript library called alloy-ts, created specifically to support the Script View, provides the instance data, including all signatures, atoms, fields, and skolems, as individual variables that are named according to their counterparts in the Alloy instance. For example, an atom named `Matrix$0` in an instance will be exposed in the scripting environment as a variable named `Matrix$0` and a field named `vals` will be exposed as a variable named `vals`. Projections are fully supported in the Script View through the same UI components used to navigate projections in the Graph View, and the variables created by the alloy-ts library are automatically updated to reflect the current projection. Furthermore, the alloy-ts library provides a JavaScript API that is useful for exploring and manipulating instance data. As in Alloy, all variables created by the alloy-ts library are considered sets, deriving from a common class called `AlloySet`. This class, using JavaScript Proxies, redefines the behavior of the lookup/assignment operators, dot (.) and bracket ([]), to perform a join operation instead. For example, the expression `Matrix$0.vals` will perform a dot join, as it would in Alloy,

rather than attempt to look up the `vals` property of the `Matrix$0` object, as it would in vanilla JavaScript. This feature gives the scripting view a more Alloy-like feel and leads to less verbose scripts, particularly when chaining join operations, as the operators can be used in lieu of method calls.

Scripts written in the text editor are executed in a sandbox environment that contains, as global variables, a reference to the rendering stage, any libraries the user has requested, and the instance data from the alloy-ts library. After a script is edited, it must be manually executed by clicking the "Execute" button or by pressing `Ctrl+Enter`. If it runs without error, the script will be automatically rerun each time a new instance is generated or the projection is changed. As such, the Script View enables an iterative modeling process and supports visualization of stateful models.

As demonstrated in Section 4.2, the Sterling Script View promotes the iterative design process by giving the modeler tools to create domain specific visualizations. In doing so, the modeler is able to more easily extract relevant information from instances and counterexamples in order to identify modeling issues.

Outside of modeling sparse matrices, we have also leveraged the Script View for models of physical simulations. In a study investigating an extension made to an ocean circulation model, ADCIRC, the authors explore implementation choices and ensure soundness of the extension using Alloy [11]. In the models, finite element mesh topologies are represented using vertices and triangles as basic building blocks. While positional attributes such as nodal coordinates in three-dimensional space cannot be modeled, the allowable mesh topologies are constrained to include only those that have a planar embedding, ensuring that the mesh is physically meaningful. As such, a planar embedding of an instance is one possible realization of that mesh topology if physical coordinates were to be assigned to the mesh nodes. Inferring the planar embedding given only the topologies of the model relations, then, is perhaps the most difficult step in interpreting a mesh instance. Indeed, the authors state that "more than any extension to Alloy, what would have benefited our study most is a tool capable of automatically producing planar embeddings of meshes from Alloy instances, which proved to be tedious to do by hand."

Using the Script View, we extract topologies from mesh instances and automatically calculate and render a planar embedding. Because these views so closely match the actual physical layout of a mesh, the spatial relationships of an instance are intuitively obvious. These scripts have proven useful enough to warrant the development of a Mesh View in Sterling, shown in Figure 4.23. As dynamics are added to the model and the mesh representations are embellished with attributes such as water surface elevations and wet-dry status, deeper integration with Sterling makes exploring of instances more straightforward. Rather than manually edit variables in scripts, we use a purpose-built user interface to control visual properties of the mesh and control animations. In doing so we can, for example, visualize inundation produced by advancing storm surge fronts.

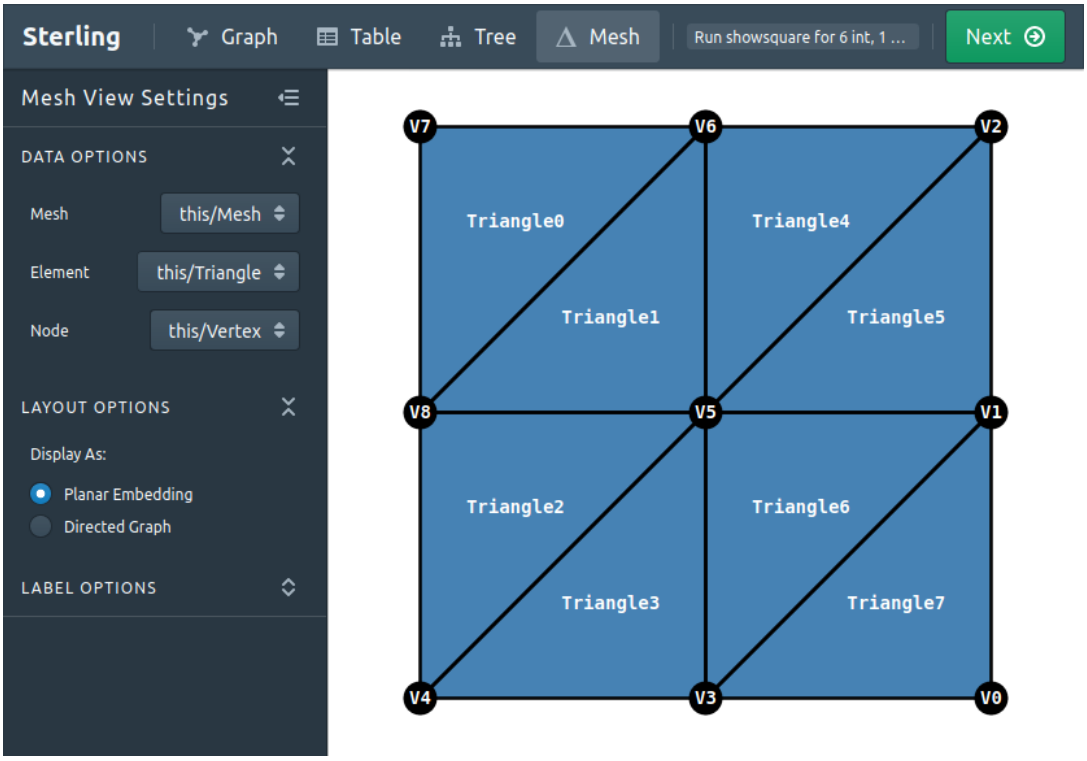


Figure 4.23 The Sterling Mesh View.

CHAPTER

5

CONCLUSIONS

The field of scientific computing faces many challenges related to reliability, reproducibility of results, and productivity. Not merely anecdotes, numerous empirical studies reveal cases of software "thwarting attempts at repetition or reproduction of scientific results" [74], and they make it clear that existing practices have led to a *productivity crisis* resulting in "frustratingly long and troubled software development times" [32]. While numerous approaches have been proposed to address the unique challenges facing the field of scientific computing, we believe that separating concerns, along the lines suggested here, should allow state-based methods to find productive use in a domain that could benefit from the kind of modeling and push-button analysis they provide.

In the presented research we demonstrate how and why state-based formal methods fit into the broader context of scientific computation, using Alloy to perform case studies. In these studies we identify challenges that stem from the adaptation of formal methods to our domain, and address them with new methods and tools that make the approach more accessible. General purpose tools like Alloy provide no out-of-the-box methods for modeling the types of bounded iteration commonly used in scientific computing, for instance, so we introduce a new idiom that makes reasoning about these types of computations possible. In addition, the visualizations generated by Alloy are not well suited to our domain, as they are not capable of capturing the types of spatial relationships that are characteristic of scientific software. These properties are critical to an *intuitive* understanding of the models we develop, so we introduce a web-based interface called Sterling that focuses on spatial relationships and facilitates the development of domain specific visualizations.

In terms of practicality, the methods and tools developed in this research are lightweight and automatic, making them convenient for spot checking and reasoning about portions of large-scale numerical code bases. By viewing comments in source code as a specification, for instance, we are

often able to verify an implementation in Fortran by building and checking a small Alloy model. A recent example of doing so in ADCIRC [60], a finite element code for simulating hurricane storm surge, was motivated by a desire to use different linear algebra solvers with different sparse matrix format requirements. Because of complex interactions between these formats and ADCIRC's custom assembly and boundary condition routines, lightweight models can be used to increase confidence in a new implementation. We have also applied these ideas to existing code used in production, and in one case found a discrepancy between an implementation that imposes boundary conditions and its purported behavior, based on comments that appear in the source code. Although further inspection revealed the flaw to be in the documentation itself and not the code, the ease with which this kind of spot checking can be performed provides some additional evidence of its usefulness.

Given the unique challenges and broad range of applications in scientific computing, though, we recognize that no single approach or tool is likely to bridge the communication gap that separates it from computer science. Rather, a wide variety of methods and development practices will need to contribute to the gradual improvement of the quality, reliability, and maintainability of scientific software. Some tools may prove more difficult than others to adapt to the domain, but the initial effort in these cases is an important step towards making the approaches more accessible, and therefore able to be used in practice by scientists and engineers.

Regarding future work, development of Sterling is ongoing, and will continue at Brown University through a postdoctoral research opportunity in the Department of Computer Science. This activity will include work on both the visualization interface for Alloy, as well as a new, primary visualization interface for a state-based teaching tool called Forge, developed at Brown. These research directions were initiated in spring 2020 in the context of a Logic for Systems course, taught at Brown, and future directions will focus on leveraging user studies to better understand the role of visualization and user interaction in state-based modeling. Additional directions include continued development of the custom Mesh View to support models of finite element software, development of "syntactic sugar" to make certain elements of the approach more convenient, e.g., use of the tabular idiom, and the application of techniques from the operations research community like Modeling to Generate Alternatives [20] to enhance user-guided exploration of a model's solution space.

BIBLIOGRAPHY

[1] Alloy models from the paper, https://people.engr.ncsu.edu/jwb/alloy/.

[2] Abadi, M. & Lamport, L., "The existence of refinement mappings," *Theoretical Computer Science*, **82**, no. 2, pp. 253–284, 1991.

[3] Ackroyd, K. S., Kinder, S. H., Mant, G. R., Miller, M. C., Ramsdale, C. A. & Stephenson, P. C., "Scientific software development at a research facility," *IEEE Software*, **25**, no. 4, pp. 44–51, 2008.

[4] Altuntas, A. & Baugh, J., "Adaptive subdomain modeling: A multi-analysis technique for ocean circulation models," *Ocean Modelling*, **115**, pp. 86–104, 2017, https://doi.org/10.1016/j.ocemod.2017.05.009.

[5] ——, "Hybrid theorem proving as a lightweight method for verifying numerical software," *Proceedings of the Second International Workshop on Software Correctness for HPC Applications, Correctness'18*, Dallas, TX, USA: IEEE, 2018, pp. 1–8.

[6] Andoni, A., Daniliuc, D., Khurshid, S. & Marinov, D., "Evaluating the 'small scope hypothesis'," *POPL '02: Proceedings of the 29th ACM Symposium on the Principles of Programming Languages*, 2002.

[7] Arnold, G., Hölzl, J., Sinan Köksal, A., Bodík, R. & Sagiv, M., "Specifying and verifying sparse matrix codes," *ACM SIGPLAN Notices*, **45**, pp. 249–260, 2010.

[8] Bartlett, R. A., "Integration strategies for computational science engineering software," *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, 2009, pp. 35–42.

[9] Baudet, G. M., "Asynchronous iterative methods for multiprocessors," *Journal of the ACM (JACM)*, **25**, no. 2, pp. 226–244, 1978.

[10] Baugh, J. & Altuntas, A., "Modeling a discrete wet-dry algorithm for hurricane storm surge in Alloy," *Abstract State Machines, Alloy, B, TLA, VDM, and Z: 5th International Conference, ABZ 2016*, Lecture Notes in Computer Science 9675, Springer, 2016, pp. 256–261.

[11] ——, "Formal methods and finite element analysis of hurricane storm surge: A case study in software verification," *Science of Computer Programming*, **158**, pp. 100–121, 2018.

[12] Baugh, J. & Dyer, T., "State-based formal methods in scientific computation," *Abstract State Machines, Alloy, B, TLA, VDM, and Z: 6th International Conference, ABZ 2018*, Lecture Notes in Computer Science 10817, Springer, 2018, pp. 392–396.

[13] Baugh, J. & Liu, S., "A general characterization of the Hardy Cross method as sequential and multiprocess algorithms," *Structures*, **6**, pp. 170–181, 2016.

[14] Bertsekas, D. P., "Distributed asynchronous computation of fixed points," *Mathematical Programming*, **27**, no. 1, pp. 107–120, 1983.

[15] Bientinesi, P., Gunnels, J. A., Myers, M. E., Quintana-Ortí, E. S. & Geijn, R. A., "The science of deriving dense linear algebra algorithms," *ACM Transactions on Mathematical Software (TOMS)*, **31**, no. 1, pp. 1–26, 2005.

[16] Biere, A., "Lingeling, Plingeling and Treengeling entering the SAT competition 2013: Solver and benchmark descriptions," *Proceedings of SAT competition*, **B-2013-1, University of Helsinki**, 2013.

[17] Bolton, C., "Using the Alloy Analyzer to verify data refinement in Z," *Electronic Notes in Theoretical Computer Science*, **137**, no. 2, pp. 23–44, 2005.

[18] Borgida, A., Mylopoulos, J. & Reiter, R., "On the frame problem in procedure specifications," *IEEE Transactions on Software Engineering*, **21**, no. 10, pp. 785–798, 1995.

[19] Boyatt, R. & Sinclair, J., "Experiences of teaching a lightweight formal method," 2008.

[20] Brill, E. D., Flach, J. M., Hopkins, L. D. & Ranjithan, S, "MGA: A decision support system for complex, incompletely defined problems," *IEEE Transactions on Systems, Man, and Cybernetics*, **20**, no. 4, pp. 745–757, 1990.

[21] Brooks, F. J., "No silver bullet: Essence and accidents of software engineering," *Computer*, **20**, pp. 10–19, 1987.

[22] Brooks, F. P., *The Mythical Man-Month (Anniversary Ed.)* USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[23] Brunel, J., Chemouil, D., Cunha, A. & Macedo, N., "The Electrum Analyzer: Model checking relational first-order temporal specifications," *33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)*, ACM Press, 2018.

[24] Carver, J., Heaton, D., Hochstein, L. & Bartlett, R., "Self-perceptions about software engineering: A survey of scientists and engineers," *Computing in Science Engineering*, **15**, no. 1, pp. 7–11, 2013.

[25] Carver, J. C., "Report: The Second International Workshop on Software Engineering for CSE," *Computing in Science & Engineering*, **11**, no. 6, pp. 14–19, 2009.

[26] Chadha, H. S. & Baugh, J. W., "Network-distributed finite element analysis," *Advances in Engineering Software*, **25**, no. 2-3, pp. 267–280, 1996.

[27] Collberg, C., Proebsting, T., Moraila, G., Shi, Z. & Warren, A. M., "Measuring reproducibility in computer systems research," Tech. Rep., 2014.

[28] Couto, R. *et al.*, "Improving the visualization of Alloy instances," *Electronic Proceedings in Theoretical Computer Science*, **284**, 37–52, 2018.

[29] Cross, H., "Analysis of continuous frames by distributing fixed-end moments," *Proceedings of the American Society of Civil Engineers*, 1930, pp. 919–928.

[30] Dijkstra, E. W., Feijen, W. H. J. & Van Gasteren, A. J. M., "Derivation of a termination detection algorithm for distributed computations," *Information Processing Letters*, **16**, pp. 217–219, 1983.

[31] Dyer, T., Altuntas, A. & Baugh, J., "Bounded verification of sparse matrix computations," *Proceedings of the Third International Workshop on Software Correctness for HPC Applications, Correctness'19*, Denver, CO, USA: IEEE/ACM, 2019, pp. 36–43.

[32] Faulk, S., Loh, E., Van De Vanter, M. L., Squires, S. & Votta, L. G., "Scientific computing's productivity gridlock: How software engineering can help," *Computing in Science & Engineering*, **11**, no. 6, pp. 30–39, 2009.

[33] Franz, M., Lopes, C. T., Huck, G., Dong, Y., Sumer, O. & Bader, G. D., "Cytoscape.js: A graph theory library for visualisation and analysis," *Bioinformatics*, **32**, no. 2, pp. 309–311, 2015. eprint: `https://academic.oup.com/bioinformatics/article-pdf/32/2/309/6689178/btv557.pdf`.

[34] Gammaitoni, L. & Kelsen, P., "Domain-specific visualization of alloy instances," *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, Ait Ameur, Y. & Schewe, K.-D., Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 324–327.

[35] Geleßus, D. & Leuschel, M., "ProB and Jupyter for logic, set theory, theoretical computer science and formal methods," *Rigorous State-Based Methods*, Raschke, A., Méry, D. & Houdek, F., Eds., Cham: Springer International Publishing, 2020, pp. 248–254.

[36] Golub, G. & Van Loan, C., *Matrix Computations*, fourth. Johns Hopkins University Press, 2013.

[37] Graf, S. & Saïdi, H., "Construction of abstract state graphs with PVS," *International Conference on Computer Aided Verification*, Springer, 1997, pp. 72–83.

[38] Hannay, J., MacLeod, C., Singer, J., Langtangen, H. P., Pfahl, D. & Wilson, G., "How do scientists develop and use scientific software?" *Software Engineering for Computational Science and Engineering, ICSE Workshop on*, **0**, pp. 1–8, 2009.

[39] Hatton, L., "The chimera of software quality," *Computer*, **40**, no. 8, 2007.

[40] Hatton, L. & Roberts, A., "How accurate is scientific software?" *IEEE Transactions on Software Engineering*, **20**, no. 10, pp. 785–797, 1994.

[41] Hoang, T. S., Fürst, A. & Abrial, J.-R., "Event-B patterns and their tool support," *Software & Systems Modeling*, **12**, no. 2, pp. 229–244, 2013.

[42] Hoare, C. A. R., "Proof of correctness of data representations," *Acta Informatica*, **1**, no. 4, pp. 271–281, 1972.

[43] Hudak, P., "Arrays, non-determinism, side-effects, and parallelism: A functional perspective," *Workshop on Graph Reduction*, Springer, 1986, pp. 312–327.

[44] Hughes, J., "Why functional programming matters," *The Computer Journal*, **32**, no. 2, pp. 98–107, 1989.

[45] Jackson, D., "Object models as heap invariants," *Programming Methodology*, Springer, 2003, pp. 247–268.

[46] ——, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2012.

[47] Jackson, D. & Damon, C. A., "Elements of style: Analyzing a software design feature with a counterexample detector," *IEEE Transactions on Software Engineering*, **22**, no. 7, pp. 484–495, 1996.

[48] Jackson, D. & Wing, J., "Lightweight formal methods," *Computer*, **29**, no. 4, p. 21, 1996.

[49] Kelly, D. F., "A software chasm: Software engineering and scientific computing," *IEEE Software*, **24**, no. 6, pp. 120–119, 2007.

[50] Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B. E., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J. B., Grout, J., Corlay, S., *et al.*, "Jupyter Notebooks—a publishing format for reproducible computational workflows," *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, IOS Press, 2016, pp. 87–90.

[51] Kotlyar, V., Pingali, K. & Stodghill, P., "A relational approach to the compilation of sparse matrix programs," *Euro-Par'97, European Conference on Parallel Processing*, Springer, 1997, pp. 318–327.

[52] Ladenberger, L., Bendisposto, J. & Leuschel, M., "Visualising Event-B models with B-Motion Studio," *Formal Methods for Industrial Critical Systems*, Alpuente, M., Cook, B. & Joubert, C., Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 202–204.

[53] Ladenberger, L. & Leuschel, M., "BMotionWeb: A tool for rapid creation of formal prototypes," *Software Engineering and Formal Methods*, Springer International Publishing, 2016, pp. 403–417.

[54] Lamport, L., "Processes are in the eye of the beholder," *Theoretical Computer Science*, **179**, no. 1, pp. 333–351, 1997.

[55] ——, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[56] Launchbury, J. & Jones, S. L. P., "State in Haskell," *Lisp and Symbolic Computation*, **8**, no. 4, pp. 293–341, 1995.

[57] Leuschel, M. & Butler, M., "ProB: An automated analysis toolset for the B method," *International Journal on Software Tools for Technology Transfer*, **10**, no. 2, pp. 185–203, 2008.

[58] Li, G., Palmer, R., DeLisi, M., Gopalakrishnan, G. & Kirby, R. M., "Formal specification of MPI 2.0: Case study in specifying a practical concurrent programming API," *Science of Computer Programming*, **76**, no. 2, p. 65, 2011.

[59] Linz, P., "A critique of numerical analysis," *Bulletin of the American Mathematical Society*, **19**, no. 2, pp. 407–416, 1988.

[60] Luettich, R. A. & Westerink, J. J., *Formulation and numerical implementation of the 2D/3D ADCIRC finite element model version 44. XX*. 2004, https://adcirc.org/files/2018/11/adcirc_theory_2004_12_08.pdf.

[61] Macedo, N., Brunel, J., Chemouil, D., Cunha, A. & Kuperberg, D., "Lightweight specification and analysis of dynamic systems with rich configurations," *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016, New York, NY, USA: Association for Computing Machinery, 2016, 373–383.

[62] Macedo, N., Cunha, A., Pereira, J., Carvalho, R., Silva, R., Paiva, A. C. R., Ramalho, M. S. & Silva, D., "Experiences on teaching alloy with an automated assessment platform," *Rigorous State-Based Methods*, Raschke, A., Méry, D. & Houdek, F., Eds., Cham: Springer International Publishing, 2020, pp. 61–77.

[63] Macedo, N. *et al.*, "Sharing and learning Alloy on the web," *ArXiv*, **abs/1907.02275**, 2019.

[64] Masci, P., Oladimeji, P., Zhang, Y., Jones, P., Curzon, P. & Thimbleby, H., "PVSio-web 2.0: Joining PVS to HCI," *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, Kroening, D. & Păsăreanu, C. S., Eds. Springer International Publishing, 2015, pp. 470–478.

[65] Misue, K., Eades, P., Lai, W. & Sugiyama, K., "Layout adjustment and the mental map," *Journal of Visual Languages & Computing*, **6**, no. 2, pp. 183–210, 1995.

[66] Morgan, C., *Programming from Specifications*. Prentice Hall, 1994.

[67] Nikhil, R. S., *ID Reference Manual, Version 90.1*, Computation Structures Group Memo 284-2, MIT, 1991.

[68] Rayside, D., Chang, F. S.-H., Dennis, G., Seater, R. & Jackson, D., "Automatic visualization of relational logic models," *ECEASST*, **7**, 2007.

[69] Reynolds, J. C., *The Craft of Programming*. Prentice Hall PTR, 1981.

[70] Sanders, R. & Kelly, D., "Dealing with risk in scientific software development," *IEEE Software*, **25**, no. 4, pp. 21–28, 2008.

[71] Schrage, L., *LINDO: An Optimization Modeling System*, Fourth. The Scientific Press, 1991.

[72] Siegel, S. F., Mironova, A., Avrunin, G. S. & Clarke, L. A., "Combining symbolic execution with model checking to verify parallel numerical programs," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, **17**, no. 2, 10:1–10:34, 2008.

[73] Sörensson, N. & Een, N., "Minisat v1.13 – a SAT solver with conflict-clause minimization," *SAT 2005 Competition*, 2005.

[74] Storer, T., "Bridging the chasm: A survey of software engineering practice in scientific programming," *ACM Computing Surveys (CSUR)*, **50**, no. 4, 47:1–47:32, 2017.

[75] Sugiyama, K., Tagawa, S. & Toda, M., "Methods for visual understanding of hierarchical system structures," *IEEE Transactions on Systems, Man, and Cybernetics*, **11**, no. 2, pp. 109–125, 1981.

[76] Sullivan, A., Wang, K., Khurshid, S. & Marinov, D., "Evaluating state modeling techniques in Alloy," *SQAMIA*, 2017.

[77] Umarji, M., Seaman, C., Koru, A. G. & Liu, H., "Software engineering education for bioinformatics," *Software Engineering Education and Training, 2009. CSEET'09. 22nd Conference on*, IEEE, 2009, pp. 216–223.

[78] Wadler, P., "A new array operation," *Workshop on Graph Reduction*, Springer, 1986, pp. 328–335.

[79] Werth, M. & Leuschel, M., "VisB: A lightweight tool to visualize formal models with SVG graphics," *Rigorous State-Based Methods*, Raschke, A., Méry, D. & Houdek, F., Eds., Cham: Springer International Publishing, 2020, pp. 260–265.

[80] West, H. H., *Analysis of Structures: An Integration of Classical and Modern Methods*. John Wiley & Sons, 1980.

[81] Wickens, C. D., Lee, J., Liu, Y. D. & Gordon-Becker, S., *Introduction to Human Factors Engineering (2nd Edition)*. USA: Prentice-Hall, Inc., 2003.

[82] Wilson, G. V., "Where's the real bottleneck in scientific computing?" *American Scientist*, **94**, no. 1, pp. 5–6, 2006.

[83] Wood, W. A. & Kleb, W. L., "Exploring XP for scientific research," *IEEE Software*, **20**, no. 3, pp. 30–36, 2003.

[84] Woodcock, J. & Davies, J., *Using Z: Specification, Refinement, and Proof*. Prentice Hall International, 1996.

[85] Zaman, A. *et al.*, "Improved visualization of relational logic models," University of Waterloo, Tech. Rep. CS-2013-04, 2013.

[86] Zave, P., "A practical comparison of Alloy and Spin," *Formal Aspects of Computing*, **27**, no. 2, pp. 239–253, 2015.